Automated Support for the Elaboration and Analysis of Goal-Use Case Integration Models

by

Tuong Huan Nguyen

Thesis submitted in fulfillment
of the requirements for the degree of
Doctor of Philosophy

Faculty of Science, Engineering and Technology

Swinburne University of Technology

2015

Abstract

Combining goal and use case modeling has been recognized as a key approach for understanding and analyzing software requirements specifications. However, there are a number of challenges that prevent the successful application of such approach. First, little consensus exists among the current goal and use case integrated modeling (GUIM) approaches regarding the types of artifacts and relationships to be modeled, making it difficult for requirements engineers to decide what to capture and how to specify them in a goal and use case integrated model. Second, while natural language requirements documents are critical source of goals and use cases, manually analyzing such documents to identify these artifacts is a time-consuming and error-prone process due to the complexity of natural languages. Third, the existing GUIM approaches generally offer limited capability of analyzing the models of goals and use cases for the 3Cs problems (problems regarding the consistency, completes, and correctness of such models). Such limitations pose a serious threat of producing problematic requirements specifications.

To address the above discuss problems, we introduce a novel **G**oal-**U**se Case **I**ntegration **F**ramework (GUI-F) that supports the modeling and analysis of goal and use case integrated models. GUI-F consists of a meta-model that serves as a conceptual foundation for goal and use case integrated modeling. It provides a comprehensive classification of artifacts and relationships in a goal-use case model, and specifies the constraints and dependencies between them. It also offers a set of specification rules that govern how these defined artifacts should be specified.

GUI-F also offers a set of rule-based techniques to provide automated support for extracting goal-use case integrated models from unconstrained textual requirements documents. Our extraction techniques are able to automatically locate artifacts and relationships buried in text, ensure the extract artifact specifications to be well-formed, and classify the artifacts into appropriate categories.

Moreover, GUI-F provides an ontology-based technique that automates the detection and resolution of the 3Cs problems in goal-use case models. Such technique relies on the use of ontologies to capture domain knowledge and semantics, functional

grammar for semantically parameterizing artifact specifications, and our developed meta-model for constraints and dependencies between artifacts and relationships.

We have conducted a number of cases studies and benchmark validations to evaluate the performance of all GUI-F components. The positive results obtained from these evaluations indicated GUI-F could effectively support requirements engineers in modeling and analyzing goal-use case integrated models.

Acknowledgement

<A nice acknowledgement to be filled in here>

Declaration

This is to certify that this thesis contains no material which has been accepted for the award of any other degree or diploma and that to the best of my knowledge this thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis. Where the work is based on joint research or publications, I have disclosed the relative contributions of the respective workers or authors.

Tuong Huan Nguyen

Table of Contents

List of Figures

List of Tables

List of Publications

The following publications are derived from the research reported in this thesis:

[1] **Tuong Huan Nguyen**, John Grundy, and Mohamed Almorsy, "Integrating Goal-oriented and Use Case-based Requirements Engineering: The Missing Link", ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems. Ottawa, 2015, ACM/IEEE.

[2] **Tuong Huan Nguyen**, John Grundy, and Mohamed Almorsy, "Rule-Based Extraction of Goal-Use Case Models from Text", 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Bergamo, 2015, ACM.

[3] **Tuong Huan Nguyen**, John Grundy, and Mohamed Almorsy, "Ontology-based automated support for goal–use case model analysis", Software Quality Journal, 23(3), p.1-39, 2015.

[4] **Tuong Huan Nguyen**, John Grundy, and Mohamed Almorsy, "GUITAR: An ontology-based automated requirements analysis tool", Requirements Engineering Conference (RE), Karlskrona, 2014, IEEE.

The following publications were published during my PhD, however are not directly related to the research reported in this thesis:

[1] **Tuong Huan Nguyen**, Bao Vo, Markus Lumpe, and John Grundy, "KBRE: a framework for knowledge-based requirements engineering", Software Quality Journal, 22(1): p. 87-119, 2014.

[2] **Tuong Huan Nguyen**, Bao Vo, Markus Lumpe, and John Grundy, "REInDetector: a framework for knowledge-based requirements engineering", IEEE/ACM International Conference on Automated Software Engineering, 2012, ACM.

Chapter 1

Introduction

1.1 Goal and Use Case Integrated Modeling in Requirements Engineering

Requirements Engineering (RE) is normally the first step in the software development life cycle. It is an iterative process of eliciting, structuring, specifying, analyzing, and managing requirements of a software system [151]. The ultimate objective of this process is to ensure the requirements of stakeholders in a software system are correctly gathered, understood, and agreed on. The agreement between stakeholders and the development team regarding the functionalities and constraints of the system are often documented in a textual software requirements specification document (SRS) [163].

Scenario and Use case-driven Requirements Engineering (UDRE) [33, 155] have been recognized as key approaches for capturing, understanding, analyzing requirements and facilitating early system designs. Use cases and scenarios, with the emphasis on the interactions between actors and system to achieve specific tasks, are considered important requirements artifacts. UDRE offers an approach in which stakeholders can express requirements in terms of operations and processes, and hence closer to their way of conceptualizing a system [7]. In addition, by identifying and independently analyzing different use cases, requirements engineers may focus on one narrow aspect of the system usage at a time, and thus cope with the complexity of the requirements

analysis process. However, UDRE suffers from a number of limitations. First, use cases (and scenarios[1]) pertain to examples and illustrations (i.e., they describe specific cases in which a system is used), and thus need to be generalized to obtain more complete requirements [133]. Second, use cases are fragmented in the sense that different single uses of the system are described while the underlying rationale and relationships between them are not always known or expressed.

Goal-oriented Requirements Engineering (GORE) [6, 23, 49, 161, 178] is another key approach in RE. GORE is concerned with the use of goals as the centralized concepts in RE activities [161]. The idea of GORE is derived from the recognition that goals are the root from that all requirements of the software system are defined. A goal is generally defined as a high-level objective regarding what the software system is expected to achieve. In a goal model, each goal (called parent-goal) should be refined by other more specific goals (called sub-goals), which means the parent-goal describes the underlying objective or rationale for having the sub-goals while the sub-goals provide details as to how the parent-goal can be achieved in the system. The modeling of such refinement relationships provides an important means for capturing, structuring, understanding, and analyzing requirements.

Based on this concept, GORE helps overcome the problems of requirements fragmentation (i.e., each piece of requirement describes an example/illustration of system use without much connection between them) and lack of rationale capture in UDRE. However, GORE also has its own limitations. First, domain experts often find it hard to deal with fuzzy concepts of goals [6]. Second, many goals do not readily exist in practice, and the discovery and refinement of them can be challenging (i.e., interviewed stakeholders are not aware of certain goals) [133]. Finally, it is often not easy for stakeholders to express goals at the required level of abstraction [164].

Due to the strengths and weaknesses of both UDRE and GORE, it can be seen that either of them alone is not adequate to comprehensively support the RE process. To overcome this problem, Potts [123] suggested that the two concepts should be combined because they complement each other. In fact, the aspects in which GORE has

[1] In RE, a scenario is a sequence of interactions between an actor and a system to accomplish a certain task. A collection of scenarios is a use case.

limitations are where UDRE's benefits come from, and vice versa. Many researchers share this view and have developed methods for combining goals and use cases together to leverage the benefits while overcoming the limitations of either approach [7, 76, 90, 91, 97, 133, 154, 164]. Such methods are referred to as Goal-Use Case Integration Modeling (GUIM) in our work. The combination enables the use of goals to guide the elicitation and structuring of use cases while having use cases to operationalize goals [7, 76, 90, 91, 97, 133, 154, 164]. GUIM methods also help achieving greater modeling strengths that are not achievable with each approach alone [7, 76, 90, 91, 97, 133, 154, 164].

1.2 Research Motivation

Despite of the benefits achievable from goal and use case integrated modeling, there are a number of key challenges that prevent the successful application of GUIM approaches in practice. Below, we briefly discuss these challenges.

Lack of a Conceptual Foundation to Guide the Integrated Modeling of Goal and Use Cases

Little consensus exists among the current GUIM approaches in regards to what should be modeled in a general goal and use case integrated model. For instance, in order to support the modeling of a goal and use case integrated model, requirements engineers should be given guidance as to what types of artifacts should be modeled, how they are specified, classified, and connected to each other. However, our literature review indicates that the existing GUIM techniques are generally isolated. In fact, different approaches, with different foci, have different ways to specify, classify, and connect artifacts (i.e., goals, use cases). For instance, Cockburn [26] defines the summary, user, and sub-function levels of abstraction for goals and use cases while Rolland et al. [133] categorize goals into business goals, design goals, service goals, and system goals. Meanwhile, Lee et al. [91] classify goals under three facets: rigid vs. soft goals, actor-specific vs. system-specific goals and functional vs. non-functional goals; and use cases can be connected to goals in the intersecting type of functional, actor-specific and rigid. Such isolation creates the confusion regarding what to model, and also makes it difficult if models created in different approaches were to be combined.

Moreover, no existing GUIM approach is comprehensive enough to model both goals and use cases as each lacks support for certain artifact types and relationships. For instance, the approaches developed by Cockburn [26] and Rolland et al. [133] do not tackle the modeling and classification of non-functional goals while functional goals are not modeled in Supakkul and Chung's work [154]. In addition, although these approaches were proposed to combine goals and use cases, the dependency between goals and use cases in these works is loose since the operationalization relationship (i.e., a use case/scenario operationalizes a goal) is the only one dependency between goals and use cases modeled in most approaches. There are also other important dependencies between integrated goals and use cases that should be modeled as well. For example, there needs to be a decomposition of non-functional constraints from the goal level to the use case level.

From these problems, it can be seen that requirements engineers do not have sufficient support when modeling goals and use cases together. The problems also suggest that there should be a conceptual foundation that can provide guidelines as to what types of artifacts should be captured in general goal and use case integrated models, what relationships and dependencies these artifacts should have, and how they should be specified.

Lack of Automated Support for the Creation of Goal and Use Case Integrated Models from Text

Since SRSs are usually the documentation of the system's objectives, functionalities, and constraints acquired in each iteration of the RE process [163], they provide a valuable source from which goals and use cases can be acquired (besides other important sources such as stakeholder interview transcripts, legacy systems' documents). However, extracting and modeling goals and use cases from SRSs is a non-trivial process. In fact, in textual requirements documents, goals are normally buried among other (non-goal) sentences and written in unstructured styles. Furthermore, frequently, descriptions of use cases are not clear in requirements documents. Multiple use case steps are commonly combined together as one (i.e., by conjunction). Moreover, data or non-functional constraints are often mixed up with use case steps, making it hard for requirements engineers to locate the information they need.

Due to such complexities, manual goal and use case identification and modeling can be a tedious, time-consuming, and error-prone process, especially for inexperienced requirements engineers and large requirements documents. Automated support is thus needed to assist requirements engineers with the extraction process. However, our literature review indicates that no existing information extraction technique is capable of comprehensively supporting the creation of goal and use case integrated models.

Lack of Automated Support for Analyzing Goal and Use Case Integrated Models for the 3Cs Problems

While analysis is a critical task to verify and improve the quality of requirements, the lack of agreed types of artifacts to be modeled in GUIM approaches prevents goal-use case integrated models being adequately analyzed. Such models need to be analyzed for defects such as incompleteness, inconsistency, and incorrectness (the 3Cs problems). Although some GUIM approaches have their own ways of analysis, they do not sufficiently address key questions such as: how to verify if an artifact is not properly specified? How to check if artifacts are not correctly connected? How to ensure a use case is matched with its associated goal? How to detect if a required artifact has not been elicited?

Moreover, although various requirements analysis techniques exist, ranging from formal methods [111, 165], natural language [51, 152] to ontology-based techniques [72, 128], our investigation pointed out that they suffer from one or more of the following problems (a detailed discussion about these problems is provided in Chapter 2):

- Lack of comprehensive support for the analysis of goal-use case models: most automated analysis techniques have been proposed for tackling problems in goal models (i.e., [161], [165], [49]), use case/scenario models (i.e., [111], [59]) or general requirements (i.e., [51], [152]). They thus are not able to validate the alignment between goals and use cases. In addition, many of the above questions are not handled in these approaches.
- Require the use of complicated requirements specification languages: most formal approaches to automated analysis of requirements require the use of complicated formal modeling languages (i.e., [161], [165], [49], [20]). However, many researchers have criticized that formal languages are not suitable for

practical use since they are generally hard to read and write, and getting adequate, consistent, and complete specifications requires considerable expertise and training [43, 149].

- Lack of support for identifying semantic-related problems: identifying semantic-related incompleteness, incorrectness and inconsistency in goal-use case models requires the meaning of terms and knowledge in the requirements domain to be captured and used by computers in the analysis processes. Without such information, various types of 3Cs problems are not identifiable. For instance, consider the use case describing the interaction between a user and the system for creating a review article. Assume that banned users are not allowed to create any review article in the domain and that no exception (or pre-condition) has been created in the use case to handle this matter, then the use case specification is incomplete. However, such incompleteness would not be identifiable if the domain knowledge about the restriction in creating review articles is unknown. A number of ontology-based approaches have been proposed to tackle semantic requirements defects [72, 128, 145], however, these techniques are neither adequately mature nor suited to goal-use case model analysis (i.e., focus on limited range of problems and/or target only general requirements).

1.3 Research Objectives

From the discussed motivation, we aim to develop a framework that comprehensively supports the elaboration and analysis of goal and use case integrated models. Specifically, the objectives of our research are as follows:

- ***RO1:*** *To develop a conceptual foundation for goal and use case integrated modeling*. Such a conceptual foundation should be able to provide guidance as to what types of artifacts should be modeled, classified, and connected to each other in a general goal-use case integrated model. It also should provide rules or guidance as to how artifacts should be specified.
- ***RO2:*** *To develop a technique to automatically extract goal and use case integrated models from textual requirements documents*. Such a technique should be able to analyze text to identify artifacts and relationships between these artifacts. It also needs to classify the identified artifacts according to the

artifact and relationship categorization developed for objective RO1. Moreover, the technique should be able to automatically ensure such artifacts to be properly specified (according to the specification rules defined for objective RO1).

- ***RO3:*** *To provide a technique to automatically analyze goal and use case integrated models for inconsistency, incompleteness, and incorrectness*. Such a technique should allow users to work directly with natural languages. In other words, it should provide a method to interpret and analyze textual artifact specifications. Moreover, the technique should be able to automatically incorporate domain-specific knowledge and semantics into the analysis of goal-use case models. Furthermore, it should also provide explanations to identified problems and effectively suggest resolution options to such problems.

1.4 Research Contributions

In this research, we have developed a goal-use case integration framework that provides semi-automated support for the modeling and analysis of goal and use case models specified in natural language. Below, we summarize the key contributions of our research in the area of Requirements Engineering.

A Meta-Model for Goal and Use Case Integrated Modeling

We have developed a novel meta-model for goal and use case integrated modeling (called *GUIMeta*) to serve as a conceptual foundation for modeling goals and use cases together. GUIMeta was proposed to address the problem that, there is no consensus among existing goal and use case integrated modeling approaches regarding what concepts and relationships to capture, and how goal-use case model can be systematically analyzed.

GUIMeta contains two layers. The *artifact layer* provides a comprehensive classification of goals and structure of use cases, and defines relationships and constraints between them. The *specification layer* provides specification rules for these defined artifacts. We adopted an approach to semantically parameterizing the textual artifact specifications using functional grammar [40]. Functional grammar provides a method to decompose a textual specification into different parts (called *semantic functions*); each of them has a unique semantic role (e.g., *agent*, *object*, *beneficiary*).

Such parameterization provides a consistent way to interpret the semantic of each group of words in a specification. This functional grammar-styled parameterization offers a basis on which the artifact specification rules are defined. For instance, each rule offers the guidelines as to what semantic functions should and should not be included in specifications of each type of artifacts.

The combination of such artifact classification, functional grammar-styled parameterization, and specification rules provides a framework that governs specification and analysis of goal and use case integrated models. Moreover, we have also established the correspondence between GUIMeta's artifacts and those in other GUIM approaches to enable the unification of models specified in different approaches. GUIMeta, accompanied with its evaluation results, were published in the MODELS 2015 conference [109]. The details of GUIMeta are provided in Chapter 4 in this thesis.

A Rule-based Technique for Automated Extraction of Goal-Use Case Integrated Model from Requirements Documents

We have developed a novel rule-based technique to semi-automatically extract goal and use case integrated models from uncontrolled text with the objective of minimizing manual efforts in identifying goals and use cases from requirements documents. Our approach is based on linguistic analysis techniques and a set of extendable extraction rules that helps identify goals, use cases and their relationships from text.

The key novelty of our approach is that, it is the first technique, to the best of our knowledge, to combine the syntactic and semantic aspects of text in specifying extraction rules to identify necessary details while ignoring unneeded content from uncontrolled natural language text. The fact that our technique can work directly with uncontrolled natural language text potentially enhances its applicability, compared to a number of existing information extraction approaches in requirements engineering that rely on constrained input text. Importantly, our technique is accompanied with a clearly defined rule syntax and parser and thus, makes it possible for users to modify and extend the extraction rule collection.

In addition, our technique is able to ensure the extracted artifacts to be well-formed according to the specification rules defined in our meta-model GUIMeta. Moreover, such extracted artifacts can also be automatically classified, allowing the most activities

in the entire extraction process to be automated (only some manual pre-processing is needed). The entire extraction approach, together with its evaluation results, were published in the FSE 2015 conference [108]. They are introduced in Chapter 5 and 6.

An Ontology-based Technique for Automated Analysis of Goal-Use Case Integrated Model

We have developed a novel ontology-based technique to semi-automatically analyze natural language-based goal and use case integrated models for inconsistency, incompleteness, and incorrectness (most tasks are fully automated, except for problems resolutions that need users to manually make selections). Our approach goes beyond the existing ontology-based requirements analysis techniques by offering a method to incorporate sophisticated domain knowledge into the analysis process (not just domain-specific terms and their dependencies). Moreover, the use of functional grammar allows textual goal and use case specifications to be semantically parameterized. Such parameterization provides a consistent way to interpret the semantics of each group of words in a textual specification. Individual words in a specification are then linked to their corresponding terms in the ontologies, allowing the entire semantics of textual specifications to be captured, represented, and interpreted during the analysis process.

In addition, the parameterized specifications can be transformed into Manchester OWL Syntax [62] statements, making it possible to integrate them with ontologies to perform analysis. The fact that all of these parameterization and transformation can be automated by our techniques enables requirements engineers to work directly with natural language-based specifications with minimal required expertise of functional grammar and Manchester OWL Syntax. Moreover, our technique allows the automated generation of comprehensive explanations for detected problems and is capable of suggesting repairing alternatives. Our analysis technique has been published in the RE 2014 conference [106] and SQJ Journal [107]. It is discussed in Chapter 7.

1.5 Thesis Organization

The remaining chapters of this thesis are organized as follows.

Chapter 2: Literature Review – In this chapter, we present some background on RE and how the concepts of goals and use cases are used in RE. We also discuss our literature

review on goal and use case integrated modeling, automated requirements extraction, and automated requirements analysis. From our discussion, we identify the key research gaps that this thesis then addresses.

Chapter 3: Approach – In this chapter, we provide a motivating scenario to illustrate the research problems we target to address in this thesis. It is then followed by our problem analysis to establish the requirements for possible solutions to such problems. We then describe the approach that we took to fulfill these requirements. We also present the overview of our entire approach to automated extraction and analysis of goal and use case integrated models. Our evaluation approach, accompanied with an introduction to the case studies used in our validations, are also provided in this chapter.

Chapter 4: Goal and Use Case Integration Meta-model – In this chapter, we present our meta-model for goal and use case integrated modeling. We discuss its components, including the artifacts, relationships, and the constraints and dependencies between them. We also provide a background on functional grammar and its use in semantically parameterizing artifact specifications. Based on that, we present our artifact specification rules. Additionally, our evaluation for GUIMeta and its results are also discussed.

Chapter 5: Rule-based Goal and Use Case Integrated Model Extraction – In this chapter, we present our rule-based technique to identify goal-use case model artifacts and relationships from text. We first provide a background on natural language processing techniques used in our work. Based on the background, we provide details about the extraction rules, including their syntax, their components, and how they can be used to locate artifacts and relationships.

Chapter 6: Artifact Specification Polishing and Classification – In this chapter, we discuss our modifying rules that are used to ensure the well-formedness of extracted artifact specifications. We also discuss the mechanism behind Mallet classifier and its limitations, and our extensions to adopt Mallet for classifying artifact specifications in goal and use case integrated models. Moreover, we discuss the evaluations conducted to validate our entire extraction approach.

Chapter 7: Ontology-based Goal-Use Case Model Analysis – In this chapter, we introduce our ontology-based technique to analyze goal-use case integrated models for incompleteness, inconsistency, and incorrectness. We first present the structure of ontology in our work to capture domain-specific knowledge and semantics, accompanied with some discussion regarding the quality assurance of ontologies. We then provide the classification of the 3Cs problems in our work and discuss various algorithms used to identify such problems. Several evaluations conducted to assess the performance of our analysis techniques, together with their results, are also provided.

Chapter 8: Conclusion and Future Work – This chapter concludes the thesis. It provides a summary of the key contributions that we have made in our research. It also discusses the limitations of our work, and the future work that we plan to carry out.

Chapter 2

Background and Literature Review

In this chapter, we present some background on requirements engineering and how the concepts of goals and use cases are used in requirements engineering. Since the focus of this research is on the extraction of goal and use case integrated models from text and the analysis of such models for inconsistency, incorrectness and incompleteness (the 3Cs problems), our literature review covers a discussion on state-of-the-art in the areas of goal-use case integration, automated requirements extraction, and automated requirements analysis. In this discussion, we describe the existing approaches to requirements extraction and analysis, and identify their benefits and limitations. From this, we identify the key research gaps that this thesis then addresses. In the following sections, when the term "requirement" is used, it is understood as a requirement for a software system.

2.1 Requirements

According to Kotonya and Sommerville [84], *"requirements are defined during the early stages of a system development as a specification of what should be implemented. They are descriptions of how the system should behave, constraints on the system's operation, or specifications of a system property or attribute. Sometimes they are constraints on the development process of the system."* Preferably, a requirement should specify what the system should satisfy instead of how it should be done. In other words, a requirement should contain a description about a functionality or constraint of the

system and it should not provide information as to how the system should be designed to satisfy such functionality or constraint.

Based on the characteristics of requirements, they are commonly classified into two categories: functional requirements which specify the functionalities of a system, and non-functional requirements which define the constraints on the way a system satisfies its functional requirements, or on the way it should be developed [163].

Since requirements serve as the input into the later stages of a system development [172], it is critical to ensure that the requirements of a system are of high quality. In fact, requirements should be assessed against a variety of qualities, including consistency, completeness, correctness, unambiguity, verifiability, and so on [30].

In practice, requirements can take various forms, depending on the system development methodology used, or the concern over the requirements at different stages during the system development. Unrestricted natural language is the primary means for specifying requirements since it is easy to write and understanding by human, and thus no special training is required. However, one of the key problems with natural language requirements specifications is that, it is sometimes ambiguous, making a single requirement interpreted differently by different people [163].

To avoid such a problem, in many cases, requirements can be found written in controlled natural languages [48] which specify rules on how requirements should be written. In addition, to facilitate the automated analysis and verification of requirements, they can be specified in formal languages that can be interpreted, processed, and reasoned by computers [49, 165, 173]. Moreover, requirements can also be represented with diagrammatic notations, each with its own focus and benefits. Some examples include context diagrams, frame diagrams, entity-relationship diagrams, use cases and scenarios, sequence diagram, goal models, and so on [163].

2.2 Requirements Engineering

We start with one of the oldest definitions of requirements [137]:

> *"Requirements definition is a careful assessment of the need that a system is to fulfill. It must say why a system is needed, based on current or foreseen*

> *conditions, which may be internal operations or external market. It must say what system features will serve and satisfy this context."*

It implies that requirements are concerned with the objectives of a software system, the functionalities it must have to satisfy such objectives, and the constraints on how the software must be designed and implemented [85].

Zave [180] put forward a related definition of RE:

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

It can be seen that, the key concerns of RE are the functions and constraints of a software system being considered, and the goals and objectives behind these functions and constraints.

Pohl's definition [120] further clarifies these points:

> *"Requirements engineering is the process of eliciting individual stakeholder requirements and needs and developing them into detailed, agreed requirements documented and specified in such a way that they can serve as the basis for all other system development activities"*

This definition makes it clear that the functions and constraints of a software system must come from the elicited requirements and needs of its stakeholders. It also carries the important characteristics that such requirements should be agreed on and documented, and that such requirements will serve as the input for the next activities in the software development life cycle (e.g., design, implementation, testing).

In line with this definition, Sommerville [151] takes a step further to note that "*in practice, requirements engineering is an iterative process."* An important characteristic of RE is that, it is an iterative process instead of a one-time task. In this process, there are a number of interleaved activities. These activities depend on the application

domain, the people involved, and the organization developing the requirements. However, the following generic activities are common to all RE processes: requirements elicitation, requirements analysis, requirements validation and requirements management [151]. We discuss a typical RE process in the following sub-section.

2.2.1 Requirements Engineering Process

Figure 2-1 presents the spiral requirements engineering process recommended by Van Lamsweerde [163]. According to the figure, each spiral starts with the domain understanding and requirements elicitation phase in which requirements engineers study the domain in which the system-as-is will operate and gather requirements from system stakeholders (1). The elicited requirements are then analyzed for problems, and negotiated with stakeholders if necessary (2). The analyzed and agreed requirements are then specified and documented in a requirements document (3). Such a requirements document is then verified to ensure its quality (4). Those steps can be repeated in another spiral in case changes need to be made in the requirements. Below, we discuss the requirements engineering process in details.

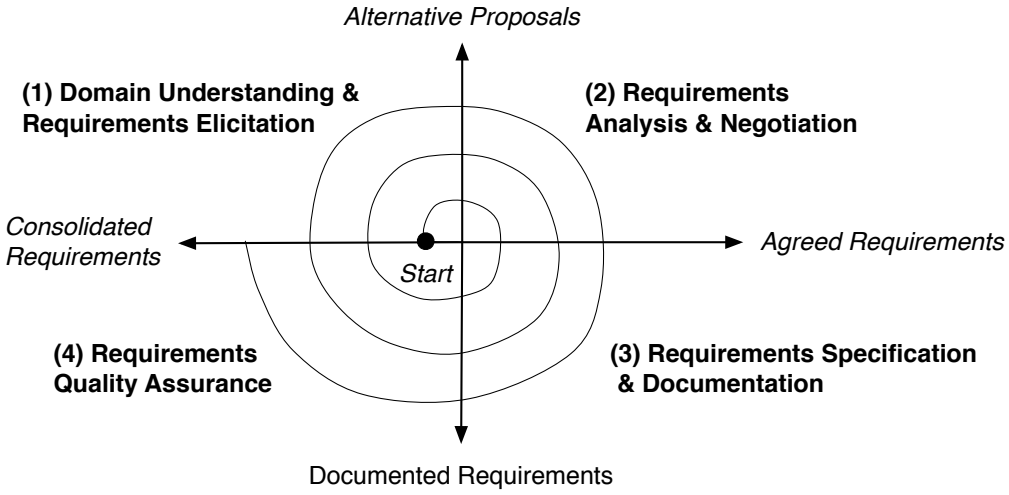


Figure 2-1. The Requirements Engineering Process [163]

Domain Understanding and Requirements Elicitation

Domain understanding and requirements elicitation is the process of seeking, uncovering, acquiring, and elaborating requirements for software systems [182]. Requirements elicitation focuses on learning and understanding the needs and requirements of the

stakeholders of the system, with the ultimate objective is to communicate these requirements to the system developers. Typically, this process involves the intertwined activities such as domain study, stakeholder analysis, and requirements identification.

Requirements Analysis and Negotiation

Requirements analysis and requirements negotiation are closely related activities in requirements engineering. Requirements analysis aims to get deeper understanding in the elicited requirements and identify problems within such requirements [84, 163]. Negotiations between requirements engineers and stakeholders are usually involved whenever problems are detected in the analysis to ensure requirements are correctly collected and understood. After such negotiations, new or updated requirements may be obtained. Some of the key concerns in requirements analysis include consistency and completeness checking, necessity checking, feasibility checking, risk analysis, and alternative discovery [163].

Requirements Specification and Documentation

After the requirements have been elicited, analyzed and confirmed with stakeholders, they need to be correctly specified and documented. This phase is generally intertwined with the previous ones. For instance, it may start as soon as some elicited materials are obtained and agreed [163]. Therefore, the input into the requirements specification phrase generally contains a collection of agreed statements in different types: general system objectives, system requirements, user requirements, environmental assumptions, domain properties and concept definitions. The output of this phrase is the early version of the requirements document of the system.

Requirements Quality Assurance

The requirements quality phase is aimed to ensure the items specified in the requirements documents meet various qualities including completeness, consistency, adequacy, unambiguity, measurability, etc. The requirements quality phrase consists of the following steps: detecting defects, reporting defects, analyzing their causes, and undertaking appropriate actions to fix them [163]. The outcome of this phase is a consolidated requirements document in which defects have been fixed.

Requirements Evolution

The term "requirements evolution" here comes from van Lamsweerde's terminologies [163]. It matches the term "requirements management" in Sommerville's work [151]. RE is not a one-time task. It is necessarily an iterative process since requirements of a system are not static. Instead, they evolve. In fact, requirements and assumptions that have been elicited, analyzed, specified and documented may need to be changed for a variety of reasons. For instance, there may be defects in the requirements document to be fixed, project fluctuation in terms of priorities and constraints, environmental changes, or new needs arrive. The objective of the requirements evolution phase is to handle such changes at various stages of the project to ensure the set of requirements is of high quality. Requirements evolution is at the heart of the requirements engineering process as it can trigger a new cycle in the spiral process shown in Figure 2-1.

2.2.2 Requirement Modeling

Requirements modeling play a central role in requirements engineering. It is defined as the process of *"building abstract descriptions of the requirements that are amenable to interpretation"* [116]. The existing system/organization as well as the possible alternative configurations for the system-to-be are typically modeled [85]. These models serve as a basic common interface to the various RE activities described above [160].

There are several benefits of modeling requirements. First, modeling supports requirements elicitation by helping requirements engineers figure out what types of requirements to be obtained, or surface hidden requirements [116]. Second, modeling can help requirements engineers measure the progress of requirements elicitation. For instance, if the model of current requirements is not complete, it is likely that there should be more requirements to acquire. Third, modeling can help reveal problems in requirements. For example, if an inconsistency is found in the model, it may indicate conflicting and/or infeasible requirements [85]. Kof [82] also highlighted that system modeling is a primary means to identify problems in natural language-based requirements. Fourth, modeling can help getting deeper understanding of the elicited requirements. For instance, goal models enable requirements engineers to focus on the "why" aspects of requirements and thus help them understand the underlying objectives of and relationships between requirements [176]. Models also provide a basis for

requirements documentation and evolution. While informal models are analyzed by humans, formal models of system requirements allow for precise analysis by both the software tools and humans [85].

Since our research focus is on the modeling of requirements using goals and use cases, we discuss the use of goals and use cases in requirements engineering in the following sub-sections.

2.2.3 Goal-oriented Requirements Engineering

The major disadvantage of traditional requirements engineering approaches is its inadequacy in dealing with complex software systems [38, 136, 139]. These approaches simply consider requirements as processes and data and do not capture the rationale behind them, thus making it difficult to understand requirements with respect to some high-level concerns in the problem domain [85]. Moreover, most techniques focus on modeling and specification of the software alone and therefore, lack of support for reasoning about the composite system comprised of the system-to-be and its environment [139].

The Goal-oriented Requirements Engineering (GORE) approach [6, 23, 49, 161, 178] was developed to overcome the above discussed problems. GORE is concerned with the use of goals for eliciting, elaborating, structuring, specifying, analyzing, negotiating, documenting, and modifying requirements [161]. The idea of GORE is derived from the recognition that goals are the root from that all requirements of the software system are defined. In fact, the current system under consideration is typically analyzed in its organizational, operational and technical settings. As soon as problems are pointed out and opportunities are identified, high-level goals are identified and refined to address such problems and meet the opportunities; requirements are then elaborated to meet those goals [161]. Requirements are then refined into more specific requirements to meet the higher-level requirements. Other requirements would then be generated by repeating this process. GORE addresses the issues of lacking of requirements' rationale by providing both top-down and bottom-up links between requirements so that from a specific requirement, it is possible to point out the underlining reason of it (through the links to higher-level requirements) and also how it is realized by the system being developed (through the links to lower-level requirements).

The main claimed benefits of GORE include:

- A goal model provides traceability links from high-level strategic goals to low-level technical requirements [177].
- Goals are normally more stable than the requirements to achieve them [8].
- Goal refinement provides a natural mechanism for structuring complex requirements documents for increased readability [161].
- Goals provide a criterion for verifying the completeness of requirements. For instance, the specification is complete with respect to a set of goals if all the goals can be proved to be achieved from the specification and the properties known about the domain considered [179].
- Goals help avoiding irrelevant requirements. For instance, if a requirement is identified not contributing to a satisfaction of any goal, it can be considered irrelevant [179].
- GORE provides a means for identifying and exploring alternative system proposals [160].

Despite of the above benefits, GORE suffers from a number of limitations including:

- Goal discovery is a complicated task since practical experience shows that goals are not readily given in software projects [6, 133].
- Domain experts often find it hard to deal with fuzzy concepts of goals [16].
- It is often not easy for stakeholders to express goals at the required level of abstraction [164]
- The goal modeling process is initiated by collecting enterprise goals. However, these goals often do not reflect the actual situation, but instead an idealized environmental one. Therefore, this may lead to ineffective requirements [133].

2.2.4 Use case-driven Requirements Engineering

Use case-driven requirements engineering (UDRE) (i.e., [33, 53, 155, 156]), also referred to as scenario-driven requirements engineering, is another approach to requirements elicitation and modeling based on the analysis of the wider context in which the system will operate [132]. It focuses on capturing examples, scenes, narrative descriptions of contexts, use cases and illustrations of agent behaviors [134]. At the core

of this approach is the concepts of use cases and scenarios. Use case is defined as a collection of possible sequences of interactions between the system under discussion and its actors, related to a particular goal. Scenario is one of the sequences of interactions in a use case [26]. An actor of a use case can be a system, or a person, a group of people who have a specific role played in the system [33]. From this definition, a use case can also be understood as a specific way of using the system by using some parts of its functionality [68]. In practice, a use case can be specified in various forms including narrative text, structured text, images, animation or simulations [97].

The application of use cases in requirements capturing and analysis was first introduced by Jacobson [68]. After that, they have become a key component in RE since use cases (and scenarios) naturally facilitate the communication between requirements analysts and stakeholders [33]. In fact, use cases offer an approach in which stakeholders can express requirements in terms of operations and processes, and hence closer to their way of conceptualizing a system [7].

Various benefits that can be achieved by the UDRE approach, including: eliciting requirements in envisioned situations [124], discovery of exceptional cases [124, 156], deriving conceptual object-oriented models [33, 69, 138, 139], understanding user needs through scenario prototyping [65], to reason about design decisions [158] and so on. In comparison with goals, typically use cases and scenarios are easier to get in the first place. Goals can be made explicit only after deeper understanding of the system has been gained [134]. In addition, according to an industrial practice survey within the CREWS project, scenarios are useful in particular when abstract modeling fails [170].

However, UDRE also has its own problems. First, use cases (and scenarios) pertain to examples and illustrations, and thus need to be generalized to obtain complete requirements [133, 155]. Second, use cases are fragmented in the sense that different single uses of the system are described [97] while the underlying rationale and relationships between them are not always known or expressed. Although each use case is associated to a goal, tracking and identifying the relationships between these goals are not a focus in UDRE approaches. Third, since use cases cannot be tightly related to each other, it is difficult to structure and manage use case models, especially with large systems [93, 131].

2.2.5 Combining Goal and Use Case Modeling in Requirements Engineering

From the the earlier discussions, it can be seen that although both GORE and UDRE approaches possess important capabilities, either of them is not adequate to comprehensively support the requirements engineering process.

To overcome this problem, Potts [123] suggested that it is *"unwise to apply goal-based requirements methods in isolation"* and that the two concepts should be combined because they complement each other. In fact, the aspects in which GORE has limitations are where UDRE's benefits come from, and vice versa. Many researchers share this view and have developed methods to integrate goals and use cases together [7, 76, 90, 91, 97, 133, 154, 164]. These attempts can be classified into two tightly related categories: the works that used goals and use cases (or scenarios) to derive each other to improve the requirements elicitation process, and the work that combined goals and use cases (or scenarios) to comprehensively model and analyze requirements. Below, we discuss both categories of goal and use case integration approaches.

Coupling goals and use cases for requirements elicitation

Most approaches to coupling goal and use case modeling commonly share the idea of having goal elaboration and use case elicitation are intertwined processes. This means a concrete scenario description may prompt the elicitation of its underlining goals and a goal specification may prompt the elaboration of scenario descriptions that illustrate or validate it.

Rolland et al. [133] were among the first researchers who focused on this stream of research. They developed a technique called L'Ecritoire for guiding the elicitation of goals using scenarios based on the concept of requirement chunks. Each chunk contains a goal and a scenario in which the scenario operationalizes its associated goal. Thus, a requirement chunk is considered a possible way of achieving a goal. There are three abstraction types of requirement chunks: contextual, system interaction, and system internal level and four abstraction types of goals: business, design, service, and system goals. At each level of requirement chunks, a goal is operationalized into a scenario and thus a new chunk is formed. An appropriate scenario step is then selected as a goal and

operationalized further. The definition of abstraction levels is one of the important strengths of this work. It helps provide requirements engineers with a means to differentiate artifacts on different levels of abstraction, and suggests the idea of when the top-down operationalization process can be terminated. Following this approach, the goals and scenarios of the system-to-be would be elicited. This work was then extended by Kim et al. [76] who developed guidelines for transferring goals and scenarios into use case models. However, these works only targeted functional goals and did not support non-functional goals.

Sharing this view, Watahiki and Saeki [169] developed a method to guide the requirements elicitation using both goals and use cases. In this work, similar to Rolland et al.'s way of operationalizing goals using requirement chunk, each step in a use case is considered as a goal for the next round of operationalization. The authors also proposed the use of constraints which are linked to goals to express their non-functional concerns.

In the context of the KAOS framework, van Lamsweerde and Willemet [164] proposed a way to use scenarios as typical examples of system usage. In this work, scenarios are represented using the semi-formal event trace diagram notations. Based on that, a formal method, which is based on temporal logic, is used to infer additional goals and requirements in the KAOS language.

Integrating goals and use cases for comprehensive requirements modeling and analysis

Various techniques in this stream of research have been proposed. Most of them share the idea of combining the concepts of goals and use cases to have their modeling strengths to complement each other. Such a combination is intended to obtain some modeling capability which is not achievable with each concept individually.

Cockburn [26] suggested the use of goals as a way to structure use cases. He defines three abstraction levels of artifacts. Summary goals are for system-level objectives. User goals specify user tasks in the system. Sub-functions refine user goals and can be steps in the use cases that operationalize user goals. Such structure allows use cases to be managed and traced effectively, especially in large systems. Non-functional goals are kept out of scope of this work.

Lee et al. [91] developed another approach called *goal-driven use cases* to structure use cases by goals, handle non-functional goals and analyze interactions between goals and use cases. They defined a number of artifacts and relationships. For instance, goals are classified in 3 facets: *rigid* vs. *soft*, *system-specific* vs. *actor-specific* and *functional* vs. *non-functional*. A use case that achieves an original goal (defined as the intersection of rigid, actor-specific and functional goals) is called an original use case while extension use cases are those maintain or optimize *soft*, *system-specific* or *non-functional goals* that constrain original goals. However, this work does not provide the internal structure of use cases and thus it is not clear how extension use cases and original use cases are different. Moreover, except for *constrain* relationships, other interactions between goals are not defined.

Santander and Castro [141] introduced an approach to integrate the i* modeling framework with use cases. In this work, the i* framework [175], with the support of the strategic dependency model and the rationale dependency model, is used to capture the rationale behind requirements and the task decomposition in a system. Use cases are then used to model the sequence of actions between actors and the system to accomplish the tasks. The authors also provide a number of heuristics to support the deriving of use cases from i* models.

Supakkul and Chung [154] proposed an approach to better integrate functional and non-functional requirements (NFR). In this work, use cases, which capture functional requirements, are associated to non-functional goals specified using the NFR framework [23]. To facilitate the integration, a set of NFR association points was proposed. For instance, an *actor association* point is used to link an actor of a use case to his/her desired non-functional properties of the system while a use case association point is used to link a use case with its required non-functional constraints. The approach also provides a set of propagation rules to enable the traceability across a goal-use case model. The limitation of this work is its lack of support for functional goals, and thus the interactions between functional goals, non-functional goals and their related use cases cannot be captured.

Kim et al. [77] proposed a multi-view approach to requirements modeling using goals and scenarios. Their approach was based on the argument that a multi-view approach

would facilitate the gathering, organization, analysis, and understanding of the various dimensions of requirements of a large complex system. It consists of four views of requirements: structure view, abstraction view, function view, and quality view. The structure view is achieved by capturing the "business description" of the system. A "business description" in this work is a group of identified agents, activities, objects, events, candidate goals, and structured objects. The abstraction view is captured by the top-down and bottom-up refinement relationships between artifacts. The function view is achieved by the system-user interactions within scenarios. The quality view is achieved by connecting functional goals to quality attributes that the goals need to satisfy. Besides the original top-down refinement process from goals to scenarios, this work also provided some guidance on the bottom-up process in which goals can be derived by the scenarios that operationalize them.

Liu and Yu [97] put forward a combined use of goals and use cases to represent design knowledge of web-based systems. In this work, goals and use cases are visualized using the Goal Requirement Language (GRL) [166] and the Use Case Map (UCM) [5], respectively. The authors have demonstrated a great combination of GRL and UCM in which GRL provides a rich set of elements (i.e., actor, task, resource, belief) to model the intents, motivations and rationales of requirements while UCM is responsible for visualizing the behavioral aspects of the designed system. Based on such a combination, the approach facilitates the transition from high-level system objectives to specific requirements and to high-level design. It also provides a comprehensive view of the entire system requirements and early design, which enables requirements engineers to review and possibly discover new requirements of the system.

2.2.6 Limitations of Current Goal-Use Case Integration Approaches

In this section, we discuss the limitations of the current goal and use case integration modeling (GUIM) approaches from the perspective of modeling and analyzing goal-use case integrated models specified in natural language, which is the primary means of requirements specification in practice [98].

Firstly, little consensus exists among the current GUIM approaches in regards to what to be modeled in a general goal and use case integrated model. For instance, in order to

support the modeling of a goal and use case integrated model, requirements engineers should be given guidance as to what types of artifacts should be modeled, how they are specified, classified, and connected to each other. However, according to our above discussion, it can be seen that the existing GUIM techniques are generally isolated. In fact, different approaches, with different foci, have different ways to specify, classify and connect artifacts (i.e., goals, use cases). For instance, Cockburn [26] defines the summary, user and sub-function levels of abstraction for goals and use cases while Rolland et al. [133] categorize goals into business goals, design goals, service goals, and system goals. Meanwhile, Lee et al. [91] classify goals under three facets: rigid vs. soft goals, actor-specific vs. system-specific goals and functional vs. non-functional goals; and use cases can be connected to goals in the intersecting type of functional, actor-specific and rigid. Such isolation makes it difficult if models created in different approaches were to be combined.

Moreover, no existing GUIM approach is comprehensive enough to model both goals and use cases as each lacks support for certain artifact types. For instance, the approaches developed by Cockburn [26], Rolland et al. [133], and Kim et al. [76] do not tackle the modeling and classification of non-functional goals while functional goals are not a focus in Supakkul and Chung's work [154]. In addition, the work of Kim et al. [77] lacks the classification of goals and their relationships. Moreover, the work of Santander and Castro [141], which is based on the i* modeling framework, does not support the categorization of goals across levels of abstraction as provided in the work of Rolland et al. [133] or Cockburn [26]. Such abstraction-based goal classification plays an important role in guiding the goal operationalization process [132, 162].

Second, although the above approaches were proposed to combine goals and use cases (or scenarios), the dependency between goals and use cases (or scenarios) in these works is loose. In fact, the operationalization relationship (i.e., a use case/scenario operationalizes a goal) is the only one dependency between goals and use cases in most of these approaches. In our view, more dependencies between integrated goals and use cases should be modeled. For instance, there needs to be a decomposition of constraints from the goal levels to the use case level.

Figure 2-2 shows an example which demonstrates this point. In this example, the goal G1 (*"travelers shall be able to write reviews"*) has a non-functional constraint which is C1 (*"travelers should be able to write review quickly."* G1 is operationalized by a use case UC1 which specifies the interactions between a traveler and the system to have a review created. The step 7 (*"system validates the review"*) in this use case has a constraint which is *"the validation should be completed within 1 second."* In this case, it is necessary to model the relationship between C1 and SC1 (i.e., SC1 is a refinement of C1) since SC1 can be understood as a lower-level constraint which helps satisfying the higher-level constraint C1 (both are concerned with speed).

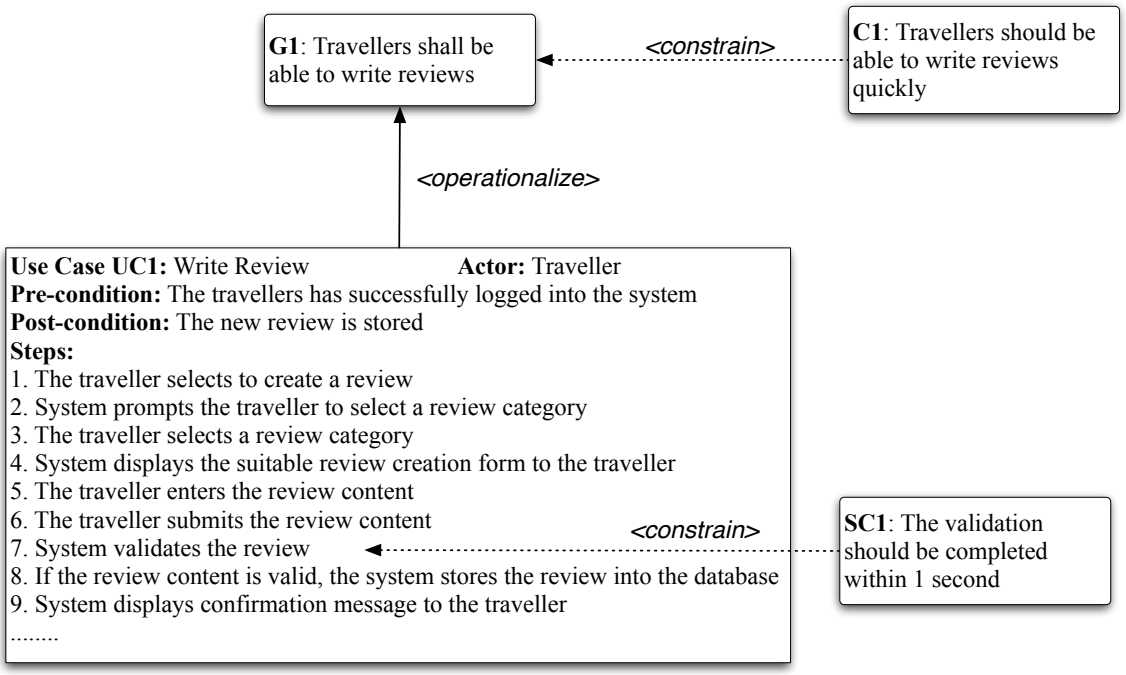


Figure 2-2. Example of Cross-level Non-Functional Constraint Dependency

Supakkul and Chung [154] also share the idea of linking non-functional constraints to functional goals. However, their study does not cover the dependency between constraints on goal levels and and those on use case levels.

Third, the lack of agreed types of artifacts to be modeled prevents goal-use case models to be adequately analyzed. Such models need to be analyzed for defects such as incompleteness, inconsistency, and incorrectness (the 3Cs problems). Although some GUIM approaches have their own ways of analysis, they do not sufficiently address key questions such as: how to verify if an artifact is not properly specified? How to check if artifacts are not correctly connected? How to ensure a use case is matched with its

associated goal? How to detect if a required artifact has not been elicited? To address these questions, an approach needs to satisfy the following 4 requirements:

(1) Providing the definitions of what types of artifacts to be model
(2) Providing guidelines or rules that suggest how artifacts of a specific type should be specified so they can be verified for correctness
(3) Providing the definitions of what dependencies/relationships can/must be specified between particular types of artifacts
(4) Providing the definitions and characterization of 3Cs problems that may exist in a goal-use case model (i.e., how to define, classify, and detect them). This requirement can be achieved if the other 3 requirements are me because if it is known what types of artifacts to model and how they should be specified, then it it possible to identify what types of defects a model of them could have in terms of the 3Cs.

No existing goal and use case integration approach satisfies all these four requirements. For instance, although Rolland et al. [133] provides the classification of goals and some templates for writing goals, such templates are general and remain on a high level and thus do not particularly guide how each type of goals (i.e., business goal, design goal) should be specified. Similarly, Cockburn provides a general template for writing use case steps while no template provided for goals. Moreover, although KAOS [164] provides powerful formal methods for detecting obstacles and conflicts in goal models, it does not provide guidelines on specifying artifacts. Furthermore, no support for use cases or scenario analysis is provided.

The sum up, despite of the claimed benefits of GUIM, little consensus exists among the current goal and use case integration approaches. As a result, requirements engineers would not be well supported when combining goals and use cases, in regards to modeling guidelines and analysis support. Therefore, a conceptual foundation should be established for GUIM to support for both the modeling and analysis process. Such a foundation should unify the existing goal and use case integration approaches and satisfy the requirements (1)-(4) discussed above.

Table 2-1 summarizes our analysis of the existing goal and use case integration approaches. The work of Liu and Yu [97] is excluded from this summary since goals

and use cases in this work are specified by GRL and UCM while we target natural language based goal and use case specifications.

Table 2-1. Summary of Goal and Use Case Integration Approaches Analysis

Approach	Support Functional Goals	Support Non-functional Goals	Artifact Classification	Relationship Definition	Specification Guidelines	3Cs Problem Classification
Rolland et al. [133]	√	X	√	X	√ (high level)	X
Kim et al. [76]	√	X	√	X	√ (high level)	X
Watahiki and Saeki [169]	√	√	X	√	X	X
van Lamsweerde and Willemet [164]	√	√	√	√	X	X
Cockburn [26]	√	X	√	X	√ (high level)	X
Lee et al. [91]	√	√	√	X	X	X
Santander and Castro [141]	√	√	√	√	X	X
Supakkul and Chung [154]	X	√	√	√	X	X
Kim et al. [77]	√	√	√	X	X	X

2.3 Information Extraction in Requirements Engineering

Information Extraction is the process of automatically extracting structured information from unstructured and/or semi-structured machine-readable documents [31]. In most cases, this process involves the processing of human language texts by means of natural language processing (NLP). In addition, the general form (i.e., template) of the information to be extracted should be defined to guide the extraction process [70].

In the context of RE, Ryan [140], in 1993, claimed that NLP had not been and would not be ready for use in this process in the foreseeable future mainly due to its lack of

ability to "understand" the text. This argument still holds with the current state-of-the-art in the NLP field, and thus it is not possible for computer systems to entirely replace human engineers in the RE tasks. However, Kof [80] later argued that the goal of most NLP-based approaches to RE is not to understand the text, but instead is to extract concepts contained in requirement-related documents, and thus NLP techniques with syntactic parsing and semantic parsing are still valuable to requirements engineers in supporting (but not replacing) them with their tasks.

In fact, several NLP-based techniques have been developed from this perspective to provide requirements engineers with automated support in different stages of the RE process, such as requirements elicitation (e.g., [34, 99]), specifications (e.g., [100, 111, 129]), or analysis (e.g., [110, 167]). In this section, we limit the discussion to the information extraction techniques in RE. The NLP-based requirements analysis approaches are later discussed in section 2.4.

The efforts of applying information extraction techniques in requirements engineering can be classified into two groups. The first group is concerned with mining early aspects from requirement-related documents while the second one contains the approaches to extracting requirement-related models from text. Most of these approaches focus on extracting information from textual requirement-related documents (i.e., requirements specification documents, stakeholder interview scripts) since most important details of software projects are embedded in documents and thus studying such documents is the key of getting information about projects and systems.

Our research falls into the second group and thus, techniques in this group are more directly related to our work reported in this thesis. We discuss the techniques within both groups in the following sub-sections.

2.3.1 Early Aspects Extraction

This area of research has its root in the field of aspect-oriented requirements engineering (AORE). AORE is an area in requirements engineering which focuses on the separation of crosscutting functional and non-functional properties at the requirements level [129]. The idea of such separation of these crosscutting properties is to support effective determination of their mapping and influence on artifacts at later

development stages. In AORE, early aspects are defined as a broadly-scoped property, represented by a single requirement or a coherent set of requirements (e.g., security requirements), that affects multiple other requirement in the system so that it may constrain the specified behavior of the affected requirements or influence the affected requirements in order to alter their specified behavior [9].

Most existing effort of using information extraction in AORE target on providing automated support for the identification of early aspects and crosscutting relationships from requirement-related documents written in natural languages.

Rosenhainer [135] developed a technique to identify cross-cutting requirements from requirements specification documents using regular expressions. The author defined patterns for each non-functional requirement category and specified such patterns with regular expressions. Requirements documents would then be processed to identify the sentences that match the regular expressions.

Baniassad and Clarke [11] proposed the Theme approach to support the capturing of early aspects at the requirement and design levels. In this approach, views of requirements specifications are created to expose the relationships between system behaviors and reveal crosscutting functionality. Ali and Kasirun [2] enhance this method by developing a technique to automate the identification of crosscutting requirements. This technique is based on the use of natural language processing tools to identify verbs in each requirement. Requirements with same verbs are considered as candidate crosscutting requirements. This work may generate a considerable number of false positives since requirements with the same verbs may not necessarily be relevant and crosscut each other. It thus need to be improved with some semantic analysis support to minimize false positives and redundancies.

2.3.2 Automated Requirement Models Extraction

One of the main focuses of our research is the extraction of goal and use case integration models from textual requirements documents. However, to the best of our knowledge, no attempt has been made in the literature to automatically or semi-automatically extract such models from text although there exist some approaches dealing with goals and use cases individually. In this section, we provide a discussion

on these approaches. We also summarize the related extraction approaches for UML models and software product line variability models.

2.3.2.1 Goal Extraction

Antón [6] proposed a set of guidelines and heuristics based on the Inquiry Cycle model [124] to manually extract goals from domain descriptions and other natural language sources. Such heuristics are a set of questions that requirements engineers should consider to answer when identifying goals from different sources. However, no automation support was provided for the goal identification process.

In line with this work, Kof [81] proposed two rules for identifying goals from stakeholders' dialog: *"Phrases like 'have to' and 'in order to' may directly show a goal. If the first sentence of a paragraph does not contain any of the above phrases, the first sentence states the reason why the previous paragraph is problematic. In this case, the negation of this sentence shows the stakeholder's goal."* The author reported an experiment in which goals were manually extracted using these rules and pointed out the challenges of such extraction. However, no technique was proposed to automate the extraction process.

As can be seen later in Chapter 5 and 6, with the extraction technique developed in our work, we are able to incorporate all these guidelines by writing extraction rules that reflect them. To the best of our knowledge, there is no technique developed to automate the identification or extraction of goals from natural language documents. Variability models in the field of software product line share some similarities with goal models. We thus discuss the approaches to extracting such models from text in the next section.

2.3.2.2 Product Line Variability Model Extraction

Software product line (SPL) is a software engineering paradigm that considers software variability as a first-class entity [121]. One of the main concerns of the requirements engineering process in SPL is to elicit and model the variability of requirements in different single software products within the same product line. For instance, what the common requirements for the entire product line are and what the specific requirements for each individual product are.

Unsurprisingly, some NLP-based techniques have been proposed to support such variability modeling tasks. Niu and Easterbrook [111] developed an approach to semi-automatically extract product line orthogonal variability model from natural language requirements documents. The authors employed the notion of functional requirement profile which is defined by the combination of a verb and an object in a requirement. In this work, given a requirement, the OpenNLP library's part-of-speech tagger [104] is used to identify functional requirement profiles from requirements documents. Such profiles would then be used to manually construct a variability model. This work, however, does not deal with non-functional requirements. Moreover, functional requirements are extracted only in the form of "verb-object" combination.

Weston et al. [171] followed another approach of extracting feature models from natural language-based requirements documents. In their work, similar requirements are first clustered into features based on some certain similarity measures. The requirements are then analyzed by their tool to identify variability based on a set of keywords (called *variability lexicon*) and grammatical patterns. Requirements engineers are required to manually make modeling decision based on the tool's output. A key limitation of this work is its limited level of automation since most important modeling decisions have to be manually made by users.

The variability models in these efforts can be considered similar to goal models in the sense that they are all concerned with capturing artifacts on different levels of abstraction and linking such artifacts with decomposition relationships. However, due to having different objectives and their own limitations, these techniques cannot address most problems we want to tackle in this research regarding goals extraction. For instance, they have no or limited support to identify and classify goals and their relationships from text, or ensure goals to have proper specifications.

2.3.2.3 Use Case Extraction

In this section, we discuss the existing approaches to automatically extracting use cases from text, or understanding/processing textual use case specifications.

Kamalrudin et al. [73] developed a light-weight technique to extracting essential use cases from text. Essential use cases are shorter, simpler and technology-free version of

conventional use cases that primarily capture the sequences of abstract system-user interactions. To facilitate the extraction, the authors develop a library of essential interactions expressed as textual phrases, phrase variants, and regular expressions. Based on string manipulations and regular expression matching techniques, they are able to identify textual segments that can be mapped into more abstract representations. Our research, however, focuses on extracting full conventional use case specifications from text. These include details such as pre/post-conditions, extensions, use case steps, and non-functional constraints on these steps.

Rauf et al. [130] introduced a framework to support the automated identification of logical structures (i.e., use cases, business rules) from natural language-based requirements documents. To accomplish this task, the authors proposed a number of templates for specifying logical structures (i.e., templates outlining how a use case and its components are written). Based on such templates, they developed algorithms for identify the components in the specifications of logical structures. A key limitation of this work is that it requires the logical structures to be written in specific templates and thus, does not deal with uncontrolled textual specifications.

Santos et al. [142] proposed a technique to semi-automatically extract use cases from freely-styled text. Their approach is based on NLP techniques to identify the syntactic verb, object and subject of a sentence. Based on such details, use case name (which contains verb and object) and use case actor (which is subject) can be identified. The detected use case details would be then manually verified by users to determine their appropriateness. Using the similar techniques of identifying verbs, objects and subjects from sentences, Deeptimahanti and Sanyal [37] are able to extract use case diagrams and design class models from text. These works, however, do not support the extraction of use cases' contents (i.e., pre/post-conditions, use case steps, extensions). Moreover, no mechanism was provided to prevent irrelevant details from being processed. For instance, every sentence that contains a subject, verb, and object would be processed regardless whether it describes a functionality of the system. This thus results in heavy manual verification on the user side.

Drazan and Mencl [41] developed a technique to identify use case steps from textual use case descriptions in some formats defined by their set of premises. This technique

enables the identification of use case steps, including the ones combined in a single sentence by coordinating conjunctions, based on the use of NLP tools to detect the subjects, verbs, direct objects, and indirect objects in sentences. The limitation of this work is that it assumes use case descriptions are written in a restricted natural language. In addition, it does not allow the extraction of other use case components (i.e., pre/post condition or extensions).

Sinha et al. [149] introduced a linguistic analysis engine to get the understanding of textual use case descriptions. The engine is able to identify components in use case steps (actor, action) and classify steps into a number of predefined semantic classes (i.e., update, input, output) using shallow parsing and a domain dictionary. In this domain dictionary, each commonly occurring verb is associated to one or more semantic classes. This work, however, does not deal with the cases when multiple artifacts (i.e., steps and/or constraints) are mixed together. In addition, it lacks support for various use case components (i.e., pre/post-conditions, extensions).

Rago et al. [127] focused on extracting sequenced use case steps from textual use cases and identifying duplication among them. In this work, the initial textual use case specifications are syntactically analyzed by some NLP tools (i.e., sentence splitter, part-of-speech tagger, and stemmer) to obtain syntactical details. Based on such details, each step is assigned a semantic class using some machine-learning techniques. Similar to Sinha et al.'s work [149], this work only targets on extracting use case step sequences.

2.3.2.4 UML Model Extraction

A majority of information extraction-based techniques in RE is about generating UML models (i.e., class diagram, sequence diagram) from text. Since UML models are important software artifacts that can help with the requirements analysis and facilitate early design, successful automation approach would potential be greatly beneficial software projects.

Harmain and Gaizauskas [57] were among the first researchers to develop a technique to automate the construction of class diagrams from textual requirements. In this work, requirements are first parsed to obtain its semantic representation in which the dependencies between words are known. It is then used to identify the candidate classes,

attributes and relationships for a class diagram. For instance, all nouns would be considered as candidate classes while action verbs become relationships and possessive verbs indicate the existence of attributes.

Mala and Uma [100] proposed another technique to extract class diagrams from text. In this work, sentences are split and verb phrases are extracted from each sentence. Using part-of-speech tagging techniques and a collection of extraction rules, the authors are able to identify object oriented elements (i.e., class, attribute) from the verb phrases. This work, however, makes an assumption that all sentences being processed contain information for a class diagram. It thus does not provide any mechanism to identify and ignore irrelevant sentences (i.e., sentences that do not contain any details that should be extracted as class diagram components). Its application is therefore limited since users need to manually ensure that only pertinent text is considered for the extraction.

Sharma et al. [144] generated class diagrams from structured textual requirement using a set of heuristics. In this work, requirements are required to be specified in templates defined by the authors and words used in each requirement must come from a set of pre-defined set of words (glossary) that can be used in the domain of interest. To facilitate the extraction of class diagram, each requirement is first transformed into a set of tokens, each is then associated to a word in the glossary. Based on that, the requirement is then automatically classified in to their pre-defined categories. Heuristic rules, which are specifically developed for each requirement category, are then used to turn the tokens into candidate classes and relationships. The key limitations of this work are that it only works with requirements specified in constrained formats and that the words used in such requirements must already exist in a glossary.

Omar et al. [117] semi-automated the extraction of entity relationship (ER) diagrams from textual requirements specifications. Their technique relies on the use of part-of-speech tagging and a collection of heuristic rules that guide the identification of ER diagram components such as entities, attributes, and relationships. These heuristics are defined based on the part-of-speech tags words in sentences. Each heuristic is also given a weighting score in order to resolve possible conflicting heuristics in some certain cases. User intervention would be needed to guide the extraction process in cases of conflicts.

2.3.3 Discussion

As can be seen through our discussions, the existing information extraction techniques in RE have been greatly beneficial from the advancement of NLP technologies. Most techniques strongly rely on the use of NLP tools for various important tasks such as tokenizing, part-of-speech tagging, syntactic parsing, and semantic parsing. Some of these tasks can now be done with very high accuracy (i.e., many NLP parsers achieved over 90% accuracy rates). Moreover, most NLP tools are accompanied with training support that allows users to further train the tools to improve their accuracy and/or suitability in some specific domains.

However, automated information extraction still remains a difficult task in RE due to the following two main reasons. First, due to the flexibility and expressiveness of natural languages, requirement-related specifications and documents can be written in various ways, making them ambiguous and thus difficult to extract details from them. Second, some extraction tasks require the understanding of text to be properly done.

Most approaches discussed in this section overcome the ambiguity of natural languages by making an assumption that the input text into the information extraction process is written in a constrained format (i.e., [41, 130, 144]). This means the requirement-related documents should be written using certain constrained languages. However, the use of constrained languages is considered impractical by many authors [87, 98]. One of the key reasons is that, the restriction on the syntax and semantics of languages results in losing the flexibility of natural languages. In addition, users may have to be trained on the languages and remember their restrictions and syntaxes. Moreover, in large projects, documents may be shared and edited by different individuals across teams, departments, or organizations. It thus is extremely difficult to ensure such documents to completely conform to certain restrictions. Therefore, a new technique is needed to tackle input text in a freely-styled format.

In order to extract necessary details from text, a common method is using heuristic rules to guide what to extract based on the identified grammatical structures of sentences. This method has been adopted in most approaches. For instance, subjects are identified as use case actors while predicates, direct objects, and indirect objects hint the extraction of use case names or use case steps in [37, 41, 142].

However, grammatical structures alone cannot guarantee precise extractions. For example, there may be a sentence that does not semantically contain details that should be extracted (in other words, that sentence should not be considered in the extraction). Such a sentence may still have the required grammatical structure. That means it would be included in the extraction results while it should not be. Most discussed approaches avoid this problem by assuming that the input text is always valid, which means there is no "irrelevant" content. However, that comes at the cost that users have to ensure the validity of the input text by manually verifying the text before the extraction. Such verification can be sometimes very labor-intensive and may limit the applicability of the extraction techniques. Therefore, there should be a better method to deal with input text that includes invalid content.

Current approaches have limited capability in extracting goal and use case integrated models from text. Although some heuristics for goal identification have been developed, no automated support has been provided. Moreover, existing efforts for use case extraction focus on high-level use case specifications. They thus do not provide any mechanism to identify use case internal components such as pre/post-conditions, extensions, or constraints. Such detailed extraction is important in our work since one of the key objectives in our research is to automate the analysis of goal-use case models.

2.4 Requirements Analysis

At the beginning of every software project, some kinds of requirements documents are usually written. According to a survey by Luisa et al. [98], a majority of such documents are written in natural languages. However, this brings up an issue that these requirements documents may likely be imprecise, incomplete, and inconsistent since these quality are considered extremely difficult to achieve using mere natural language as the main presentation means [54, 74].

According to Boehm [12], in software development, the later an error is found, the more expensive its correction. Therefore, it is a critical task in requirements engineering, specifically in requirements analysis, to identify and resolve such defects to ensure the success of the software project. In this section, we start with a discussion on problems regarding the consistency, completeness, and correctness of requirements (the 3Cs problems). We then discuss the existing efforts in automated requirements analysis.

2.4.1 The 3Cs Problems in Requirements

2.4.1.1 Consistency

Consistency is the property that no two or more requirements in a specification contradict each other [181]. In other words, if there exist two or more requirements that cannot be fully satisfied altogether, then such requirements are considered inconsistent with each other. The inconsistency of requirements is considered one of the most serious problems in RE since it may prevent part of the stakeholders' expectation being unsatisfied and thus, hamper the success of the software development project [86].

According to Nuseibeh [115], the causes of requirements inconsistency may arise different language usage, varying development strategies, different views held by participants, and the overlap of the concerns of different stakeholders. Inconsistency is viewed as an internal property of a certain body of knowledge [181] since it is only concerned with the relationships among the requirements that have already been explored and specified.

2.4.1.2 Completeness

Boehm [13] defined three fundamental characteristics of a complete requirements documents: (1) No information is left unstated or "to be determined," (2) The information does not contain any undefined objects or entities, (3) No information is missing from this document. The first two properties are referred to as *internal completeness* since they are concerned with the existing requirements. The third property is classified as *external completeness* since it aims to ensure all the information required for the problem of interest is found within the specified requirements. This demonstrates the complexity of ensuring the completeness of requirements since it is impossible to guarantee no extra requirement is needed for a certain system [181].

Zowghi and Gervasi [181] proposed that, to determine the degree of completeness of requirements, it is more meaningful to verify whether a specification is *sufficiently complete*. The decision on how a requirements specification is sufficiently complete would have to be defined with respect to the type of system being implemented. For instance, in safety-critical systems, the important consideration to define sufficient completeness are the system safety design constraints and requirements derived from

hazard analysis [94]. Similar to inconsistency, incompleteness is also a serious issue in RE since it results in missing features in the system-to-be [3].

2.4.1.3 Correctness

According to Zowghi and Gervasi [181], *"correctness is usually meant to be the combination of consistency and completeness."* In other words, a specification is correct if it is both consistent and complete. Therefore, it is incorrect if it is inconsistent or incomplete.

Zowghi and Gervasi [181] also indicate very important inter-relationships between these properties. In fact, resolving inconsistency between requirements may involve the removal of one or more requirements from the specification and thus results in the diminish the requirements completeness and correctness. On the other hand, adding new requirements may introduce new inconsistency in the specification and therefore also reduce its correctness. Due to such characteristics, it is a well-known fact that requirements are often incomplete and inconsistent during most of their life. Therefore, requirements engineers should be able to keep track of the statuses regarding the incompleteness and inconsistency (and thus, incorrectness) of requirements and have appropriate actions (i.e., either tolerate the problems, or resolve them) at different stages during the requirements evolution.

Since modeling requirements is one of the key approaches to understanding and analyzing requirements. As discussed in section 2.2.2, different types of modeling languages have been proposed to provide means for viewing requirements from different perspectives (i.e., goal models, use case models). In such modeling approaches, there is another notion of correctness that is about ensuring the models to be generated correctly. For instance, the models need to conform to certain constraints and rules of the associated modeling languages and the models must correspond to the requirements being modeled. Since our research concentrates on goal and use case integrated modeling, this is the notion of correctness that we focus on, besides consistency and completeness problems.

2.4.2 Automated Requirements Analysis Techniques

One of the key concerns of our research is developing automated support for the identification of the 3Cs problems in goal and use case integrated models specified in natural languages. In this section, we provide a review on existing efforts in automated requirements. We limit the scope of our discussion to the approaches to detecting the 3Cs problems in natural language requirements, and goal-use case models. In these approaches, only few address problems specific to the integration of goals and use cases. In addition, limited work has been done on detecting inconsistency between goals and associated use cases or missing relationships between goals and use cases.

All automated techniques for analyzing requirements share the same concept, that is, the requirements have to be transformed to a certain formal modeling language which can be interpreted and processed by computers. However, due to different concerns and objectives, different approaches may follow different designs and techniques. We classify the research efforts in this area into three categories that are discussed in detailed in the following sub-sections: formal methods, ontology-based methods, and natural language processing-based methods.

2.4.2.1 Formal Methods

Formal methods have a long history. They were probably the first approach in automated requirements analysis with the first proposal was made at least in the 1970s according to a 1977 literature review on requirements analysis techniques [157]. The principle of formal methods is to formalize requirements with a formal language that can be interpreted and processed by computers and thereby, facilitating the automated detection of requirements problems. Various formal requirement specification languages have been proposed to tackle requirements from different perspectives. Some of the key formal languages include logic-based (i.e., propositional, first-order, or temporal logic) specification languages, state-based specification languages (i.e., Z [122, 153], VDM [42], B [1], Alloy [67], and OCL [168]), event-based specification languages (i.e., SCR [58], RSML [95], LTS [159], and Petri nets [105]).

In the following sub-sections, we discuss the commonly used formal analysis techniques and some of their applications in goal and use case model analysis.

2.4.2.1.1 Formal Requirements Analysis Techniques

In this section, we discuss the commonly used techniques for verifying 3Cs problems in requirements in formal methods.

Language Checks

This technique is the simplest one. It mainly deals with the correctness of requirements specifications. It relies directly on the specification syntax and semantics of a specific formal language being used [163]. For example, a specification is considered incorrect if it does not conform to the language's grammar, or it has a variable used outside its scope (i.e., in Z specification language). Depending on the language used, the types of checks can be different.

Dedicated Consistency and Completeness Checks

This technique is based on the checking of input-output relations that describe the expected services or behaviors of the system [163]. For instance, to check consistency, an algorithm can be developed to verify if there is any situation in which there are more than one output corresponds to a single input (which would create an undesirable non-deterministic system behavior). To check incompleteness, an algorithm can be built to verify if there is an output situation that does not have any corresponding input (which would result in some valid state or behavior of the system never being achieved).

Model checking

Model checking is a formal technique to verify finite-state concurrent systems in which specifications are written in temporal logic [25]. In RE, model checking is normally used to verify if a formal model satisfies some desired properties such as requirements, assumptions, or domain properties [163]. The key idea of model checking is to systematically analyze a model to identify property violations. If a violation is found, the algorithm produces a counter-example that does not satisfy the property.

Theorem Proving

Different from model checking technique which is algorithmic, theorem proving technique is used to verify requirements specifications based on the inference rules defined in the formal languages underlying such specifications. In this technique, a

certain property is first selected as a candidate theorem. The existing formal specifications are considered as axioms. A theorem prover tool is then used to verify if the candidate theorem can be proved based on the inference rules defined in the specification language [163].

2.4.2.1.2 Formal Methods in Goal and Use Case Model Analysis

KAOS [161, 165] is a framework aimed to support the process of Requirements Engineering including the management of requirements conflicts. In KAOS, requirements are formalized in linear temporal logic (LTL) to enable the conflicts detection using formal reasoning methods and heuristics. Based on the use of LTL, some algorithms have also been developed to identify obstacles to goal satisfaction to improve the model's completeness [3]. However, the use of LTL requires requirements engineers to be sufficiently familiar with LTL and be able to correctly specify requirements. Thus, its applicability in practice can be fairly limited [43]. In addition, although KAOS supports the capture of domain knowledge through the use of its conceptual meta-model, the concepts and relationships in the problem domain and the semantics of requirements cannot be adequately expressed and reasoned in LTL [15].

Tropos [49] provides formal analysis of early requirements. The analysis in Tropos is done using model checking with the support of a specification language called Formal Tropos which is derived from the i* modeling language [21] and the KAOS temporal specification language. Similar to KAOS's specification language, Formal Tropos is considered complicated to be used in practice. Moreover, although supporting inconsistency detection between goals, both frameworks do not provide explicit automated supports to detect inconsistency between goals and use cases and incorrectness and incompleteness among the artifacts. Furthermore, use case specification and analysis are not supported in both KAOS and Tropos.

Lauenroth and Pohl [86] proposed an approach for dynamically checking consistency among behavioral invariants in product line variability models. In this work, behaviors of the product line are specified by a set of automata. A contradiction function is also developed to serve as a basis for the inconsistency verification. The automatic checking approach is based on the model checking technique. Due to having a different objective from our research, this work does not tackle our problems in goal and use case analysis.

Sikora et al. [146] developed a framework to specify scenarios at different levels of abstraction (called system level and component level) and support the automated consistency verification of such scenarios based on message sequence charts (MSC). In this work, the consistency check is carried out for each pair of scenarios on the system level and component level that are related by a 'refines' link. Such check is based on a transformation of the specified MSCs to interface automata and the computation of differences between the automata or, respectively, the regular languages associated with the automata. This work, however, does not deal with semantic errors and other problems such as incorrectness and incompleteness among goals and use cases.

Lee et al. [111] proposed a method to formalize use case specifications using Petri nets in order to automatically identify inconsistency and incompleteness of use cases. Petri nets provide a means to formally specify use cases as a set of events and states. A number of inconsistency and incompleteness rules were developed to support the automated verification of use cases. Many approaches followed a similar technique to automate the analysis of use cases. They formalized use cases in different formal languages. For instance, Hsia et al. [65] used BNF-like grammar, Dano et al. [32] used a combination of a tabular notation and Petri nets, or Glinz [52] used statecharts. However, the main drawbacks of these approaches include the complicated manual formalization process of use cases and the lack of semantic capturing which makes semantic problems undetectable.

2.4.2.1.3 Challenges of Formal Methods for Requirements Analysis

Formal methods provide great support for requirements specification and analysis. In fact, due to the formally defined syntax and semantics, formal languages eliminate the problems of ambiguous problem of natural language-based requirements specifications. In addition, formal languages allow a way to automate the requirements analysis since they can be interpreted and processed by computers.

However, a major practical challenge of using many formal methods is the use of complicated formal languages. Many researchers have criticized that formal languages are not suitable for practical use [43, 149] since they are generally hard to read and write, and getting adequate, consistent, and complete specifications requires considerable expertise and training. Due to such complexity, formal languages cannot

be used as an effective communication means between requirements engineers and stakeholders. Moreover, formal languages generally have limited expressive power compared to natural languages [163]. They mostly only address the functional aspects of systems (except some languages that can express temporal properties).

2.4.2.2 Natural Language-based Methods

Natural language-based techniques (i.e., [152, 59, 78]) for requirements analysis take another direction. Instead of representing requirements in formal languages, they focus on the use of natural languages as the principle representation language for requirements. Different types of linguistic techniques can be applied to these requirements to achieve some automated analysis support. Generally, these techniques fall into two categories. The first one contains the approaches that rely only on linguistic techniques to analyze textual requirements. The second category includes the efforts that involve the transformation of textual requirements into formal specifications to perform automated analysis. Below, we discuss efforts in both categories.

Approaches that Rely on Only Linguistic Techniques

Lami et al. [152] proposed an approach with tool support (QuARS) for analyzing textual requirements specifications. The authors developed a quality model that defines the indicators and metrics for detecting quality problems (i.e., ambiguity, subjectivity) in textual requirements. Based on this model, problems can be automatically detected based on keyword-matching techniques (i.e., finding if a requirement contains one or more quality problem indicators and calculate metrics). The limitation of this work is that, its analysis support is limited to syntax-related issues while the actual meanings of requirements are not considered.

Also following a light-weight natural language technique, Kamalrudin et al. [59] developed a to identify inconsistencies in Essential use case (EUC) models based on a set of EUC patterns. The method allows natural language requirements to be extracted into abstract interactions and transformed into EUCs for automated analysis. This work, however, does not deal with semantic-related problems, and incompleteness and incorrectness detection in goals and use cases.

Kim et al. [78] introduced a method to semi-automatically support the identification of conflicts in goal-scenario models using linguistic techniques. They assume that each specification can be parameterized into verb, object, and resource. A number of rules have been developed to hint the conflict detection. For instance, if two specifications have the same verb and resource while having different objects, then they are considered potentially conflicting since they may indicate two activities that use the same resource. However, the accuracy of this approach is limited since they do not have any support for checking the actual meaning of specifications. Moreover, goal and scenario specifications may contain additional details, and thus, analyzing them based on only verb, object and resource may not be sufficient. In addition, there is no support to automate the identification of these parameters from textual specifications.

Sinha et al. [148] introduced a technique to analyze textual use case specifications. At the core of this work is a pre-defined domain dictionary in which each action (verb) is classified into a semantic class (i.e., INPUT, CREATE). Based on such a dictionary and natural language parsing techniques, the authors can identify a number of problems in use case specifications such as missing actors, or using an object before it is created, or invoking an undefined use case. This work, however, lacks the consideration on pre/post-conditions and extensions (i.e., identify if a certain exception in use case flow has not been handled by a pre-condition or an extension).

Ferrari et al. [47] proposed a technique to measure and improve the completeness of requirements specifications. The notion of completeness targeted in this work is "backward functional completeness," which means a set of requirements is considered complete with respect to an input document if all terms from the input documents and interactions between these terms are treated in the requirements specifications. In this technique, given an input document, a natural language processing technique called "contrastive analysis" is used to identify the domain-specific terms. The relevant relations between such terms are also extracted based on the Log-likelihood metrics (basically the idea is that, terms often appear together are likely to have relations). Requirements are then analyzed to verify if they contain the extracted terms and interactions. In this way, the degree of completeness can be determined.

In our view, checking requirements completeness based on missing domain-specific terms in requirement sets can be helpful to some extent. However, a high number of false positives and false negatives can be produced (i.e., there is no need to have a requirement that has a certain term, or even all domain-specific terms are used in a requirement set, it does not necessarily mean the requirements are complete). Moreover, this technique does not take into account synonyms of terms (i.e., if a term is not contained in any requirement, however its synonym is. This thus should not be considered as incomplete). A more comprehensive approach that can incorporate knowledge in the problem domain could potentially improve the completeness checking.

Approaches that Require Formal Language Transformation

Gervasi and Zowghi [51] aim at automating the inconsistency detection of high-level natural language requirements. They developed a method that automates the transformation of natural language requirements into propositional logic statements and use default reasoning to identify inconsistencies. This work, however, does not deal with semantic-related issues. 3Cs problems analysis in the context of goals and use cases is also not supported.

Using a similar approach, Gervasi and Nuseibeh [50] enable the transformation of natural language-based requirements into specifications in a formal language (i.e., SCR), which then allow automated analysis for various problems including inconsistency. The transformation and validation of requirements are supported by the tool Cico [4]. This work, however, also lacks support for semantic-related problem detection, and analysis in the goal-use case modeling context.

Simko et al. [147] proposed a technique to automate the correctness verification of use case specification. In this work, the authors defined the syntax and semantics of an annotating language that can be used to annotate the flows of use cases and their temporal information (i.e., a certain step must occur before another). Such annotations can be automatically parsed and transformed into LTS specifications which then be converted to the Symbolic Model Verifier language for automated analysis using model checking. Inconsistency and incompleteness detection, however, are not in the scope of this work.

Holtmann et al. [60] introduced a technique of using a controlled natural language to specify high-level textual requirements. Based on the use of this language, requirements can be automatically transformed into the authors' defined structural pattern in which they can be analyzed for problems including inconsistency and incompleteness using a set of rules. However, the types of requirements targeted in this work remain on a high level. They mostly describe the system composition or enumerate which services are supported.

2.4.2.3 Ontology-based Methods

Ontology-based requirements analysis approaches rely on the idea that domain knowledge and semantics of domain-specific terms could be helpful in identifying problems in requirements specifications. In these approaches, such knowledge and semantics are generally captured in domain ontologies with OWL languages to provide automated reasoning support.

Kaiya and Saeki [72] proposed an approach of using domain ontologies to analyze inconsistency, incompleteness and incorrectness in requirements. Each term in a requirement is manually mapped to an ontological concept for getting semantic support. The analysis is done based on the relationships between concepts in the ontology. For instance, if a requirement is linked to a concept *'create,'* if *'create'* has a *'require'* relationship with 'edit' and there is no requirement linked to the concept *'edit,'* then there is incompleteness. Similarly, inconsistency is detected based on *'contradictory'* concepts. The disadvantage of this approach is that using only relationships between atomic concepts may result in a considerable number of false negatives. For instance, if there are requirements about *"create an account"* and *"edit a review,"* then it is not detectable that a requirement about *"edit an account"* is missing. In addition, the approach does not provide analysis support in the context of goal-use case integration.

Rajan and Wahl [128] support the automated analysis of natural language requirements using domain ontologies. Their tool support allows users to enter natural language requirements in a constrained format, which are then automatically transformed into some representations to allow automated reasoning. This work provides similar ontological support to Kaiya and Saeki and thus shares the same disadvantages. Several authors have proposed similar approaches, for instance [44, 66, 92].

RAT [167] is an ontology-based requirements analysis tool that allows requirements to be input in natural language. Based on a set of standardized requirements specification syntax and user-defined glossaries, the tool enforces requirements to be written in a systematic way to avoid linguistic defects (i.e., ambiguity, terminology inconsistency). In addition, the tool offers a way to identify potential conflicting requirements by categorizing requirements into different pre-defined ontology classes whose relationships are captured. The key disadvantages of this work include its lack of support for detection of logical inconsistency, inconsistency involving more than two artifacts and several goal-use case models' specific types of incompleteness and incorrectness.

Körner and Brumm [83] proposed an approach to improving the completeness of natural language requirements specifications with the help of ontologies and a part-of-speech tagger. Their approach can detect possible requirements incompleteness such as incompletely specified process words (i.e., a verb is used while its subject is missing), the use of modal verbs of necessity (i.e., the word "must" is used), incompletely specified conditions (i.e., there are "if" and "then" clauses specified in a sentence while there is no "else" clause). This technique, however, does not tackle more complicated tasks such as identifying if an extension is missing in use cases, or a goal is missing a certain constraint.

2.4.3 Discussion

Each approach to automated requirements analysis has its own benefits and limitations. Formal methods are promising approaches due to the strength of formal languages in terms of machine-understandability. However, such methods suffer from two main problems: limited expressive power, and being complicated for practical use. Natural language-based techniques allow users to work directly with natural language. However, they lack the capability of automating complicated checking as supported with formal methods. Ontology-based approaches leverage the analysis of requirements by providing a way to incorporate domain knowledge and semantics that have been proved as an important part in verifying requirements. This technique generally requires words in requirements to be associated to the corresponding ontological terms. However, such association needs to be done manually in most existing ontology-based

approaches, which considerably diminishes their level of automation. Moreover, current ontology-based techniques mainly support the incorporation of the semantics of domain-specific term into requirements analysis. No work has been done to examine the possibility of capturing and using more sophisticated domain knowledge (i.e., domain rules, tasks, constraints, assumptions) with ontologies.

With the objective of providing automated analysis support for goal and use case integrated models specified in natural languages, we identified the following additional research gaps:

- No work has been done to provide comprehensive automated analysis for goal and use case models. Although there are a number of techniques developed for goal and use case models individually, no work has tackled the possible problems (i.e., inconsistency) when modeling goals and use cases together.
- No work has been done to comprehensively capture the semantics of natural language-based requirements (formal methods are excepted since our objective is to work with textual requirements specifications) and take such semantics into account during the requirements analysis. Although some techniques attempted to model the semantics of requirements, they only do that partially by considering only key parameters of requirements such as actors, actions, and objects (i.e., [45, 78, 83]). In our view, other parameters (e.g., beneficiary, means, location) are critical for understanding requirements as well since they contribute to the meanings of requirements.

2.5 Summary of Research Gaps

In this chapter, we have provided some background on requirements engineering and discuss our literature review on the related efforts in goal-use case modeling, requirements extraction, and requirements analysis. We summarize the identified research gaps as follows:

Goal and Use Case Integrated Modeling (GUIM)

Different approaches, with different foci, define different set of concepts and relationships to model. In addition, little consensus exists among the existing GUIM approaches in regards to what to be modeled in a general goal and use case integrated

model. Moreover, no existing GUIM approach is comprehensive enough to model both goals and use cases as each lacks support for certain artifact types and dependencies. Such problems potential result in the following issues:

- It is difficult for requirements engineers to decide what to capture in a general goal-use case model
- It is difficult for requirements engineers if models created in different approaches were to be combined.
- The lack of agreed types of artifacts to be modeled prevents goal-use case models to be adequately analyzed. Existing GUIM approaches do not sufficiently address key analysis questions such as: how to verify if an artifact is not properly specified? How to check if artifacts are not correctly connected? How to ensure a use case is matched with its associated goal? How to detect if a required artifact has not been elicited?

In this research, we target to develop a goal and use case integrated meta-model to address the discussed problems.

Automated Goal and Use Case Model Extraction

Current approaches have limited capability in extracting goal and use case integrated models from text. Although some heuristics for goal identification have been developed, no automated support have been provided. Moreover, existing efforts for use case extraction focus on high-level use case specifications. They thus do not provide any mechanism to identify use case internal components such as pre/post-conditions, extensions, or constraints.

Many techniques overcome the ambiguity problem of natural languages by making an assumption that the input text into the information extraction process is written in a constrained format. However, constrained languages are considered unsuitable for use in practice, making such extraction techniques less applicable.

Other techniques, on the other hand, deal with freely-styled text by using heuristic rules to guide what to extract based on the identified grammatical structures of sentences. However, grammatical structures alone cannot guarantee precise extractions. For instance, a sentence containing irrelevant details may be extracted if it satisfies the

required grammatical structure. The semantics of text thus needs to be also taken into account during the extraction process to avoid (or minimize) invalid extraction results.

In our research, we target to develop automated support for the extraction of goal and use case integrated models from text. We also aim to overcome the above discussed problems by ensuring our technique to (1) support the model extraction from freely-styled text, and (2) consider semantics of text during the extraction process.

Goal and Use Case Model Analysis

No work has been done to provide comprehensive automated analysis for goal and use case models. Although there are a number of techniques developed for goal and use case models individually, no work has tackled the possible problems when modeling goals and use cases together (i.e., inconsistency between a goal and its connected use case).

Moreover, no work exists to comprehensively capture the semantics of natural language-based requirements (so that the entire requirements can be interpreted by computers) and take such semantics into account during the requirements analysis. In addition, although domain knowledge and semantics have been recognized as an important part in requirements analysis, existing techniques (i.e., ontology-based techniques) mainly consider the capturing and use of domain-specific terms and their relationships without providing support for more complicated knowledge (i.e., assumption, domain tasks, or dependencies between tasks).

In our research, we aim to develop an approach to automating the analysis of goal and use case integrated models for the 3Cs problems. We target at supporting natural language-based goal and use case specifications while overcoming the above discussed problems by: (1) providing a technique to systematically capture the semantics of requirements so that they can be interpreted and processed by computers (so that the analysis can be automated), and (2) providing a technique to comprehensively capture and use domain knowledge and semantics with ontologies.

Chapter 3

Approach

In this chapter, we provide an overview of our approach to goal and use case integrated model extraction and analysis. Section 3.1 presents a motivating scenario (which is an extension of a real-word case study later presented in section 3.5.2) to illustrate the research problems that we target in this thesis. Section 3.2 discusses our analysis of the problems. It presents the set of requirements that we established for possible solutions to such problems. Section 3.3 provides an overview of the approach that we took to build a goal and use case integration framework to address these solution requirements. Section 3.4 describes the goal-use case modeling and analysis process in our framework. Section 3.5 discusses our evaluation approach which includes an introduction to the evaluations that we conducted, and the set of case studies that we used for evaluations in this research. From this chapter to the rest of the thesis, we use the term *'artifact'* to refer to the elements being captured (except relationships) in a goal-use case integrated model (i.e., goals, use case components, constraints).

3.1 Motivating Scenario

Consider a group of requirements engineers who are distilling the requirements for a Traveler Social Networking system, that is intended to improve the quality of travel planning (of travelers). They intend to use goal and use case modeling to comprehensively capture, understand, and analyze the requirements. In the following sub-sections, we discuss the problems that they face during this process.

3.1.1 Determining what should be modeled

Assume at the current stage, they have obtained the requirements document for the system and start to build a goal and use case integrated model. Figure 3-1 presents some selected parts in a requirements document that contain the descriptions of goals and use cases. The requirements engineers need to extract the details of goals and use cases from this piece of requirement text to build a goal-use case model. However, one of the key challenges in this step is to determine what to model and how to model them. Below, we provide a closer look into this challenge.

(S1) This software system will be a Social Networking System for travellers around the world. **(S2)** This system will be designed to improve the quality of travel planning by providing tools to assist in facilitating the communication between travellers. **(S3)** To improve the quality of travel planning, the system supports travelers to share and gain experiences while remaining easy to understand and use. **(S4)** More specifically, travelers shall be able to participate in forum discussions, and create reviews and travel articles. **(S5)** They should also be supported to create reviews quickly.
...
Use case: Create Reviews
Brief Description: **(S6)** A traveler creates a review
Pre-condition: **(S7)** Before this use case can be initiated, the traveler has successfully logged into the system.
Main Success Scenario
Step 1. **(i)** The traveler selects to create a review.
Step 2. **(i)** System prompts the traveler to select a review category.
Step 3. **(i)** The traveler selects review category. **(ii)** The list of categories includes hotel, attraction, restaurant and tour.
Step 4. **(i)** System displays the suitable review creation form to the traveler.
Step 5. **(i)** The traveler enters the review content and submits it.
Step 6. **(i)** System validates the review. **(ii)** The validation should be completed within 1 second.
Step 7. **(i)** If the review content is valid, the system stores the review into the database and display confirmation message to the traveler; **(ii)** else the system displays the errors to the traveller and the traveler repeats step 5.
Post-condition: **(S8)** After this use case is successfully finished, the new review is stored

Figure 3-1. Motivating Scenario for Goal-Use Case Model Extraction

Although general understanding of requirements and goals may help identify what artifacts should be extracted from text (i.e., sentence (S1) should not be considered as an artifact since it is an introduction to a system), to ensure the consistency of goal-use case modeling, fundamental guidelines are needed to instruct the requirements engineers on what artifacts to be modeled, what their roles are and how they should be specified. For instance, should the sentences Step 3-(ii) and Step 5-(ii) be modeled as artifacts? From our analysis, they are important artifacts (data and quality constraints) within the given use case. However they are not supported in many existing GUIM approaches (i.e., [91], [27]).

In addition, although the artifact *"Facilitate the communication between travelers"* (from sentence (S2)) can be modeled in several GUIM techniques, it is classified into different categories by those techniques. For instance, it is categorized as a rigid goal in [91], summary goal in [27], and design goal in [76]. Therefore, we need to unify them in a conceptual foundation. Moreover, artifacts in a model need to be consistently specified. For example, should the entire sentence (S2) be a specification for an artifact or should it be split into two artifacts *"Improve the quality of travel planning"* and *"Facilitate the communication between travelers"*? Unfortunately, the existing GUIM approaches provide insufficient guidance on how to specify such artifacts.

Moreover, the existing GUIM techniques lack support for several important relationships between artifacts at goal levels and use case level. For instance, the *refine* relationship between the goal *"Travelers shall be able to quickly create reviews"* (from (S5)) and the use case constraint *"The validation should be completed within 1 second"* (from Step 6-(ii)) (a short validation time would help travelers quickly create reviews) is neglected. Furthermore, since different techniques use different artifact categories, different sets of relationships are defined. Thus, we need to study the correlation of these relationships and unify them under a conceptual foundation.

3.1.2 Goal-Use Case Integrated Models Extraction

Assume that the requirements engineers now have a framework that indicates what types of artifacts and relationships to be modeled. Figure 3-2 shows an example of their desired extracted model from the text provided in Figure 3-1. In this example, BG1 is a business goal. NPG1 and NPG2 are non-functional product goals. FFG1, FFG2, etc. are functional feature goals. FSG1, FSG2, etc. are functional service goals. NSG1 is a non-functional service goal. DC1 is a data constraint and NUUC1 is a non-functional use case constraint. The characteristics of these types of goals are not important in this scenario. They are discussed in details in Chapter 4. The key purpose of having these goals here is to illustrate that the extracted goals should be placed on appropriate levels of abstraction.

Moreover, three types of relationships are included in this example, including *refine*, *operationalize*, and *constrain*. A *refine* relationship specifies an artifact supports the satisfaction of another artifact. An *operationalize* relationship denotes a use case

describing steps to achieve a goal. A *constrain* relationship represents that an artifact specifies a data or non-functional constraint of another artifact. Below, we discuss the key challenges of the requirements engineers in extracting such a model from the requirements text. For readability reason, we have placed labels underneath each artifact or on each relationship to indicate in which example(s) that artifact or relationship is referred to. For instance, Ex2 means that the corresponding artifact or relationship is to be mentioned in Example 3-2.

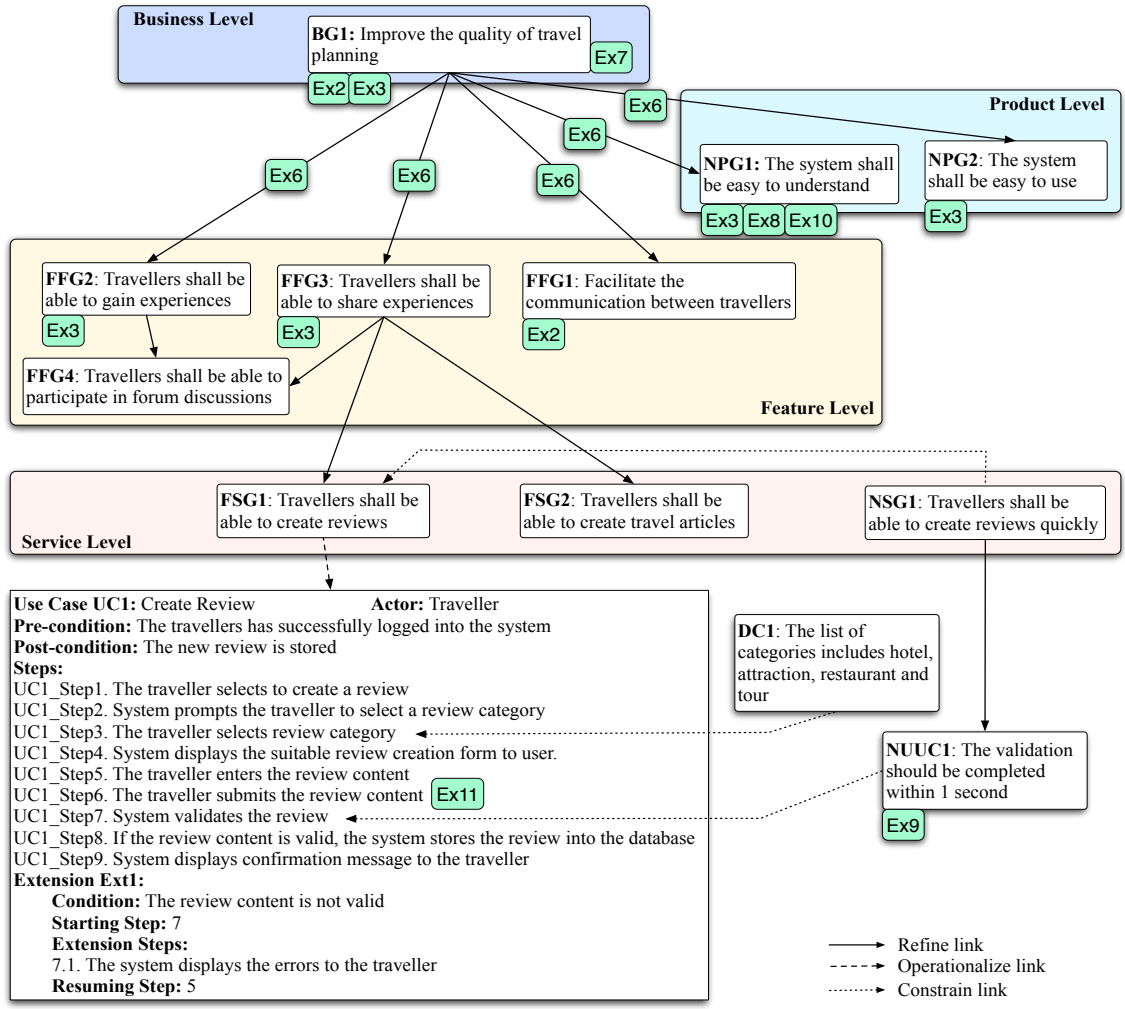


Figure 3-2. Extracted Goal-Use Case Model

Identification of Artifacts from Text

Identifying artifacts from text is one of the key challenges due to the following reasons. First, not all sentences and not all parts of a sentence in a requirement document contain goal or use case descriptions. Second, multiple goals or use case steps may be mixed up in a single sentence. Third, use case steps are often combined with constraints. Four,

alternative paths in a use case are often combined as a single use case step. Thus, automatically filtering important information from text is needed. So is the automatic separation of different artifacts mixed up together. We illustrate these challenges with the following examples.

Example 3-1: The sentence (S1) – *"This software system will be a Social Networking System for travelers around the world"* in Figure 3-1 does not contain a goal description. It is rather an introduction to the system. It thus should be ignored.

Example 3-2: In the sentence (S2), the phrase *"This system will be designed to"* has no significance regarding the objective, functionality or quality of the system. Its role in the sentence is to highlight the more important part which is *"improve the quality of travel planning by providing tools to assist in facilitating the communication between travelers."* For a similar reason, it can be seen that the phrase *"providing tools to assist in"* is unimportant. Therefore, the artifacts to be extracted from this sentence are: *"Improve the quality of travel planning"* and *"Facilitate the communication between travelers"* (they are goals BG1 and FFG1 respectively in Figure 3-2).

Example 3-3: The sentence (S3) is a typical example of a single sentence that contains multiple artifacts. In fact, five artifacts can be extracted from this sentence, including: BG1 (*"Improve the quality of travel planning"*), FFG2 (*"Travelers shall be able to gain experiences"*), FFG3 (*"Travelers shall be able to share experiences"*), NPG1 (*"System shall be easy to understand"*), and NPG2 (*"System shall be easy to use"*).

Example 3-4: The step 3 of the use case in Figure 3-1 contains a use case step in the first part (Step 3-(i) – *"The traveler selects review category"*) and a data constraint (about manage options) in the second part (Step 3-(ii) – *"The list of categories includes hotel, attraction, restaurant, and tour"*). They need to be distinguished to guarantee a correct extraction of use cases. Additionally, step 5 is a combination of two separated steps (*"The traveler enters the review content"* and *"The traveler submits the review content"*). Moreover, step 6 consists of a use case step (Step6-(i)), and a non-functional constraint (Step6-(ii)).

Example 3-5: Step 7 (in Figure 3-1) combines a use case step with an extension specification. It should be extracted into the extension Ext1 in Figure 3-2.

Identification of Artifact Relationships

A goal-use case model requires the relationships between the artifacts to be specified. These relationships are often implicitly mentioned in requirements specifications.

Example 3-6: In the sentence (S2), the structure *"by providing…"* implies a *refinement* relationship between *"improve the quality of travel planning"* and *"providing tools to assist in facilitating the communication between travelers."* This relationship should be extracted as showed between the goals BG1 and FFG1 in Figure 3-2. Similarly, in the sentence (S3), we have the refinement relationships between BG1 (*"Improve the quality of travel planning"*) and the following goals: FFG1 (*"Travelers shall be able to gain experiences"*), FFG2 (*"Travelers shall be able to share experiences"*), NPG1 (*"System shall be easy to understand"*), and NPG2 (*"System shall be easy to use"*). Note that although both sentence (S2) and (S3) contain the text *"Improve the quality of travel planning,"* they should be merged into a single artifact (BG1) to avoid redundancy.

Classification of Artifacts

In requirements engineering, goals need to be classified as functional or non-functional (This is a common way to categorize goals, besides other classifications such as hard vs. soft). Moreover, they need to be classified based on how abstract or concrete they are. Both such classifications are important to understand and analyze goal models [132, 162]. In addition, other artifacts (components in use cases) also need to be classified to differentiate one from another. The following examples clarify these challenges.

Example 3-7: The goal BG1 *"Improve the quality of travel planning"* describes a business objective, not a functionality or quality of the system. It thus should be placed on a *business level*.

Example 3-8: The goal NPG1 *"The system shall be easy to understand"* should be classified as a non-functional goal since it describes a usability quality that the system must meet. Moreover, NPG1 should be placed on a *product level* since it is concerned about the system as a whole, rather than a specific feature.

Example 3-9: The artifact NUCC1 *"The validation should be completed within 1 second"* (from Step6-(ii) in Figure 3-1) should be classified as a non-functional constraint since it describes a performance constraint of a use case step specified in Step6-(i) (*"System validates the review"*).

Ensure Artifacts Are Properly Specified

When identifying artifacts, relevant text in the requirements document is located. However, they are often fragments of the sentences in which they are contained and thus in many cases, cannot be used as descriptions for stand-alone artifacts. Therefore, we need to ensure those artifacts are rewritten in a sensible way after being extracted.

Example 3-10: In sentence (S3), it is identified that *"remaining easy to understand"* contains a goal description. However, this phrase, by itself, is not a meaningful goal description. The context of the whole sentence needs to be considered to obtain a proper specification. As showed in Figure 3-2, this phrase should be changed to *"The system shall be easy to understand"* to properly specify the goal (labeled as NPG1).

Example 3-11: There are two use case steps identified in the step 5 (*"the travel enters the review content and submits it"*) of the use case description which are *"The traveler enters the review content"* and *"The traveler submits it."* The context of the sentence is needed to recognize which noun phrase the pronoun *"it"* refers to. A replacement of *"it"* by *"the review content"* is necessary for a proper use case step specification (UC1_Step6 in Figure 3-2). This process is referred to as *coreference resolution* in the field of natural language processing.

3.1.3 Goal-Use Case Integrated Model Analysis

Assume that the requirements engineers have extracted and built a goal-use case model from the given text. They then continue to build up the model with the details from other parts of the requirements document. However, the model now contains a number of incorrectness, incompleteness and inconsistency that they need to be able to identify and resolve. They are shown in Figure 3-3. In this section, we discuss some of these problems in details. Similar to Figure 3-2, for the sack of readability, the labels underneath each artifact indicates in which example(s) that artifact is mentioned.

Incorrectness

In our work, correctness refers to the correspondence between artifact specifications and the system's needs and constraints. In other words, it is concerned with the soundness of an artifact specification. In the following examples, the obtained goals and use cases may be incorrect due to their unsound specifications or relationships.

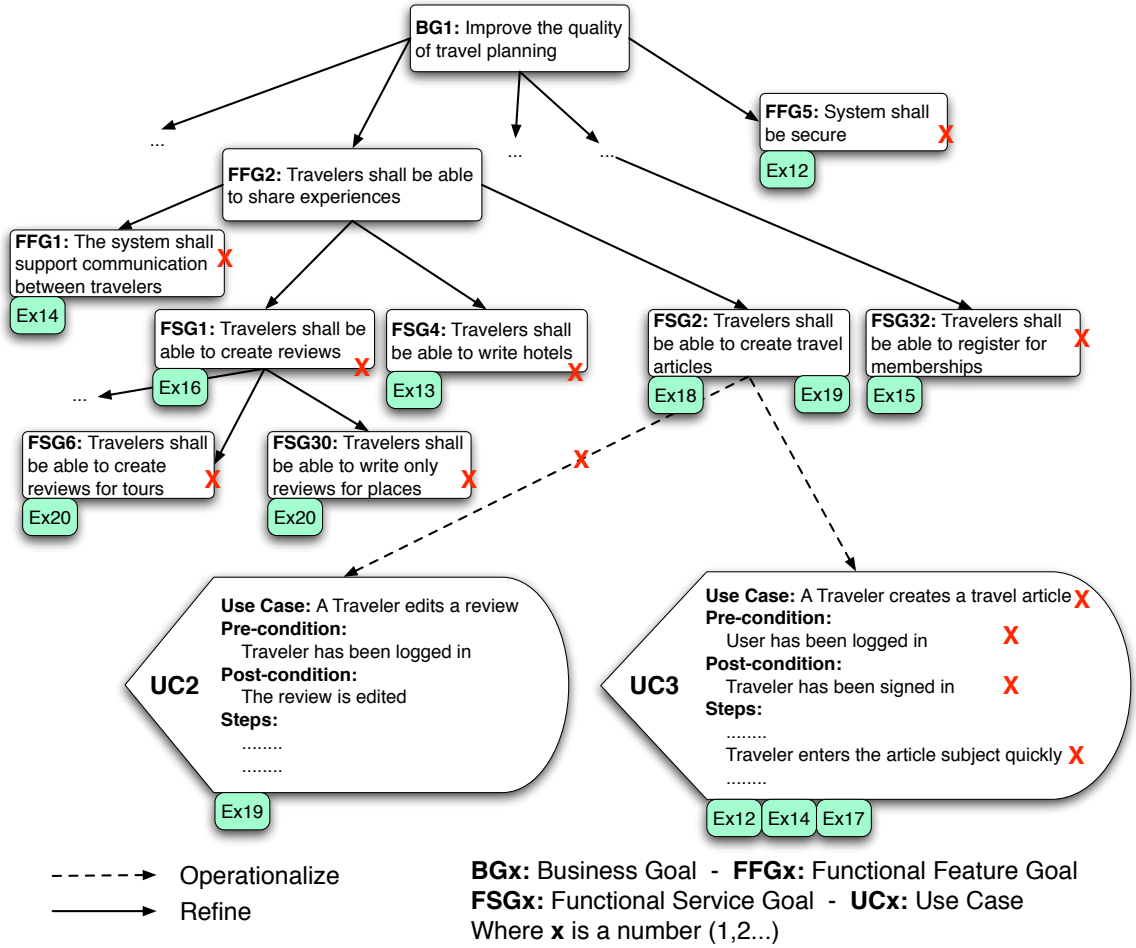


Figure 3-3. Motivating Scenario for Goal-Use Case Model Analysis

Example 3-12: Consider the functional goal FFG3 "*System shall be secure*" and UC2's use case step "*Traveler enters article subject quickly.*" These specifications are malformed for their types. The former should describe a system's functionality rather than quality while the later should not state how a user achieves a task.

Example 3-13: Consider the goal FSG4 defined as "*Travelers shall be able to write hotels."* It is not correct semantically because *hotels* cannot be *written*.

Example 3-14: Consider use case UC3 which has the pre-condition "*User has been logged in*" and the post-condition "*Traveler has been signed in.*" Such use case specification is invalid as its post-condition is identical to its pre-condition, given that *user* is equivalent to *traveler* and *logged in* is identical to *signed in* in this domain.

Incompleteness

Completeness is the quality that indicates whether all the needs and constraints for the system have been identified. In the following examples, incompleteness may include missing artifacts that need to be elicited and modeled, missing parts in artifact specifications or missing relationships between modeled artifacts.

Example 3-15: Consider two goals: FSG32 "*Travelers shall be able to register for memberships*" and FFG1 "*System shall support communication between travelers.*" The former requires an operationalizing use case to describe the needed system-user interaction for membership registration. The later needs to be further refined into a specific goal since it is necessary to identify more specific goals that specify what types of communication to be supported by the system. However, these needed artifacts have not been specified.

Example 3-16: Consider the goal FSG1 "*Travelers shall be able to create reviews.*" There is a missing goal about "*editing reviews*" given the general domain knowledge that any *content writing* activity is usually accompanied with a *content editing* activity.

Example 3-17: Consider use case UC3 ("A *traveler creates a travel article*"). Assume that in this domain a *banned user* is not allowed to create a travel article. Thus, if no pre-condition or extension defined to handle this case, there is possibly incompleteness.

Example 3-18: Consider goal FSG2 "*Travelers shall be able to create travel articles*" and use case UC3 *"Traveler creates a travel article."* Assume they are not connected, then that is incompleteness since UC3 describes the step to realize the goal specified by FSG2.

Inconsistency

Consistency refers to the requirements model quality indicating that no two or more specifications contradict each other. In the following examples, inconsistency can be the discrepancies between a goal and its associated use case's specification, illogical relationships between artifacts, or conflicting artifact specifications.

Example 3-19: The goal FSG3 "*Travelers shall be able to create travel articles*" is operationalized by the use case UC2 which has the description of "*Traveler edits a*

review." This is considered an inconsistency since the use case needs to describe the steps to achieve the goal it operationalizes.

Example 3 -20: Consider two goals, FSG6 "*Users shall be able to create reviews for tours*" and FSG30 "*Users shall be able to write only reviews for places.*" They are inconsistent given "*write reviews*" and "*create reviews*" are equivalent activities while *tour* and *place* are disjoint concepts in the domain.

3.2 Problem Analysis

Although the examples of problems provided in the motivating scenario appear to be simple when considered individually, manually addressing them is not a trivial task in case the number of artifacts is large, and importantly, such task is not sufficiently and effectively supported by the existing GUIM approaches. Therefore, the requirements engineers are not well supported when modeling goals and use cases together. From this we establish the following research questions that we address in this thesis:

RQ1: *How can we establish a conceptual foundation for goal and use case integrated modeling?* This conceptual foundation should clearly indicate what artifacts and relationships to be captured in a goal-use case integrated model, what the dependencies and constraints between them, and how they should be specified.

RQ2: *How can we develop a technique to automate the extraction of goal and use case integrated models from textual requirements documents?* To reduce the manual efforts of developing goal and use case integrated models from text, some automated support should be provided. This question is concerned with developing a technique that can automatically (or semi-automatically) locate goals and use cases from text, classify them, and identify the relationships between them to generate integrated models.

RQ3: *How can we develop a technique to automate the detection and resolution of incompleteness, inconsistency, and incorrectness in goal and use case integrated models?* This question is concerned with developing a technique that can automatically (or semi-automatically) identify both syntactic and semantic 3Cs problems and propose recommendations to users to resolve such problems.

To address the above research problems, we formulate the set of requirements for a possible solution as follows.

R1: There should be a conceptual foundation for goal and use case integrated modeling. Such a foundation should satisfy the following requirements.

- ***R1.1: The conceptual foundation should provide guidance as to what types of artifacts should be modeled, classified, and connected to each other.*** Clear definitions and classifications must be given to the artifacts and relationships between artifacts. Moreover, the possible constraints or dependencies between these artifacts and relationships must be indicated (i.e., a certain relationship should not be used with a certain type of artifacts).
- ***R1.2: The conceptual foundation should provide rules or guidance as to how artifacts should be specified.*** This research aims to support the natural language-based specifications of goal and use case integrated models. Therefore, to ensure artifacts are specified in a consistent way, it is helpful to have a set of rules on how such textual specifications should be written. Moreover, to ensure high specification quality, such specification rules should conform to the existing relevant specification practices for textual software requirements. For instance, functional requirements should be specified with action-verbs, or the use of adjective-preposition phrase (e.g., *"users are **capable of** creating reviews"*) should be avoided [152, 172].

If satisfying the above requirements, the targeted conceptual foundation would be both helpful in the extraction and analysis of goal and use case integrated models. For instance, it would help determine what to extract from a natural language-based requirements document and in which form the extracted contents should be to build a goal and use case integrated model. Moreover, the foundation would be able to provide a basis on which artifacts and relationships between them can be analyzed. For instance, it would be able to verify if an artifact is not correctly specified (based on the specification rules), or if artifacts are not correctly connected (based on the constraints and dependencies defined for artifacts and relationships).

R2: The technique should be able to automatically extract goal and use case integrated models from natural language requirements documents. This requirement is divided into the smaller requirements as follows.

- ***R2.1: The technique should be able to automatically locate goals, use case components and their relationships from uncontrolled text.*** Such a technique needs to be able to cope with the complexity of freely-styled text in requirements documents. It is also required to identify and ignore irrelevant text (i.e., non-goal or non-use case component). In addition, the technique should be capable of recognizing relationships between the identified artifacts.
- ***R2.2: The technique should be able to ensure the well-formedness of the extracted artifacts.*** As discussed in the motivating examples in section 3.1, the extracted artifact specifications may be fragments of sentences and thus are not grammatically correct artifact specifications (i.e., incomplete sentence, or improper tense used). In addition, an extracted specification may not conform to requirements specification practices (i.e., a functional goal specification is not written using an action verb). Therefore, a well-formedness assurance technique is necessary to ensure that the extracted artifacts are in well-formed English grammar and written in conformance to our defined artifact specification rules (discussed in requirement R1.2). This helps guarantee the correctness of the extraction process.
- ***R2.3: The technique should be able to automate the classification of these artifacts.*** After being extracted from a requirements document, artifacts need to be classified when being placed into a goal and use case integrated model. The classification of some use case components, such as step and condition, is generally not necessary, since use cases are normally described in a certain structure in a requirements document. However, goals and constraints that are related to use case steps (i.e., data constraints, non-functional constraints) need to be clearly classified. The classification of goals should cover the key characteristics of goal modeling [163], which includes details as to whether a goal is a functional or non-functional, and which abstraction level it should be on. For example, *"Users shall be able to send text messages to others"* (FSG1) and *"System shall support communication"* (FFG1) should be classified as

functional goals, while *"System shall be secure"* (NPG1) should be categorized into a non-functional goal class. Moreover, FFG1 should be placed on a higher level of abstraction than FSG1 as it describes an abstract requirement regarding a *'communication'* feature while FSG1 provides a requirement about a specific function to support *'communication.'*

R3: The technique should be able to automatically analyze goal-use case integrated models for 3Cs problems. Apart from identifying inconsistency, incompleteness and incorrectness, in order to improve the quality of analyzing models, the technique should be able to provide explanations for detected problems and suggest correction or improvement options. This requirement is divided into the following sub-requirements.

- ***R3.1: There should be a way to consistently structure artifact specifications so that their semantics can be effectively captured and analyzed by computers.*** Each artifact specification should be structured in a way so that computers can recognize its meaning, i.e., which action is referred to, which object to be targeted by the action, how the action to be carried out. In other words, the semantic composition of a specification should be known to computers. In addition, such composition needs to be consistent in the sense that semantically equivalent specifications should have the same semantic components even if they are written differently. Moreover, the transformation of textual specifications into such structured specifications needs to be fully automated or semi-automated to ensure the applicability of the approach in practice. While this requirement is concerned with how a specification is structured for semantic capturing, requirement R1.2 has the focus on how each type of artifacts is specified. There thus is a tight relationship between these two requirements regarding the specification of artifacts. Therefore, R3.1 should be considered when developing a solution to address requirement R1.2.
- ***R3.2: The technique should be able to incorporate and use the captured knowledge and semantics in a certain domain.*** Such a technique would enable the automated identification of semantic problems. It would allow not only the meaning of individual vocabularies used in artifact specifications but also the relevant knowledge in the domain to be available to and processable by computers. For instance, consider two goals, FSG6 "*Users shall be able to*

create reviews for tours" and FSG30 "*Users shall be able to write only reviews for places*" (discussed in Example 3-20), the inconsistency between them would not be detected if it is unknown that *"create reviews"* and *"write reviews"* are the same activity while "tour" and "place" are different concepts. Similarly, consider the use case for "*User creates a travel article*" which has no pre-condition or exception path defined to handle the case that a *banned user* is trying to create a travel article (discussed in Example 3-17). Such incompleteness would not be revealed if it is not known that banned users are not allowed to create travel articles in this particular domain. Therefore, the proposed solution needs to provide a way to retrieve and use captured semantics and knowledge in particular domains.

3.3 Our Approach

We first focused on addressing the requirement R1 and its sub-requirements regarding developing a conceptual foundation for goal and use case integrated modeling. Since requirements R3.1 and R1.2 are tightly related, we started with addressing R3.1 when developing a solution to satisfy R1.2. We have chosen functional grammar [40] as the theoretical foundation to parameterize textual artifact specifications. Functional grammar allows a specification to be structured into different components called *semantic functions*; each holds a unique semantic role (i.e., *agent*, *object*, *beneficiary*, *manner*). Such parameterization provides a consistent way to interpret the semantic of each group of words in a specification, and thus offers a means to analyze specifications with their meanings known. For instance, the goal *"Users shall be able to create reviews easily"* is parameterized as *"Agent(users) + Verb(create) + Object(reviews) + Manner(easily) + Negation(False) + Tense (Present)."*

In order to establish the conceptual foundation, we developed a meta-model called *GUIMeta* (**G**oal and **U**se Case **I**ntegration **Meta**-model). GUIMeta consists of two layers. The *artifact layer* defines the classification and descriptions of commonly used artifacts and their relationships in a goal and use case integration model. In this layer, goals are classified based on their functional – non-functional characteristics and levels of abstraction. It also captures the commonly used components of use case specifications. Relationships between different types of artifacts and their dependencies were also taken into consideration when creating the artifact layer.

The *specification layer* provides the specification rules for the defined artifacts in the artifact layer. It governs how each type of artifacts should be specified in a goal-use case model. The formation of artifact specification rules in the specification layer is built upon the discussed functional grammar-styled parameterization. Each rule contains the sets of semantic functions that may or may not appear in a specification of a certain artifact type. The specification rules are defined in the form of boilerplates[2] that can be used to provide specification templates for artifacts in a goal-use case integration model. A discussion on how functional grammar is used to parameterize artifact specifications and details of GUIMeta are provided in Chapter 4.

In order to meet the requirement R2.1, we developed an extendable set of *artifact and relationship extraction rules* based on the dependencies between words in a sentence (these dependencies are defined according to Stanford's dependency collection [35]). This idea was derived from our observations that although text in a requirements document is normally in uncontrolled formats, the identification of artifacts and relationships from sentences, and the recognition of unimportant sentences or parts of a sentence (i.e., those contain no artifact specification) follow certain patterns.

For example, in sentences that have the form of *"the system is designed to do something,"* the phrase *"the system is designed to"* should always be ignored since it does not contain valuable details of a goal. The important thing in such sentences is the *"do something"* phrase. The Stanford parser [79] was chosen as a natural language text parser to support the rule-matching detection that enables the identification of artifacts and relationships from text. We have extended the Stanford parser to enable to ability of progressive training. Specifically, users are able to train the parser with their new data, without re-train it with the entire dataset as required by the original version. In addition, the Stanford Coreference Resolution System [89] is employed to resolve coreference in sentences. We have developed the syntax and parser for extraction rules, making it possible to extend the set of extraction rules in any domain. Chapter 5 describes our artifact extraction technique.

[2] In requirements engineering, boilerplates refer to templates for writing textual requirements

In order to satisfy the requirement R2.2, we developed a two-step process of artifact specifications polishing to ensure their properness. In the first step, a specification is ensured to be grammatically correct based on a number of linguistic analysis techniques. In the second step, it is aimed to guarantee that the specification conforms to our developed boilerplates for goal and use case specifications. The conformance of specifications to our boilerplates also ensures them to satisfy a number of recommended specification practices (i.e., functional requirements/goals are generally encouraged to be written using action verbs instead of adjective-preposition phrases).

To accomplish this, we developed a set of extendable *modifying rules* that are able to automatically modify a specification to ensure its conformance to the boilerplates. For instance, the specification *"users are capable of creating reviews"* can be automatically rewritten as a proper specification of *"users shall be able to create reviews."* These modifying rules share the same concepts as the discussed extraction rules. The collection of modifying rules can also be extended by users to leverage the polishing process. Chapter 6 provides a detailed discussion on the artifact polishing process.

To address the requirement R2.3 about the automated classification of extracted artifacts, we investigated existing requirements classification tools. Unfortunately, we found that no tool was available to be incorporated into our framework. Mallet [102], a machine-learning based general text classifier, was identified as the most suitable tool to be adopted in our work. However, we found a number of problems that reduced Mallet's accuracy to identify the suitable class for a goal specification. For instance, it considers every word in a specification equivalently in its calculation to determine the probability of a specification for being classified into a class. However, some words may be better indicators for a certain class than others, and thus they should be considered differently. We then extended Mallet to improve its accuracy. The extension and evaluation to prove such improvement is discussed in Chapter 6.

In order to satisfy the requirements R3 and its sub-requirements, we developed a categorization of inconsistency, incompleteness and incorrectness problems that may exist in goal-use case integrated models based on the definitions and classification of artifacts and relationships in GUIMeta. This classification serves as the basis on which goal-use case integrated models are analyzed. We combined the use of functional

grammar and ontology to provide automated analysis support. Firstly, we developed a technique to automate the discussed functional grammar-based parameterization of artifact specifications. Such parameterization result is called a *parameterized specification* in our work. Secondly, ontology is employed to capture domain semantics and knowledge due to a number of their benefits: the suitability for capturing semantics and knowledge, reusability, the increasing availability of ontologies in a wide-range of domains and the existence of automated support for reasoning with ontologies. In order to incorporate semantics into goal-use case models, words in an artifact's parameterized specification are linked to individual concepts in the ontology. For instance, given the goal *"Users shall be able to create reviews easily"* that is parameterized as *"Agent(users) + Verb(create) + Object(reviews) + Manner(easily) + Negation(False) + Tense (Present),"* *'users'* is linked to the *active entity* concept *'User'* while *'reviews'* is linked to the *inactive entity* concept *'Review,'* and *'easily'* is linked to the quality adverbial property *'Easily'* in the ontology. Moreover, we developed a technique to automatically transform structured specifications into Manchester OWL Syntax (MOS) representations. Such representations allow the full integration between artifact specifications and ontologies written in OWL languages (the most popular languages for capturing ontologies currently) and thus enable the fully automated analysis of artifact using the Pellet ontology reasoner [150].

The full automation of the transformation from textual specifications into parameterized specifications and from parameterized specifications into MOS statements allows users to work directly with natural language specifications. That helps overcome the complexity of formal modeling languages which is a key disadvantage of formal analysis approaches [43]. Chapter 7 discusses our analysis techniques.

All above techniques are placed under a single framework called GUI-F (**G**oal-**U**se Case **I**ntegration **F**ramework). We have developed two tools: GUEST (**G**oal-**U**se case model **E**xtraction **S**upporting **T**ool) that realizes our entire artifact extraction approach, and GUITAR (**G**oal-**U**se Case **I**ntegration **T**ool for **A**nalysis of **R**equirements) that implements our analysis techniques. The two tools are fully integrated to offer the seamless process of modeling and analysis of goal-use case models. The integrated tool is called GUITARiST (a combination of **GUITAR** and GUE**ST**). The process of modeling and analysis is described in the following section.

3.4 The GUI-F Process

Figure 3-4 shows an overview of the process of goal-use case modeling and analysis using the GUI-F framework. The modeling of goals and use cases is supported by the GUEST tool while the analysis process is enabled by the GUITAR tool. Input is a software requirements specification document. The aim of the modeling process is to produce a well-formed goal-use case integrated model from the document. In the analysis process, such model is validated for inconsistency, incompleteness and incorrectness. Resolution options are then generated for each detected problem and presented to users for their decisions. In the below sub-sections, we discuss each step (activity) in the framework.

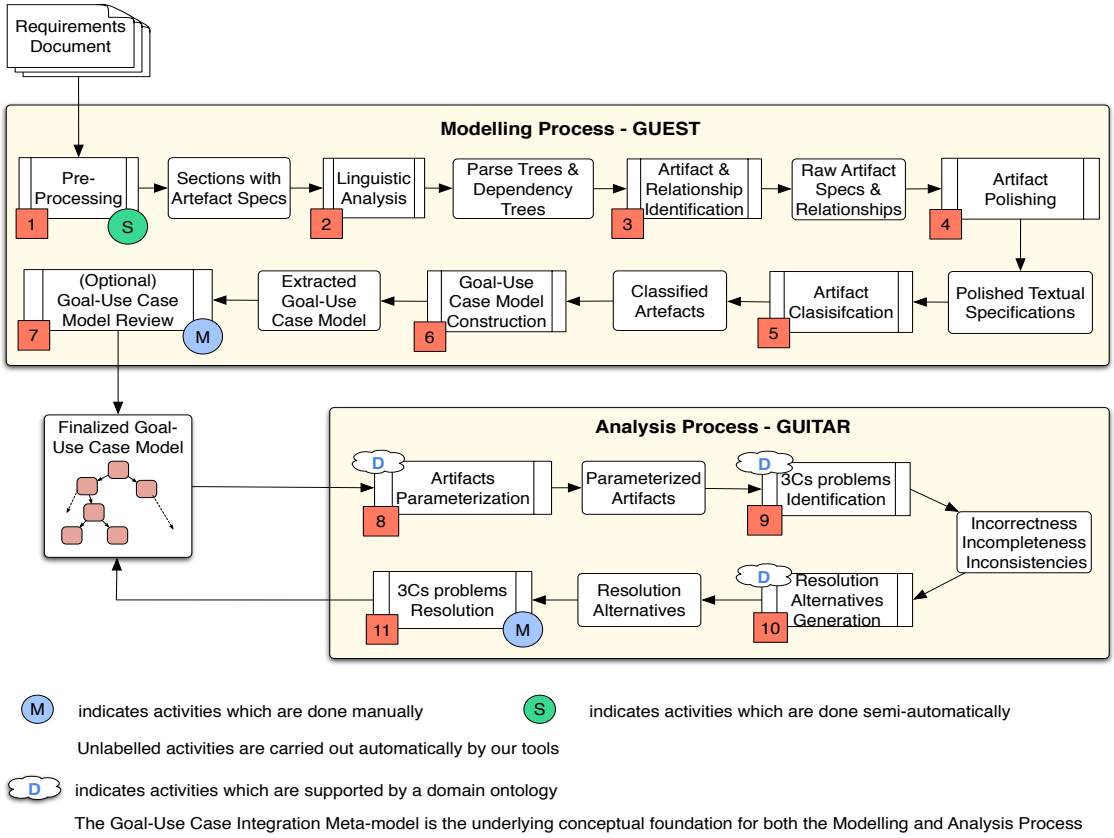


Figure 3-4. Overview of GUI-F Process

3.4.1 Modeling Process

GUI-F supports the modeling process by providing automated assistance to extract goals, use cases and their relationships from a requirements document. Specifically, given the input requirements document, our GUEST tool is able to generate an initial

goal-use case model. Users can optionally take an additional step to further complete that extracted model. The steps taken in this extraction process are as follows.

Step 1: Pre-processing. A requirement document may consist of a number of sections and not all of them include specifications of goals and/or use cases. Therefore, in this first step, the target is to identify the sections that possibly contain goals and/or use cases. This step is done through a semi-automated process.

First, users are required to manually fill in a list of section indicators that indicates whether each section should be processed to look for artifacts of particular types (i.e., functional goals, non-functional goals) (such sections are called *important sections*) or should be ignored since it contains irrelevant information (such sections are called *unimportant sections*).

Second, the tool then automatically analyzes the text in each *important section* to remove unnecessary details such as pictures, brackets, or multiple whitespaces. Figure 3-5 presents an example of a simplified section indicator list in XML format. In GUEST, the details of this list depend on the content of the requirements document being processed.

```
<goal>
    <indicator>scope of project</indicator>
    <indicator>product scope</indicator>
</goal>
<nf_goal>
    <indicator>user interfaces</indicator>
    <indicator>security requirements</indicator>
</nf_goal>
<use_case>
    <indicator>stimulus/response sequences</indicator>
    <indicator>scenario</indicator>
</use_case>
<ignored_document_section>
    <indicator>overview of document</indicator>
    <indicator>potential risks</indicator>
</ignored_document_section>
```

Figure 3-5. Example of Section Indicator List

Step 2: Linguistic Analysis. In this step, sentences in each identified section are split and analyzed to resolve coreference if any. For instance, if there is a sentence specified as *"The traveler enters the review content and submits it,"* it would be rewritten as *"The traveler enters the review content and submits the review content."* In addition, each sentence is then parsed to obtain a parse tree, part-of-speech (POS) and dependencies between its tokens (words). These tasks are supported by the Stanford Coreference Resolution System [88] and Stanford parser [79].

Step 3: Artifact and Relationship Extraction. The extraction of artifacts and relationships is supported by our extendable set of extraction rules. Each extraction rule specifies which part in a matching sentence contains an artifact specification. Depending on its type, a rule can also indicate the relationships between the artifacts contained in such a sentence. In order to facilitate this process, a rule-checking engine is used to analyze the parse trees, POS of words and dependencies obtained from step 2 to identified sentences that match the rules. The output of this step is the "raw" version of artifact specifications (in the sense that the artifacts' specifications are the fragments of sentences where they are extracted from and no edition has been made on them) and their relationships.

Step 4: Artifact polishing. As discussed in section 3.3, in many cases, a specification of an artifact may be mined from a fragment of a sentence and thus may not be completely stand-alone and/or grammatically correct. In addition, extracted specifications may not be written in accordance with the boilerplates defined by our goal-use case meta-model. Therefore, the purpose of this step is to correct such specifications to ensure their properness. It takes the "raw" version of artifact specifications extracted from step 3 and produces well-formed textual specifications, which are guaranteed to be grammatically correct stand-alone sentences, and conforming to the boilerplates, which implement the requirements specification guidelines.

Step 5: Artifact classification. An important concern of a goal-use case model is how to determine the types of the modeled artifacts. In this step, we address this concern. Apart from some use case components (i.e., steps, conditions) whose classes might have been determined in previous steps (we assume that use cases are written in a certain structure in a requirements document), other artifacts (i.e., goals, constraints) need to be categorized. The extracted and polished artifact textual specifications are taken as inputs into this step. The output indicates the classes they should be in (i.e., business goals, feature goals, non-functional goals, data constraints). This is achieved by our extended version of Mallet, as mentioned in section 3.3.

Step 6: Goal-Use case model construction. After the artifacts and their relationships have been extracted, polished, and classified, a model can be constructed. However, there may be cases that the same artifact is repeated in different sentences in the

document, leading to duplication. This step is intended to deal with this problem. In addition, in this step, GUEST also generates a report as to what problems and issues have been identified through the extraction process that may need users' attention. For instance, a sentence in a requirements document may match multiple extraction rules, and thus lead to different extraction alternatives. The outcome of this step is an initial goal-use case model.

Step 7: Goal-Use case model review. After an initial model is produced, users are required to manually review the generated reports and take any necessary actions to correct errors. For instance, users can review the alternatives of the model presented to them and select one of those alternatives if they believe it is more appropriate. In addition, users may need to review the correctness of the extracted model and make necessary modifications. At the end of this step, a goal-use case model is finalized.

3.4.2 Analysis Process

In GUI-F, the analysis process focuses on the validation and improvement of a goal-use case integrated model. GUI-F automatically provides the detection of 3Cs problems in a goal-use case model and suggests resolutions for those problems. The following steps are taken in this process.

Step 8: Artifact parameterization. Before the detection of 3Cs problems commences, it is necessary to have all artifacts parameterized using functional grammar so that they can be formalized later for automated problem verification. This step is automatically done by our *artifacts parameterizer* (called *FGParam*). It takes the textual specification of each artifact and generates a structured specification. In addition, it establishes the mapping between each individual term in the textual specification and a concept in the associated domain ontology. In case a term does not exist in the ontology, the parameterizer suggests options to add it into the ontology.

Step 9: 3Cs problem detection. In this step, the goal is to identify as many 3Cs problems as possible, with the support from the captured knowledge and semantics in an associated domain ontology. The detection process involves two stages.

First, the model is checked for syntactic problems, which are about its conformance to GUIMeta's constraints regarding artifacts and relationships. For instance, a relationship

is specified between two goals while it is not allowed by GUIMeta, or an artifact specification contains a disallowed semantic function according to GUIMeta.

In the second stage, the focus is on semantic problems that are about the sensitivity of artifact specifications in a given domain. In this stage, the domain ontology is utilized for obtaining the supporting knowledge and semantics. In order to facilitate the corporation of domain ontology and automated analysis, the structured specifications are automatically transformed into Manchester OWL Syntax statements. The explanations for each detected problem are also generated in this step.

Step 10: Resolution alternative generation. In this step, the goal is to support users in resolving the detected problems by generating a list of resolution alternatives. The generation of such resolutions is based on the analysis of the problem's explanations produced in the previous step. There are two important concerns in this process. First, a single resolution may introduce additional problems in the model. Secondly, a resolution may entail other problems to be solved. These circumstances are taken into account during the resolutions generation process to provide the rankings for resolution alternatives (i.e., an alternative leading to new problems may not be desirable as another one which does not).

Step 11: 3Cs problem resolution. The resolution alternatives generated in step 10 are presented to users for their decisions. A resolution may have an associated warning that indicates it may introduce a new problem, or it may entail an existing problem resolved. If an alternative were selected, necessary changes would be automatically made in the goal-use case model. The model would then be validated again to confirm that the problem is resolved and highlight new problems if any. In case users cannot find a suitable resolution or wish to resolve the problems at a later stage, they can make their own modifications or choose to temporarily ignore the problem. In such circumstances, our GUITAR records the situation for future improvement. GUITAR also allows users to report in case they believe the detection result is false.

3.5 Evaluation Approach

The evaluation strategy of our research is based on the use of case studies to verify the accuracy of our techniques, and a number of benchmark validations to compare and

contrast our techniques with the existing comparable approaches. Below, we briefly describe our evaluations conducted for the GUI-F components (which include GUIMeta, the model extraction technique, and the model analysis technique). The details of these evaluations are later discussed in the chapters that describe these components (Chapter 4, 6, and 7).

3.5.1 Conducted Evaluations

To evaluate our meta-model for goal and use case integrated modeling (GUIMeta), we carried out an evaluation with a number of case studies with the objective of verifying:

- *How well does GUIMeta handle different types of artifacts in goal and use case integration modeling?* We aimed to identify which goals and use case components in each case study could and could not be classified by GUIMeta's artifact layer.
- *How suitable functional grammar is for parameterizing artifacts in goal-use case integrated models?* We aimed to investigate each artifact specification in the case studies to identify if it could be sufficiently parameterized using the set of functional grammar's semantic functions adopted in our work.
- *How appropriate are GUIMeta's specification rules for goal and use case integration modeling?* We aimed to we evaluated the appropriateness of our specification rules in specifying the classified goals and use case components. An important consideration is whether the specification rules are overly restrictive, making them not expressive enough.

To evaluate our techniques for automatically extracting goal and use case integrated models from textual requirements documents, we conducted two evaluations. The first one was concerned with the performance of our extended version of the Mallet classifier. We carried out a benchmark validation to compare it with the original version of Mallet and a state-of-the-art requirements classifier developed by Casamayor et al. [18]. The second evaluation was intended to assess the accuracy of our entire extraction techniques. We conducted this evaluation by applying our GUEST tool into two case studies to obtain the extraction results. The results were then manually verified by us to determine its validity. We used the precision and recall metrics to measure GUEST's performance in this validation.

To evaluate our techniques of automatically identifying the 3Cs problems in goal and use case integrated models, we carried out three evaluations. First, we focused on measuring the accuracy of our automated parameterization technique. To do that, we gathered 310 textual requirements, used our tool to automatically parameterize them, and then verified the result manually.

Second, we evaluated GUITAR's performance in detecting 3Cs problems in goal-use case models based on a number of selected case studies. In this evaluation, we first obtained goal and use case models from the case studies. We then used GUITAR to identify 3Cs problems in these models and verified the results manually to measure its performance (using recall and precision metrics).

Third, we conducted a benchmark evaluation to compare and contrast GUITAR's performance with other industrial and academic requirements analysis tools. With tools that were available to download and use, we set up and tested them with our case studies. For tools that were not downloadable but having their demonstrative data available, we ran GUITAR with such data and compared its results with the reported outcomes of other tools.

There were three industrial case studies and one requirements repository used in our evaluations of the GUI-F components, including a Traveler Social Networking System, an Online Publishing System, a Split Payment System, and the PROMISE Requirements Repository [14]. These case studies and repository were chosen since they contain requirements from different domains, and thus diversified our validation data. For the validation of our automated parameterization technique, apart from the PROMOSE requirements data, we have also collected 110 textual requirements from various sources in the literature to further diversify the testing data. These requirements were not used in other evaluations since those evaluations requires tightly related requirements to generate goal-use case models while these requirements came from different projects.

Each single case study was used differently in the evaluations of different GUI-F components. For instance, in the validation for our GUIMeta meta-model, we manually identified goals, use cases, and other artifacts from the case studies and verified if they can be classified and specified properly using GUIMeta's artifact definitions,

classification, and specification rules. In the validation of GUEST regarding the extraction techniques for goal-use case integrated models, we prepared the requirements documents in the case studies so that they satisfied GUEST's formatting requirements. We then applied GUEST into the documents and verified its extraction results. The benefit of using of the same set of case studies in most evaluations was that we could get better understanding of the case studies (as opposed to using different case studies for different types of evaluation). This enabled us to minimize mistakes in the manual verification of the results produced by our proof-of-concept tools.

Figure 3-6 summarizes the evaluations that we have conducted in our research, grouped by the GUI-F components, and the case studies that were used in each evaluation. In the next section, we introduce these case studies.

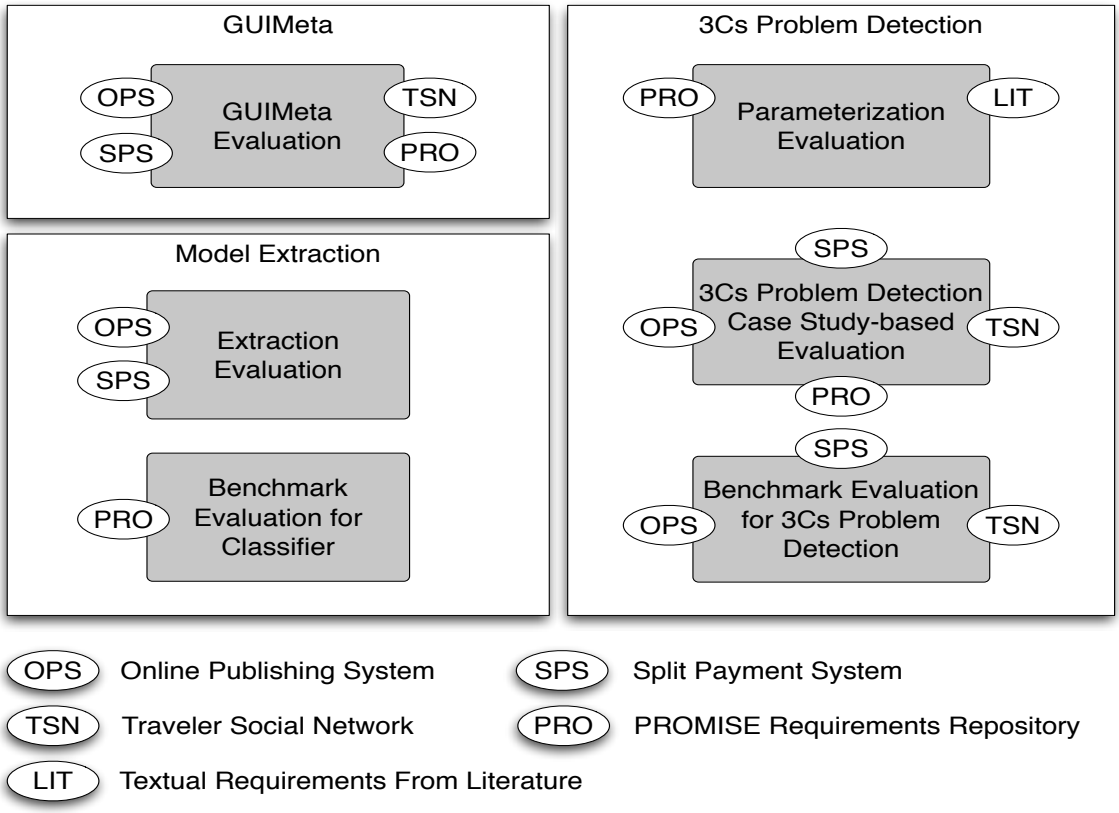


Figure 3-6. Summary of Conducted Evaluations

3.5.2 Case Studies

Below, we provide a brief introduction to these case studies and requirements repository. The requirements documents from these case studies can be found in the

Appendix A1. Due to the large size of the PROMISE requirements repository, interested readers can find it on our supporting webpage at http://goo.gl/gCUofM.

The PROMISE Requirements Repository

The PROMISE (**PR**edict**O**r **M**odels **i**n **S**oftware **E**ngineering) Requirements Repository [14] is a collection of requirements specifications from 15 software projects in a Master's level class at DePaul University, which was made publicly available. The PROMISE dataset have been used as evaluation data in various requirements engineering research. This repository contains 625 requirements. Among them, 255 items are marked as functional requirements and the remaining 370 non-functional requirements are classified into 11 categories, such as *security*, *performance* and *usability*.

The PROMISE dataset was used in several evaluations in our research. First, it was used as part of our evaluation of GUIMeta. Second, it was employed as part of our evaluation for GUITAR to evaluate the accuracy of our 3Cs problems detection techniques. Third, it was used in the validation of our automated parameterization technique. Four, it was used in the benchmark validation to evaluate the performance of our extended version of the Mallet classifier.

Due to the large size of the PROMISE dataset, we only used part of it in the first three evaluations. In the evaluations of GUIMeta and GUITAR, we randomly selected 3 projects in the 15 projects in the PROMISE dataset and used them together with the requirements from the other three case studies. These projects are in the domains of Master Scenario Events List Management, Real Estate and Nursing Training Program Administration, respectively. Moreover, in the evaluation of our automated parameterization technique, we randomly selected and used 200 requirements from the dataset.

On the other hand, the entire PROMISE dataset was used in our benchmark evaluation for requirements classification since the result of Casamayor et al.'s classifier (which was chosen as a benchmark application in this evaluation) was produced based on the original version of the dataset. We thus used the same version of data as theirs to ensure the results were comparable.

Online Publishing System (OPS)

This case study is concerned with the development of an online publishing system that supports the process of submitting articles, reviewing articles, accepting articles, and publishing articles to journal. The objective of this system is to maximize the efficiency and productivity of the journal editors by automating the article review process. The key users are the system include editors, authors, reviewers, and readers. Some of the key features of this system include:

- Search articles: this enables readers of the journal to search for a particular article by title, authors, or category, and download the article to their machines.
- Author's support: this include features that allow authors to submit their articles and communicate with the editors.
- Review's support: this includes features that allow reviewers to submit their reviews, and communicate with the editors.
- Editor's support: most features of this system are intended for editors since they are the primary users of the system. These include: retrieving article submission, assigning reviewers to articles, updating reviewers and authors' information, checking article statuses, or publishing accepted articles.

This case study was used in most of our evaluations, including the validation of GUIMeta, GUITAR (for 3Cs problem detection accuracy, and benchmark validation), and GUEST (for model extraction accuracy). The main content of the requirements document from this case study contains 26 pages, including text, pictures and tables. According to our investigation, the document contains 63 requirements and 11 use cases, which then include 137 statements such as use case steps and conditions.

Split Payment System (SPS)

This case study is concerned with the development of an Android split payment application, which is aimed to facilitate an efficient process of tracking and settling shared expenses. By using the synchronization and notification features, the application enables users to be updated with the latest status of the payments. The key features of this application are as follows:

- *User registration:* which allows users to register for accounts in order to use the application
- *Group creation and management:* allows users to create groups of people who have shared expenses, add new members to groups, or delete groups
- *Uploading bills and resolving debts:* allows users to upload bills to their groups and select options for splitting expenses among members
- *Settings:* allows users to customize his/her preferences in the application

Similar to the OPS case study, SPS case study was used in most of our evaluations. The main content of the requirements document from this case study contains 40 pages. However, only 26 pages in it include requirements details, the rest of the document contains design information. We have identified from this document 67 requirements and 16 use cases, which then include 96 artifacts such as use case steps and conditions.

Traveler Social Network (TSN)

This case study is concerned with the development of a social networking system, intended to be used by travelers around the world. The key objective of the system is to provide a place where travelers can connect to each other, share and gain travel experience with the ultimate purpose of improving their travel planning quality. Some of the key features of this system include:

- *User management:* this allows users to register memberships, edit their accounts, reset their passwords. It also includes support for administrators to monitor, ban, or remove user accounts
- *User Interaction:* this includes features allowing users to send friend request, follow other users, create groups, making comments, or sending messages
- *Forum:* offers a place for asking/answering questions and sharing experience.
- *Content creation:* this involves features allowing users to write reviews for attractions or services, or create travel articles

Since this case study comes from a confidential industrial project, only part of the requirements document was available for our evaluation and publishing. The main content (cover page, table of content excluded) of this partial requirements document was 19 pages long. Within this document, we identified 124 requirements and 20 use

cases, which then contain 218 statements such as use case steps, pre/post-conditions, and so on. This case study was used in several evaluations, including the validation of GUIMeta, GUITAR (for 3Cs problem detection accuracy, and benchmark validation).

3.6 Chapter Summary

In this chapter, we provided an overview of our approach to modeling and analyzing goal-use case models. We presented a motivating scenario to illustrate the research problems and discussed the requirements for possible solutions to these problems. Based on such requirements, we presented our techniques and methods to extract goal-use case models from textual requirements documents, and analyze such models for 3Cs problems including inconsistency, incompleteness and incorrectness. An overview of the modeling and analysis process in our GUI-F framework was also provided. Moreover, we discussed our evaluation approach in this research, and introduced the set of case studies and requirements repository used in our evaluations. In the next chapters, we discuss our techniques in details. Chapter 4 presents the specification parameterization method using functional grammar and the details of our meta-model for goal and use case integration (GUIMeta). Chapter 5 and 6 describe our techniques for extracting goal and use case integrated models from natural language-based requirements documents. Chapter 7 discusses our ontology-based approach to analyzing goal and use case integrated models for inconsistency, incompleteness and incorrectness.

Chapter 4

Goal and Use Case Integration Meta-model

In this chapter, we present our Goal-Use Case Integration Meta-model (GUIMeta) that provides a conceptual foundation for integrating goal and use case modeling. GUIMeta consists of two layers. The *artifact layer* provides a comprehensive classification of goals, constraints, structure of use cases, and defines relationships between them. The *specification layer* provides specification rules for these artifacts. We have employed functional grammar as the underlying grammatical theory to parameterize artifact specifications. Such parameterization provides the basis on which the specification rules are developed. GUIMeta and the functional grammar-based parameterization approach are our solution to address both requirements R1 (regarding a conceptual foundation for goal and use case integrated modeling) and R3.1 (regarding structuring artifact specifications to effectively capture their semantics) discussed in Chapter 3.

Section 4.1 describes GUIMeta and its layers in details, including some background about functional grammar and discusses how it is used to parameterize artifact specifications in our work. Section 4.2 discusses our evaluation results. In Section 4.3, we present a discussion about the relationship between GUIMeta and the existing goal-use case integration approaches, the benefits and limitation of GUIMeta.

4.1 Goal and Use Case Integration Meta-model

In this section, we present GUIMeta, our two-layered meta-model for goal and use case integration. GUIMeta offers a conceptual foundation that governs the process of modeling and analysis of goal and use case integrated models. GUIMeta's *artifact layer* provides a comprehensive classification of artifacts and defines relationships between them. The *specification layer* provides specification rules for the defined artifacts based on the functional grammar-styled parameterization. For instance, it offers the guidelines on what semantic functions should and should not be included in specifications of each type of artifacts.

The design of our meta-model is based on over 450 goals and 190 use cases from the literature and industry[3]. These are from different domains including web applications, embedded systems, process control systems, and information systems. The development of GUIMeta was done in two steps. First, we focused on the *artifact layer*. We studied the existing goal-use case integration modeling, goal-use case coupling and requirements abstraction categorization approaches (i.e., [91], [154], [133], [76], [55]) to identify the overlaps and differences between their defined artifacts. Based on that, we developed the core categories of artifacts. We then used the exemplar goals and use cases to recognize the artifacts that are not supported by those approaches and integrated them into the categories.

Second, we analyzed individual goals, constraints and use cases. We used functional grammar to parameterize the textual specifications of these artifacts, and studied the commonalities among semantic functions usually used for each category of artifacts. From such results, we developed a collection of specification rules for each artifact category. Moreover, we also took into account some common requirement specification practices (i.e., [152, 172]) to ensure the artifact specifications to conform to standard requirements practices. For instance, functional goals should be described in the form of "<subject> + shall be able to + <action verb>," or the use of adjective + preposition phrases should be avoided. The outcome of step two is the *specification layer*.

[3] They can be found on the project website at http://goo.gl/gCUofM

Below, we describe GUIMeta's *artifact* and *specification layers*. In the discussion on the specification layer, we provide some background on functional grammar and how it is used for artifact specification parameterization.

4.1.1 Artifact Layer

The *artifact layer* provides a classification of artifacts and relationships in goal-use case integrated models. In the following sub-sections, we discuss these concepts in details.

4.1.1.1 Artifacts

Figure 4-1 illustrates the "goals and constraints" part of GUIMeta's artifact layer, which defines a wide range of goals across levels of abstractions:

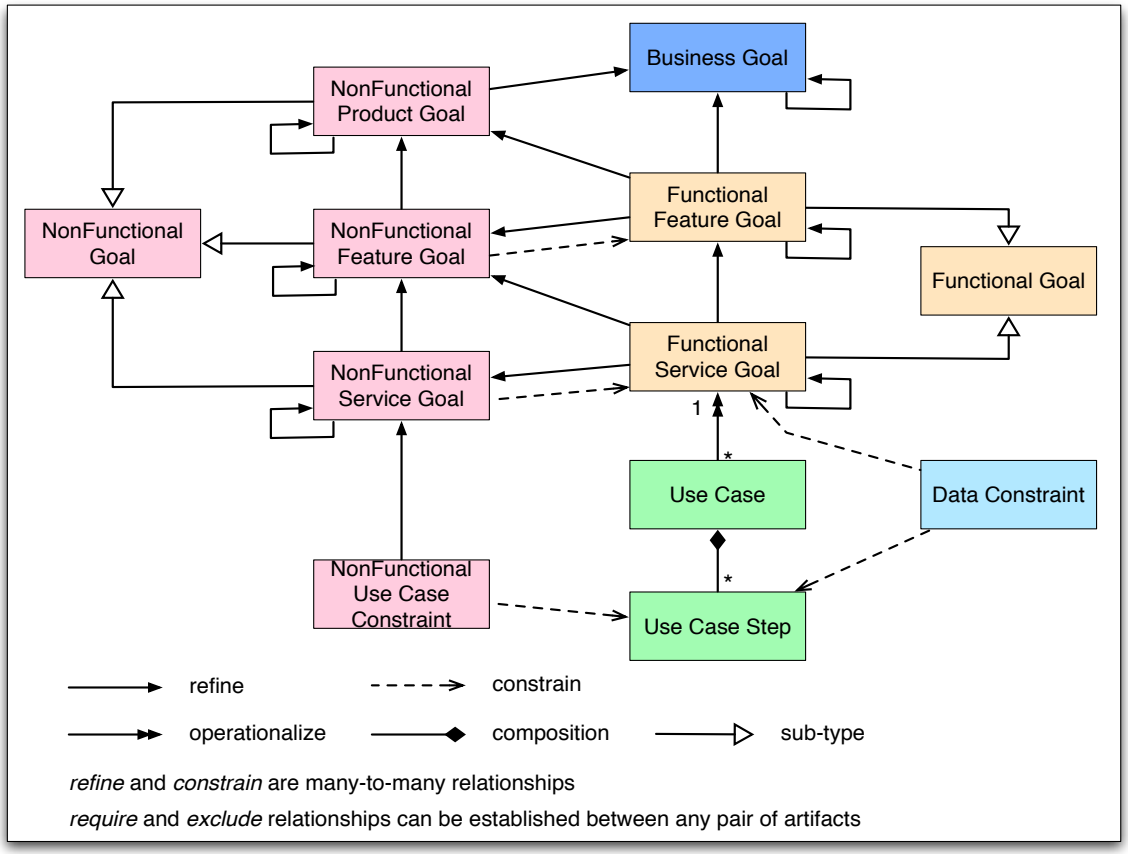


Figure 4-1. Artifact Layer ('Goals and Constraints' Part)

Business goal (BG): a BG describes the business objectives of the software system being built. A business goal does not include any information about what the system should do or how it should operate. For instance, *"Improve quality of travel planning"* or *"Reduce workload for employees."*

Functional feature goal (FFG): a FFG describes a feature the system should support to achieve one or more business goals (or higher feature goals). A goal at this level should not offer details as to what functions are needed in order for the product to support a feature; rather it should be an abstract description of the feature itself. An example of functional feature goals is *"System shall support communication."* In this example, it is known that the system shall support a feature that allows users to communicate.

However, it is not specific enough to provide information regarding what types of communication (i.e., text messages, voice mail) to be offered. In another example, the functional feature goal *"Users shall be able to share experience with others"* describes a requirement for an *"experience sharing"* feature, yet what services that the system offers for users to share their experiences are not known.

Functional service goal (FSG): a FSG provides details as to how a feature is achieved. It contains a description of what function a user can perform. The main difference between a FSG and a FFG is that, a FSG is specific enough to form a testable unit and can be operationalizable by a use case; while a FFG describes a high-level feature and thus cannot have any connected use case. For example, *"Users shall be able to send text messages to other users"* and *"Users shall be able to create a travel article"* are the FSGs that refine the above discussed FFGs.

Non-functional product goal (NPG): a NPG describes quality constraint on the entire product. It should not contain any information about particular features or services supported by the system. For instance, *"The system shall be secure"* or *"the system shall be always available to users."*

Non-functional feature goal (NFG): a NFG describes a quality constraint of a particular feature of the system (i.e., which is specified in a FFG). A NFG should not contain detailed information about how such constraint can be met by the system. For example, *"Users shall be able to share experience easily" or "Ensure users can communicate at any time."*

Non-functional service goal (NSG): a NSG describes a quality constraint on a particular service. For example, *"The article creation process is familiar to typical Internet users"* is a NSG that restricts the FSG *"Users shall be able to create travel articles."*

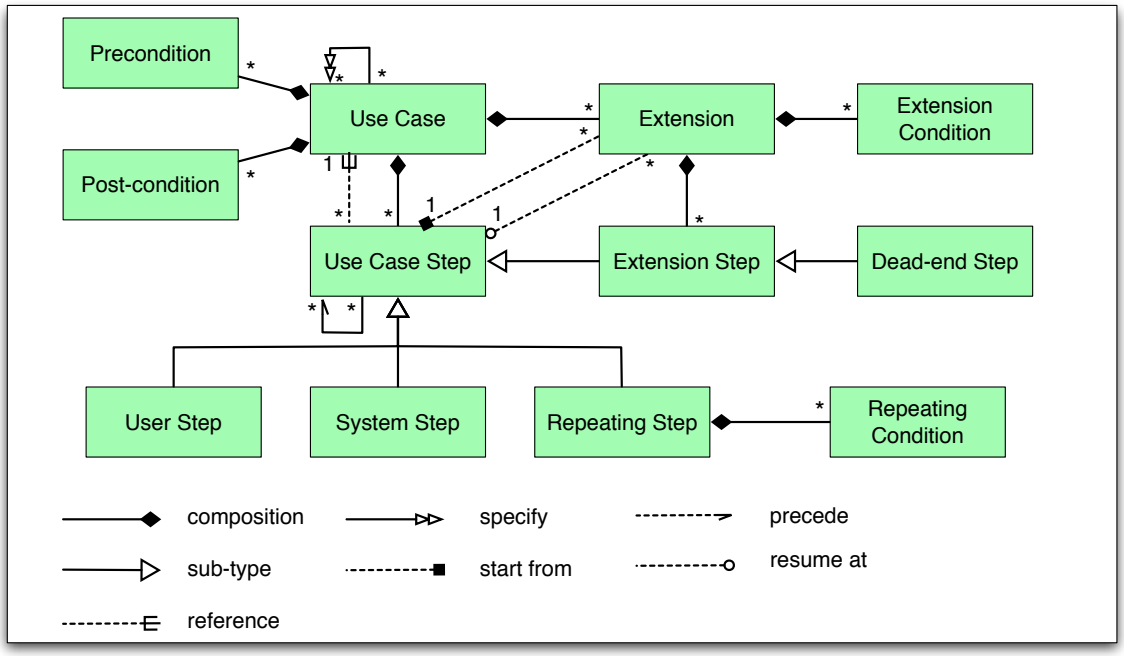


Figure 4-2. Artifact Layer ('Use Case Structure' Part)

In GUIMeta, a use case describes the interactions between a user and a system in order to achieve a functional service goal. Such connection is modeled by an *operationalize* relationship. Due to the focus on only requirements engineering, we keep the internal interactions of system components, which are more about system architecture, out of scope. In other words, the software system to be developed is considered as a "black box" and thus system-user interactions are the lowest level artifacts to be modeled in our framework. Our use case structure is adopted from Cockburn's use case template [28]. Figure 4-2 presents the structure of use case. It contains the following components:

- ***Description:*** a textual description of the use case (e.g., *"a user creates a review"*). Description is an attribute of a use case. It thus does not appear in Figure 4-2, which shows only components.
- ***Precondition:*** a condition that must be satisfied before the use case can be started. For instance, *"The user has already logged in."* A use case may have multiple preconditions.
- ***Post-condition:*** a condition that must be satisfied after the use case is completed. For instance, *"A new review is stored in the system."* A use case may have multiple post-conditions.

- ***Use case step:*** there are three types of use case steps. *User* and *system steps* describe the interactions between a user and the system with the purpose of achieving the goal associated to (*operationalized* by) the use case. *User steps* are those carried out by users while *system steps* are performed by the system. Some examples of these types of steps are: *"The user fills out the review creation form"* and *"The user submits the form"* (*user steps*), *"The system validates the review creation form"* (*system step*).

  Another type of steps is *repeating step*. Repeating steps do not describe system-user interactions, they rather control the flow of other steps in a use case. For instance, *"The user repeats step 3 to step 5 until a sufficient number of members is added to the group."* A repeating step contains three components: a starting step (i.e., step 3), an ending step (i.e., step 5) and one of more terminating conditions (i.e., *"sufficient number of members is added to the group"*) that specify when to terminate the repetition. All steps in a use case must to be strictly ordered (no cyclic order).
- ***Main success scenario:*** in our framework, main success scenario is a logical concept, but not an artifact (that is why it is not presented in Figure 4-2). A main success scenario, as the name suggests, considers only the situation when the use case runs from the first step to the last one without any interruption or exceptions.
- ***Extension:*** extensions are used to handle the exceptions in the main success scenario. For instance, when the user fills out and submits the review creation form, if the system detects that the details in the form are not valid, then an extension needs to be added into the use case to handle this problem. An extension contains the following components:
  - ***Starting step:*** the step from which the exception is invoked.
  - ***Resuming step:*** the step where the use case resumes after necessary steps to handle the exception have been taken.
  - ***Extension condition:*** the condition that must be satisfied for an extension to be invoked. For instance, *"The review subject is blank."* An extension may have multiple conditions.
  - ***Extension steps:*** extension steps are similar to the main success scenario steps. They describe the interactions needed to handle the exception.

Apart from goals and use cases, GUIMeta defines two types of lower level constraints to model the artifacts that constrain use case components. A *non-functional use case constraint* (NUUC) describes a quality constraint (i.e., security, usability) on a use case step (e.g., *"The system shall validate the form details in less than 1 second"*). A *data constraint* (DC) captures a data requirement for a particular entity mentioned in a functional service goal or use case step (e.g., *"A review contains a subject, a rating, and a comment"*).

4.1.1.2 Relationships

The artifact layer defines the commonly used relationships between artifacts in goal and use case integrated modeling [23, 71, 161]. The followings provide details about these relationships shown in Figure 4-1.

Refine: *refine* relationships are used to model the refinements of goals and constraints in goal and use case integrated models. For instance, if the functional service goal FSG1 *"Users shall be able to write reviews" refines* the functional feature goal FFG2 *"Users shall be able to share travel experiences with others,"* then that means the satisfaction of FSG1 contributes to the satisfaction of FFG2. In this case, FSG1 is called a sub-artifact of FFG2 and FFG2 is called a parent-artifact of FSG1 (we can use more specific terms *"sub-goal"* and *"parent-goal"* in this case since both FSG1 and FFG2 are goals). There are three types of *refine* relationships in GUIMeta:

- ***AND-refine relationship:*** *AND-refine relationships* are used for cases of *minimal refinement*, which means an artifact (i.e., FFG2) would only be satisfied if all the sub-artifacts (i.e., FSG1) linked to it via AND-refine relationships are satisfied
- ***OR-refine relationship:*** *OR-refine relationships* are used in cases of *alternative refinement*, which means the artifact being refined can be satisfied by fulfilling any of the sub-artifacts involved in the OR-refine relationships. Conversely, satisfying any of these sub-artifacts would fulfill the (parent) artifact.
- ***Optional-refine relationship:*** *Optional-refine relationships* are used in cases of *optional refinement*. This denotes that the sub-artifacts involved in Optional-refine relationships are the preferred options but they are not strictly required for the parent-artifact to be fulfilled. They may contribute to the realization of the

artifact being refined. However, without such sub-artifacts, the parent-artifact can still be satisfied. For instance, if FSG1 *optionally refines* FFG2, then the satisfaction of FFG2 is not affected regardless the satisfaction status of FSG1.

Figure 4-3 presents an example showing different types of refine relationships. In this example, a functional feature goal FFG3 specifies a compulsory feature (communication) to allow users to share their travel experiences with others (FFG2). The two services of *"create reviews"* and *"create travel diaries"* (FSG1 and FSG2) are alternatives of each other. At least one of them is needed to satisfy FFG2. Moreover, FFG4 specifies a *"discussion forum"* feature. As indicated by the Optional-refine relationship, this feature is considered *"nice-to-have"* yet not critical to enable the experience sharing between users.

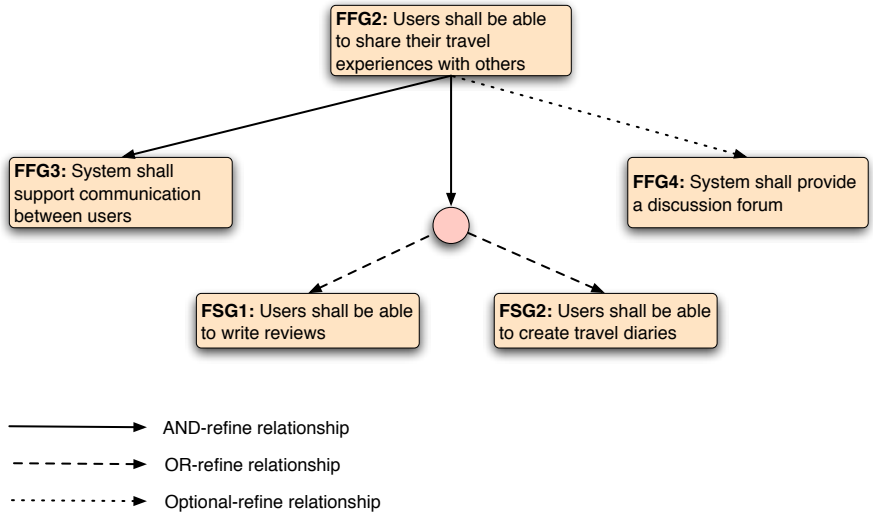


Figure 4-3. Examples of *Refine* Relationships

Only goals and non-functional use case constraints may be involved in a *refine* relationship. There are also a number of rules in regard to this type of relationships:

- A non-functional use case constraint can only *refine* a non-functional service goal.
- A non-functional goal never refines a functional goal (it instead constrains a functional goal, which is discussed later in this section).
- A non-functional goal only refines the non-functional goals that are either on the same level of abstraction or one level higher in the abstraction hierarchy (referred to as "same or one-level higher non-functional goals"). For instance, a

non-functional feature goal can only refine another non-functional feature goal or a non-functional product goal. It cannot refine a non-functional service goal.

- A functional goal only refines its "same or one-level higher functional goals" and the non-functional goals that constrain those goals. For instance, a functional service goal can refine another functional service goal, a functional feature goal, a non-functional feature goal, or a non-functional service goal. It cannot refine a business goal or a non-functional product goal.

Constrain: *constrain* relationships are used to define non-functional or data constraints on a functional goals or a use case step. The following rules are used to determine the eligibility of constrain relationships between artifacts:

- A data constraint can only be connected to a functional service goal or a use case step via a *constrain relationship*.
- A non-functional use case constraint can only be connected to a use case step via a *constrain relationship*.
- Non-functional product goals are the highest-level non-functional goals. They thus do not constrain any other artifacts (rather refine business goals).
- Non-functional feature goals can only *constrain* functional feature goals and non-functional service goals can only *constrain* functional service goals.
- An artifact can constrain and can be constrained by multiple artifacts.

Require: *require* relationships are used to describe situations in which the satisfaction of an artifact requires the satisfaction of another. For instance, the functional service goal FSG4 *"Users shall be able to edit their reviews"* requires another functional service goal FSG1 *"Users shall be able to write reviews"* since a review cannot be edited if it has not been created.

Exclude: *exclude* relationships are opposite to *require* relationships. They describe the situations in which two artifacts in the model cannot be both satisfied. In other words, the system being developed must not fulfill the requirements specified in both artifacts. An *exclude* relationship is bidirectional, which means if an artifact A *excludes* an artifact B, then B *excludes* A. As an example, FSG1 "Users shall be able to write reviews" and FSG4 *"Only admins shall be able to write reviews" exclude* each other since they cannot be both satisfied in a system.

Operationalize: an *operationalize* relationship is used to connect a use case to a functional service goal to model the situation that the use case describes the system-user interactions to achieve the goal. In GUIMeta, a use case can only operationalize one goal while a goal can be operationalized by multiple use cases. For instance, the goal FSG1 *"Users shall be able to write reviews"* can be operationalized by use case UC3 *"A user writes a review for a hotel"* and use case UC4 *"A user writes a review for an attraction."*

Below we discuss the relationships within use cases that are shown in Figure 4-2.

Specify: *specify* relationships are used to model the cases when both use cases describe similar tasks but one is more detailed than another. For instance, use case UC1 *"A user creates a content"* only describes abstractly the steps required to generate a general content while use case UC2 *"A user creates a review"* contains *specific* details about the interactions to create a travel article that is a *specific* type of content. Similarly, the use case UC3 and UC4 above are *specific* use case of UC2.

Reference: *reference* relationships model the cases that a use case uses another use case to perform one of its steps. For instance, given the use case UC1 *"A user creates a review."* Assume that in the first step of UC1, the user needs to log into the system, then we have step UC1_Step1 *"The user logs into the system."* In order to facilitate the login process, UC1 invokes another use case UC5 that describes the process for a user to log into the system. In this case, UC5 is *included* as part of UC1. GUIMeta allows this situation to be modeled by using a *reference* relationship between UC1_Step1 and UC5 (UC1_Step1 *references* UC5).

Precede: *precede* relationships are used to model the ordering of steps in a use case. For instance, if the step UC1_Step1 *precedes* UC1_Step2, then UC1_Step1 is before UC1_Step2 in the use case that they are in.

Start from: *"start from"* relationships are used in use case extensions. Since an extension defines an exceptional path to the main success scenario, it is necessary to know from which step in the main success scenario the exception is raised. A *"start form"* relationship then connects the extension to that step. Each extension must have one *"start form"* relationship.

Resume At: similar to "start *from*" relationships, *"resume at"* relationships are used to mark the point where the main success scenario is resumed from an extension. If an extension leads to a failure of the use case (the post-conditions cannot be reached), then it does not have a *"resume at"* relationship with any step in the main success scenario.

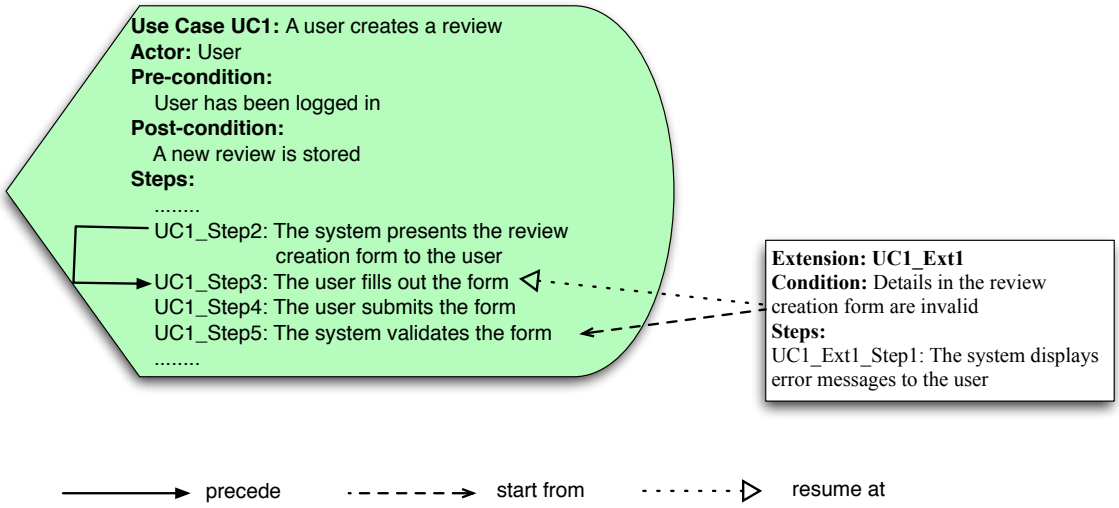


Figure 4-4. Examples for *precede - start at - resume at* Relationships

Figure 4-4 presents examples for *precede*, *"start from"* and *"resume at"* relationships. The step UC1_Step2 precedes the step UC1_Step3 since it should happen before UC1_Step3 in the use case. Similarly, UC1_Step3 precedes UC1_Step4 and so on.

The extension UC1_Ext1 is aimed to handle the cases when the system detects some invalid details in the review creation form. It is invoked from the step UC1_Step5 in which the system validates the form details. Therefore, UC1_Ext1 *starts from* UC1_Step5. In this extension, the system displays error messages to the user to help them recognize what problems are detected in the form. The main success scenario then *resumes at* step UC1_Step3 where the user needs to fills out the form again.

Since different types of relationships are defined between artifacts in GUIMeta, it is important to know how these relationships are related to each other. Table 4-1 summarizes the dependencies between the discussed relationships. One of the important characteristics is that *refine* and *constrain* are special cases of *require* relationships. Specifically, if FSG1 *refines* FFG2, then FFG2 *requires* FSG1 because the satisfaction of FFG2 depends on the satisfaction of FSG1. Similarly, if NSG1 *constrains* FSG1, then NSG1 *requires* FSG1 because if FSG1 is not satisfied, the satisfaction of its

constraints (i.e., NSG1) is unnecessary. The dependencies between relationships form the basis on which a goal-use case integrated model can be analyzed for syntactic 3Cs problems.

Table 4-1. Dependencies between Relationships in GUIMeta

Formal Description	Explanation
require(a_1, a_2) ∧ require(a_2, a_3) → require(a_1, a_3)	Require is a transitive relationship. If a_1 requires a_2, a_2 requires a_2 then a_1 requires a_3
refine(a_1, a_2) → ¬ refine(a_2, a_1)	If a_1 refines a_2, then a_2 cannot refine a_1
refine(a_1, a_2) → require(a_2, a_1)	If a_1 refines a_2, then that means a_2 requires a_1
exclude(a_1, a_2) → exclude(a_2, a_1)	Exclude is a bidirectional relationship
exclude(a_1, a_2) → ¬ require(a_1, a_2)	If a_1 excludes a_2, then a_1 does not require a_2
constrain(a_1, a_2) → require(a_1, a_2)	If a_1 constrains a_2, then a_1 requires a_2
specify(u_1, u_2) → ¬ specify(u_2, u_1)	If u_1 refines u_2, then u_2 cannot refine u_1
precede(s_1, s_2) ∧ precede(s_2, s_3) → precede(s_1, s_3)	Precede is a transitive relationship. If s_1 precedes s_2, s_2 requires s_2 then s_1 precedes s_3
isStepOfUseCase(s, u) → ¬ reference(s, u)	If s is a step in the use case u, then s cannot reference u
startsFrom(e, s) → ¬ resumesAt(e, s)	If an extension e starts from a use case step s, then it cannot resumes at s
hasDeadEndStep(e, s_1) ∧ resumesAt(e, s_2) → ∅	An extension e cannot have a dead-end step while resuming to the main success scenario
¬ hasDeadEndStep(e, s_1) ∧ ¬ resumesAt(e, s_2) → ∅	An extension e must have a dead-end step or resumes to the main success scenario
a_1, a_2, a_3 are non-use case artifacts in in goal-use case integrated model s, s_1, s_2, s_3 are use case steps u, u_1, u_2 are use cases e is a use case extension	

4.1.1.3 Non-functional Concern Categories

Each non-functional goal or constraint has a concern. For instance, the non-functional use case constraint *"The system shall validate the form details in less than 1 second"* has a *speed* concern while the non-functional service goal *"The article creation process is familiar to typical Internet users"* is concerned with the *usability* of a system. GUIMeta allows additional classification of non-functional goals and constraints based on their concerns by proving a collection of non-functional concern categories presented in Figure 4-5. These categories are derived from Lamsweerde's work on the taxonomy of non-functional goals [163] and Sommerville and Kotonya's work on the classification of non-functional requirements [84]. The non-functional concerns are divided into two groups: *quality of service* and *compliance*. Each group is then refined into sub-categories such as *security*, *reliability*, *performance* (under quality of service) or *legal* and *standard* (under compliance).

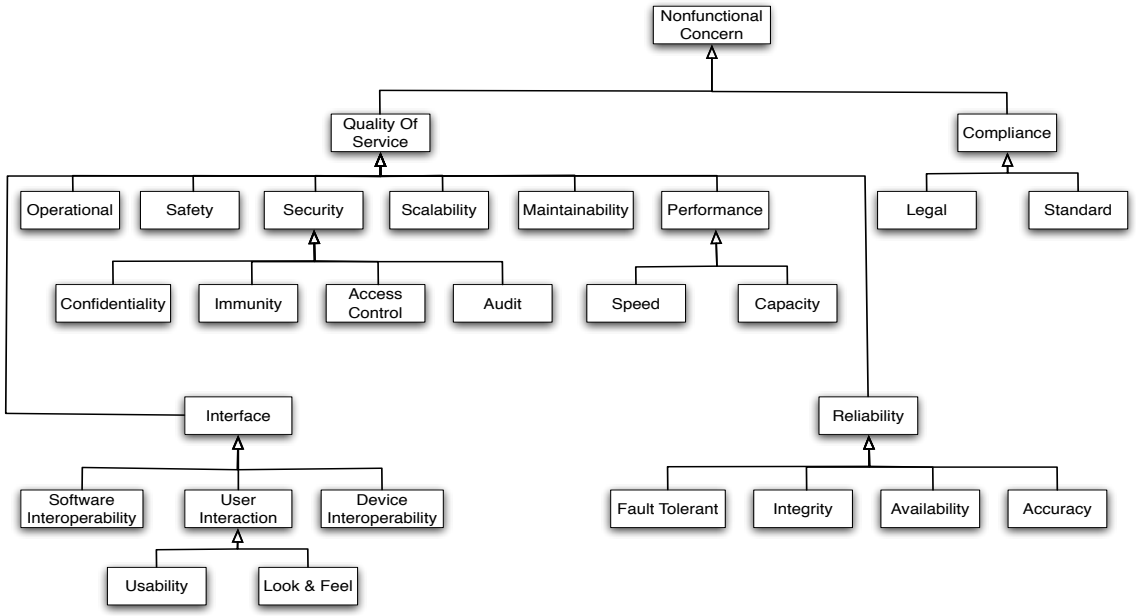


Figure 4-5. Non-functional Concern Categories

The classification of non-functional goals and constraints based on their concerns may bring a number of benefits. First, that helps identify potential relationships between artifacts. For instance, given a functional service goal FSG1 *"Users shall be able to create reviews"* which refines a functional feature goal FFG2 *"Users shall be able to share experience with others."* FFG2 is constrained by a non-functional feature goal NFG2 *"Users shall be able to share experience easily."* FSG1 is constrained by a non-

functional service goal NSG1 *"The article creation process is familiar to typical Internet users."*

Assume that no relationship has been specified between NSG1 and NFG2, then that could be alerted as incompleteness since these two goals express the same concern (usability) on the same feature (FFG2 describes an *"experience sharing"* feature while FSG1 specifies a function – create review, to support that feature). A refine relationship should be established between them in which NSG1 refines NFG2 since it specifies a more specific constraint to ensure *"users to share experience easily."*

Second, the categories of non-functional concerns can be used as means to identify potential missing artifacts in a goal-use case integrated model. For instance, if there exists a certain concern in the categories (e.g., usability) that does not have any corresponding non-functional artifact specified (an artifact that has such concern), then it can be alerted that there is a missing artifact that has that concern (e.g., a usability non-functional product goal) in the model.

4.1.1.4 Goal and Use Case Integration Modeling with GUIMeta

In this section, we present an example that shows how goal and use case integrated modeling is supported by the concepts defined in GUIMeta. Figure 4-6 illustrates a partial goal-use case model built from requirements for a traveler social networking system that is intended to improve the quality of travel planning (of travelers). As shown in the figure, the artifacts are organized into different categories that reflect the artifacts' functional-non-functional characteristic and the levels of abstraction.

The top artifact is a *business goal* (BG1) that denotes the objective of the system, which is to improve the quality of travel planning. BG1 is then refined into a number of *functional feature goals* (i.e., FFG1, FFG2) and *non-functional product goals* (i.e., NPG1, NPG2). These goals are then further refined to identify more specific features, services or constraints that the system must satisfy. For instance, the functional feature goal FFG2 *"users shall be able to share their travel experiences with others"* is refined into another functional feature goal FFG3 *"system shall support communication"* and *functional service goals* like FSG1 *"users shall be able to write reviews"* and FSG3 *"users shall be able to write travel articles."*

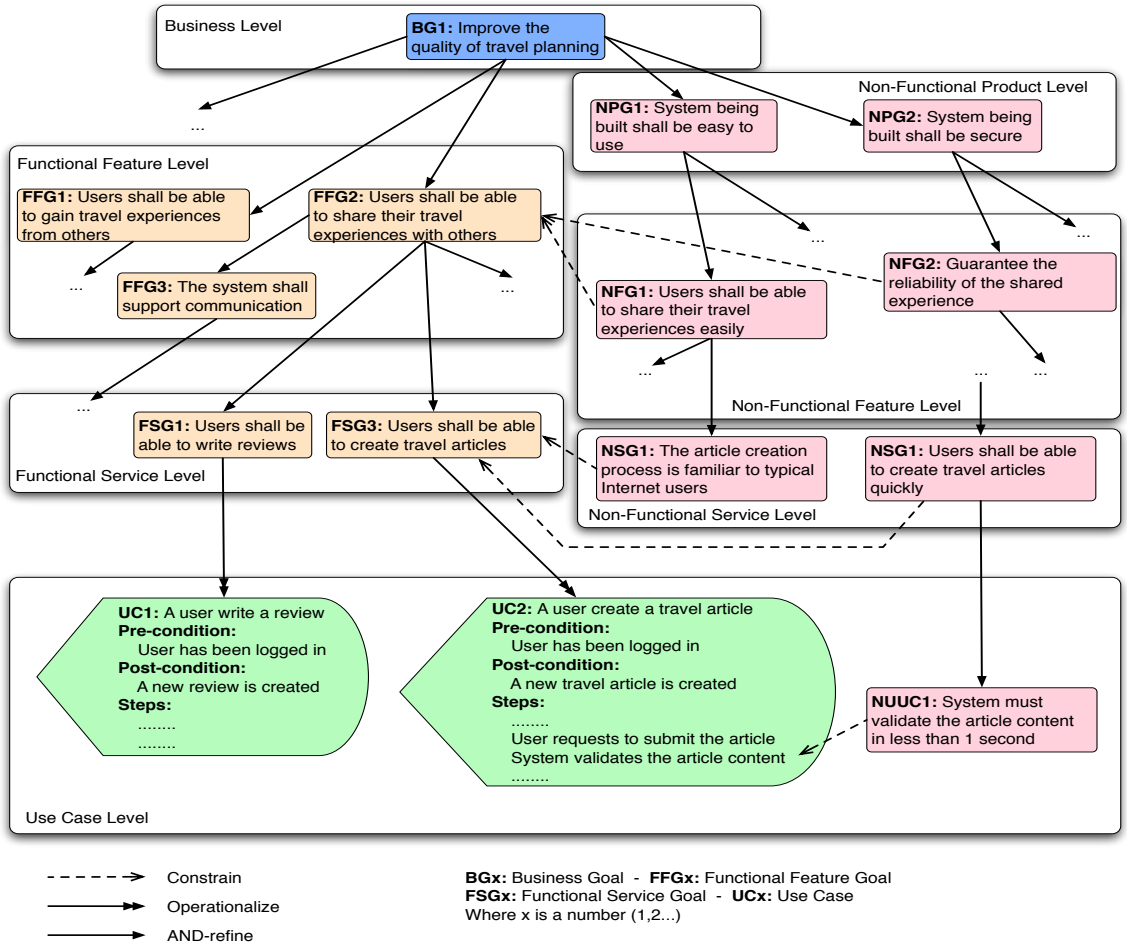


Figure 4-6. A Sample Goal-Use Case Integration Model

The identified non-functional product goals are also refined into more specific non-functional goals that provide constraints on the identified functional feature goals and functional service goals. For example, the *non-functional feature goal* NFG1 *"users shall be able to share experiences easily"* provides a usability constraint on FFG2 on the *feature level* and its sub-goal NSG1 *"the article creation process is familiar to typical Internet users"* provides a more specific usability constraint on FSG3 on the *service level*. FSG3 is also constrained by another non-functional service goal NSG2 *"users shall be able to create travel article quickly."*

The functional service goals FSG1 and FSG3 are operationalized by the use case UC1 and UC2 respectively. These use cases describe the user-system interactions in order to achieve their associated goals. In our example, a UC2's step (*"system validates the article content"*) is constrained by a non-functional use case constraint NUUC1 that specifies *"the system must validate the article content in less than 1 second."* NUUC1 is a refinement of NSG2 into the *use case level*.

4.1.2 Specification Layer

We first provide some background about functional grammar and how we used it to parameterize textual specifications of artifacts in goal-use case integrated models. We then discuss the specification rules for the artifacts defined in the artifact layer.

4.1.2.1 Functional Grammar-based Specification Parameterization

Functional Grammar is a general theory concerning the grammatical organization of natural languages [40]. The key idea of functional grammar is to structure a natural language sentence into different parts (called *semantic functions*); each of them has a unique semantic role. The combination of different semantic functions forms a meaning of a sentence and a modification of any of these functions would change the meaning of that sentence. Functional grammar is native to the field of linguistics. In requirements engineering, a number of authors have applied functional grammar in their approaches [76, 126, 133]. In our work, functional grammar provides means to consistently formalize and structure natural language-based artifact specifications. In this section and the rest of this thesis, we use the term *"artifact"* to refer to goals, constraints, use cases and use case components (i.e., steps, conditions) in goal-use case integrated models.

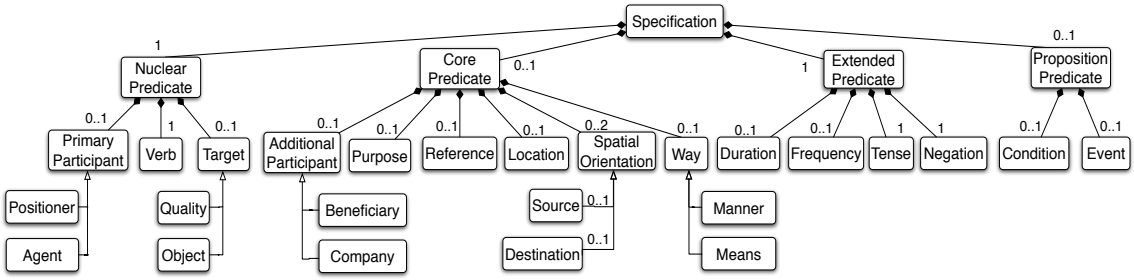


Figure 4-7. The Structure of a Specification

4.1.2.1.1 Semantic Functions

Figure 4-7 presents the components of an artifact specification in our work (i.e., goals, use case steps, use case conditions, etc.). A specification consists of four predicates, each denoting a number of semantic functions. *Nuclear predicate* contains the basic elements of a sentence. That is, which action is conducted (verb), by whom (agent), or on what target (object)? *Core predicate* enriches *nuclear predicate* with information related to the beneficiary of an action or how an action is to be performed (manner). *Extended predicate* further enhances the statement with information about the duration

or frequency of an action. *Proposition predicate* provides information about when or under what conditions an action is to be carried out. As shown in Figure 4-7, verb is the compulsory semantic function in any specification. Depending on the type of an artifact, other semantic functions can be specified. Table 4-2 provides the definitions of the semantic functions used in GUIMeta.

Table 4-2. Definition of Semantic Functions[4]

Semantic Function	Definition
Agent	The entity pursuing the action (used with action verbs)
Positioner	The entity being described (used with to-be or possessive verbs)
Object	The entity being possessed by another entity or affected by an action
Quality	A quality to be attained or preserved
Source	The entity from which something moves/ is moved
Destination	The entity towards which something moves/ is moved
Location	The entity where something is located
Beneficiary	Person or group for (or against) whose benefit the action is carried out
Purpose	The entity towards which the action is carried out
Reference	The second or third term of a relation with reference to which the relation is said to hold
Manner	The manner by which the action is taken
Means	The instrument used when an action is taken
Duration	The duration of an activity
Frequency	The frequency of an activity
Event	The event which results in the action to be carried out
Condition	The condition whose satisfaction results in the action to be carried out
Tense	The tense in which the action occurs, either *present* or *past or future*
Negation	It is to negate the statement when needed. It has a Boolean value (*true* or *false*)

[4] Tense and Negation are our self-introduced semantic functions

The following examples demonstrate the use of functional grammar in parameterizing artifact specifications. In our work, we refer to the parameterization of specifications as *parameterized specifications*.

Example 4-1: The goal *"System shall notify users immediately when new messages arrive"* is parameterized as *"Agent(system) + Verb(notify) + Object(users) + Manner(immediately) + Event(new messages arrive) + Tense(present) + Negation(false)."*

Example 4-2: The use case step *"If the review has less than 50 characters, the system shall display an error message"* is parameterized as *"Condition(the review has less than 50 words) + Agent(system) + Verb(display) + Object(error message) + Tense(present) + Negation(false)*."

Example 4-3: The non-functional goal *"The system shall be secure"* is parameterized as *"Positioner(system) + Verb(tobe) + Quality(secure) + Tense(present) + Negation(false)."*

Example 4-4: The functional goal *"System shall send update emails to users every 2 weeks"* is parameterized as *"Agent(system) + Verb(send) + Object(update email) + Beneficiary(user) + Frequency(2 weeks) + Tense(present) + Negation(false)."*

Example 4-5: The non-functional goal *"Users who do not have technical background can easily create reviews"* is parameterized as *Agent(users who do not have technical background) + Verb(create) + Object(reviews) + Manner(easily) + Tense(present) + Negation(false)* .

Example 4-6: The use case step *"System displays a prompt for amount to the user"* is parameterized as *"Agent(system) + Verb(displays) + Object(prompt for amount) + Beneficiary(user) + Tense(present) + Negation(false)."*

4.1.2.1.2 Terms

Semantic functions provide a way to determine the semantic roles of certain groups of words in a sentence. In other words, the "content" of each semantic function may include multiple words. In order to fully parameterize a specification, Prat [126]

proposed a formalization of various terms based on functional grammar to represent the internal structure of each semantic function's content in the context of goal modeling. We have extended his work (and functional grammar theory) to provide the definitions of additional terms to adequately representing goal-use case model specifications. In this section, we discuss these terms[5] in details.

Verbal Term

Verbal terms constitute the underlining structure of specifications in our work. A verbal term is made up from all components described in Figure 4-7. Due to their nature of representing verb phrases in sentences (those with a *verb* followed by additional details), verbal terms are used to provide the internal structure of condition and event semantic functions, and verbal restrictors (discussed in the section) in our work. The use of verbal terms is illustrated in the following examples. Note that the "VerbalTerm" keyword is used in these examples for demonstration purpose only. In our presentation of parameterized specifications, it is preferred not to include such keyword for the sack of simplicity and readability (the absent of such keyword generally does not cause any confusion).

Example 4-7: The event "new messages arrive" in Example 4-1 is parameterized as *"Event(VerbalTerm(Positioner(new messages) + Verb(arrive) + Tense(present) + Negation(false)))."* This is normally written as *Event(Positioner(new messages) + Verb(arrive) + Tense(present) + Negation(false))* (without "VerbalTerm").

Example 4-8: The condition "*the review has less than 50 characters"* in Example 4-2 is parameterized as *Condition(VerbalTerm(Positioner(review) + Verb(Have) + Object (less than 50 characters))*. The internal structure of the *object* semantic function should be described by a nominal term (discussed in the next section).

Nominal Term

Nominal terms are used to parameterize noun phrases that describe entities. Figure 4-8 illustrates the internal structure of a nominal term. A nominal term consists of one or

[5] While Prat uses the word "group" (i.e., nominal group) to refer to internal units of a semantic function's content, we prefer to use "term" (i.e., nominal term), which is used also in functional grammar theory, since we believe it better refers to internal structures.

more atomic nominal terms that are connected by conditional operators (i.e., and, or). Each atomic nominal term has three components as follows:

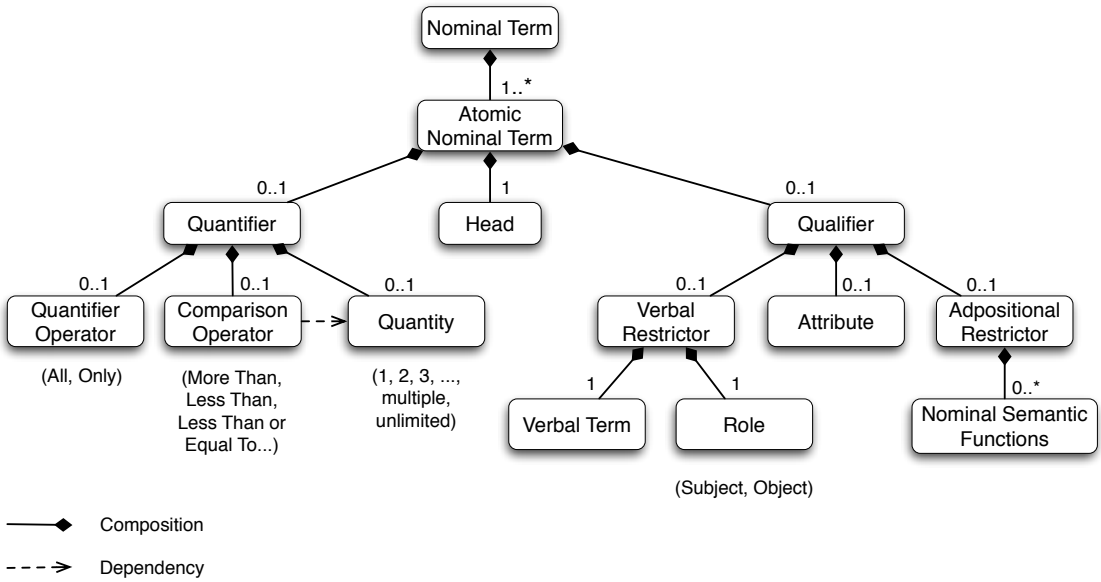


Figure 4-8. Structure of Nominal Terms

- ***Head:*** is the compulsory component of an atomic nominal term. It represents the core of a noun phrase (that is always a noun). For instance, *'characters'* is the *head* of the noun phrase *"less than 50 characters"* (in Example 4-8).
- ***Quantifier:*** is an optional component of an atomic nominal term. It is used to represent the elements that indicate quantities in a noun phrase. A quantifier optionally contains a *quantity*, a *quantifier operator* and a *comparison operator*. *Quantity* has numeric or percentage values (1, 2, 1.5, 90%…) with two exceptions that are *'multiple'* and *'unlimited.'* A *quantifier operator* can have the value of *'all'* or *'only.'* In linguistic, there may be more quantifier operators, for instance, *'many*,' *'few,'* or *'some.'*

  However, due to the focus on requirements modeling, these operators are not considered in our work since they are discouraged to be used in requirements specifications due to their ambiguity characteristics [75]. A *comparison operator* can be *"less than," "more than," "less than or equal to"* or *"more than or equal to."* A *comparison operator* cannot be used in case the quantity value is not specified. As an example, a *quantifier* component can be used to represent the sub-phrase *"less than 50"* in the noun phrase *"less than 50 characters"* (in Example 4-8).

- ***Qualifier:*** is an optional component in an atomic nominal term. It is used to represent both the pre-modifiers (by *attributes*) and post-modifiers (by *verbal restrictors* and *adpositional restrictors*) in a noun phrase. An *attribute* may hold one or more adjectives (*"and," "or"* connectors are used in case of multiple adjectives) that describe the nominal term's head (i.e., *new* in *"new messages"*). V*erbal restrictors* provide the representation of participles or relative clauses in noun phrases. For instance, a verbal restrictor can represent the relative clause *"who do not have technical background"* in the phrase *"Users who do not have technical background"* in Example 4-5.

  A verbal restrictor is parameterized into a verbal term followed by a *role* component that indicates whether the head of the nominal term is the subject or object of the activity described by the verbal restrictor's verbal term (it is *subject* in the currently considered example). Adpositional restrictors are used to represent prepositional and possessive phrases in a noun phrase. An adpositional restrictor may contain one or many nominal semantic functions; each represents a prepositional or possessive phrase (nominal semantic functions are those describe entities, i.e., *object*, *beneficiary*, *location*, *source*…). For instance, *"for amount"* in the noun phrase *"prompt for amount"* (in Example 4-6) can be described by an adpositional restrictor that has a *purpose* semantic function.

Adjectival Term and Adverbial Term

Adjectival terms are associated with adjectives, in the same way that attributes in nominal terms are linked to adjectives. An adjectival term has no internal structure. Its value is always an adjective. In our work, modifiers of adjectives are not supported (i.e., ***very*** fast, fast***er***, fast**est**, ***really*** good) since they should be avoided for ensuring unambiguity in requirements specifications [75]. As an example, the *quality* semantic function in example 4-3 is parameterized as *Quality(AdjectivalTerm(secure))* (or *Quality(secure)* in a shortened version).

Similarly, adverbial terms are associated with adverbs and have no internal structure. They are used to represent the values of *manner* semantic functions. In example 4-1, the *manner* semantic function is specified as *Manner(immediately)*. Note that it is not necessary to write *Manner(AdverbialTerm(immediately))* since manners can only be specified by adverbial terms.

Time Measurement Term

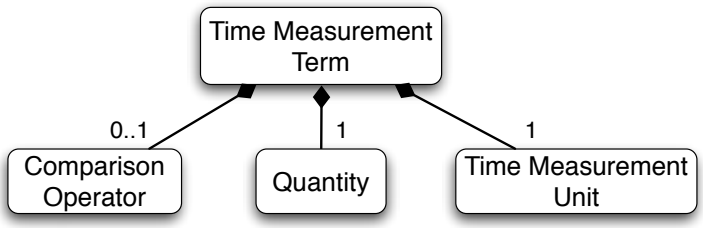


Figure 4-9. Internal Structure of Time Measurement Term

Time measurement terms are our extension to Prat's work and functional grammar theory to describe *duration* and *frequency*. Figure 4-9 illustrates the internal structure of a time measurement term, which consists of a compulsory *quantity* and *time measurement unit* (i.e., *second*, *day*, *week*), and an optional *comparison operator*. For instance, the frequency "every 2 weeks" in Example 4-4 can be structured as *"Frequency(Quantity(2) + MeasurementUnit(week))*." Note that it is not necessary to write *"Frequency(TimeMeasurementTerm(Quantity(2) + MeasurementUnit(week)))*" since frequency can only be represented using term measurement term (the same format applied to duration).

Table 4-3 presents the full parameterization of specifications in Example 4-1 to 4-6 using the discussed terms. For readability purpose, it is preferred not to specify *NominalTerm* where a nominal term is used since the *Head* keyword (and possibly *Quantifier* or *Qualifier*) is sufficient to indicate the use of a nominal term (i.e., *Agent(Head(System))* instead of *Agent(NominalTerm(Head(System)))*). Moreover, since word forms (i.e., plural vs. singular) are not important in the parameterization process, the standard form of words (called *stem*) is used.

Table 4-4 summarizes the possible terms that can be used to represent each semantic function. Nominal term is the only one that can be used to structure semantic functions describing entities. Other than nominal terms, objects can be structured using verbal terms. For instance, consider the goal *"encourage users to share travel experience."* It is parameterized as *Verb(Encourage) + Object(Agent(Head(User)) + Verb(Share) + Object(Head(TravelExperience)) + Tense(Present) + Negation(false)) + Tense (Present) + Negation(false)*. The object of the goal in this situation is an *"experience sharing"* activity carried out by users (which is *'encouraged'*). This type of object is referred to as *verbal object* in our work.

Table 4-3. Summary of Specification Parameterization

Specification	Parameterization
System shall notify users immediately when new messages arrive	Agent(Head(System)) + Verb(Notify) + Object(Head(User)) + Manner(Immediately) + Event(Positioner(Qualifier(Attribute(New)) + Head(Message)) + Verb(Arrive) + Tense(Present) + Negation(false)) + Tense(Present) + Negation(false)
If the review has less than 50 characters, the system shall display an error message	Agent(Head(System)) + Verb(Display) + Object(Head(ErrorMessage)) + Condition(Positioner(Review) + Verb(Have) + Object(Quantifier(Comparison Operator(Less Than) + Quantity(50)) + Head(Word)) + Tense(Present) + Negation(false)))) + Tense(Present) + Negation(false)
The system shall be secure	Positioner(Head(System)) + Verb(ToBe) + Quality(AdjectivalTerm(Secure)) + Tense(Present) + Negation(false)
System shall send update emails to users every 2 weeks	Agent(Head(System)) + Verb(Send) + Object(Head(UpdateEmail)) + Beneficiary(Head(User)) + Frequency(Quantity(2) + MeasurementUnit(Week)) + Tense(Present) + Negation(false)
Users who do not have technical background can easily create reviews	Agent(Head(User) + Qualifier(VerbalRestrictor(Verb(Have) + Object(TechnicalBackground) + Tense(Present) + Negation(true) + Role(Subject)))) + Verb(Create) + Object(Review) + Manner(Easily) + Tense(Present) + Negation(false)
System displays a prompt for amount to the user	Agent(Head(System)) + Verb(Display) + Object(Head(Prompt) + Qualifier(Adpositional Restrictor(Purpose(Head(Amount)))))) + Beneficiary(User) + Tense(Present) + Negation(false)

Table 4-4. Mapping between Semantic Functions and Terms

Semantic Function(s)	Term(s)
Agent/Positioner/Source/Destination/Location /Beneficiary/Reference/Means	Nominal Term
Purpose/Object	Nominal Term, Verbal Term
Quality	Nominal Term, Adjectival Term
Manner	Adverbial Term
Duration	Nominal Term, Time Measurement Term
Frequency	Time Measurement Term
Event/Condition	Verbal Term

4.1.2.2 Specification Rules

The specification layer is aimed to provide specification rules for each type of artifacts defined in the artifact layer. The benefits of these rules are twofold. First, they provide consistent guidelines for writing artifacts. In fact, each type of artifacts in a model has its own characteristics. Some artifacts are more specific while some artifacts are more abstract than others. Some artifacts are concerned with the functionality of a system while others tend to describe the objectives, quality or data constraints of a system, or conditions to be satisfied. Due to such variances, artifacts of different types are normally specified differently. Thus specification rules would help in consistently determining how each artifact should be specified by guiding what should and should not be included in its specification.

Second, since specification rules govern how an artifact should be written, they can be used as the basis for analyzing artifacts for 3Cs problems (i.e., specifications that do not follow the specification rules may be considered problematic).

As discussed in section 4.1, functional grammar can be used to parameterize artifact specifications into different parameters called semantic functions. Each semantic function has a unique semantic role in an artifact specification (i.e., *object*, *beneficiary*, *manner*). Therefore, we choose to formalize the specification rules based on such parameterization. Specifically, each rule defines which semantic functions should and should not be used in a specification of an artifact type. For instance, as business goals are usually high-level strategic statements, *condition* or *duration* should not be specified while other parameters (i.e., *beneficiary*, *destination*) are permitted.

Figure 4-10 shows the specification model for business goals that contains the compulsory and optional sets of semantic functions that can be used to describe a business goal, and the dependencies between them. The semantic functions that do not appear in the figure (i.e., *condition*, *event*) are those should not be used in a business goal specification. As shown in the figure, the compulsory semantic functions are *verb*, *tense* and *negation* (*tense* and *negation* are attributes of a specification rather than contents to be specified).

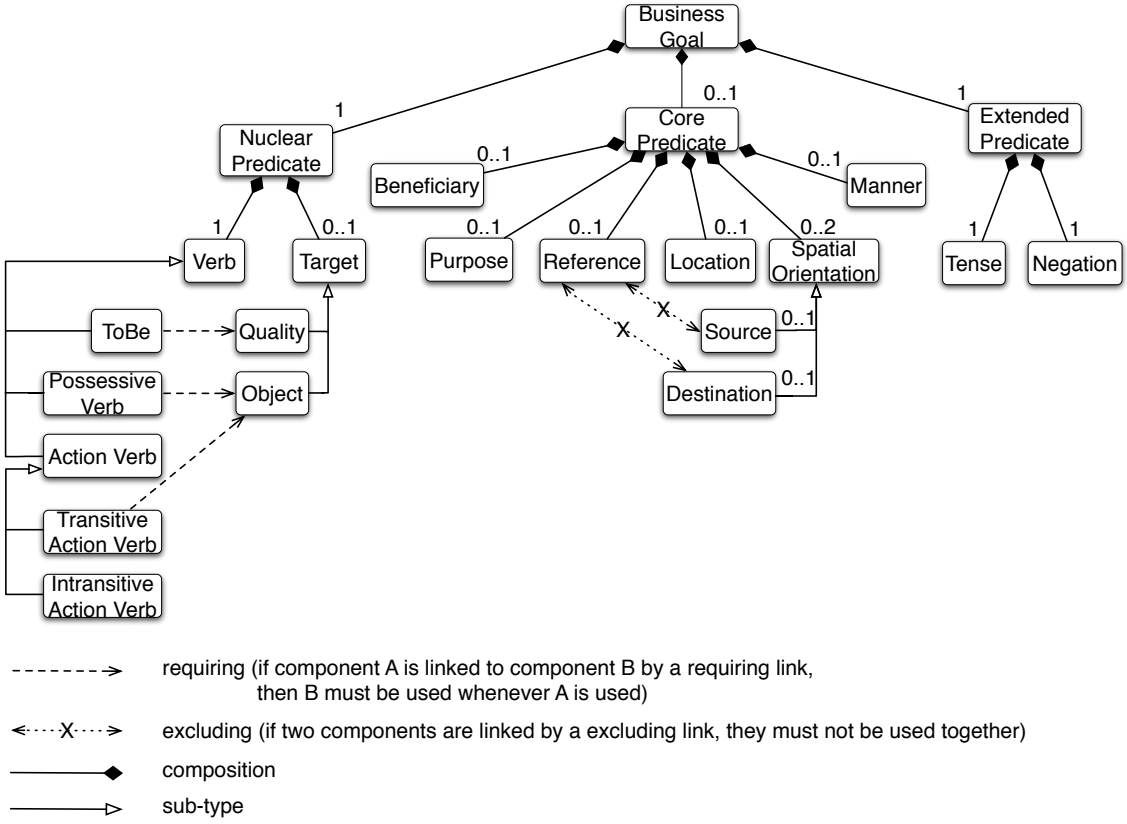


Figure 4-10. Specification Model for Business Goals

Business goal specifications accept *"to-be,"* possessive and action verbs. If the verb used in a specification is *"to-be,"* then a *quality* must be used in the specification (enforced by the *requiring* link). Similarly, if a *transitive action verb* or a *possessive verb* is used, then an *object* is needed. Other types of components in the figure are optional. *"Improve editors' work efficiency"* (parameterized as *"Verb(Improve) + Object(Editor's work efficiency)")* is an example of business goal specifications with only a *verb* and an *object*. A business goal may contain a beneficiary. For instance, *"reduce workload for employees"* is parameterized as *"Verb(Reduce) + Object(Workload) + Beneficiary(Employee)*. However, if a condition were added into the specification (i.e. *"If most employees are working overtime"*), then it would become invalid because a *condition* semantic function is not permitted here, according to the specification rule.

Another important consideration in the specification model is, not all optional semantic functions may appear together in the same specification. According to our investigation of the exemplar requirements, business goal specifications with a *reference* semantic function do not naturally contain a *source* or *destination* (depicted by the *"excluding"*

links in Figure 4-10). For instance, consider a business goal with a *reference* semantic function: *"Align business process with best IT practices."* It is unnatural to add a *source* (i.e., "from a certain location") or *destination* (i.e., "to a certain location") into this specification.

A specification model is used to define specification rules in the form of boilerplates[6]. Each rule outlines one or more ways of writing specifications of a certain type of artifacts. Figure 4-11 presents some specification rules for business goals, which are derived from the specification model in Figure 4-10.

B1. <transitive action verb> <object> ((for) <beneficiary>) ([with | to]) <reference> ([in | at | on] <location>) (for <purpose>)

B2. <transitive action verb> <object> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) (for <purpose>)

B3. <intransitive action verb> ((for) <beneficiary>) ([with | to | of]) <reference> ([in | at | on] <location>) (for <purpose>)

Note: (…) denotes optional parameters. [x | y | .. | z] denotes alternative parameters

Figure 4-11. Example Specification Rules for Business Goals

Each specification rule (or boilerplate) describes possible ways of specifying a business goal. For instance, rule B1 provides possible alternatives that a business goal specification can have a *transitive verb* and a *reference* (denoted by "([with | to]) <reference>" without a surrounding pair of brackets). In that case, an object semantic function (*'<object>'*) must be included. In addition, location and purpose semantic functions are optional (they may or may not be included in a specification of this case). Note that according to the specification model, *source* and *destination* must not be included in this case since a *reference* is already used. However, in rule B2, which specifies the cases when no *reference* is used, *source* and *destination* can be included. The words such as *'with,' 'to,' 'in'* are included in the boilerplates to provide recommendations for specifying textual artifact specifications.

[6] In requirements engineering, boilerplates refer to templates for writing textual requirements

GUIMeta offers specification rules for all defined artifacts, including all types of goals, use case components (use case steps and conditions), data constraints and non-functional use case constraints. In addition, although conditions and events are not artifacts, they are described by verbal terms that can be as complex as some specifications for artifacts. GUIMeta thus also provide specification rules for them. These specification rules are provided in Appendix A2.

4.2 Evaluation

We evaluated GUIMeta by assessing its suitability in goal and use case integration modeling. Specifically, we aimed to address the following questions:

- ***RQ1:*** How well does GUIMeta handle different types of artifacts in goal and use case integration modeling?
- ***RQ2:*** How suitable functional grammar is for parameterizing artifacts in goal-use case integrated models?
- ***RQ3:*** How appropriate are GUIMeta's specification rules for goal and use case integration modeling?

4.2.1 Evaluation Setup

To answer these questions, we employed six industrial case studies, in which three came from the PROMISE requirements repository. In each case study, we manually identified goals and use cases and used them to evaluate GUIMeta. To address RQ1, we verified the coverage of GUIMeta's artifact layer. Specifically, we aimed to identify which goals and use case components in each case study could and could not be classified by our artifact layer. To address RQ2, we investigated each artifact specification to identify if it could not be sufficiently parameterized using the set of functional grammar's semantic functions adopted in our work. To do that, we first automatically parameterized artifact by using *FGParam* – a Java library that we developed to automate the parameterization of textual artifact specifications based on functional grammar (discussed in Chapter 7). FGParam is also able to indicate the part of text that cannot be parameterized due to the lack of semantic functions to handle it. We then manually verified the generated parameterization results and ensured they were correct. The cases of parameterization failure (specifications could not be fully parameterized) were recorded.

To address RQ3, we evaluated the appropriateness of our specification rules, which are defined in GUIMeta's specification layer, in specifying the classified goals and use case components. An important consideration is whether the specification rules overly restrict specifications to small sets of semantic functions that are not enough to express them (i.e., assume that the parameterization of a functional service goal's specification of *"Users shall be able to withdraw money from their accounts via ATM machine"* has a *means* semantic function – ATM machine, while the specification rules of functional service goals does not allow a *means* to be specified). To do that, we compared each artifact's parameterization with the corresponding specification rules of such artifact to identify if the parameterization of contains a set of semantic functions that is not covered by any of the specification rules. For instance, if a *condition* semantic function existed in the parameterization of a business goal specification, then it was a problem since the *condition* semantic function is not included in the business goal specification rule.

The first three case studies we used came from the domains of traveler social network (TSN), online publication system (OPS) and split payment system (SPS) that we introduced in Chapter 3. Three further case studies were randomly chosen from the PROMISE dataset, which provided requirements in the domains of Master Scenario Events List Management (MSEL), Real Estate (REs) and Nursing Training Program Administration (NTPA). The number of artifacts in each PROMISE case study is smaller than those in the TSN, OPS and SPS because they do not contain use cases and business goals, due to the focus mainly on functional and non-functional requirements of this dataset.

In the TSN, OPS and SPS case studies, we obtained goals, constraints and use cases by investigating their requirements documents. In the PROMISE case studies, although they contained pre-extracted non-functional and functional requirements, each of these requirements was normally a combination of multiple single-sentence requirements. In such cases, we split a requirement into multiple ones. In addition, combined words such as "Program Administrators/Nursing Staff Members" were changed to single words (i.e., Program Administrators). Moreover, each sentence with coordinating conjunctions (i.e., "and") is split into two separated ones. However, if the split sentences have identical structure (i.e., *"The system shall offer the ability to pause the refresh of data"*

and *"The system shall offer the ability to resume the refresh of data"*), we only keep one of them to maintain the structural differences between artifacts. Our evaluation data can be found at http://goo.gl/gCUofM.

4.2.2 Evaluation Results

In this section, we discuss our findings for each of the above research questions.

RQ1: How well does GUIMeta handle different types of artifacts in goal and use case integrated modeling?

Table 4-5 presents our evaluation results. We successfully categorized all 800 artifacts extracted from all six case studies into our defined artifact classes in GUIMeta's artifact layer, including different types of goals, and use case steps, conditions, data and non-functional constraints.

Every artifact was classified into one and only one artifact class, which means there was no confusion regarding which class an artifact should belong to. The C rows in each case study section (i.e., TSN, OPS) in Table 4-5 show the number of artifacts classified into each artifact class. Similarly, the NC rows present the number of artifacts that were not classifiable. The NC values were zero in all case studies, which means all artifacts were successfully classified.

In order to classify the artifacts into our defined classes, we matched the meaning of each artifact with the definition of each class to identify the best suitable class for it. For instance, if an artifact describes a business objective without mentioning a functionality or constraint of a system (i.e., *"Maximize the editor's work productivity"*), then it should be classified as a business goal. In case the artifact describes an overall quality of a system (i.e., *"The application can be accessed at any time"*), then it should be a non-functional product goal.

Table 4-5. GUIMeta Evaluation Results

C: classified **NC**: not classified **RM:** rule mismatched **NP**: not parameterizable
BG: Business Goal **FFG**: Functional Feature Goal **FSG**: Functional Service Goal
NPG: Non-functional Product Goal **NFG**: Non-functional Feature Goal
NSG: Non-functional Service Goal **NUCC**: Non-functional Use Case Constraint
DC: Data Constraint **UCS**: Use Case Step **UCC:** Use Case Condition

		BG	FFG	FSG	NPG	NFG	NSG	NUCC	DC	UCS	UCC	**Total**
TSN	C	4	18	74	7	7	14	19	9	130	60	**342**
	NC	0										**0**
	RM	0										**0**
	NP	0	1	5	0	1	0	0	0	3	1	**11**
OPS	C	3	19	21	4	8	8	9	5	90	33	**200**
	NC	0										**0**
	RM	0										**0**
	NP	0	0	0	0	4	1	3	0	9	0	**17**
SPS	C	2	16	21	8	11	9	9	11	73	3	**163**
	NC	0										**0**
	RM	0										**0**
	NP	0	0	0	0	2	2	4	0	4	0	**12**
MSEL	C	0	0	15	6	2	5	5	0	0	0	**33**
	NC	0										**0**
	RM	0										**0**
	NP	0	0	0	1	0	5	1	0	0	0	**7**
REs	C	0	3	15	11	6	4	2	0	0	0	**41**
	NC	0										**0**
	RM	0										**0**
	NP	0	1	1	2	1	0	2	0	0	0	**7**
NTPA	C	0	8	39	10	9	5	0	10	0	0	**81**
	NC	0										**0**
	RM	0										**0**
	NP	0	1	0	3	1	1	0	0	0	0	**6**

The most complicated classifications we encountered were those involved the decision as to whether an artifact was abstract enough to be placed on the feature level (i.e., functional feature goal, non-functional feature goal) or specific enough to be placed on the service level (i.e., functional service goal, non-functional service goal). The key difference between a functional service goal and a functional feature goal is that the former describes a testable service of the system which can be operationalized by a use case while the later specifies an abstract feature that cannot be operationalized by a use case. In many cases, such classification was made easily by evaluating the abstraction of the language used in the artifact's specification (i.e., *"Support automated notifications"* is very abstract, it thus should be a feature goal) or checking whether a use case existed to operationalize the artifact (then it should be a service goal).

However, in some cases (especially in PROMISE case studies, where full requirements documents were not available), the level of abstraction of an artifact was not clear due to the lack of additional details about it. For instance, consider the artifact *"users shall be able to schedule appointments,"* it was not clear whether this artifact describes a feature regarding appointment scheduling which involves a sequence of activities such as availability requesting, availability sending, date nominating, each operationalized by a use case, or it refers to a simple function of a system that involves some steps such as choosing participants and sending email. In the former case, the artifact should be classified as a functional feature goal. It should instead be a functional service goal in the later case. In such situations, we made relevant assumptions to determine which class an artifact should belong to.

In PROMISE case studies, although there was no use case included, we classified some artifacts as non-functional use case constraints since we believed they might have been extracted from a use case. For example, consider *"the top 1/4 of the table will hold events that occur sequentially,"* this artifact seemed to describe a user interface constraint in a use case (i.e., "manage events" or "view events" use case).

RQ2: How suitable functional grammar is for parameterizing artifacts in goal-use case integrated models?

In order to determine the suitability of functional grammar in parameterizing artifact specifications, we tried to parameterize each specification using our adopted set of

semantic functions. If it was found that a specification was not sufficiently parameterized, we attempted to find a way to rewrite such specification so that its meaning was retained while it could be properly parameterized. For instance, while the specification *"the system shall allow users to create groups in an intuitive way"* could not be parameterizable (since our *adverbial term* used to describe *manners* cannot parameterize the phrase *"in an intuitive way"*), its equivalent version *"the system shall allow users to create groups intuitively"* could be. In case no way could be found, we recorded that the specification was not parameterizable.

The NP (not parameterizable) row in each case study in Table 4-5 shows the number of artifacts that were not parameterizable during our evaluation. According to the numbers, the majority of artifacts were successfully parameterized. There was only 60 out of 860 artifacts (7%) were not parameterizable. The non-parameterizable artifacts fell in to the following cases:

- ***Artifacts with temporal properties:*** for instance, a goal *"System logs a user off* ***after*** *15 minutes of being inactive"* or a constraint *"System shall ensure a locked account to be locked* ***until*** *an admin unlocks it"* was not parameterizable in GUIMeta. This was due to the lack of semantic functions to support for such temporal properties (i.e., after, until).
- ***Artifacts with complicated time expression:*** for instance, *"the product shall be available for use* ***24 hours per day, 365 days per yea****r"* or *"the system shall be available for use* ***between the hours of 8am and 6pm****."*
- ***Artifacts with other non-parameterizable phrases:*** there were also some cases in which artifact specifications contain parts whose semantics are not covered by our set of semantic functions. For example, "*The system will use the stored e-mail addresses* ***as a primary means*** *of communicating information to affected parties*" (the phrase of *"use something as something"* was not supported), *"on a 10x10 projection screen, 90% of viewers must be able to read event data* ***from a viewing distance of 30****"* or *"the product shall be able to distinguish* ***between authorized and unauthorized users*** *in all access attempts"* (no semantic function exists to handle the phrases *"from a distance of ..."* and "distinguish between…").

RQ3: How appropriate are GUIMeta's specification rules for goal and use case integration modeling?

The appropriateness of GUIMeta's specification rules were evaluated by verifying if there existed any parameterizable artifact whose parameterization contained semantic functions that were not included in the corresponding specification rule. As indicated by the zero value in the RM (rule mismatched) row of each case study in Table 4-5, there was no mismatch found between the artifacts' parameterized specifications and their corresponding specification rules (in the total of 800 parameterizable artifacts). This implied that our specification rules did not overly restrict artifact specifications.

4.2.3 Threats to Validity

In this section, we discuss the threats to validity from this evaluation.

External Threat: A threat to external validity was the representativeness of the selected case studies. Having good results with the selected case studies may not imply similar results in others. To reduce this threat, we diversified the data by selecting case studies from different domains while obtaining a large amount of requirements.

Internal Threat: A threat to internal validity was the human factors involved in the evaluation tasks. In RQ1, the classification of artifacts was done manually and thus might be incorrect and subjective. In RQ2 and RQ3, the parameterization of artifacts and the matching of the parameterized specifications and specification rules were also done manually.

To alleviate this, we did the tasks carefully and reviewed them twice after they had been done. In addition, in regards to RQ1, although subjectivity might be involved in the classification of a goal into the feature or service level, that did not affect the overall result of the validation, which was concerned with the possibility that a certain artifact could not be classified into any of our defined class. Moreover, to further minimize the risk of manual artifact parameterization, we used our FGParam tool (which achieved 90% accuracy in our evaluation – discussed in Chapter 8) to generate the initial parameterizations, and then manually verified the results and made necessary corrections. This threat can be further reduced if two or more people with relevant knowledge and experiences were involved in the evaluation.

4.3 Discussion

In this section, we first describe about the correspondence between GUIMeta and the related approaches to integrating goal and use case modeling and requirements abstraction. We then discuss the benefits and limitations of GUIMeta in the modeling and analysis of goal-use case integrated models.

4.3.1 Correspondence between GUIMeta and related approaches

Apart from providing a fundamental foundation for GUIM, GUIMeta was designed to unify existing GUIM approaches. In fact, we provide one-to-one mappings between their concepts and GUIMeta's. This enables the transformation of models from those approaches into our format. Based on that, the combination of models specified in different approaches can be facilitated. In this section, we discuss the correspondence and differences between GUIMeta and the existing GUIM approaches. We discuss goal and scenario coupling approach [133] and requirements abstraction model [55] in greater details since these work share the most commonalities with GUIMeta.

Goal and Scenario Coupling (USC)

As discussed in Chapter 2, the goal and scenario coupling approach was developed by Rolland et al. [133] with the objective of guiding the elicitation of goals using scenarios based on the concept of requirement chunks. There are four abstraction levels of goals defined in this work. *Business goals* describe the ultimate purposes of the system. *Design goals* describe possible manners of fulfilling a business goal. *Service goals* specify possible manners of providing services to fulfill design goals. *Internal goals* describe system component interactions to fulfill service goals. The goal abstraction levels in GUIMeta were inspired by the ones in this work. Apart from the term "business goal" remains the same in our work, "service goal" and "design goal" are termed as "functional feature goal" and "functional service goal" respectively. In addition, internal goals are not used in our work since they tend to describe the internal architecture of a system, while our focus is on requirements engineering. The key difference between GUIMeta and USC is that while USC only supports functional goals, GUIMeta offers a complete framework to integrate goals and use cases in a model. Specifically, GUIMeta provides the classification of non-functional goals, use

case structure, data constraint and non-functional use case constraints (these concepts have also not defined in other existing goal-use case integration modeling approaches). Moreover, GUIMeta defines a collection of relationships that allow artifacts across abstraction levels and categories to be linked together.

Requirement Abstraction Model (RAM)

Gorschek and Wohlin [55] proposed a requirements abstraction model in which requirements are classified into 4 abstraction levels. *Product-level requirements* describe system's high-level features that can be directly comparable to product strategies. *Feature-level requirements* describe lower-level features that a system must support. *Functional-level requirements* specify actions that users can perform with a system. *Component-level requirements* specify steps of how each function is performed. Similar to *internal goals* in USC, *component-level requirements* are not used in GUIMeta since they tend to describe internal structure of a system. The functional-level and feature-level requirements match the functional service and feature goals in GUIMeta respectively. Moreover, while product-level requirements are used to describe very abstract system features, we also classify them as functional feature goals and define refinement links so that can be further refined into more specific feature goals.

Similar to USC, RAM does not provide a classification for non-functional requirements. Thus, one of key differences between GUIMeta and RAM is our categorization of non-functional goals, data constraint, non-functional use case constraints and the definitions of relationships between artifacts.

Other GUIM approaches

GUIMeta also covers most concepts used in other GUIM approaches (i.e., [26, 91, 141, 154]). However, since different GUIM approaches have different foci, it is not possible and also not meaningful to combine all of their concepts into GUIMeta. GUIMeta instead was designed as a general-purpose meta-model for goal and use case model integration. Therefore, there are some concepts from some certain approaches (i.e., agent, object, and event models in [77], or strategic dependency model in [141]) that are not included in GUIMeta. However, GUIMeta can always be extended with new concepts and relationships.

Table 4-6. Summary of Correspondence between GUIMeta and Related Approaches

Approach	Related Approach' concept	GUIMeta's Corresponding Concept(s)
Rolland et al. [133] & Kim et al. [76]	Business goal: ultimate purpose of the system	BG
	Design goal: possible manners of fulfilling a business goal	FFG
	Service goal: possible manners of providing services to fulfill design goals	FSG
Gorschek & Wohlin [55]	Product level requirement: product strategies	FFG
	Feature level requirement: high-level feature that the product supports	FFG
	Functional level requirement: action that users can perform	FSG
Cockburn [26]	Summary goal: system objectives	FFG
	User goal: user's task	FSG
	Sub-function goal: describe user activities	Use case step

For readability purpose, Table 4-6 only provides a summary of the correspondence between concepts in some existing GUIM approaches and those in GUIMeta. The complete list of correspondence is provided in Appendix A3. In this table, BG, NPG, FFG, FSG, NFG, NSG, and NUCC refers to business goal, non-functional product goal, functional feature goal, functional service goal, non-functional feature goal, non-functional service goal, and non-functional use case constraint, respectively.

4.3.2 Benefits and Limitations of GUIMeta

GUIMeta provides a conceptual foundation that governs the process of modeling and analysis of goal and use case integrated models. As discussed in the next chapters, GUIMeta is the core component in our Goal-Use Case Integration framework that provides automated support for the extraction of goal and use case integrated models from natural language-based requirements documents and analysis of such models for inconsistency, incompleteness and incorrectness.

Moreover, developed based on the commonality of a large number of exemplar requirements, GUIMeta's specification rules in the form of boilerplates can be used as guidelines on writing goal and use case specifications. This helps to ensure the properness of artifact specification right in the stages of elicitation and modeling to save analysis effort later on. For instance, the boilerplate *"<transitive action verb><object> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) (for <purpose>)"* provides a template for writing a business goal such as *"reduce workload for users"* (*<location>*, *<source>*, *<destination>* and *<purpose>* are optional parameters).

The key limitation of GUIMeta is that it currently does not support the parameterization of specifications in some particular forms. Firstly, artifacts with temporal properties such as *'until,' 'unless,' 'after x seconds'* are not parameterizable. Secondly, we currently do not have support for specifications with complex time expressions like *"24 hours per day, 360 days per year"* or *"between 8am and 6pm."*

Thirdly, there are also some cases in which artifact specifications contain un-parameterizable phrases such as "*the system will use the stored e-mail addresses* ***as a primary means*** *of communicating information to affected parties"* or *"on a 10x10 projection screen, 90% of viewers must be able to read event data* ***from a viewing distance of 30****."* That is because these properties are not supported by functional grammar. In functional grammar, the discussed temporal properties are all captured as a general "time" semantic function, which does not sufficiently indicate the semantic difference between these properties. For instance, a phrase with *'until'* must be handled differently from a phrase with *'after'* since their semantics are different. They thus cannot be handled by the same semantic function (*'time'*). Moreover, functional grammar lacks support to handle the phrases such as *'use…as something'* and *'from a distance of…'* in the discussed examples.

We plan to extend functional grammar with additional semantic functions to accommodate these cases. For instance, each temporal property would be associated to a unique semantic function so that their semantic roles can be fully differentiated.

4.4 Chapter Summary

In this chapter, we provided a detailed discussion on GUIMeta, our goal-use case integration meta-model that was built as a conceptual foundation for the modeling and analysis of goal and use case integrated models. GUIMeta contains two layers. The artifact layer provides a comprehensive classification of goals and structure of use cases, defines relationships between them. The specification layer provides specification rules for the defined artifacts based on functional grammar-based parameterization. We also provided the details of our validation carried out to evaluate the suitability of GUIMeta in goal-use case integration modeling. We have successfully classified all artifacts in the selected case studies into our defined artifact classes. Moreover, we found that 93% of artifacts could be parameterizable using functional grammar. In the next part of this thesis, Chapter 5 and 6 describe our technique for extracting goal and use case integrated models from natural language-based requirements documents. Chapter 7 discusses our ontology-based approach to analyzing goal and use case integrated models for inconsistency, incompleteness and incorrectness.

Chapter 5

Rule-based Goal and Use Case Integrated Model Extraction

In this chapter, we present our rule-based techniques for extracting goal and use case integrated models from natural language-based requirements documents. These techniques are developed to address the requirements R2.1 and R2.2 discussed in Chapter 3. Our approach relies on the use of natural language parsing techniques and an extendable sets of rules for extracting and polishing goal and use case specifications. Section 5.1 provides a background about the natural language processing techniques related to our work. Section 5.2 describes our rule-based method for extracting goal and use case specifications.

5.1 Natural Language Processing

Natural Language Processing (NLP) is an area of research and application that explores how computers can be used to understand and manipulate natural language text or speech to do useful tasks [22]. NLP techniques have been adopted in different fields to improve the human-computer interactions. In our work, NLP plays a major role in both the extraction and analysis process of goal-use case integrated models. In this section, we provide some preliminary knowledge about natural language parsing and coreference resolution.

5.1.1 Natural Language Parsing

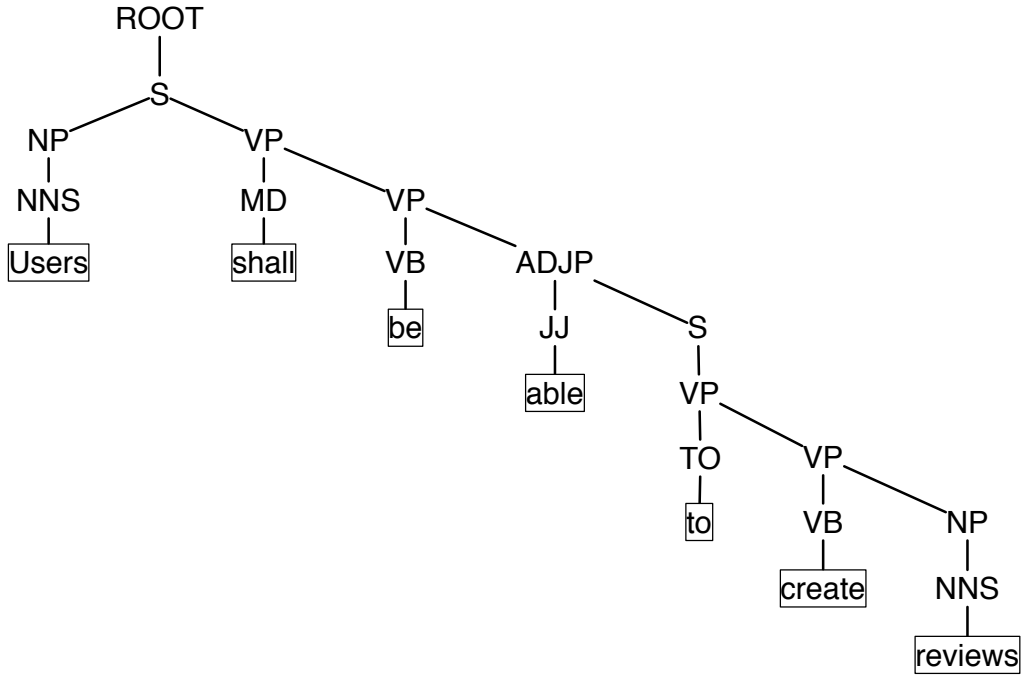


Figure 5-1. Example of Parse Trees

Natural language parsing is used extensively in applications that require the understanding and processing of natural language. It is concerned with analyzing the grammatical structure of sentences (or a sequence of words) to find out the role of each word or group of words in a sentence, and the relationships between them according to some grammar formalism. For example, which words should be grouped together as a noun phrase, verb phrase, or which word is the object or subject of a verb. In our work, we limit the scope to only specifications in English.

The outcome of natural language parsing is presented as a *parse tree*. There are two types of parse trees that are described as follows:

Constituency parse tree

A constituency parse tree breaks a sentence (or a sequence of words) into sub-phrases (i.e., verb phrase, noun phrase). Each sub-phrase is then broken into smaller sub-phrases or individual words from the parsed sentence. The non-terminal nodes in a constituency parse tree present part-of-speech (POS) tags and the terminal nodes (leaves) present words. Figure 5-1 illustrates an example parse tree of the sentence *"Users shall be able to create reviews."*

In this example, the entire sentence (denoted by the 'S' tag under 'ROOT') is divided into a noun phrase (NP) and a verb phrase (VP). The noun phrase is made up by a plural noun (NNS) *"Users."* The verb phrase contains a modal verb (MD – *"shall"*) and a sub-verb phrase that is then broken down into a verb (VB) and an adjective phrase (ADJP) and so one. The process is repeated until individual words are reached. The tags such as VP, NP, VB are called part-of-speech tags, which are used as standard tags by most natural language parsers [21, 79, 104, 119]. They are derived from the Penn Tree bank [101]. Figure 5-2 presents a textual presentation of the same parse tree.

```
(ROOT
   (S
     (NP (NNS Users))
     (VP (MD shall)
       (VP (VB be)
         (ADJP (JJ able)
           (S
             (VP (TO to)
               (VP (VB create)
                 (NP (NNS reviews)))))))))
```

Figure 5-2. Textual Presentation of a Parse Tree

Dependency parse tree

A dependency parse tree, as its name suggests, presents the grammatical dependencies (relationships) between words in a sentence being parsed. In our work, we adopted the Stanford typed dependency collection [35], which was designed to provide a simple description of the grammatical relationships between words in a sentence that can easily be understood and effectively used by people without linguistic expertise to extract textual relations. A typed dependency is a one-way binary relationship between a *governor* and a *dependent*. Figure 5-3 provides an example of a dependency parse tree.

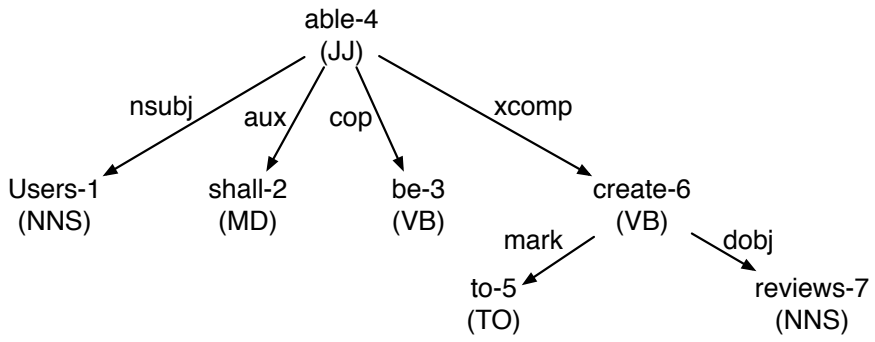


Figure 5-3. Example of Dependency Parse Trees

Each node in a dependency tree represents a word in the parsed sentence. A node is associated with a POS tag and an index (starts with 1) that indicates the position of the corresponding word in the parsed sentence. Instead of showing the phrases as in a constituency tree, a dependency tree provides the relationships between words. In this example, *"able"* is the root of the dependency tree since it only has dependency links pointing to other nodes without any links pointing to it. In other words, *"able"* is never a *dependent* in any dependency in this sentence. All other nodes in the tree are called *descendants* of the root node. *"Users"* is identified as the nominal subject (nsubj) of *"able"* (*"user"* is the dependent, *"able"* is the governor of this dependency), while *"shall"* is the auxiliary verb (aux) that supports *"able"* in the sentence. Similarly, *"to create reviews"* is a clausal complement (xcomp) of the root *"able." "Reviews"* is the direct object (dobj) of the verb *"create."*

```
nsubj(able-4, Users-1)      aux(write-6, to-5)
aux(able-4, shall-2)        xcomp(able-4, write-6)
cop(able-4, be-3)           dobj(write-6, reviews-7)
root(ROOT-0, able-4)
```

Figure 5-4. Textual Presentation of the Example Dependency Tree

Figure 5-4 presents the textual presentation of the dependency tree that uses the syntax of *"dependency_name(governor, dependent)."* Since a textual presentation does not have any visualization to indicate the root of the dependency tree, a *"root"* dependency is used to mark a node as the root. In this case *"root(ROOT-0, able-4)"* denotes that *"able"* is the root of the dependency tree. The index of 0 means a dummy word, which is only used for the governor of a root dependency.

Given such details, dependency trees complement constituency trees to provide a deeper analysis of natural language sentences. As discussed later in this chapter, dependency parsing is the underlying concept of our rule-based goal-use case model extraction approach. There are currently 42 grammatical relations (also called *typed dependencies*) between words in the Stanford typed dependency collection. Table 5-1 provides definitions of 10 most frequently encountered relations during our work. A full reference of the dependencies can be obtained from [36].

Table 5-1. List of Example Dependencies

Semantic Function	Definition
conj	Describe conjunction relation between two words (i.e., and, or)
cop	A copula is the relation between the complement of a copular verb and the copular verb
dobj	The direct object of a verb phrase is the noun phrase which is the (accusative) object of the verb
nsubj	A nominal subject is a noun phrase which is the syntactic subject of a clause
nsubjpass	A passive nominal subject is a noun phrase which is the syntactic subject of a passive clause
mark	A marker is the word introducing a finite clause subordinate to another clause
xcomp	An open clausal complement (xcomp) of a verb or an adjective is a predicative or clausal complement without its own subject
aux	An auxiliary of a clause is a non-main verb of the clause
neg	The negation modifier is the relation between a negation word and the word it modifies
prep	A prepositional modifier of a verb, adjective, or noun is any prepositional phrase that serves to modify the meaning of the verb, adjective, noun, or even another preposition

Natural language parsing tool

There are various natural language parsers that support the constituent parsing [21, 79, 104, 119] and dependency parsing [79, 103, 112]. We decided to use the Stanford parser [79] in this research since it satisfies two important criteria. First, the Stanford parser, among few other parsers, provides supports for both constituent and dependency parsing. Second, the parser reportedly achieved high accuracy in evaluations [19].

The Stanford parser is a statistical parser, which means it uses knowledge of language gained from a collection of manually-parsed sentences to build a *parser model* and based on such model, the parser tries to produce the most likely analysis of new sentences. The process of building a parser model from a collection of hand-parsed sentences is called *training* while a collection of sentences used for training is called *training data*.

One of the biggest limitations of the Stanford parser (and other statistical parsers) is that its parsing result is highly dependent on the quality of its training data. For instance, a sentence may unlikely be parsed correctly if the training data does not include sentences with similar grammatical structures to its. Therefore, the capability of training the parser is critical for our approach, which relies on the parser, to be applied in unlimited domains with different types of requirement specifications. Fortunately, the Stanford parser provides a training feature that allows users to train the parser based on their own training data.

Coreference

Coreference refers to cases in which a pronoun (i.e., *'he,' 'they,' 'her'*), possessive adjective (i.e., *'his,' 'her'*), determiner (i.e., *'this,' 'that'*) or phrase is used to describe an object under investigation with different expression. For instance, in these two consecutive sentences *"Users can create reviews in the system"* and *"Moreover, they can edit their reviews," 'they'* (and *'their'*) in the later sentence refers to *'users'* in the former one. In another example, *'it'* refers to "the form" in the sentence *"The editor fills in the form and submits it."* In the context of coreference, the term that refers to another is called *anaphor* (i.e., *'they,' 'it'* in the examples) while the term being referred to is called *antecedent* (i.e., *'users,' 'the form'*).

Coreference is a serious issue in extracting and analyzing goal-use case model from natural language text since it prevents proper goal or use case specifications to be acquired. For instance, consider again the second sentence in the first example *"They can edit their reviews"* (S1). If the coreference were not resolved (i.e., *'they'* is not changed to *'users'*), the obtained specification would not carry the complete meaning of the original sentence. This obtained specification is also not understandable since it is not known what *'they'* really means. In addition, that prevents the specification to be analyzed. For instance, if there is another specification *"Only system admins can edit reviews"* (S2), then S1 is inconsistent with S2 since S2 implies that users (users are not admins) are not allowed to edit reviews in the system. However, such inconsistency would not be detectable since it is not recognized *'they'* refers to *'users.'*

In our work, we use the Stanford Coreference Resolution System (SCRS) [89] to resolve coreference. The reasons for choosing SCRS are twofold. First, it is among the

most accurate coreference system [89]. Second, SCRS was integrated as part of the Stanford parser, making it a seamless process of resolving coreference and parsing sentences. In the discussed examples, SCRS can be used to rewrite the sentences as *"Users can create reviews in the system. Moreover, users can edit their reviews"* and *"The editor fills in the form and submits the form."*

5.2 Rule-based Goal-Use Case Integrated Model Extraction

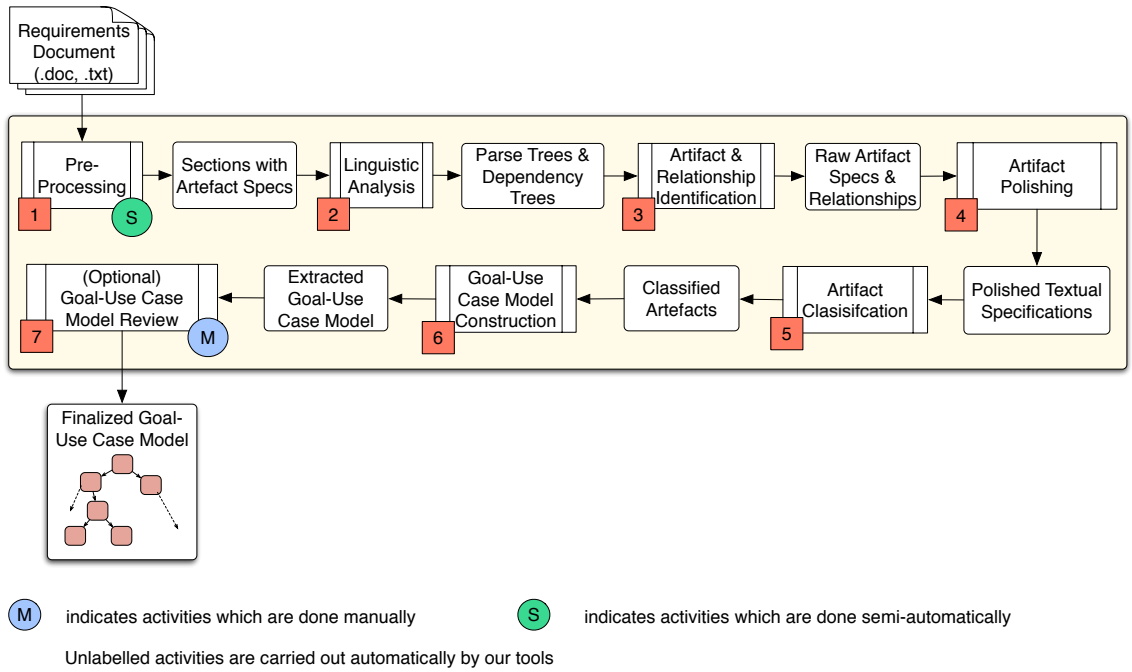


Figure 5-5. Goal-Use Case Integrated Model Extraction Process

Figure 5-5 presents an overview of our goal and use case integrated model extraction approach, which has been briefly described in Chapter 3. Our approach accepts requirements documents in .doc and .txt formats as input. The extraction process starts with pre-processing a requirements document to identify the right sections to extract goals and use cases, and remove unnecessary details such as photos, tables, brackets and symbol characters (Step 1). In step 2, a linguistic analysis is then conducted on the pre-processed text to resolve coreference and generate dependency parse trees. In step 3, a rule-based technique is applied to identify artifacts and relationships between artifacts based on the generated dependency trees and an extendable set of extraction rules. In step 4, the identified artifact specifications are polished to ensure they properly follow our boilerplates. In step 5, the polished artifact specifications are classified into our artifact categories (defined in our meta-model GUIMeta – discussed in Chapter 4). In

step 6, all extracted and classified details are combined to generate a goal and use case model. In step 7, extraction reports generated throughout the extraction process are presented to users who can then optionally make modifications to the extracted model. We have developed GUEST (**G**oal-**U**se Case **E**xtraction **S**upporting **T**ool) to implement the techniques for supporting the entire extraction process.

In this chapter, we describe our techniques for step 1, 2 and 3. The rest of the process, together with some usage examples and evaluation results, are discussed in Chapter 6.

5.2.1 Requirements Document Pre-processing

Our first step in the extraction process is concerned with pre-processing requirements documents to prepare for the linguistic analysis in the next step. The outcome of this step is a number of *document sections* (we use the term *document sections* when appropriate to avoid possible confusions when referring to these sections and the sections in this chapter) associated with information as to what types of artifacts that they contain. Unnecessary and unprocessable details are identified and removed from these document sections. In this section, we first present the format requirements for requirements documents in our approach. We then provide some background about how document section is defined in our work. That is then followed by the discussions about how artifact type indicator list can be created and what should be removed from the original requirements documents to ensure the extraction process to be properly continued.

5.2.1.1 Format Requirements

In order to allow the model extraction techniques to be applied across domains, there is no specific constraint on the structure of requirements documents to be used as input (except that they must be in .doc or .txt formats). However, although GUEST automates some tasks in pre-processing requirements documents, some manual effort is required to ensure the following criteria are met:

- **No Slashes:** A requirements document should contain no slash ("/") that is used with the meaning of conjunctions (i.e., *'and,' 'or'*). Such use of slashes is ambiguous since it is often not clear whether the authors of the documents refers to *'and'* or *'or.'* Slashes thus need to be replaced with appropriate words.

- **Section numbering:** A requirement document should have all sections numbered in a strictly ascending order. For instance, the first section must start with "1", following by "2", or "1.1". "1.1" then is followed by "2", "1.2" or "1.1.1". The section numbered "1.1.1" can be followed by another section numbered "1.1.1.1", "1.1.2", "1.2" or "2". There must not be any case such as "1.1.1" is followed by "3", "1.1", "1.3" or "1.1.3". The section numbering is critical for GUEST to correctly work since it identifies sections by investigating heading numbers. A section number must follows the format of "`(\d+\.)*\d(\.)?`". For instance, it does not matter whether a number ends with a dot (".") ("1.1" or "1.1." are both accepted). We term such strictly ascending order for numbering document sections as *section-numbering order*.
- **No "table of content":** A *"table of content"* section (if any) must be removed from a requirements document since it likely confuses GUEST when looking for document sections (since it contains section numbers).
- **Use case indicator:** Each use case must be in its own section. For instance, if a use case specification is currently in a document section (i.e., 1.1.3), such use case then needs to be within a sub-section (i.e., 1.1.3.1). Moreover, document sections that contain use case specifications must have consistent headings. For example, it should be named by a keyword such as *"use case,"* or *"stimulus/response sequence."* There is no requirement as to which keywords should be use. However, the selected keywords must be used consistently in all use cases and should be correctly included in an artifact indicator list (discussed in the next section).
- **Use case component indicators:** Use case components (i.e., main success scenario, precondition) must also have consistent indicators. For instance, a use case's list of preconditions should be indicated by the keyword *'preconditions,'* a use case extension can be indicated by *'extension,' 'exception'* and so on. Similar to use case section indicators, there is no requirement as to which keywords should be use for use case components. They only need to be consistently used and declared in an artifact indicators list.
- **Important details need to be in plain text:** Use case specifications (and other important details) must be presented in plain text. For instance, if a use case description is in tabular form, it should be transformed to plain text (formatting

such as bold, italic is not a matter). All tables will be automatically removed by GUEST during the pre-processing of requirements documents.

- **Use case step numbering:** Use case steps should not be labeled by only numbers such as "1", "2" since that may confuse GUEST when the tool is looking for document sections. They must be labeled by "Step 1", Step 2" and so on.

5.2.1.2 Section Definition

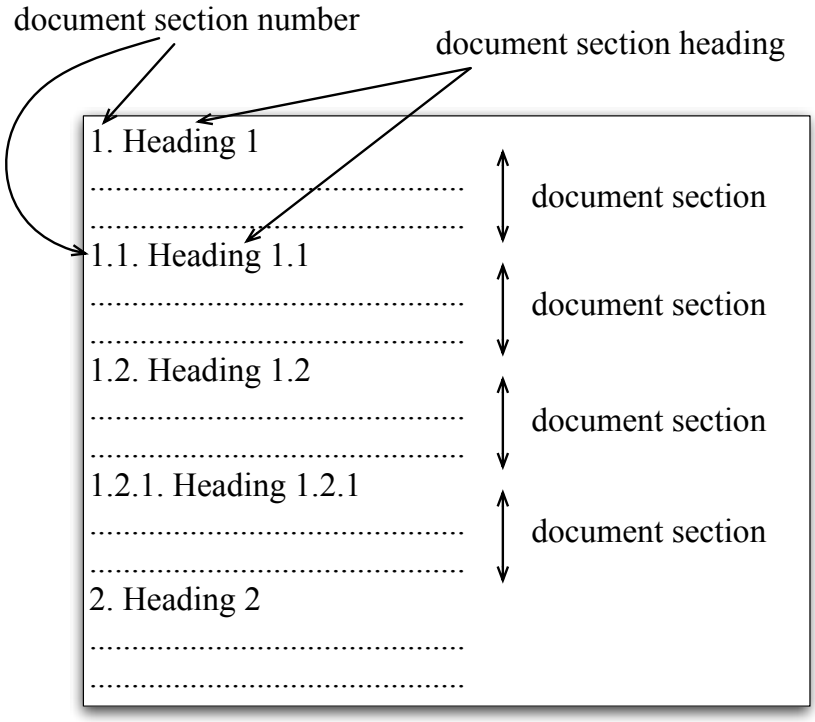


Figure 5-6. Example of How Document Section is Determined

In this section, we provide our definition of document section in GUEST. A document section is defined as a part of document that starts from a document section heading and ends before the *next document section heading*. A heading h_2 (labeled with the number n_2) is considered the next heading of h_1 (labeled with the number n_1) if n_2 is the next number of n_1 according to the *section-numbering order*. The document section s_2 with heading h_2 is then called the *next section* of the section s_1 with heading h_1. Reversely, s_1 is the *previous section* of s_2. Figure 5-6 illustrates our definition of document sections. In this example, the portion of document from "1. Heading 1" to right before "1.1. Heading 1.1" is considered as a document section. Similarly, the portion from "1.2. Heading 1.2" to "1.2.1. Heading 1.2.1" is another section. Since there is no

definition for *"sub-section,"* the document section with number "1.1" is completely separated from the document section numbered "1". This demonstrates why it is critical in GUEST that sections in a requirements document must be numbered in accordance with the *section-numbering order*. For instance, if section 1.1 instead were numbered as "1.2", then the entire document would be considered as a single section (with section number "1" and heading "Heading 1").

5.2.1.3 Artifact Indicator List

```
<non-use_case>
    <indicator>scope of project</indicator>
    <indicator>product perspective</indicator>
</non-use_case>
<nf_goal>
    <indicator>user interfaces</indicator>
    <indicator>security requirements</indicator>
</nf_goal>
<use_case>
    <indicator>stimulus/response sequences</indicator>
</use_case>
<pre-condition>
    <indicator>preconditions</indicator>
</pre-condition>
<use_case_desc>
    <indicator>brief description</indicator>
</use_case_desc>
<main_scenario>
    <indicator>basic path</indicator>
</main_scenario>
<extension>
    <indicator>exceptions</indicator>
</extension>
<use_case_goal>
    <indicator>goal in context</indicator>
</use_case_goal>
<ignored_section>
    <indicator>overview of document</indicator>
    <indicator>potential risks</indicator>
</ignored_section>
```

Figure 5-7. A Sample Section Indicator List

One of the main challenges in extracting goals and use cases from uncontrolled requirements documents is that, not every section in a requirements document contains details about goals and use cases. A requirements document may include sections such as introduction, glossaries or timeline that unlikely contain any details for a goal-use

case model. In addition, use cases and their components may have different indicators in different documents. For example, a use case may be indicated by the keyword "use case" or "stimulus/response sequence," use case extensions can be signified by the keyword "extensions," "exception" or "exceptional paths." Therefore, some manual effort is needed at the beginning of the extraction process to study the documents to identify which sections of the documents being considered potentially contain goal and/or use case specifications (we call such sections *important sections*), which sections should be ignored because they contain unneeded details (we call such sections *unimportant sections*) and which keywords (indicators) are used to indicate use cases and use case components. In GUEST, such information is required to be specified in a so-called *artifact indicators list* in XML format. Figure 5-7 presents an example of a simplified artifact indicator list.

In this example, "scope of project" and "product perspective" are specified as the indicators of sections that contains non-use case artifacts. That means they may contain functional, non-functional goals and data constraints. The <*non-use case*> tag is used when it is not sure what type(s) of artifacts can be found in a section. In such case, the extracted artifacts from that section will then be automatically classified by GUEST (discussed in Chapter 6). Note that since use cases must have their own section numbers and it is assumed that a use case does not contain goals, there is no situation in which a section contains both a use case and goals (an exception is that a use case may contains the goal directly *operationalized* by it. Such goal is normally defined under the "use case goal" component of a use case). A use case specification may contain non-functional use case constraints and data constraints of its steps. They can be automatically identified by GUEST in the extraction process. Therefore, a <*use_case*> tag is used to indicate sections that contain use cases (and their constraints). In case it is sure that a certain section contains only artifacts of a particular type (i.e., non-functional goals), a narrower-scope tag can be used. For instance, it is specified in Figure 5-7 that the *"user interface"* section only contains non-functional goals (<*nf_goal*>).

The sample indicator list also includes indicators for use case components such as precondition (<*pre-condition*>), main success scenario (<*main_scenario*>), use case goal (<*use_case_goal*>), description (<*use_case_desc*>), and extensions (<*extension*>). The <*ignored_section*> tag is used to indicate the sections that should

be ignored in the extraction process. Alternatively, these sections can be manually removed from the considered requirements document to avoid the need to specify their indicators. However, the remaining sections in the document must be guaranteed to have heading numbers in our defined *section-numbering order*.

In case a section heading is not included as an artifact type indicator, it is implicitly understood that such a section has the same type of artifacts as its *previous section*. The list of artifact type indicators varies depending on the content of the requirements document being considered. If the same style of requirements document writing is used by a company in different projects, such list can be reused with little modifications.

5.2.1.4 Automated Pre-processing

After modifications have been made to ensure a requirements document to meet the format requirements and a list of indicators is created, GUEST provides extra support to pre-process the document to remove unnecessary and unprocessable details before the linguistic analysis can be started. The pre-processing steps are described as follows:

- **Extract plain text from the requirements document:** this step is not needed if the requirements document is in .txt format. In case it is in .doc format, GUEST extracts the entire document into plain text by using the Apache POI library[7]. In this step, all figures and tables (if any), which are not processable by GUEST, are removed.
- **Extract important section text:** in this step, GUEST manipulates through the extracted plain text to associate each section in the document with the type(s) of artifacts expected to be extracted from that section based on the manually generated artifact type indicators. The sections whose headings matching the "ignored section" indicators are ignored during this process.
- **Remove other unnecessary details:** in this step, GUEST removes all necessary and unprocessable details from the text extracted from the previous steps. These details include: parentheses, brackets, braces and their contained details, multiple-whitespaces, ellipses, slashes and mathematical expressions. Question marks and exclamations are replaced with periods (".").

[7] https://poi.apache.org/

5.2.2 Linguistic Analysis

After all section text are associated with the types of artifacts and pre-processed, they are analyzed to resolve coreference and producing parse trees. The steps taken in linguistic analysis are as follows:

- **Resolve coreference:** in this step, the Stanford coreference resolution system is run on the text of each section to identify the coreference within and between sentences. Based on the identified coreference, GUEST automatically rewrites the sentences in the section text by replacing the anaphora and their antecedents. For example, the sentences *"Users can create reviews in the system"* and *"Moreover, they can edit their reviews"* would be rewritten as *"Users can create reviews in the system. Moreover, users can edit their reviews"* while "*The editor fills in the form and submits it"* would be rewritten as *"The editor fills in the form and submits the form."* Note that the coreference process needs only to be done for each section instead of the entire (pre-processed) requirements document since coreference is not likely to occur between texts in different sections.
- **Split sentences:** after coreference is resolved, text in each section is split into sentences. This task is supported by the Stanford parser.
- **Parse sentences:** each sentence is then parsed using the Stanford parser to generate dependency parse trees.

5.2.3 Rule-based Artifact and Relationship Identification

The use of rules to extract artifacts and relationships is inspired by our observation that although requirements specification text is freely styled and unstructured, the identification of unimportant phrases or goal relationships usually follow certain patterns. For instance, consider the sentence (S): *"The system will be designed to improve the quality of travel planning by facilitating the communication between travelers,"* the phrase *"The system is designed to"* should be ignored because it contains no important information. The role of this phrase is to introduce an intention following it in the sentence (i.e., *"improve the quality of travel planning"*). If this phrase were used in another sentence with the same role, it should still be ignored. In addition, the words in this phrase do not equally contribute to its unimportance. In fact,

'system,' 'is,' 'designed' and *'to'* are more important than *'the.'* This leads to the conclusion that the phrase *"system <be> designed to"* (do something) (with *<be>* refers to the use of *'is,' 'are,' 'will be,' 'shall be'* and so on) should be ignored in any sentence containing it. Moreover, the refinement relationship between *"facilitate the communication between travelers"* and *"improve the quality of travel planning"* can be recognized by the structure *"do something by doing something"* detected in the sentence. Our observation showed that refinement relationships could be extracted by this structure in most cases.

Note that normal textual comparison cannot guarantee correct extractions. For instance, if we identify refinement relationships by looking for the exact match of *"<verb>+<object>+by+<verb_ing>,"* then we would fail to reveal the relationship in "*improve the quality of travel planning by efficiently facilitating the communication between travelers*" because *'efficiently'* is now between '*by'* and *<verb_ing>*, making the structure unmatched. It is the dependencies between the words that matter, rather than the order they appear in the sentence. In fact, the most important factor in this example is the *'prepc_by'* relationship between *'maximize'* and *'automating'* (the full list of dependencies from sentence (*S*) is presented in Figure 5-8). This relationship would still remain unchanged regardless of what details are added into the related verb phrases of maximize and automating (the relationship between the two words would only change if the connector *'by'* is removed, or either of them is changed, or the sentence structure is modified). This demonstrates that dependencies between words can be used to identify both artifacts and relationships between artifacts from text.

```
det(system-2, The-1)                    nsubjpass(designed-5, system-2)
aux(designed-5, will-3)                 auxpass(designed-5, be-4)
root(ROOT-0, designed-5)                dobj(improve-7, quality-9)
aux(improve-7, to-6)                    nn(planning-12, travel-11)
det(quality-9, the-8)                   xcomp(designed-5, improve-7)
det(communication-16, the-15)           prep_of(quality-9, planning-12)
prepc_by(improve-7,                     prep_between(communication-16,
facilitating-14)                        travelers-18)
dobj(facilitating-14,
communication-16)
```

Figure 5-8. Dependencies from Sentence (*S*)

In our work, extraction rules are used to analyze the dependency parse tree of a sentence to identify if some certain conditions are satisfied. If a rule's conditions were met, some actions would then be executed to extract artifacts and possibly relationships from such a sentence. The outcome of a rule execution is one or more sub-trees of the original dependency tree, accompanied with a relationship between them. The textual specifications of artifacts would then be produced from the extracted sub-trees in the artifact polishing step (to be discussed in Chapter 6). In this section, we discuss our extraction rules in details and present how they are used to identify artifacts and relationships from textual sentences.

5.2.3.1 Extraction Rules

5.2.3.1.1 Extraction Rule Syntax

Table 5-2 presents the generic syntax and an example of extraction rules. An extraction rule is comprised of two components: a *matching condition* and a list of *actions*. A matching condition contains two parts, including a list of *variable declarations* and a list of *matching dependencies* specified based on the declared variables. These matching dependencies are used to determine if a match exists between a dependency tree and a rule. If a sentence has a dependency tree that matches a rule, then we say that sentence matches the rule. During the extraction process, if a dependency tree matches a rule, then the rule's actions are executed. How actions are executed depends on their types.

Table 5-2. Extraction Rule Syntax and Example

Generic Syntax	Example
<Variable Declarations> <Matching Dependencies> -> <Action Declarations>	X={design|aim|intend|target} Y/VB root(X) nsubjpass(X, {system|project}) auxpass(X, {be}) xcomp(X, Y) -> root(Y)

The presented example shows a rule that can be used to identify that the phrase such as *"the system is designed to"* in sentence (S) should be ignored during the artifact extraction. In this example, there are two variables declared. The variable `X` is defined

as any node in a dependency tree that has the *stem* of *'design,' 'aim,' 'intend'* or *'target'* (*stem* is the standard form of a word). The variable Y is defined as a node that has the POS tag of VB in a dependency tree. In other words, Y is a standard verb that can have any value.

A dependency tree matches this rule if and only if there exist a pair of nodes in this tree that matches X and Y, and its set of dependencies cover the list of dependencies specified by the rule. For example, the dependency tree of sentence (*S*) matches this rule since it is found that there exists a set of nodes and dependencies in the tree that completely matches the variables and matching dependencies in the rule. Note that the use of dependencies between specific words in sentences enables us to consider not only the syntactic but also semantic aspects of text in the extraction. For instance, apart from taking into account the grammatical relationships between words, our syntax also allows us to specify the conditions regarding which specific words should be involved in the dependencies or which specific words should be included in the sentence.

Table 5-3 shows the details of these matches. Note that `root(X)` is the shorter form of `root(?, X)`, which means `X` is the root of the dependency tree. The action of this rule is `root(Y)`, which means the root of the tree should be moved to Y, resulting all the nodes that are not part of the tree rooted at Y to be removed (thus, the phrase *"the system is designed to"* is removed). A detailed discussion about rule actions is provided in the next section.

Table 5-3. Matching Variables and Dependencies

Rule Condition	Matching Components
`X={design|aim|intend|target}`	`designed-5`
`Y/VB`	`improve-7`
`root(X)`	`root(ROOT-0, designed-5)`
`nsubjpass(X,{system|project})`	`nsubjpass(designed-5,system-2)`
`auxpass(X, {be})`	`auxpass(designed-5, be-4)`
`xcomp(X, Y)`	`xcomp(designed-5, improve-7)`

Based on the described dependency matching technique, this rule can be applied in similar cases to identify phrases that should be ignored. Table 5-4 presents some of these cases.

Table 5-4. Example Sentences with Ignorable Phrases

Example Sentence	Phrase can be removed
The system shall be intended to improve the quality of travel planning	"The system shall be intended to"
This project is aimed to reduce the workload for employee	"This project is aimed to"
Our system is targeted to automate the artifact publishing process	"Our system is targeted to"

Table 5-5. Syntax for Variable and Dependency Declarations

	Syntax	Meaning & Example(s)
Variable declaration	`X={a} or X={a|b}`	Variable X has a value which then has the stem of a, or one of the stems in a, b… Example: `X={design|aim}`
	`X/AB or X/{AB|CD}`	Variable X has the POS tag of AB, or one of the POS tags in AB, CD… Example: `X/{NN|NNS}`
Dependency Declaration	`root(X)`	Specify that X is the root of a dependency tree
	`dep_name(X, ?)`	There is a *dep_name* dependency between X and any node. Example: `dobj(X, ?)`
	`dep_name(X, Y)`	There is a *dep_name* dependency between X and Y. Example: `xcomp(X, Y)`
	`dep_name(X,?/{AB|CD})`	There is a *dep_name* dependency between X and any node having the POS tag of AB, CD … Example: `nsubj(X, ?/{NN|NNS})`
	`dep_name(X, ?/AB)`	There is a *dep_name* dependency between X and any node having the POS tag of AB Example: `nsubj(X, ?/NN)`
	`dep_name(X,{a|b})`	There is a *dep_name* dependency between X and any node that has a value

	Syntax	Meaning & Example(s)
		which has the stem of a or b Example: `nsubjpass(X,{system|project})`
	`dep_name(X,{a})`	There is a *dep_name* dependency between X and any node that has a value which has the stem of a Example: `nsubjpass(X,{system})`
	`not dep_name(X, Y)`	There is no *dep_name* dependency between X and Y. Example: `not xcomp(X, Y)`

In table 5-5, we present the syntax for specifying variables and dependencies. An important feature in variable declarations is that the value of a variable is defined based on the standard form of words (stem). The advantage of this is that, we do not need to enumerate all possible forms of words that a variable may have. For instance, `X={be}` is used to eliminate the need of specifying `X={be|is|are|been}` or `X={improve}` is used instead of `X={improve|improves|improved}`.

Since goals and use case specifications are normally located in separated sections in a requirements document and the extraction is done section by section, the extractions of them are carried out separately (except that a use case description sometimes contains information about the goal it operationalizes). We thus developed separated sets of extraction rules for goals and use cases.

5.2.3.1.2 Goal Extraction Rules

As discussed in the previous section, a goal extraction rule contains a condition and a list of actions. In many cases, a rule only has one action that is called a *primary action*. The primary action must be restricted to the type that the rule belongs to. For instance, a *root* action (as used in the example rule in Table 5-2) can only be used for *navigation* goal extraction rules. A rule may have additional actions that are all called *secondary* actions. A secondary action represents a new dependency to be added to a matched dependency tree. For instance, a secondary action `nsubj(X, Y)` suggests that a new `nsubj` dependence (with the governor and dependent are the nodes that match the variables X and Y respectively) should be added into the dependency tree. A secondary

action may be in the form such as `nsubj(X, dobj(Y))`, which indicates that the dependency being added has `X` as the governor and the dependent is the governor of an existing `dobj` dependency whose dependent is `Y` (i.e., assume that such `dobj` dependency already exists). The number of secondary actions in a rule is not limited. There are four types of goal extraction rules as follows.

Ignorance Rules

Ignorance rules are used to recognize a sentence or parts of a sentence that have no important information. They thus should be ignored during the extraction process. An ignorance rule must have an *ignorance action* as its primary action. An ignorance action has the syntax of `ignore(X)` with `X` is a variable declared in the rule's variable declaration part. Assume if a dependency tree matches this rule and the node *N* in such tree matches the variable X, then the execution of this ignorance action is executed would remove the sub-tree rooted at *N* from the dependency tree. For instance, if node *"Node 7"* in Figure 5-9 matches X, then *"Node 7," "Node 10," "Node 11"* and all its descendant nodes and dependencies would be removed from the dependency tree. In case the root of the tree (i.e., *"Node 1"*) matches X, then that indicates the entire dependency tree is cleared, which means the entire sentence is ignored.

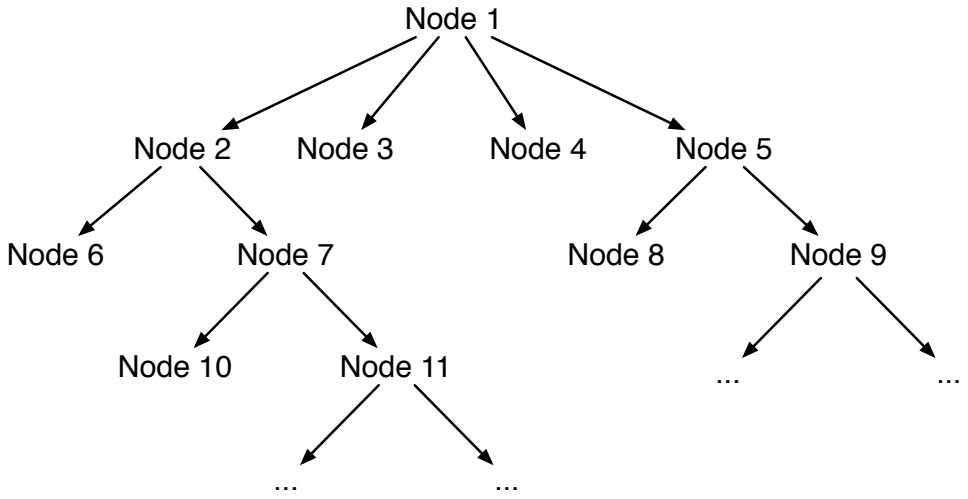


Figure 5-9. A Generic Dependency Tree

Table 5-6 presents some examples of ignorance rules. The first rule implies that the word *"specifically"* which is used as an adverbial modifier (`advmod`) of a verb in a sentence should be ignored. The second rule implies that a sentence should be

completely ignored if it is in the form such as *"database contains (or includes) something"* since such a sentence is architectural and design-oriented. The second column in the table provides examples of sentences that match the corresponding rules. The lists of dependencies of these sentences are also presented with the matched dependencies highlighted (bold).

Table 5-6. Examples of Ignorance Rules

Ignorance Rule	**Example Sentence**

`X={specifically}`
`advmod(?/{VB|VBZ|VBN}, X)`
`-> ignore(X)`
 	
Matching variables:
`X=Specifically-1`
	Specifically, the system shall allow travelers to participate in forum discussions
	`advmod(allow-6, Specifically-1)`
	`det(system-4, the-3)`
	`nsubj(allow-6, system-4)`
	`aux(allow-6, shall-5)`
	`root(ROOT-0, allow-6)`
	`nsubj(participate-9,travelers-7)`
	`aux(participate-9, to-8)`
	`xcomp(allow-6, participate-9)`
	`nn(discussions-12, forum-11)`
	`prep_in(participate-9, discussions-12)`

`X={contain|include}`
`Y={database}`
`root(X)`
`dobj(X, Y)`
`-> ignore(X);`
 	
Matching variables:
`X=database-2`
`Y=contains-3`
	The database contains two tables
	`det(database-2, The-1)`
	`nsubj(contains-3, database-2)`
	`root(ROOT-0, contains-3)`
	`num(tables-5, two-4)`
	`dobj(contains-3, tables-5)`

Navigation Rules

A navigation rule is used to move the root of a dependency tree to a certain node, which entails every node that is not part of the new root's sub-tree will be removed. A navigation rule must have a *root action* as its primary action. A root action has the syntax of `root(X)` with `X` is a variable declared in the rule's variable declaration part. Consider again Figure 5-9, assume that *"Node 5"* matches the variable `X`, then when the

root action is executed, all nodes except *"Node 5," "Node 8," "Node 9"* and its children nodes are removed. *"Node 5"* then becomes the root of the tree, which means the dependency `root(ROOT-0, Node5)` is added.

The discussed rule in Table 5-2 and Table 5-3 is an example of navigation rules. In the presented case, the execution of the rule's action (*"root(Y)"*) moves the root of the dependency tree to *"improve"* (that matches Y). As a result of this execution, *"improve"* becomes the new root of the dependency tree and the phrase *"the system will be designed to"* is removed from the sentence.

Relationship Rule

Relationship rules are concerned with extracting goals while identifying relationships between them. We support the specification of rules to identify *refinement*, *require* and *relevant* relationships. Goals are considered relevant when they are related, but no additional information to infer more detailed relationship between them. A relationship rule must have a *relationship action* as its primary action. A relationship action has the syntax of `rel_name(goal(X), goal(Y))` with `rel_name` can be `sub_goal`, `require` or `relevant`. For instance, `sub_goal(goal(X), goal(Y))` means that the goal extracted from a tree rooted at `X` is a sub-goal of the goal extracted from a tree rooted at `Y`.

Table 5-7 presents some examples of relationship rules. The first rule is used to identify sub-goal relationships if a sentence is written in the format of *"do something by doing something."* In the corresponding example sentence, this rule can help identify that *"Facilitating the communication between travelers"* is a sub-goal of *"Improve the quality of travel planning."* The second rule is used to reveal require relationships in sentences that are written in the format of *"before being able to do something, someone have/need to (or must) do something."* Based on this rule, it can identified from the corresponding example sentence that the goal *"Users are able to create reviews" requires* the goal *"Users register for memberships."*

Table 5-7. Examples of Relationship Rules

Relationship Rule	Example Sentence
X/{VB|VBD|VBG|VBN|VBP|VBZ} Y/VBG root(X) prepc_by(X, Y) -> sub_goal(goal(Y), goal(X)) **Matching Variables:** X=Improve-1 Y=facilitating-8	Improve the quality of travel planning by facilitating the communication between travelers **root(ROOT-0, Improve-1)** det(quality-3, the-2) dobj(Improve-1, quality-3) nn(planning-6, travel-5) prep_of(quality-3, planning-6) **prepc_by(Improve-1, facilitating-8)** det(communication-10, the-9) dobj(facilitating-8, communication-10) prep_between(communication-10, travelers-12)
X={have|need|must} Y={able} cop(Y, {be}) prepc_before(X, Y) xcomp(Y, ?/VB) nsubj(X, ?) root(X) not nsubj(Y, ?) -> require(goal(Y), goal(X)), nsubj(Y, nsubj(X)); **Matching Variables:** X=have-9 Y=able-3	Before being able to create reviews, users have to register for memberships **cop(able-3, being-2)** **prepc_before(have-9, able-3)** aux(create-5, to-4) **xcomp(able-3, create-5)** dobj(create-5, reviews-6) nsubj(have-9, users-8) **root(ROOT-0, have-9)** aux(register-11, to-10) xcomp(have-9, register-11) prep_for(register-11, memberships-13)

Splitting Rule

Splitting rules are used in case coordinating conjunctions (i.e., and/or) are used in a sentence. They allow a sentence to be split into two goals with a *sibling* (if *"and"* is used) or *alternative* (if *"or"* is used) relationship between them. These relationships mean that the goals should share the same parent goal via AND-refine relationships (in case of *sibling*) or OR-refine relationships (in case of *alternative*). A splitting rule must have a *splitting action* as its primary action, which has the syntax of `splitting_type`

`(goal(X), goal(Y))` with `splitting_type` can be either `split_sibling` or `split_alternative`. For instance, `split_sibling(goal(X), goal(Y))` means that the goal extracted from a tree rooted at `X` is a sibling-goal of the goal extracted from a tree rooted at `Y`.

Table 5-8 presents an example of splitting rules. Based on this rule, the sentence in the second table column can be split into two sibling goals: *"Users can search reviews by author names"* and *"Users can search reviews by destinations."*

Table 5-8. Examples of Splitting Rules

Ignorance Rule	**Example Sentence**

`X/{NN|NNS|NNP|JJ}`
`Y/{NN|NNS|NNP|JJ}`
`conj_and(X, Y)`
`-> split_sibling(X, Y)`
 	
Matching Variables:
`X=names-7`
`Y=destinations-9`
	Users can search reviews by author names and destinations
	`nsubj(search-3, Users-1)`
	`aux(search-3, can-2)`
	`root(ROOT-0, search-3)`
	`dobj(search-3, reviews-4)`
	`nn(names-7, author-6)`
	`prep_by(reviews-4, names-7)`
	`conj_and(names-7, destinations-9)`

5.2.3.1.3 Use Case Extraction Rules

In a document section that potentially contains use cases, the use case components (i.e., use case name, steps, extensions) can normally be identified using an artifact indicator list presented in Figure 5-7. For instance, the term *"pre-conditions"* indicates the precondition specifications of a use case, or the term *"basic path"* indicates the main success scenario of a use case. However, in many cases, the specification of a component may contain extra information, or different components are mixed up with each other (i.e., two use case steps may be combined together, or a use case step may be mixed up with a data constraint). Thus, we developed a set of extraction rules to identify these components. Below, we discuss different types of use case extraction rules.

Similar to goal extraction rules, a use case extraction rule may optionally contain a number of secondary actions. However, some types of rules such as extension extraction rules may have more than one primary action (which are compulsory).

Use Case Ignorance Rule

Similar to goal specifications, there are also details that should not be considered when extracting use case specifications from text. However, due to the different concerns between goals and use cases, the set of ignorance rules for use cases are different from that for goals.

Table 5-9. Examples of Use Case Ignorance Rules

Use Case Ignorance Rule	Example Sentence
X={case} Y={use} Z={end} nn(X, Y) nsubj(Z, X) root(Z) -> uc_ignore(Z); **Matching Variables:** X=case-3 Y=use-2 Z=ends-4	The use case ends here det(case-3, The-1) **nn(case-3, use-2)** **nsubj(ends-4, case-3)** **root(ROOT-0, ends-4)** advmod(ends-4, here-5)
X={case} Y={use} Z={initiate|start} nn(X, Y) mark(Z, {before}) auxpass(Z, {is|be}) nsubjpass(Z, X) advcl(?, Z) -> uc_ignore(Z); **Matching Variables:** X=case-4 Y=use-3 Z=initiated-7	Before this use case can be initiated, the traveler has successfully logged into the system **mark(initiated-7, Before-1)** det(case-4, this-2) **nn(case-4, use-3)** **nsubjpass(initiated-7, case-4)** aux(initiated-7, can-5) **auxpass(initiated-7, be-6)** **advcl(logged-13, initiated-7)** det(traveler-10, the-9) nsubj(logged-13, traveler-10) aux(logged-13, has-11) advmod(logged-13, successfully-12) root(ROOT-0, logged-13) det(system-16, the-15) prep_into(logged-13, system-16)

A use case ignorance rule only has one primary action, which has the syntax of `uc_ignore(X)`. Similar to the action in goals' ignorance rules, such an action means the dependency sub-tree rooted at `X` must be removed from the original dependency tree. If `X` is the root of the original dependency tree, the sentence is completely ignored.

Table 5-9 shows some examples for use case ignorance rules accompanied with the sentences that they can be applied in. The example sentence is followed by its textual dependency tree in the table's second column. The variable-word matching is provided under each rule to present how the rule matches the structure of its corresponding sentence. In the first example, the entire sentence is ignored according to the rule. In the second example, only part of the sentence that corresponds to the dependency sub-tree rooted at `Y` (which matches the word '*initiated'*) is ignored. This results in a new sentence of *"The traveler has successfully logged into the system."*

Step Extraction Rule

Step extraction rules are designed to extract use case steps combined in one single sentence (i.e., by *'and/or,'* or *'after/before'*). Using a step extraction rule, not only the steps are extracted, but also their relationships (i.e., *precede relationship*) are identified. In case two steps are alternatives of each other (i.e., when the *'or'* conjunction is used), a new extension is then created to establish an alternative use case path.

A step extraction rule has one primary action which as the syntax of `rel_name(statement(X),statement(Y))` with `rel_name` can be either `precede` or `alternative_path`. For instance, `precede(statement(X), statement(Y))` means that the step (which is a use case statement) extracted from a tree rooted at `X` precedes the step extracted from a tree rooted at `Y`.

Table 5-10 presents an example of step extraction rules. This rule specifies a matching condition in which a `conj_and` (*'and'* conjunction) relationship exists between two verbs while no `prep_between` relationship exists between them and the verbs are not involved in any *'if'* clause (which avoids the case that these verbs are part of a condition clause). If a dependency tree matches this rule, two consecutive use case steps can be extracted. In the corresponding example sentence, 2 steps: *"The user fills the form"* and *"The user submits the form"* are extracted in which the former precedes the later.

Table 5-10. Examples of Step Extraction Rules

Step Extraction Rule	Example Sentence
X/{VB|VBD|VBG|VBN|VBP|VBZ} Y/{VB|VBD|VBG|VBN|VBP|VBZ} conj_and(X, Y) not prep_between(?/{NN|NNS}, X) not prep_between(?/{NN|NNS}, Y) not mark(X, {if}) not mark(Y, {if}) -> precede(statement(X), statement(Y)); **Matching Variables:** X=fills-3 Y=submits-7	The user fills the form and submits the form det(user-2, The-1) nsubj(fills-3, user-2) root(ROOT-0, fills-3) det(form-5, the-4) dobj(fills-3, form-5) **conj_and(fills-3, submits-7)** det(form-9, the-8) dobj(submits-7, form-9)

Extension Extraction Rule

Although extensions in a use case should be placed in the extension (or exception) paths within the use case specification, our observation showed that simple extensions (with one or two extension steps) are frequently described within the main success scenario of the use case. An extension may be mixed up with a use case step to describe an exceptional case regarding such a step (i.e., using *"if - else"* clauses). An extension may also be specified alone as a use case step (although it is not a use case step) to describe an alternative to another step. Extension extraction rules are used to deal with such situations.

The execution of such rules is intended to identify extension condition(s), extension step(s) and separate them from the normal use case step in case the extension is mixed up with a use case step. Based on its definition, an extension extraction rule must have two primary actions: `extension_condition(statement(X))` and `extension_step(statement(Y))`. The first action indicates that the extension condition is extracted from the dependency tree rooted at `X`. The second action means the extension step is extracted from the dependency tree rooted at `Y`. A rule may contain a secondary action `use_case_step(statement(Z))` in case it is needed to identify the use case step that is mixed up with the extension. Another possible secondary action is `extension_start(W)` which is used to extract the information regarding where the extracted extension is invoked from the use case.

Table 5-11 shows some extension extraction rules accompanied with the sentences that they can be applied in. In the first example, the extension condition is the negation of the statement extracted from a dependency sub-tree rooted at `Y` (`extension_condition(neg(statement(Y)))`), which is *"the review content is **not** valid."* The extension step is extracted from a dependency sub-tree rooted at `Z`, which is *"The system displays the errors to the user."* The notion "`Y\Z`" indicates the portion of the dependency tree generated by stripping the dependency sub-tree rooted at `Z` from the dependency sub-tree rooted at `X` (`Z` is a descendant of `X`). Therefore, the extracted use case step is *"If the review content is valid, the system stores the review into the database."*

In another example, the extension condition is identified as *"The review content is invalid."* The extension step is *"The system displays the errors to the user"* and *"Step 5"* is marked as where the extension is invoked (`extension_start(Z)`).

Table 5-11. Examples of Extension Extraction Rules

Extension Extraction Rule	Example Sentence

`X/{VB|VBD|VBG|VBN|VBP|VBZ}`
`Y/{VB|VBD|VBG|VBN|VBP|VBZ|NN|NNS|JJ|JJS|JJR}`
`Z/{VB|VBD|VBG|VBN|VBP|VBZ}`
`mark(Y, {if})`
`advcl(X, Y)`
`root(X)`
`advmod(Z,{else})`
`parataxis(X, Z)`
`->`
`extension_condition(neg(statement(Y))),`
`extension_step(statement(Z)),`
`use_case_step(statement(X\Z));`
 	
Matching Variables:
`X=stores-10`
`Y=valid-6`
`Z=displays-20`
	If the review content is valid, the system stores the review into the database; else the system displays the errors to the user
	`mark(valid-6, If-1)`
	`det(content-4, the-2)`
	`nn(content-4, review-3)`
	`nsubj(valid-6, content-4)`
	`cop(valid-6, is-5)`
	`advcl(stores-10, valid-6)`
	`det(system-9, the-8)`
	`nsubj(stores-10, system-9)`
	`root(ROOT-0, stores-10)`
	`det(review-12, the-11)`
	`dobj(stores-10, review-12)`
	`det(database-15, the-14)`
	`prep_into(stores-10, database-15)`
	`advmod(displays-20, else-17)`
	`det(system-19, the-18)`
	`nsubj(displays-20, system-19)`
	`parataxis(stores-10,displays-20)`
	`det(errors-22, the-21)`

Extension Extraction Rule	Example Sentence
	dobj(displays-20, errors-22) det(user-25, the-24) prep_to(errors-22, user-25)
X/{VB|VBD|VBG|VBN|VBP|VBZ} Y/{VB|VBD|VBG|VBN|VBP|VBZ|NN|NNS|JJ|JJR|JJS} Z/CD prep_in(X, {step}) num({step}, Z) mark(Y, {if}) advcl(X, Y) root(X) -> extension_condition(statement(Y)), extension_step(statement(X\Y)), extension_start(Z); **Matching Variables:** X=displays-14 Y=invalid-10 Z=5-3	In step 5, if the review content is invalid, the system displays the errors to the user **prep_in(displays-14, step-2)** **num(step-2, 5-3)** **mark(invalid-10, if-5)** det(content-8, the-6) nn(content-8, review-7) nsubj(invalid-10, content-8) cop(invalid-10, is-9) **advcl(displays-14,invalid-10)** det(system-13, the-12) nsubj(displays-14, system-13) **root(ROOT-0, displays-14)** det(errors-16, the-15) dobj(displays-14, errors-16) det(user-19, the-18) prep_to(errors-16, user-19)

Constraint Extraction Rule

Constraint extraction rules are used to identify if a use case should instead be classified as a non-functional constraint or a data constraint of the use case it is in. This type of rules is aimed to provide an early classification of use case components in the extraction stage. Another round of classification would then be conducted in the artifact classification step (to be discussed in Chapter 6). Since in many cases the grammatical structures of non-functional and data constraints are considerably different from normal use case steps (i.e., a data constraint may describe what information to be included, or a non-functional constraint may specify the time should be taken for a system to complete a step), this type of rules could be greatly useful to identify such constraints.

A constraint extraction rule only has one primary action which has the syntax of `constraint_type(statement(X))` with `constraint_type` can be either `uc_non_functional_constraint` or `uc_data_constraint`. For instance, `non_functional_constraint(statement(X))` indicates that the statement extracted from the dependency sub-tree rooted at X is a non-functional constraint.

Table 5-12. Examples of Constraint Extraction Rules

Constraint Extraction Rule	Example Sentence
X={include|contain} Y={list|choice|option} Z/{NN|NNS|NNP} root(X) nsubj(X, Y) dobj(X, Z) conj_and(Z, ?/{NN|NNS|NNP}) -> uc_data_constraint (statement(Y)); **Matching Variables:** X=includes-5 Y=list-2 Z=hotel-6	The list of categories includes hotel, attraction and tour. det(list-2, The-1) **nsubj(includes-5, list-2)** prep_of(list-2, categories-4) **root(ROOT-0, includes-5)** **dobj(includes-5, hotel-6)** **conj_and(hotel-6, attraction-8)** conj_and(hotel-6, tour-10)
X/{VB|VBD|VBG|VBN|VBP|VBZ} Y={second|minute|hour} Z={than} W/CD root(X) mwe(Z, {less|more}) quantmod(W, Z) num(Y, W) prep_in(X, Y) -> uc_non_functional_constraint (statement(X)) **Matching Variables:** X=validate-3 Y=second-11 Z=then-9 W=1-10	System should validate the review content in less than 1 second. nsubj(validate-3,System-1) aux(validate-3, should-2) **root(ROOT-0, validate-3)** det(content-6, the-4) nn(content-6, review-5) dobj(validate-3,content-6) **mwe(than-9, less-8)** **quantmod(1-10, than-9)** **num(second-11, 1-10)** **prep_in(validate-3,second-11)**

Table 5-12 presents some examples of constraint extraction rules. The first rule indicates that a sentence written in the format of *"list (or choice or option) includes (or contains) something"* is a data constraint. The second rule indicates that if a sentence has the form of *"do something in less (or more) than a number of second (or minute, or another time unit)"* is a non-functional constraint. In GUEST, we make the assumption

that the identified constraint has a *"constrain"* relationship with the step that is immediately before it in the use case being considered.

Use Case Relationship Extraction Rule

Use case relationship extraction rules are used to identify the "include" relationships between use cases when such relationships are implicitly specified as steps in the main success scenario of a use case. Consider a use case step *"Use case 'Register for membership' is performed"* in Table 5-13. This is obviously not a normal use case step. It is instead a specification that another use case (*'Register for membership'* use case) should be invoked at this step. The rule specified in the first column can be used to reveal this relationship.

Each use case relationship extraction rule only has one primary action which has the syntax of `uc_include(X)` with `X` is the root of the dependency sub-tree where the included use case's name can be extracted.

Table 5-13. Examples of Constraint Extraction Rules

Constraint Extraction Rule	Example Sentence

`X={case}`
`Y={use}`
`Z={perform}`
`W/{VB|VBD|VBG|VBN|VBP|VBZ|NN|NNP|NNS}`
`nn(X, Y)`
`nsubjpass(Z, X)`
`ccomp(Z, W)`
`auxpass(Z, {is|be})`
`root(Z)`
`-> uc_include(W);`
Matching Variables:
`X=case-2,Y=Use-1,Z=performed-9`
	Use case 'Register for membership' is performed
	`nn(case-2, Use-1)`
	`nsubjpass(performed-9, case-2)`
	`ccomp(performed-9, Register-4)`
	`prep_for(Register-4, membership-6)`
	`auxpass(performed-9, is-8)`
	`root(ROOT-0, performed-9)`

Repeating Step Extraction Rule

Repeating step extraction rules are used to verify whether a use case step belong to the *"repeating step"* sub-class. A rule of this type must have three primary actions. First, the action `repeating_step_condition(statement(X))` indicates that the

condition of that repeating step is extracted from the dependency sub-tree rooted at X. Secondly, the action `repeating_step_start(Y)` indicates the ID number of the first repeated step (i.e., 3). Similarly, the action `repeating_step_end(T)` indicates the ID number of the last repeated step.

Table 5-14 presents an example of repeating step extraction rules. According to the matching details showed in the table, the following information is extracted from the example sentence (in the second column):

- Repeating step condition: *"A sufficient number of members is added to the group"*
- Starting step: *"Step 3"*
- Ending step: *"Step 5"*

Table 5-14. Examples of Repeating Step Extraction Rules

Repeating Step Extraction Rule	**Example Sentence**
X={repeat} Y={step} Z={step} W/CD T/CD U/{VB|VBD|VBG|VBN|VBP|VBZ|NN|NNS|JJ|JJR|JJS} root(X) num(Y, W) prep_to(X, Z) num(Z, T) mark(U, {until}) advcl(X, U) -> repeating_step_condition(statement(U)), repeating_step_start(W), repeating_step_end(T) **Matching Variables:** X=repeats-3 Y=step-4 Z=step-7 W=3-5 T=5-8 U=added	The user repeats step 3 to step 5 until a sufficient number of members is added to the group det(user-2, The-1) nsubj(repeats-3, user-2) **root(ROOT-0, repeats-3)** dobj(repeats-3, step-4) **num(step-4, 3-5)** **prep_to(repeats-3, step-7)** **num(step-7, 5-8)** **mark(added-16, until-9)** det(number-12, a-10) amod(number-12, sufficient-11) nsubjpass(added-16, number-12) prep_of(number-12, members-14) auxpass(added-16, is-15) **advcl(repeats-3, added-16)** det(group-19, the-18) prep_to(added-16, group-19)

5.2.3.2 Extraction Rule Creation Process

GUEST currently provides a set of over 250 extraction rules. However, since requirements documents can be written in various styles, there may be cases in which such extraction rules are not adequate to handle certain sentences. For instance, a certain unimportant part of a sentence is not identified and thus extracted as part of an artifact specification. A relationship between goals embedded in a sentence may not be recognized. To deal with this problem, GUEST allows users to extend the existing set of rules by developing their own extraction rules using a rule editor. Currently in GUEST, extraction rules are manually developed based on the analysis of how some certain phrases are ignored, or how artifacts and relationships are extracted from certain sentences. The discussions regarding the steps taken to write new extraction rules and the quality analysis of such rules are provided in the Appendix A4.

5.2.3.3 Extraction Rule Usage

In this section, we describe how extraction rules are used in our work to support the identification of artifacts and relationships from text. Since use cases are specified in their own sections in a requirements document, there is no case in which goal and use case extraction rules need to be executed in the same section, except when the goal of a use case is specified within that use case specification. However, even in such cases, goal and use case extraction rules are also used independently. In the following, we discuss the extraction processes for goals and use cases in which goal and use case extraction rules are used respectively. The details regarding the algorithms behind these extraction processes are provided in the Appendix A5.

Goal Extraction

A certain sentence may match different rules and the execution of each of these rules may produce different results. To solve these problems, rules of different types are given different priorities. Specifically, the goal extraction rules are prioritized in the following order: ignorance rules (having the highest priority), navigation rules, relationship rules, and splitting rules. In addition, GUEST supports the quality analysis of extraction rules to ensure rules' correctness, redundancy and consistency. Furthermore, in case a sentence is found matching multiple rules, the alternative extraction results will be recorded and presented to users at the end of the extraction

process for their actions. Such actions may include selecting the preferred extraction outputs, or repairing the rules in case one or more rules are found incorrectly specified.

To extract goals from sentences in a section, we use an iterative process to analyze each sentence in consideration of the rule priorities. In each iteration, GUEST searches for the rule with highest priority that matches the dependency parse tree of such a sentence. If there were such a rule, it would be executed to generate an outcome that contains one or more new dependency trees (which are sub-trees of the original dependency tree) and possibly a relationship between these trees (if a relationship or splitting rule is used). The resulting dependency trees are then considered in the next iteration and so on. The process ends when no matching rule is found. The final resulting dependency trees are used to produce textual goal specifications.

In the following, we use an example to demonstrate the process of extracting goals from the sentence "*The system will be designed to improve the quality of travel planning by facilitating the communication between travelers"*, which we refer to as the *original sentence.* Table 5-15 shows the two extraction rules used in this example. R1 is a navigation rule while R2 is a relationship rule.

Table 5-15. Extraction Rules Used in Goal Extraction Example

Rule R1	X={design|build|aim|intend|target} Y/VB root(X) nsubjpass(X, {system|project|software|application}) auxpass(X, {be}) aux(Y, {to}) xcomp(X, Y) -> root(Y)
Rule R2	X/VB Y/VBG root(X) prepc_by(X, Y) -> sub_goal(goal(Y), goal(X))

Table 5-16 illustrate the iterative process of extracting goals and their relationships from the original sentence. Rule R1 is applied in the first iteration since it takes precedence

over R2 that then is applied in the second iteration. The new dependency trees Tree 1 and Tree 2 are generated from the extraction, in which Tree 2 *refines* Tree 1.

Table 5-16. Extraction Process of the Original Sentence

<table>
<tr><td>Original Dependency Tree</td><td colspan="2">det(system-2, The-1)
aux(designed-5, will-3)
root(ROOT-0,designed-5)
aux(improve-7, to-6)
det(quality-9, the-8)
det(communication-16, the-15)
dobj(facilitating-14, communication-16)
prep_between(communication-16, travelers-18)
prepc_by(improve-7, facilitating-14)
nsubjpass(designed-5, system-2)
auxpass(designed-5, be-4)
xcomp(designed-5,improve-7)
dobj(improve-7, quality-9)
nn(planning-12, travel-11)
prep_of(quality-9, planning-12)</td></tr>
<tr><td rowspan="2">Iteration 1</td><td colspan="2">Rule R1 matched and applied, resulting in the new dependency tree:</td></tr>
<tr><td colspan="2">root(ROOT-0,improve-7)
aux(improve-7, to-6)
det(quality-9, the-8)
det(communication-16, the-15)
dobj(facilitating-14, communication-16)
prepc_by(improve-7, facilitating-14)
dobj(improve-7, quality-9)
nn(planning-12, travel-11)
prep_of(quality-9, planning-12)
prep_between(communication-16, travelers-18)</td></tr>
<tr><td rowspan="2">Iteration 2</td><td colspan="2">Rule R2 matched and applied, resulting in two new dependency trees: Tree 1 and Tree 2 in which Tree 2 refines Tree 1</td></tr>
<tr><td>Tree 1</td><td>Tree 2</td></tr>
<tr><td></td><td>root(ROOT-0,improve-7)
aux(improve-7, to-6)
det(quality-9, the-8)
dobj(improve-7, quality-9)
nn(planning-12, travel-11)
prep_of(quality-9, planning-12)</td><td>root(ROOT-0, facilitating-14)
det(communication-16, the-15)
dobj(facilitating-14, communication-16)
prep_between(communication-16,travelers-18)</td></tr>
</table>

Use Case Extraction

Since use case specifications are normally structured with component indicators (i.e., use case name, pre-condition), there are only a few cases in which multiple types of rules may be applicable in the same piece of text. For instance, in extracting the pre-conditions and post-conditions from a use case specification, only ignorance rules (under the category of use case extraction rules) are applicable (i.e., to identify parts of a condition description that should be ignored). In addition, no extraction rule is needed for mining the use case's name (since it is normally described by only a few words).

Table 5-17. Summary of Use Case Component - Rule Category Mapping

Use Case Component	Applicable Rule Categories
Use Case Name, Actor	None
Pre-condition & Post-condition	Ignorance Rules
Use Case Goal	Goal extraction rules
Use Case Step	All use case extraction rules
Non-functional constraint & data constraint (This is for cases when these constraints are explicitly specified using some indicators in use case specification. Normally they are embedded in use case steps)	Ignorance Rules, Constraint Extraction Rules
Use Case Extension	All use case extraction rules

Use case step specifications are where all extraction rules are to be applied since it is where other types of components (i.e., extension, constraints) are embedded. Use case goal is a special component in a use case specification. It requires goal extraction rules to be used to extract the goal(s) of the use case from the text. In case there is a single goal, or multiple goals with sibling or alternative relationships (between them) are found from the text, an operationalize relationship will then be established between each of them and the use case being considered. If there are multiple goals with other relationships (i.e., refine), or no goal (i.e., the entire goal description is ignored according to an ignorance rule) is extracted, then the problem is recorded and reported to users. Table 5-17 summarizes the applicable categories of use case extraction rules for each use case component.

Based on this mapping, the use case extraction process is done by inspecting each specified component in a use case (by looking for the use case component indicators specified in the artifact indicator list at the beginning of the entire extraction process). Based on each type of component, a suitable set of rules is then invoked to analyze each sentence in the specification of such component. The process of searching for matching rules and executing these rules is similar to that of goal extraction.

5.3 Chapter Summary

In this chapter, we provided a detailed discussion about our rule-based approach to extracting goal and use case integrated models from natural language-based software requirements specification documents (SRS). We started the chapter with some background knowledge on natural language processing that is relevant to our techniques. We then discussed the formatting requirements of SRSs to be used with our extraction techniques. We described the pre-processing and linguistic analysis process that make the text ready for the extraction of artifacts and relationships. The use of extraction rules to identify artifacts and relationships from text were discussed in detailed. In Chapter 6, we continue our discussion about extraction techniques with the focus on artifact polishing and classification, and model construction. Chapter 6 also provides details about our GUEST tool, some usage examples, and the evaluation results.

Chapter 6

Artifact Specification Polishing and Classification

In this chapter, we describe the polishing and classification of artifact specifications used in our rule-based approach to goal and use case integrated model extraction. In section 6.1, we discuss our artifact specification polishing techniques. Section 6.2 provides details about the Mallet text classifier and how it was extended and used to support the classification of the extracted artifact specifications. Section 6.3 describes how a goal and use case model is constructed from information collected from the previous steps. Section 6.4 provides a discussion about GUEST, a tool developed to implement our entire extraction approach, accompanied with some usage examples. We conclude this chapter with details about the validation we have conducted to evaluate the performance of GUEST in extracting goal and use case models from text.

6.1 Artifact Specification Polishing

We present our technique to polish the artifact specifications resulted from the *"artifact and relationship identification"* step (discussed in Chapter 5). The objective of artifact specification polishing is to ensure the extracted specifications to conform to our defined specification boilerplates discussed in Chapter 4.

6.1.1 Overview

The outcome of the artifact and relationship identification step is a set of dependency trees and their relationships. Each dependency tree is a sub-tree of the dependency tree (called *original dependency tree*) of the original sentence. Since the dependencies contain the words and indexes (positions) of these words in the original sentence, they can be used to form a textual specification for an extracted artifact. Table 6-1 presents the extracted dependency trees obtained from the original sentence (*"The system will be designed to improve the quality of travel planning by facilitating the communication between travelers"*) and the textual artifact specifications created from them.

Table 6-1. Example of Generating Textual Specification from Dependencies

Dependency Collection	Textual Specification

`root(ROOT-0, improve-7)`
`aux(improve-7, to-6)`
`det(quality-9, the-8)`
`prep_of(quality-9, planning-12)`
`nn(planning-12, travel-11)`
`dobj(improve-7, quality-9)`
	To improve the quality of travel planning

`root(ROOT-0, facilitating-7)`
`det(communication-16, the-15)`
`dobj(facilitating-14, communication-16)`
`prep_between(communication-16, travelers-18)`
	Facilitating the communication between travelers

However, since these extracted dependency trees are only fragments of the original dependency tree, in many cases they do not readily represent a grammatically correct, complete and stand-alone specification. For example, the first specification in Table 6-1 has an unneeded word *'to'* at the beginning while the second specification has the main verb in the present participle form (*'facilitating'*) while it should be in its standard form (*'facilitate'*).

In addition, in order to guarantee artifacts are consistently specified in a goal and use case integrated model, the extracted artifact specifications need also to conform to our defined specification boilerplates presented in Chapter 4. Although specification boilerplates are defined specifically for each artifact type, in this step we aim to ensure

the extracted specifications to satisfy the general properties of these boilerplates. The reasons are twofold. Firstly, the categories of the extracted artifacts (except for use case components) have not been determined at this stage (they will be classified in the next step – artifact classification). Therefore, it is not possible to verify such artifacts against any categories. Secondly, the verification as to whether an artifact specification meets the boilerplates for its type is not necessarily in the scope of artifact extraction. It instead belongs to the analysis phase that is to be discussed in Chapter 7.

There are three general properties of our specification boilerplates that are described as follows. These properties come from the general requirement specification guidelines that we adopted [152, 172]. The artifact specification polishing process focuses on ensuring these properties in the extracted specifications.

- **P1:** Any specification, except for use case conditions, must be in active voice.
- **P2:** The use of adjective-preposition phrases instead of verb phrases must be avoided. For instance, a specification such as *"editors **are capable of** entering new reviewers"* should not be used. It should be instead written as *"editors shall be able to enter new reviewers."*
- **P3:** Each goal specification, which has a subject, must be written in the form of *"subject + shall/may be + adjectival phrase/noun phrase"* or *"subject shall/may be able to + verb phrase."* If the specification does not have a subject, it must start with a verb that is in its standard form. Use case component specifications do not have this property.

In order to address these needs, we developed a four-step specification polishing process. In each step, a dependency tree is input and a new dependency tree is produced as a result. The first step is concerned with ensuring an extracted specification to be correct grammatically (i.e., verb is in the right form). The second step ensures the specification (if it is not a use case condition) is in active voice. A passive voice-to-active voice transformation is needed if the specification is found to be in passive voice. It thus guarantees that the property P1 is satisfied. The third step is used to rewrite the specification if an adjective-preposition phrase is used instead of a verb phrase (property P2). In this step, GUEST also provides a feature to help automatically rewrite the specification in a way preferred by users (but still conforms to the boilerplates). For instance, the specification *"users without technical background can easily use the*

system" can be rewritten as *"users who do not have technical background can easily use the system."* In the last step, if the specification is a goal, it is ensured to have the appropriate modal auxiliary verb as discussed in property P3.

In the following sections, we discuss the techniques used in each of these steps.

6.1.2 Grammatical Error Correction

Grammatical errors are usually caused when a sentence has part of it trimmed (i.e., by ignorance rules) while the remaining part of such a sentence, which is usually a phrase, is not readily a stand-alone sentence. Similarly, errors may be produced in the case where a sentence is split into multiple specifications (by goal relationship or splitting rules, use case step extraction rules or extension extraction rules, etc.). These types of grammatical errors commonly fall into the following cases:

Incorrect tense: this type of errors refers to the cases in which the main verb of a specification is not specified in its suitable form. The specification *"Facilitating the communication between travelers"* presented in Table 6-1 is an example of this type of errors. To fix this problem, we developed an algorithm to verify the root verb (call it `x`) of a specification and make modifications according to the following rules:

- If there does not exist a `nsubj` or `nsubjpass` dependency that has `x` as its governor (which means there is no subject or passive subject of the verb), then the verb must be in its standard form.
- If there is a `nsubjpass` dependency that has `x` as its governor and if the specification is a use case condition, then the verb must be in the past participle form. However, if the specification is not a use case condition, no change needs to be made on the verb since it will be updated in the next step (when the entire specification is transformed to active voice).
- If there is a `nsubj` dependency that has `x` as its governor and if the specification is a goal, then no change is needed since the verb value will be updated in the last step of this polishing process (adding modal auxiliary verbs). If the specification is a use case pre-condition, then the verb is ensured to be in the simple past or simple present tense depending on the currently used tense. For instance, if the verb's POS tag is VBN (past participle), it will remain

unchanged. If the past perfect tense or past continuous tense is used (detected by checking the existence and POS tag of its auxiliary verb), the auxiliary verb will be removed and the root verb is changed to its past participle form. In other cases or if the specification belongs to another type of use case component, the verb is ensured to be in the simple present tense. In this case, the value of the verb depends on whether the subject is in singular or plural form. This can be detected by checking the POS tag of the subject (NNS and NNPS indicate plural and NN and NNP indicate singular). The SimpleNLG library[8] is used to get the value of a verb based on its forms.

Unneeded words: according to our observation, the specification (*"To improve the quality of travel planning"*) presented in Table 6-1 represents the only case in which an unneeded word (i.e., *'to'*) is left in an extracted specification. To fix this problem, we verify if an `aux(X, {to})` dependency exists in the extracted dependency tree (`X` is the root verb) and *'to'* does not appear in any other dependencies in that tree. If that is the case, then that `aux` dependency is then removed.

6.1.3 Passive Voice to Active Voice Transformation

A passive voice specification can be recognized by verifying that its associated dependency tree contains the following dependencies:

- **nsubjpass and auxpass:** these dependencies always appear in a dependency tree of a specification. `nsubjpass` dependency is used to represent the connection between the subject and verb of a passive clause. Semantically, the subject is the object of the action specified by the verb. `auxpass` dependency is used to represent the connection between this verb and a passive auxiliary verb in the same passive clause. This auxiliary verb is normally a *'to-be'* verb. Table 6-2 shows a dependency tree of a passive specification. The discussed nsubjpass and auxpass dependencies are highlighted.
- **agent:** An agent is the complement of a passive verb that is introduced by the preposition *"by"* and does the action specified by the verb. Therefore, it is the semantic subject of the action. Since a passive specification may not contain a

[8] https://code.google.com/p/simplenlg/

"by" clause, a dependency agent may not appear in a passive dependency tree. Table 6-2 also shows an example of an agent dependency.

Table 6-2. Example for Passive Dependencies

Sale reports are generated monthly by the system	nn(reports-2, Sale-1) root(ROOT-0, generated-4) advmod(generated-4, monthly-5) det(system-8, the-7) **nsubjpass(generated-4, reports-2)** **auxpass(generated-4, are-3)** **agent(generated-4, system-8)**

Since these dependencies can be considered as the indicators of a passive specification, we term them as *"passive dependencies."* In order to transform a passive specification into active voice, we employ an algorithm that removes or replaces the passive dependencies with their corresponding active dependencies. Specifically, a `nsubjpass` dependency should be replaced with a `dobj` dependency while the related auxpass dependency is removed. An `agent` dependency should be replaced with a `nsubj` dependency. In addition, the algorithm updates the positions of these dependencies' *governors* and *dependents* to reflect the transformation. For instance, the object (*"sale reports"*) of the verb (*"generated"*) is moved from the position behind the verb to the front of it. Furthermore, since the verb is no longer in a passive clause, it should be updated to the standard form (i.e., "*generate"*). Table 6-3 illustrates the result of these modifications. The crossed out dependencies are those removed and the highlighted (bolded) dependencies are those added.

Table 6-3. Result of the Transformation

The system generates sale reports monthly	det(system-2, The-1) root(ROOT-0, generates-3) nn(reports-5, sale-4) advmod(generates-3, monthly-6) **nsubj(generates-3, system-2)** **dobj(generates-3, reports-5)** ~~nsubjpass(generated-4, reports-2)~~ ~~auxpass(generated-4, are-3)~~ ~~agent(generated-4, system-8)~~

6.1.4 Artifact Specification Structure Review

In this step, the dependency tree is reviewed to check if it describes an artifact specification in an undesirable way. Other than ensuring the specification is not written using adjective-preposition phrase, GUEST enables users to define rules for modifying the structure of the specification according to their specific formatting requirements (given that these requirements conform to the defined specification boilerplates).

The process of artifact specification structure review is done based on a set of *modifying rules*. The syntax of modifying rules is adopted from that of extraction rules. Each modifying rule also contains a list of matching conditions specified by variables and dependencies and a list of modifying actions. Since modifying rules are intended to manipulate and update existing dependency trees, an action used in a rule can also be used in another. That makes the syntaxes of different types of rules may not be clearly distinguished. Therefore, different from extraction rules, modifying rules are not clearly categorized. The behavior of each rule entirely depends on the actions that it has. These actions (termed as *"dependency manipulating actions"*) are discussed in the following sections.

Dependency Adding Action

Dependency adding actions are used to add a new dependency into the dependency tree being considered. It has the syntax of `dep_name(governor, dependent)` in which `dep_name` is the name of the dependency being added, `governor` and `dependent` refer to the variable name of a node that is intended to be the governor and dependent of this dependency respectively. An example of this type of rule is `dobj(W, Y)`. In this example, a `dobj` dependency is being created between the nodes `W` and `Y`. Dependency adding actions are frequently used in extraction rules as secondary actions.

Dependency Removal Action

A Dependency removal action is used to remove a certain dependency from a dependency tree. It has the syntax of `remove(dep_name(governor, dependent))` in which `dep_name(governor, dependent)` is the dependency to be removed. For example, the action `remove(dobj(W, Y))` means that if the `dobj` dependency exists between the nodes `W` and `Y`, it would be removed from the dependency tree.

Dependency Collection Removal Action

Dependency collection removal actions are used to remove multiple dependencies with a single action. This type of actions takes two or more variables. The first one is called *origin* that is the node where the removal begins. Other variables are called *terminal points*. A terminal point denotes the node where the removal ends. A terminal point must be a descendant node of the origin. When executing a dependency collection removal action, all descendant nodes of the origin and their dependencies are removed except for the dependency sub-trees rooted at the terminal points. Dependency collection removal actions have the syntax of `remove_col(origin, terminal_point,…, terminal_point)`.

Figure 6-1 illustrates an example of how this type of action works. Assume the variable `X` refers to the node `capable-4`, `Y` refers to `users-2`, and `Z` refers to `creating-5`. Then the action `remove_col(X, Y, Z)` would remove the following dependencies: `nsubj(capable-4,users-2)`, `cop(capable-4, are-3)` and `prepc_of(capable-4,creating-6)`.

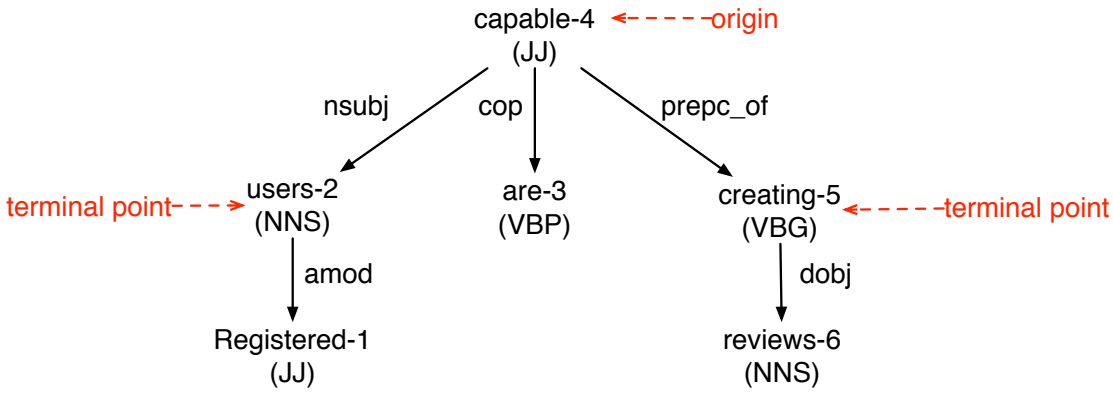


Figure 6-1. Example of Dependency Collection Removal Action

Node Adding Action

A *"node adding"* action is used to create a new node in a dependency tree (a node in a dependency tree corresponds to a word in its textual specification). Since the action's objective is only creating a new node, it must be used in conjunction with other actions that make use of the newly created node. Therefore, this type of actions is considered as a *"helper"* action or *"dependent"* action. Table 6-4 presents the syntax of this type of actions and an example showing how it is used.

Table 6-4. Syntax and Example of Node Adding Actions

Syntax	`new(name, {value}, POS_tag, node_before, node_after)` **Variables:** **`name:`** the name given to the node. It is also called as *"variable name"* It is used in other actions to refer to this node. i.e., `X, Y` **`Value:`** the standard value of the node. i.e., `{who}`, `{create}`, `{review}` **`POS_tag:`** the POS tag assigned to the node, it is used later to determine the actual value of the node. For instance, if `POS_tag` is `VBG,` then the node will have the value of `{creating}` if its standard value is `{create}` **`node_before:`** the node that should be placed before the node being created when the dependency tree is transformed to a textual specification. It is 'null' if the new node should be the first word in the specification. **`node_after:`** the node that should be placed before the node being created when the dependency tree is transformed to a textual specification. It is 'null' if the new node should be the last word in the specification.
Example	`new(Z, {who}, WP, X, Y)` In this example, a new node named `Z` with value of "who" and POS tag of WP is created. In the corresponding textual specification of the dependency tree, the word "who" (the node's value) should be placed after the word associated to node `X` and before the word associated to node `Y`.

Verb Adding Action

Verb adding actions are a subset of node adding actions. They are used to create a verb that can then be used to create a verb phrase in a text specification. They can also be used to modify an existing verb (i.e., change POS tag, add negation). The verb being created or modified can optionally (using a variable) be marked as the root of the dependency tree being considered. Verb adding actions are primarily used for replacing an adjective-preposition phrase with a verb phrase and adding a relative clause into a noun phrase. Table 6-5 presents the syntax of this type of actions and an example showing how it is used.

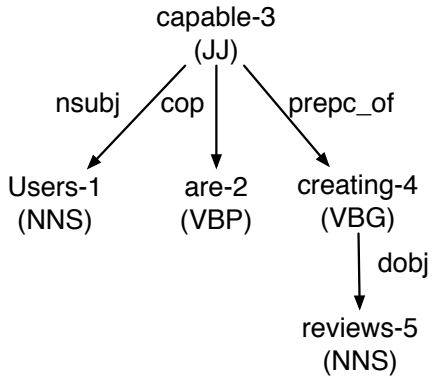
Table 6-5. Syntax and Example of Verb Adding Actions

Syntax	`new_verb(name, {value}, POS_tag_ref, POS_tag_subject, node_before, node_after, is_negated, is_root)` **Variables:** **`name:`** the name given to the node. It is also called as *"variable name"* It is used in other actions to refer to this node. i.e., `X, Y` **`value:`** the standard value of the node. i.e., `{write}, {create}.` If the node already exists, then the value of the variable is 'null.' **`POS_tag_ref:`** the node that is referenced to determine the POS tag of the node being added as the root. For instance, if the referenced node has the POS tag of VBN, then the new node will has a VBN POS tag. **`POS_tag_subject:`** in case no node is available to reference for a POS tag, then a POS tag subject is needed. This variable refers to the node that is the subject of the verb to be added. Based on the POS tag of the subject, the verb's POS tag can be determined (i.e., if the subject has the NNS POS tag, then the verb has the VBP POS tag). If both POS_tag_ref and POS_tag_subject variables are 'null,' then it is assumed the verb has a VB POS tag **`node_before:`** the node that should be placed before the node being created when the dependency tree is transformed to a textual specification. This variable has a 'null' value if the new node should be the first word in the specification, or it is an existing node. **`node_after:`** the node that should be placed before the node being created when the dependency tree is transformed to a textual specification. This variable has a 'null' value if the new node should be the last word in the specification, or it is an existing node. **`is_negated:`** a Boolean value indicating whether the phrase being considered should be negated. If so, a `neg` dependency should be added into the dependency tree in a suitable way. **`is_root:`** a Boolean value indicating whether the verb should be marked as the root of the dependency tree. If so, the existing root dependency must have its dependent node changed to the verb being added/modified.
Example	`new_verb(Z, {create}, W, null, X, Y, false, true)` In this example, a new node named `Z` that has the standard value of "create" and the same POS tag as that of W. In the corresponding textual specification of the dependency tree, the word "create" (the node's value) should be placed after the word associated to node `X` and before the word associated to node `Y`. There is no negation needs to be added. The verb is marked as the root of the universal dependency tree.

Below we provide some examples to illustrate how the discussed actions are used.

Example 6-1: Consider the specification *"users are capable of creating reviews."* The goal is to rewrite it as *"users create reviews"* (which will then be updated with modal verbs in the next step and become *"users shall be able to create reviews"*). The dependency tree of the specification and the modifying rule used to modify it are presented in Table 6-6.

Table 6-6. Dependency Tree and Modifying Rule for Example 6-1




```
X={capable}
Y/{NN|NNS|NNP|NNPS}
Z/VBG
T={be}
nsubj(X, Y)
cop(X, T)
root(X)
prepc_of(X, Z)
-> new_verb (Z, null, null, Y,
null, null, false, true),
remove_col(X, Y, Z),
nsubj(Z, Y);
```

There are three actions in this modifying rule. The first is a verb adding action. Since Z is an existing node (it is *'creating'*), the value, `node_before` and `node_after` parameters (the second, fifth and sixth parameters) have no value. Z is set to have a POS tag that suits to have Y as its subject (the forth parameter). The current POS tag of Y is NNS (plural noun). Therefore, Z should have the POS tag of VBP and its value should be changed to *'create.'* In addition, no negation needs to be added (the seventh variable). Lastly, Z is marked as the root of the dependency tree. That means the root dependency will have its dependent node changed to Z (which becomes `root(ROOT-0, create-4)`).

The second action is a dependency collection removal action. It is intended to remove a set of dependencies with the origin is X and the terminal points are Y and Z. As a result of this action execution, the dependencies `nsubj(capable-3,users-1)`, `cop(capable-3, are-2)` and `prepc_of(capable-3,creating-5)` are all removed.

In the last step, the `nsubj(create-4, users-1)` is created and added into the dependency tree. Table 6-7 summaries the outcome through each of these steps. In each step, the newly added or modified parts of the dependency trees are highlighted.

Table 6-7. Summary of Action Execution in Example 6-1

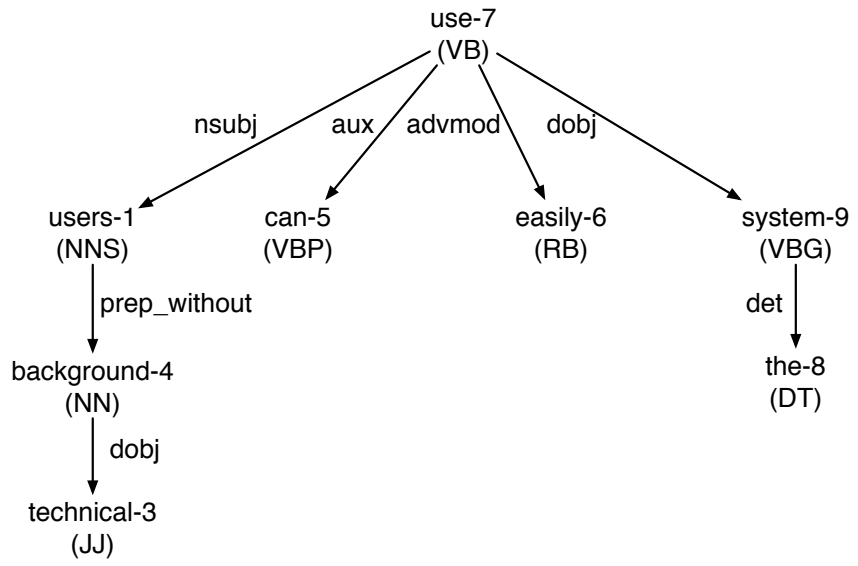
Action	Resulting Dependency Tree

`new_verb (Z, null, T, null, null, null, false, true)`	`nsubj(capable-3, users-1)`
	`cop(capable-3, are-2)`
	`root(ROOT-0, **create-5**)`
	`prepc_of(capable-3, **create-5**)`
	`dobj(**create-5**, reviews-6)`

`remove_col(X, Y, Z)`	`root(ROOT-0, create-5)` `dobj(create-5, reviews-6)`

`nsubj(Z, Y)`	`root(ROOT-0, create-5)`
	`dobj(create-5, reviews-6)`
	`**nsubj(create-5, users-1)**`

Table 6-8. Dependency and Modifying Rule for Example 6-2



`X=users-1, Y=background-4`

```
X/{NN|NNP|NNS|NNPS}
Y/{NN|NNP|NNS|NNPS}
prep_without(X, Y)
-> new(Z, {who}, WP, X, Y), new_verb(W, {have}, null, X, Z, Y, true, false), remove(prep_without(X, Y)), nsubj(W, Z), rcmod(X, W), dobj(W, Y);
```

Example 6-2: Consider the specification *"users without technical background can easily use the system."* The goal is to rewrite it as *"users who do not have technical background can easily use the system."* The dependency tree of the specification and the modifying rule used to modify it are presented in Table 6-8.

There are six actions in this modifying rule. The first is a new node action that is intended to create a new node Z with the value of *'who,'* POS tag of WP and is placed after X and before Y.

The second action is a verbal root adding action that creates a new verb W that has the standard value of *'have.'* W has X as the POS tag subject. Since X has the NNS POS tag, W has the POS tag of VBP and thus its value remains *'have.'* The fifth and sixth variables indicate the W should be placed before Z and after Y when the dependency tree is transformed into a textual specification. The seventh variable indicates that a negation should be added with the verb. Therefore, the dependencies `aux(W, {do})` and `neg(W, {not})` are added into the dependency tree. The addition of negation dependencies depends on both the POS tag of the verb (i.e., to determine it should be *"do not"* or *"does not"*) and whether modal verbs are in use. For instance, in cases when *"shall"* is used, then only the `neg(W, {not})` dependency is added.

The rest of the actions are concerned with removing and adding dependencies. Table 6-9 summaries the state of the dependency tree after each action is executed.

Table 6-9. Summary of Action Execution in Example 6-2

Action	Resulting Dependency Tree

`new(Z, {who}, WP, X, Y)`	`nsubj(use-7, users-1)`
	`amod(background-4, technical-3)`
	`prep_without(users-1, background-4)`
	`aux(use-7, can-5)`
	`advmod(use-7, easily-6)`
	`root(ROOT-0, use-7)`
	`det(system-9, the-8)`
	`dobj(use-7, system-9)`
	`Z={who}/WP created`

`new_verb(W, {have}, null, X, Z, Y, true, false)`	`nsubj(use-7, users-1)`
	`amod(background-4, technical-3)`
	`prep_without(users-1,`

Action	Resulting Dependency Tree
	background-4) aux(use-7, can-5) advmod(use-7, easily-6) root(ROOT-0, use-7) det(system-9, the-8) dobj(use-7, system-9) **W={have} created** **aux(W, {do})** **neg(W, {not})**
remove(prep_without(X, Y))	nsubj(use-7, users-1) amod(background-4, technical-3) aux(use-7, can-5) advmod(use-7, easily-6) root(ROOT-0, use-7) det(system-9, the-8) dobj(use-7, system-9) aux(W, {do}) neg(W, {not})
nsubj(W, Z)	nsubj(use-7, users-1) amod(background-4, technical-3) aux(use-7, can-5) advmod(use-7, easily-6) root(ROOT-0, use-7) det(system-9, the-8) dobj(use-7, system-9) aux(W, {do}) neg(W, {not}) **nsubj(W, Z)**
rcmod(X, W)	nsubj(use-7, users-1) amod(background-4, technical-3) aux(use-7, can-5) advmod(use-7, easily-6) root(ROOT-0, use-7) det(system-9, the-8) dobj(use-7, system-9) aux(W, {do}) neg(W, {not}) nsubj(W, Z) **rcmod(X, W)**
dobj(W, Y)	nsubj(use-7, users-1) amod(background-4, technical-3) aux(use-7, can-5) advmod(use-7, easily-6)

Action	Resulting Dependency Tree
	root(ROOT-0, use-7) det(system-9, the-8) dobj(use-7, system-9) aux(W, {do}) neg(W, {not}) nsubj(W, Z) rcmod(X, W) **dobj(W, Y)**

Modifying Rule Creation and Quality Analysis

Similar to extraction rules, the creation of modifying rules is currently done manually with the support of the rule editor in GUEST. Modifying rules are normally created by analyzing a sentence and determine the changes needed to modify such a sentence so that it is written in a desired way. A detailed discussion on the creation and quality analysis of modifying rules is provided in Appendix A6.

6.1.5 Modal Auxiliary Verb Addition

This step is only required for goals since use case components do not need to be specified with modal auxiliary verbs according to the specification boilerplates. The input into this step is a dependency tree that has been processed through the previous steps. Therefore, it is ensured that the specification corresponding to this dependency tree is grammatically correct (except that the root verb may not be in a correct form since it was intentionally left to be considered in this step), in active voice and specified properly using a verb phrase. The following rules are used to consider adding modal auxiliary verbs into this dependency tree.

R1: If the root verb already has the required modal auxiliary verbs, then nothing should be updated.

R2: If a modal verb has been used, however it is not the required one, then such a modal verb is replaced with the appropriate one. Table 6-10 presents how the replacement is done.

R3: If the root verb is *"to-be"* (i.e., *"is," "are"*), then *"shall"* should be added before the verb (and after the subject if any) while the *"to-be"* verb is transformed to its standard form (*"be"*).

R4: If the root verb is a normal verb, then *"shall be able to"* is added before the verb (and after the subject if any) while the verb is transformed to its standard form.

R5: If the specification contains a negation (i.e., *"users can**not** create reviews"*), then the above modification must be adjusted accordingly. For instance *"shall be able to"* becomes *"shall not be able to."* Negations can be detected by verifying if a `neg` dependency (i.e., `neg(create, not)`) exists in the dependency tree.

Table 6-10. Modal Verb Replacement

Original Text	Replacement Text
Must be/will be/should be	Shall be
Must/will/shall/should + Verb	Shall be able to + Verb
May + Verb	May be able to + Verb

All these modifications are done on the dependency tree (which then entails the corresponding textual specification to be updated). For instance, to add the phrase *"shall be able to*" to the textual specification, the following dependencies are added into the dependency tree: `aux({able}, {shall})`, `cop({able}, {be})`, `root({able})`, `aux(X, {to})`, `xcomp({able}, X)` and `nsubj({able}, Y)` with `X` is the root verb and `Y` is the subject of this verb (if it exists). Note that *"able"* is the root of the new dependency tree, therefore a number of relevant existing dependencies (i.e., `root(X))` must be removed. In addition, the ordering among the added words (i.e., *"able"* is after *"be"*) and between them with the existing words (*"to"* is before `X`) need to be entered into the dependency tree so a proper textual specification can be later generated.

Table 6-11 provides some examples about how these rules are used to add suitable modal verbs into specifications. In this Table, the bold and italic dependencies indicate those created by replacing modal verbs (discussed in rule R2), bold dependencies indicate those added to while crossed out dependencies indicate those removed from the dependency tree.

Table 6-11. Modal Verb Replacement Examples

Original Spec & Dependency Tree	New Spec & Dependency Tree
The system must be secure det(system-2, The-1) nsubj(secure-5, system-2) aux(secure-5, must-3) cop(secure-5, be-4) root(ROOT-0, secure-5)	The system shall be secure det(system-2, The-1) nsubj(secure-5, system-2) ***aux(secure-5, shall-3)*** cop(secure-5, be-4) root(ROOT-0, secure-5) **Note:** "shall" replaces "must"
Users shall create reviews nsubj(create-3, Users-1) aux(create-3, shall-2) root(ROOT-0, create-3) dobj(create-3, reviews-4)	Users shall be able to create reviews **nsubj(able, Users-1)** **aux(able, shall)** **cop(able, be)** **root(ROOT-0, able)** **aux(create-3, to)** **xcomp(able, create-3)** dobj(create-3, reviews-4) ~~nsubj(create-3, Users-1)~~ ~~root(ROOT-0, create-3)~~ ~~aux(create-3, shall-2)~~ **Added ordering:** before(Users,shall), before(shall,be), before(be,able), before(able, to), before(to,create)
Banned users cannot write articles amod(users-2, Banned-1) nsubj(write-5, users-2) aux(write-5, can-3) neg(write-5, not-4) root(ROOT-0, write-5) dobj(write-5, articles-6)	Banned users shall not be able to write articles amod(users-2, Banned-1) **nsubj(able, users-2)** **aux(able, shall)** **neg(able, not)** **cop(able, be)** **root(ROOT-0, able)** **aux(write-5, to)** **xcomp(able, write-5)** dobj(write-5, articles-6) ~~nsubj(write-5, users-2)~~ ~~aux(write-5, can-3)~~ ~~neg(write-5, not-4)~~ ~~root(ROOT-0, write-5)~~ **Added ordering:** before(users, shall), before(shall,not), before(not,be), before(be, able), before(able, to), before(to, write)

6.2 Artifact Classification

The goal specifications extracted from the previous steps need to be classified to determine what artifact type (defined in GUIMeta's artifact layer) they belong to so they can be placed in a goal and use case integrated model. Such classification should contain the information about whether a goal specification is a business, functional or non-functional goal and which level of abstraction it belongs to (i.e., business, product, feature or service level).

In addition, if a goal is classified as a non-functional goal, it is useful to identify what non-functional category it belongs to (i.e., *usability*, *security*). Moreover, while most components in a use case do not need to be classified (since they normally come with indicators that indicate their roles in a use case), use case steps need to be classified to verify if they should be categorized as a non-functional or data constraints instead. This is critical even there are constraint extraction rules that can be used to recognize these constraints from step specifications because these extraction rules may be insufficient to identify all constraints.

In this section, we first provide some background about Mallet [102], a machine learning toolkit that supports text classification on which our classification technique is based. We then present our extension to Mallet to improve its accuracy in classifying textual specifications in goal and use case integrated models. Lastly, we discuss how such the extended version of Mallet supports the classification of artifacts in our work.

6.2.1 Text Classification with Mallet

Mallet supports text classification based on a range of machine learning algorithms including Naïve Bayes, Maximum Entropy and Decision Tree. Mallet allows the training of a classification model based on a set of sentences (or sequences of words) manually labeled to be certain topics or classes. Based on such a model, Mallet enables the calculation for the probability that a particular sentence can be classified into a certain class.

In order to perform model training and text classification, Mallet transforms each textual sentence into a so-called feature vector. Each feature vector contains the mapping of an integer number the represents a word in the sentence and the number of

times that this word appears in the sentence. When training is performed, these feature vectors are used to compute the probability of each word to be classified into each class. When classifying a sentence, the feature vector generated from such a sentence enables the calculation of probability for a sentence to be classified into each class based on the probabilities obtained during previous trainings. Table 6-12 presents the process of transforming textual sentences into feature vectors. We take the sentence *"Users shall be able to create reviews"* as an illustrating example.

Table 6-12. Sentence to Feature Vector Transformation Process

Steps	**Example Outcome**
1. Tokenize sentence	{"Users", "shall", "be", "able", "to", "create", "reviews"}
2. Convert token values to lowercase	{"users", "shall", "be", "able", "to", "create", "reviews"}
3. Convert tokens to features	{"users" (132), "shall" (12), "be" (45), "able" (18), "to" (76), "create" (165), "reviews" (321)}
4. Convert features to feature vectors	{132 – 1, 12 – 1, 45 – 1, 18 – 1, 76 – 1, 165 – 1, 321 – 1}

In the first step, the sentence is tokenized to obtain the set of words contained in the sentence. In step 2, these tokens (words) are converted into lower case. If this step is done during text classification and if an obtained token is not found in the set of trained tokens, it will be removed. In step 3, each token is assigned an integer number. If this step is done during training and if the token has already been assigned a number (i.e., it has been found in a sentence that has been processed), then the same number would be given. Otherwise the token is assigned the next integer number that has not been allocated to any other tokens. If this step is done during text classification then the token will then be given the number that previously assigned to the same token during training. In the last step, the assigned number is mapped to the number of times the token appears in the sentence (they are all 1 in this example since each word only appears once). Mallet provides an option to remove stop words (i.e., *"a," "an," "the," "that"*…) from the token list. For instance, if using this option, *"shall," "be," "able"* and *"to"* would not be considered. That enables the classifier to focus on only important words to classify sentences.

6.2.2 An Extended Version of Mallet

Since Mallet implements algorithms for general text classification, it does not well fit into the task of classifying goal specifications. In this section, we discuss our extensions to tailor the classifier to the needs of artifact specification classification.

Mallet's limitations in artifact specification classification

L1: Mallet considers different forms of a word differently. For instance, *"improves"* is considered as a different word from *"improve"* or *"improved."* Similarly, *"reviews"* is different from *"review."* However, from the semantic point of view, different forms of a word still share the same meaning. They thus should be treated equivalently. In addition, there exist cases in which a word can be both verb and noun (i.e., *"request," "link"*). They need to be differentiated.

L2: Mallet treats words individually without considering multiple word expressions. A multiple word expression is a group of words that are usually used together in a sentence. For example, *"user interface"* is a commonly used expression in usability and look-and-feel non-functional goals or constraints. It is also less likely to be used in other types of specifications. Therefore, if *"user interface"* is considered as a single word, it would be more probable that a specification containing this word to be correctly classified compared to the case that the word is split into *"user"* and *"interface."* That is because *"user"* and *"interface"* may appear in different classes of specifications (especially *"user"*). It thus may be less likely the specification is correctly classified (unless the number of usability or look-and-feel specifications is considerably higher than the number of specifications containing the word *"user"* and/or *"interface"*).

L3: Mallet lacks support for using keywords or patterns to enhance the classification of text, specifically, artifact specifications. In fact, there are keywords that can hint the category of the artifact specifications in which they are used. For instance, a specification containing the word *"productivity"* or *"efficiency"* may likely be a business goal (i.e., *"improve employee's productivity."* A specification that contains *"color scheme"* or *"color code"* may probably be a look-and-feel non-functional goal or constraint. Similarly, specifications with *"hours per day"* or *"hours per week"* may likely describe the availability of a system. In addition, a pattern such as *"in x seconds"*

or *"under x minutes"* may indicate non-functional specifications concerning speed of a system. Being able to utilize these keywords and patterns would probably improve the accuracy of artifact classification.

Addressing these limitations

In order to overcome these limitations, we provide additional processing into the transformation of sentence into feature vectors. Firstly, to address the limitation L1, each token is converted to its standard form (i.e., *"reviews"* would becomes *"review"*). In addition, to handle the cases in which a single word may be used with different roles (i.e., verb, noun), a POS tag is added after each token (i.e., *"review_vb," "review_nn"*). Secondly, to address the limitations L2 and L3, we build a set of keywords that are commonly used in specifications of the artifact classes defined in GUIMeta. Each keyword is associated to one or more artifact classes, implying that a specification containing such a keyword may likely belong to one of these classes. A keyword may contain multiple words. In this case, a dash (*"-"*) is used between words. For instance, *"user-interface," "operating-system"* or *"easy-to-learn."* A multi-word keyword can also represent a pattern. For example, *"every-second"* and *"under-minute"* are used to denote the patterns *"in x seconds"* and *"under x minutes"* respectively. Such keywords can be used to find specifications that contain the patterns since numbers (i.e., x in these patterns) would be removed since they are considered stop words.

To facilitate the use of keywords and patterns, the list of keywords is used as an additional training dataset during the training of a classification model. This ensures the feature created based on each keyword has a high probability for being classified into the classes associated to the keyword. In addition, the way a sentence is tokenized needs to be updated to accommodate multi-word tokens. For instance, if a sentence contains the word *"user interface,"* then this word only corresponds to a single token instead of two tokens holding *"user"* and *"interface"* separately.

Table 6-13 presents two examples for the new transformation process from sentences to feature vectors. In this new process, the sentence tokenization and lowercase conversion are combined as a single step. In this step, the following processing is done:

- A sentence is parsed using the Stanford parser to obtain a list of tokens, and each word is labeled with a POS tag.
- The sequence of tokens is examined to verify if multi-word keywords are used in this sentence, if a multi-word keyword is found, then the involved individual tokens are connected to each other with dashes ("-").
- Stop words are then removed from the token list
- The remaining tokens are then converted to their standard form, then to lower case and concatenated with an underscore ("_"), and then their standard POS tag (multi-word tokens do not need to have a POS tag assigned). For instance, *"validates"* in the first example is a verb and thus is converted to *"validate_vb."*

The other two steps (*"Convert tokens to features"* and *"Convert features to feature vectors"*) remain unchanged.

Table 6-13. New Transformation Process from Sentence to Feature Vector

Considered sentence: *"System validates the review content in 1 second"*	
Steps	**Example Outcome**
1. Tokenize sentence	{"system_nn", "validate_vb", "review_nn", "content_nn", "in-second"}
2. Convert tokens to features	{"system_nn" (198), "validate_vb" (139), "review_nn" (321), "content_nn" (476), "in-second" (683)}
3. Convert features to feature vectors	{198 – 1, 139 – 1, 321 – 1, 476 – 1, 683 – 1}
Considered sentence: *"System shall be easy to use"*	
Steps	**Example Outcome**
1. Tokenize sentence	{"system_nn", "easy-to-use"}
2. Convert tokens to features	{"system_nn" (198), "easy-to-use" (956)}
3. Convert features to feature vectors	{198 – 1, 956 – 1}

6.2.3 Artifact Classifiers

Our extended version of Mallet plays a key role in the development of our *classifier* to classify artifact specifications. Our classifier is composed of two separated classifiers. The *horizontal classifier* is used to determine if a specification describes a business objective, functional or non-functional property or data constraint. For a non-functional

goal or constraint, it also provides the prediction as to which non-functional category the specification belongs to (i.e., *security*, *usability*). Currently we support the identification of 20 non-functional categories defined in GUIMeta's artifact layer (discussed in Chapter 4). The *vertical classifier* is used to identify the abstraction level a goal should belong to (product, feature, or service level).

The rationale for using two separate classifiers are twofold. Firstly, the classification of use case step specifications only requires the horizontal classifier. Secondly, this separation of classification concerns can help reducing training data size. For instance, if a single classifier were used, then we would need sufficient training data for 23x3 classes, as opposed to only 23 classes (business goal, functional goals, data constraints and 20 non-functional goal or constraint categories) for the *horizontal classifier* and 3 classes (product, feature and service) for the *vertical classifier*. Additionally, the training data can be shared between the two classifiers (i.e., a specification is classified as a functional goal and belong to the service level of abstraction), given they are appropriately labeled for each classifier.

Up to now we have trained the classifiers with over 2500 requirements collected from multiple sources including online resources, literature and books. GUEST allows the classifiers to be further trained or re-trained. In addition, we have created a set of over 350 keywords for supporting the classification of artifacts. This set of keywords can also be extended and modified depending on the need of particular projects.

Figure 6-2 presents the high-level algorithm for the classification process goals. The classification result is usually the class that the goal has the highest probability of belonging to. However, since it may exist cases in which multiple classes have the same probability, the classification result may sometime contain multiple classes. In this classification process, a goal specification is first classified by the horizontal classifier. If the goal is classified as a business goal or a data constraint, then the classification is terminated (the goal's class has been determined). If the result is a functional goal or a non-functional goal, then the vertical classifier is used to identify the level of abstraction the goal should be on (i.e., feature, service). The combination of classification results from these two classifiers would then be used to determine the class of the goal.

The classification of use case steps is similar to that for goals. However, it is much simpler since only the horizontal classifier is required.

```
Function classifyGoal(spec)
  Input: a goal specification
  Output: a list of class the goal specification can belong
  to // multiple classes are returned instead of a single
  class when the specification has equivalent highest
  probability of being classified into different classes

  classes = empty list of goal classes
  horiontalClasses = classify(horizontalClassifier, spec)
  IF horizontalClasses = NULL || isEmpty(horizontalClasses)
     return NULL
  END IF
  FOR EACH class in horizontalClasses
     IF class = 'business goal' || class = 'data constraint'
        classes.ADD(class)
     ELSE
        verticalClasses = classify(verticalClassifier, spec)
        IF verticalClasses = NULL ||isEmpty(verticalClasses)
           return NULL
        END IF
        Classes.ADD(CombineGoalClasses(class,
                                       verticalClasses))
     END IF
  END FOR
  return classes
END Function
```

```
Function classify(classifier, spec)
  Input: a classifier and a specification to be classified
  Output: a list of class the specification can belong to

  return classifier.classify(spec)
END Function
```

```
Function combineGoalClasses(horizontalClass, verticalClasses)
  Input: a horizontal class and a list of vertical classes
  Output: a list of class the specification can belong to

  classes = empty list of goal classes
  FOR EACH verticalClass in verticalClasses
     class = getClass(horizontalClass, verticalClass) //get
     goal class based on horizontal and vertical
     classification. i.e., if horizontalClass = 'functional
     goal' & verticalClass = 'service goal' then class =
     'functional service goal.' If horizontalClass =
```

```
        'functional goal' while verticalClass = 'product goal'
        then class = NULL
        IF class != NULL
            classes.ADD(class)
        END IF
    END FOR
    return classes
END Function
```

Figure 6-2. Classification Algorithm

6.3 Model Construction

The outcomes of previous steps in the model extraction process are documented in the form of an extraction report, which contains *extraction results* and an *extraction log*. *Extraction results* contain a set of extracted and classified specifications and the relationships between these specifications. In this step, these details are synthesized to build a model. For instance, if a specification is classified as a business goal, then a business goal is generated based on such a specification.

The relationships that are extracted between these specifications would also be used to form the relationships between artifacts in a model. One of the main issues in the synthesis of extraction results is that the extracted artifacts may be the duplicates of each other. GUEST is able to identify duplicate artifacts if they have the same specifications. In such cases, only one copy of the duplicated artifact is included in the model. GUEST by itself does not handle the duplication of artifacts that have semantically equivalent specifications (but written differently). Such cases are addressed by our GUITAR tool, which provides semantic analysis support in the form of goal-use case integrated models. GUITAR is discussed in Chapter 7. Since GUEST and GUITAR are fully integrated in a single tool called GUITARiST, users are provided an option to request that a model is analyzed immediately after it is extracted. In this case, an analysis report generated by GUITAR would be provided to users.

Extraction results may contain multiple sets of specifications and relationships in case alternative results were obtained. For instance, a goal can be classified into multiple artifact classes, or multiple extraction rules can be applied into a single sentence, leading to different artifacts and relationships extracted. In such cases, GUEST uses one of the alternatives to create a model. However, the alternative results are also presented

to users. They then can choose to create another model based on an alternative extraction result if such an alternative is more desirable. GUEST also provides full model editing features to enable users to make necessary corrections or manually extend the model when additional details are available.

The extraction results are dependent on the quality of various components including the natural language parser (Stanford parser in this case), the coreference resolution system, the artifact classifier and the set of extraction rules, modifying rules. Therefore, there are cases the obtained extraction result is not correct due to the errors produced by one or more of these components. For instance, if the natural language parser does not correctly parse a sentence, then that would lead to an incorrect result in the entire extraction related to that sentence, no matter how the quality of the other components are. Similarly, if no extraction rule exists to identify that a sentence actually contains two goal specifications with a certain relationship between them, then the extraction result would also be incorrect.

In order to help users monitor and get more insights into the extraction process, GUEST produces an *extraction log* that contain the results produced by each component in each step of the extraction and details of problems encountered during the entire extraction process. In case users identify an error in an extracted model, they can review the entire extraction process (based on this extraction log) to locate the source of problems and make corrections. For instance, if it is found that the natural language parser incorrectly parsed a sentence, then users can train the parser with a correct manually generated parse tree (for that sentence) to ensure a correct parse if a similar sentence is encountered in the future. The details of an extraction log are described as follows:

- **Pre-processing details:** each sentence (called an original sentence) in the original requirements document is associated to its pre-processed version (the version that was used for the extraction). This association indicates the details modified or removed in the original sentence during the pre-processing step. This information helps users identify if important details were removed or modified and thus make adjustments in the requirements documents to avoid this incorrectness.

- **Constituent parse trees and dependency trees:** the constituent parse tree and dependency parse tree of each processed sentence in a requirements document are included. They help users determine if a sentence is not correctly parsed.
- **Coreference resolution details:** contain information about the identified coreference and changes made in the relevant sentences to resolve such coreference.
- **Used extraction rules:** contain the information about the extraction rules applied for each sentence and the outcome produced after each rule was executed. The cases in which a single sentence matches with multiple extraction rules are also highlighted. These details enable users to get insights into how alternative extraction results are produced. These can also help them verify if some certain extractions were not done properly (i.e., due to a rule that was not specified correctly).
- **Polishing details:** contain the information about how a specification is polished. For instance, is such a specification needed to be transformed from passive voice to active voice, or which modifying rules have been used.
- **Classification results:** the probabilities calculated by the artifact classifier for each artifact specification are all recorded. That helps users verify the correctness of the artifact classifier and get insights into alternative extraction results are produced (if any).

6.4 GUEST: Goal and Use Case Extraction Supporting Tool

We describe our tool GUEST that supports the extraction of goal and use case integrated models from textual requirements documents based on the discussed extraction techniques. We first discuss the architecture of the tool. We then provide some usage examples to demonstrate how GUEST supports the extraction process.

6.4.1 Tool Architecture

GUEST is developed as an Eclipse Rich Client Platform tool. Figure 6-3 presents an overview of GUEST's architecture. The tool consists of three main components. The *extraction controlling module* is responsible for the entire process of extracting goal-use case models from natural language-based requirements documents. It contains a number of sub-modules, each is responsible for a step in the extraction process. The *document*

processing module extracts requirements documents into plain text (handled by the *document extractor*), associates document sections to artifact types and pre-process the section text (handled by the *text preprocessor*). The *linguistic analysis module* contains the *Stanford coreference resolver* that helps resolving coreference in the processed text and the *Stanford parser* that is used to produce constituent and dependency parse trees.

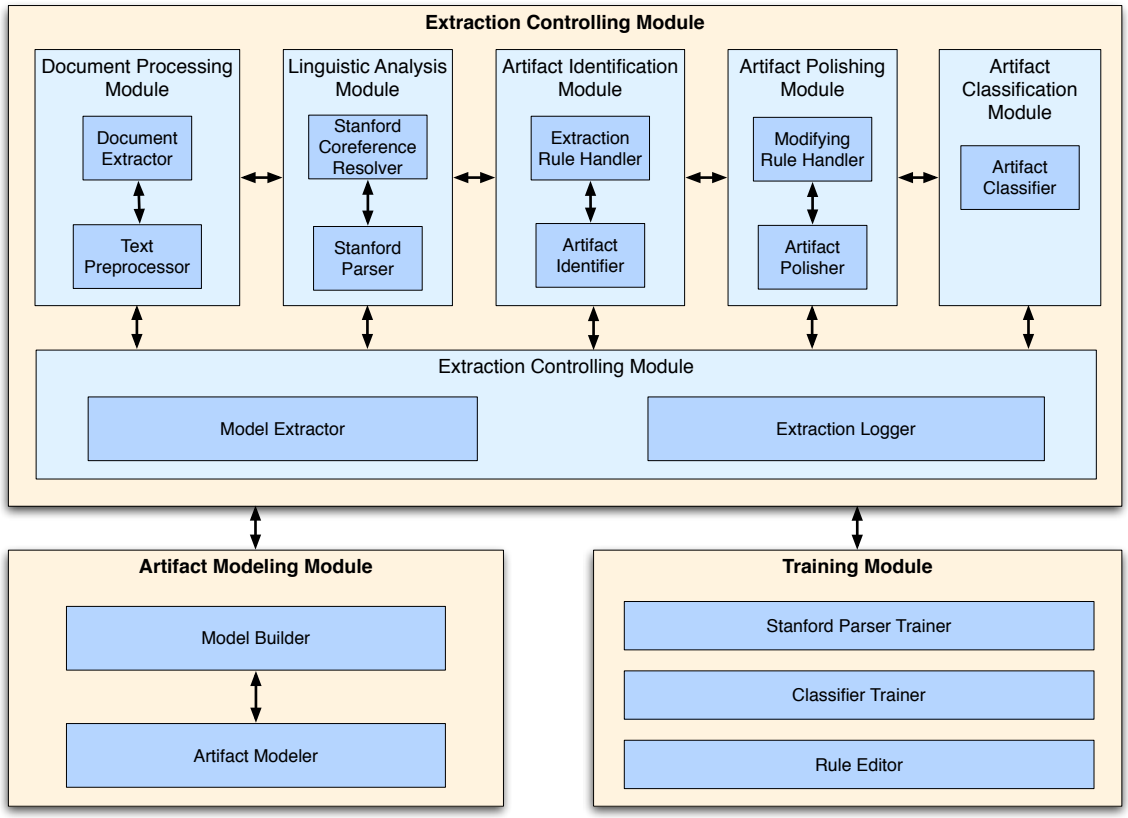


Figure 6-3. GUEST Architecture

The *extraction rule handler* and *artifact identifier* are the components of the *artifact identification module*. The former is responsible for parsing extraction rules, identifying matching rules for a sentence and executing these rules to produce outcomes. The later manages the process for identifying goals, use case components and relationships with the support of the former component. Similarly, the *artifact polisher* in the *artifact polishing mo*dule manages the entire process of polishing an artifact specification (i.e., transform passive voice to active voice, replacing adjective-preposition phrases) with the support of the *modifying rule handler* which handles the identification and execution of modifying rules.

In addition, the *artifact classifier* (from the artifact classification module), which is built based on Mallet classifier, enables artifact specifications to be classified. Lastly, the *extraction controlling module* provides two components. The *model extractor* controls the entire extraction process, gathers the outcomes from all steps and produces extraction results. The *extraction logger* records the outcomes from each step and all problems that occur during extraction.

The artifact modeling module receives the extraction results from the extraction controlling module to build goal-use case models (handled by the *model builder*). Moreover, the artifact modeling module enables the editing of goal-use case models after they have been extracted (handled by the *artifact modeler*). This module also allows users to select between alternative extraction results and produces alternative goal-use case models. The training module allows users to train the Stanford parser and artifact classifier with additional data (supported by the *Stanford parser trainer* and *classifier trainer*). It also provides a rule editor that enables the editing and adding of extraction and modifying rules.

6.4.2 Usage Examples

In this section, we provide some examples to illustrate the use of GUEST in extracting goal and use case integrated models. These examples are based on a scenario in the domain of traveler social networking system.

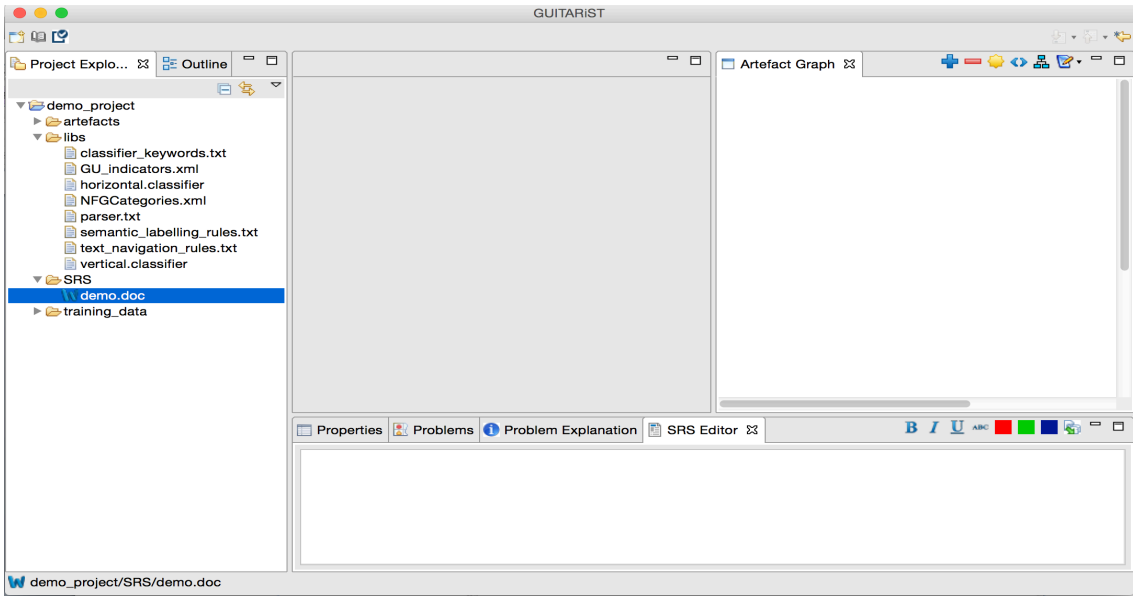


Figure 6-4. Key Components in a Project

Figure 6-4 shows a screenshot that displays the key components of a project in GUEST (since GUEST and GUITAR tools are integrated into a single tool called GUITARiST, the window in the figure has its name as *GUITARiST*. The GUITAR tool is discussed in Chapter 7. The directory *'SRS'* contains the requirements documents to be extracted. The directory *'libs'* contains the supporting libraries for the extraction and modeling of goals and use cases. These include the Stanford parser model, classifier files, the collection of extraction and modifying rules (both are referred to as *text navigation rules*). The *'training_data'* directory stores the training data for the parser and classifiers. The extracted model is stored in an XML file placed in the *'artifacts'* directory.

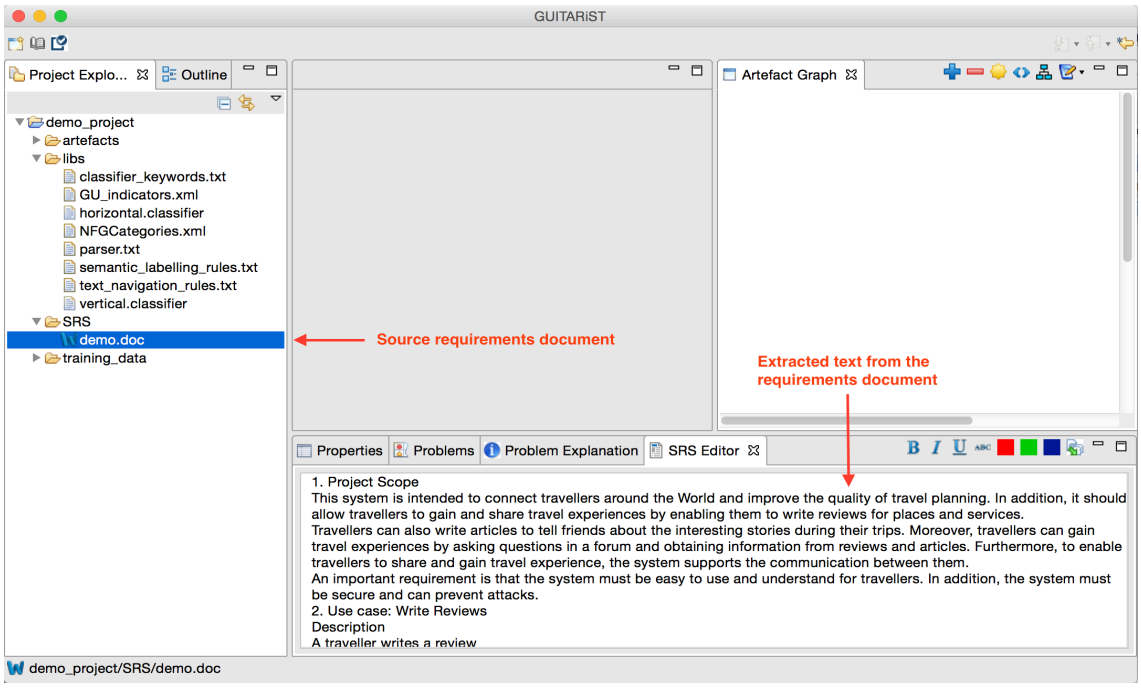


Figure 6-5. Extract Text from a Requirements Document

Once the demo.doc file (which is a requirements document to be extracted) is double clicked, GUEST pre-processes the text in this document and associates each section in this document to certain artifact types according to the artifact type indicator list that was created and included in the *"libs"* directory of the project. Figure 6-5 shows the text extracted and pre-processed from the requirements document. GUEST provides a SRS editor to enable users to further edit the extracted text if needed. Figure 6-6 shows the artifact indicator lists created for this document. The full extracted text is showed in Figure 6-7. In this figure, the annotators (i.e., S1, S2) are added to help referring to the sentences in the following discussion.

```
<gu_indicators>
    <non_use_case>
        <indicator>project scope</indicator>
    </non_use_case>
    <use_case>
        <indicator>use case</indicator>
    </use_case>
    <pre-condition>
        <indicator>pre-condition</indicator>
    </pre-condition>
    <post-condition>
        <indicator>post-condition</indicator>
    </post-condition>
    <use_case_desc>
        <indicator>description</indicator>
    </use_case_desc>
    <main_scenario>
        <indicator>main success scenario</indicator>
    </main_scenario>
</gu_indicators>
```

Figure 6-6. Artifact Indicator List

GUEST is able to automatically extract the model from this text based on a collection of extraction rules. Figure 6-8 presents the partial extracted model in which a number of artifacts are identified, polished and classified. For instance, the functional feature goal FFG2 (*"Travelers shall be able to gain travel experiences"*) and its sub-goals (FFG1, FFG5, FSG1, FSG2, FSG3, FSG4, and FSG5) are extracted. These goals (and the relationships between them) are derived from three sentences: S2, S4, and S5. Firstly, FFG2, FSG1 (*"travelers shall be able to write reviews for places"*), and FSG2 (*"Travelers shall be able to write reviews for services"*) are extracted from the sentence S2 (*"In addition, it should allow travelers to gain and share travel experiences by enabling them to write reviews for places and services"*). Secondly, from the sentence S4 (*"Moreover, travelers can gain travel experiences by asking questions in a forum and obtaining information from reviews and articles"*), FSG3 (*"Travelers shall be able to ask questions in a forum"*), FSG4 (*"Travelers shall be able to obtain information from reviews"*), and FSG5 (*"Travelers shall be able to obtain information from articles"*) are identified as sub-goals of FFG2. Thirdly, FFG5 (*"The system shall be able to support the communication between travelers"*) is extracted as another sub-goal of FFG2 from the sentence S5 (*"Furthermore, to enable travelers to share and gain*

travel experience, the system supports the communication between them"). Note that the text referring to the goal FFG2 (*"Travelers shall be able to gain travel experiences"*) is repeated in all three mentioned sentences. GUEST is able to identify such duplication and only includes one copy of such goal in the extracted goal-use case model.

In addition, for traceability purpose, GUEST provides a link between each obtained artifact and the sentence from which it is extracted. For instance, when clicking on the functional service goal FSG4, the sentence where this goal was extracted from is highlighted in the SRS editor.

1. Project Scope

(S1) This system is intended to connect travelers around the World and improve the quality of travel planning. **(S2)** In addition, it should allow travelers to gain and share travel experiences by enabling them to write reviews for places and services.

(S3) Travelers can also write articles to tell friends about the interesting stories during their trips. **(S4)** Moreover, travelers can gain travel experiences by asking questions in a forum and obtaining information from reviews and articles. **(S5)** Furthermore, to enable travelers to share and gain travel experience, the system supports the communication between them.

(S6) An important requirement is that the system must be easy to use and understand for travelers. **(S7)** In addition, the system must be secure and can prevent attacks.

2. Use case: Write Reviews

Description

A traveler writes a review

Pre-condition

The traveler has successfully logged into the system.

Post-condition

The new review is stored

Main Success Scenario

Step 1. The traveler selects to create a review.

Step 2. System prompts the traveler to select a review category.

Step 3. The traveler selects review category. The list of categories includes hotel, attraction and tour.

Step 4. System displays the suitable review creation form to the traveler.

Step 5. The traveler enters the review content and submits it

Step 6. System validates the review

Step 7. If the review content is valid, the system stores the review into the database and displays a confirmation message to the traveler; else the system displays the errors to the traveler and the traveler repeats step 5.

Figure 6-7. Full Extracted Text

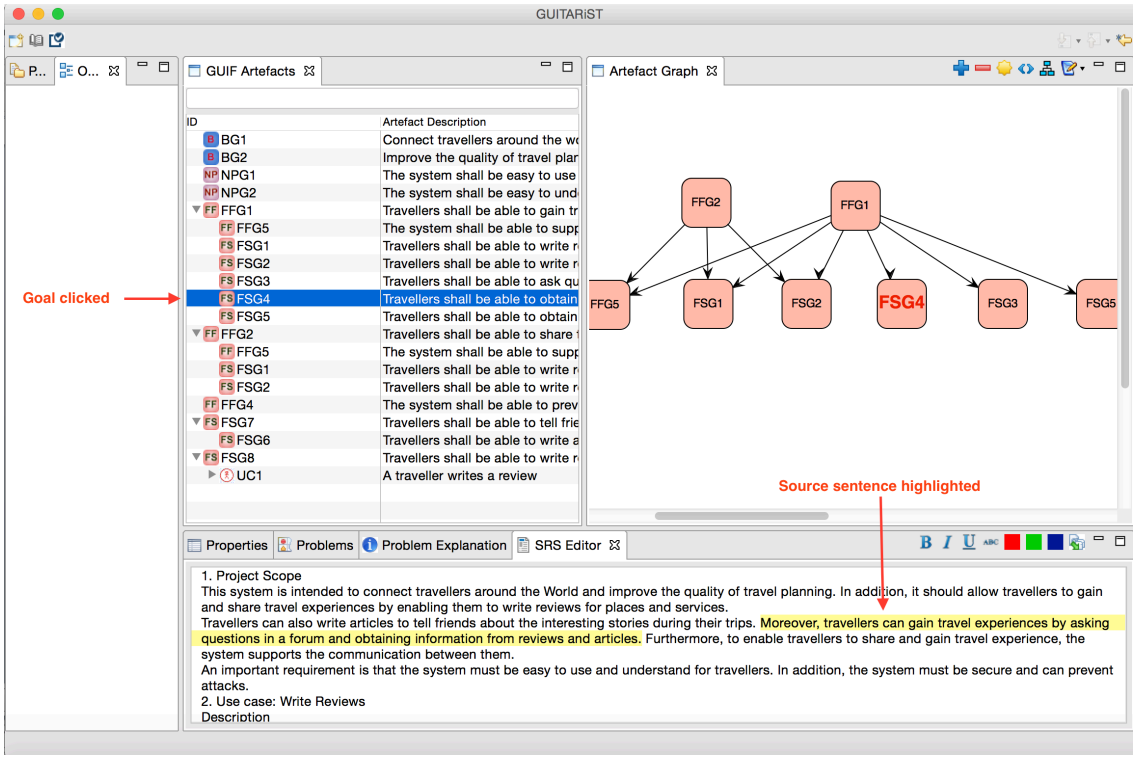


Figure 6-8. An Extracted Business Goal and its Sub-goals

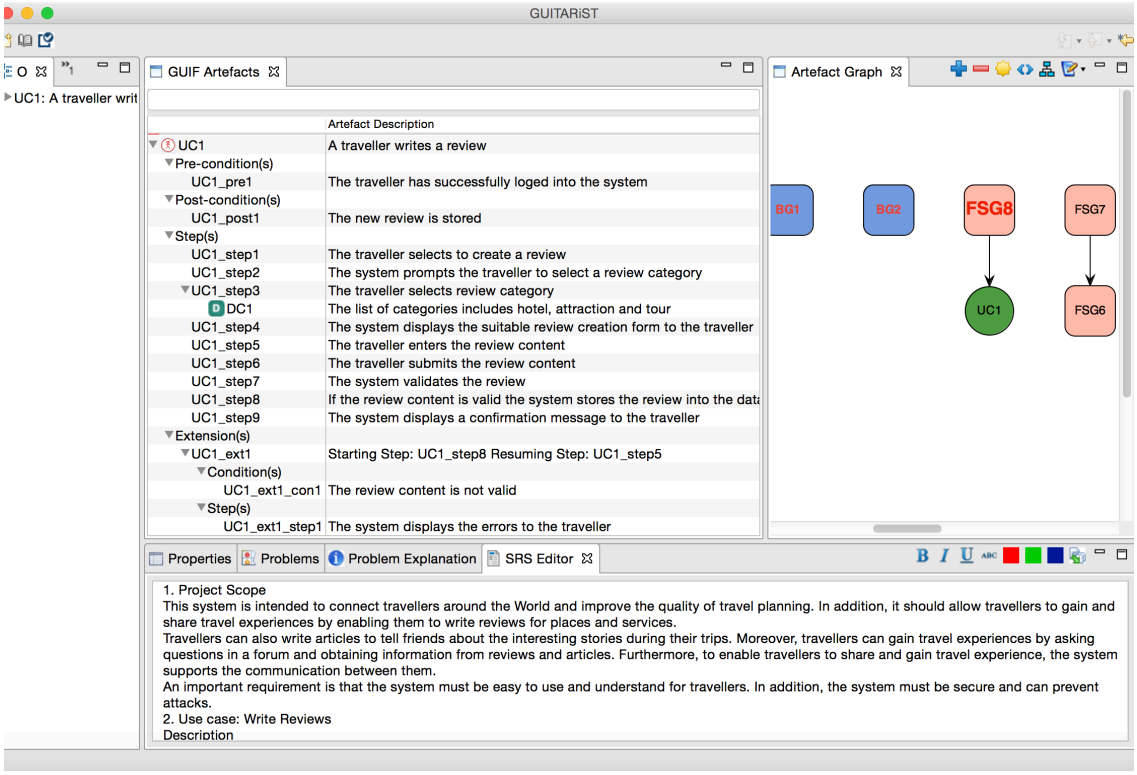


Figure 6-9. An Extracted Use Case and its Components

Figure 6-9 presents an extracted use case from the use case description in the extracted text (in Figure 6-7). Based on the brief description of the use case (*"a traveler writes a*

review"), a functional service goal FSG8 (*"Travelers shall be able to write reviews"*) is created and linked to this use case.

In this example, the pre-condition *"the traveler has successfully logged into the system"* and the post-condition *"the new review is stored"* is extracted based on the corresponding indicators (*'pre-condition'* and *'post-condition'*) in the use case specification (cf. Figure 6-7). Moreover, although no extension is explicitly specified in the text, it is implicitly stated in the sentence *"If the review content is valid, the system stores the review into the database and displays confirmation message to the traveler; else the system displays the errors to the traveler and the traveler repeats step 5."*

As shown in Figure 6-9, GUEST extracts the statements *"if the review content is valid, the system stores the review into the database"* and *"the system displays confirmation message to the traveler"* as steps of the use case. GUEST also identifies an extension of the use case that handles the case when the review content is not valid. Such extension includes the step *"the system displays the errors to the traveler,"* the starting step (UC1_Step7 – which is the use case step that deals with the opposite case of the one handled by this extension) and resuming step (UC1_Step5 – which is specified by the sentence *"the traveler repeats step 5"*).

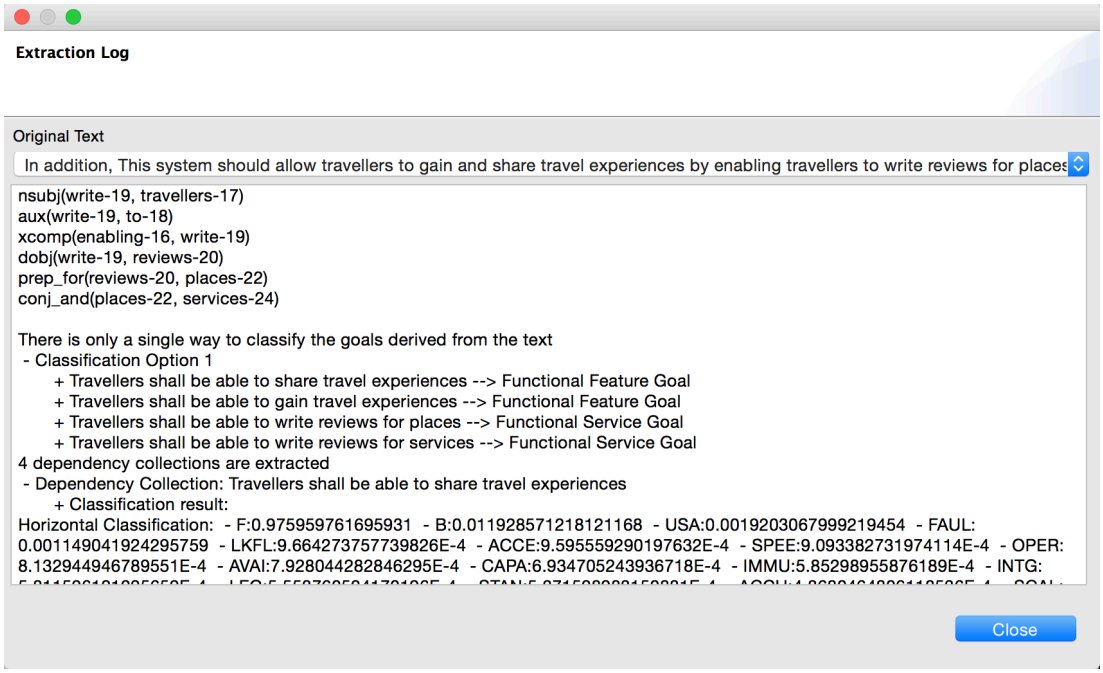


Figure 6-10. An Extraction Log

GUEST also enables users to view extraction log that contains the details as to how each sentence in the text is processed and analyzed to extract artifacts and relationships. Figure 6-10 shows an example for such a log for the sentence S3. The portion of log in this figure shows the classification results of each artifact extracted from this sentence.

Since the quality of extraction depends on the accuracy of the Stanford parser, artifact classifiers and the collection of extraction and modifying rules, GUEST enables users to further train the parser and classifiers, create new rules and modify existing rules. Figure 6-11 presents a dialog in which the horizontal classifier is incrementally trained with new data. The data should be in the format of `<label>[space]<text>` in which `<text>` refers to a sentence or sequence of word and `<label>` refers to the class into which such a sentence should be categorized. In case it is needed to completely retrain the classifier, the relevant training data file in the *"training_data"* directory in the project tree should be removed before new training data are entered into the dialog.

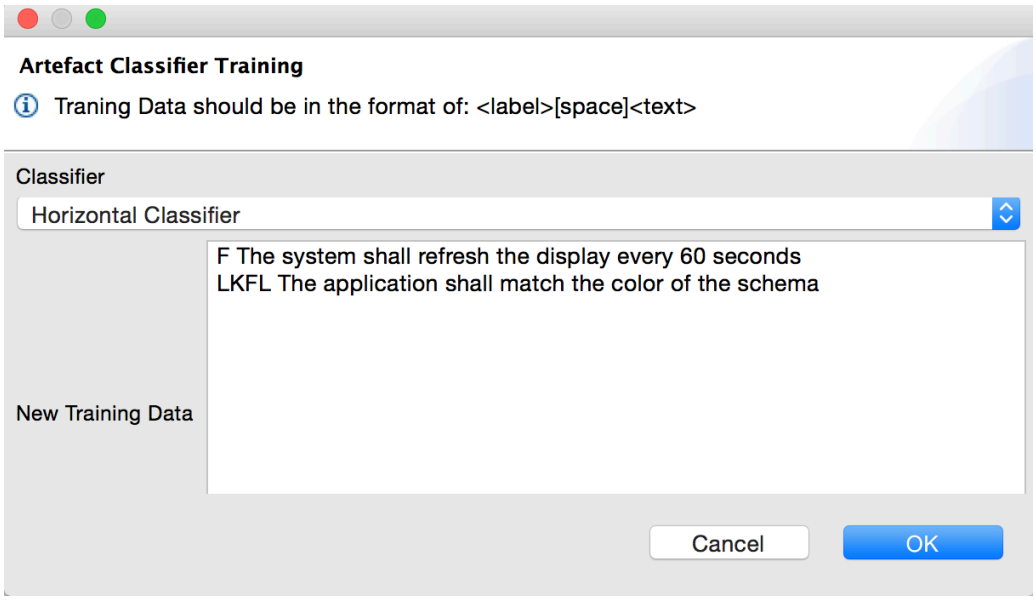


Figure 6-11. Training a Classifier

Figure 6-12 shows a dialog in which the Stanford parser is further trained with new data. Parser training data should be in the format of a constituent tree.

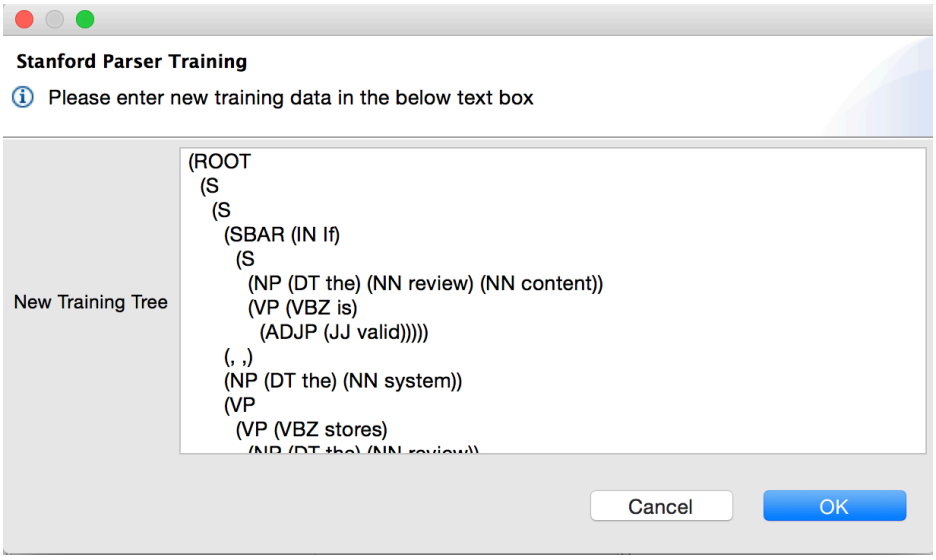


Figure 6-12. Training the Stanford Parser

Figure 6-13 presents a dialog in which a new extraction rule is added into the rule collection.

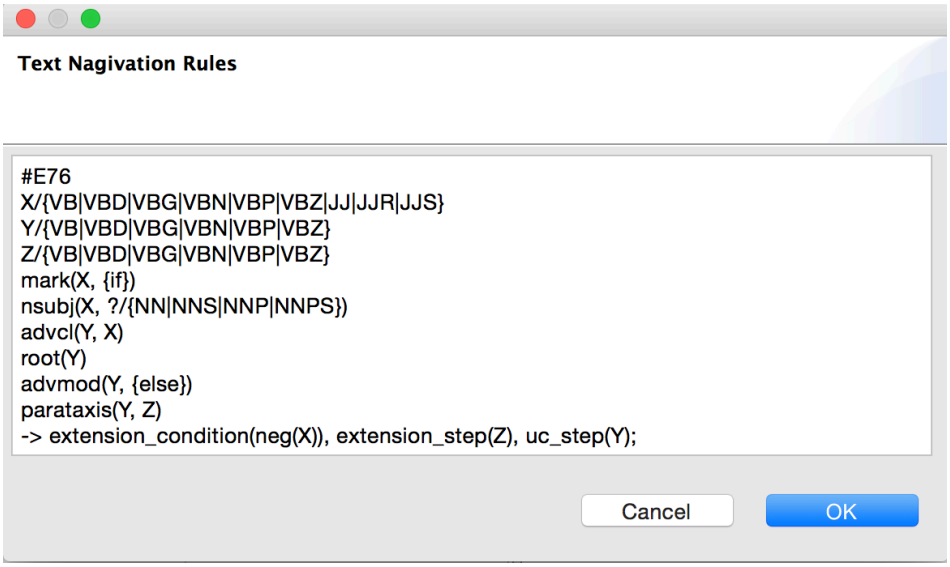


Figure 6-13. Creating an Extraction Rule

6.5 Evaluation

In this section, we present the evaluations conducted to evaluate the effectiveness of GUEST in extracting goal and use case integrated models from requirements documents. Specifically, we seek to answer the following research questions:

- **RQ1:** How accurately does GUEST classify artifacts by their textual specifications?
- **RQ2:** How accurately does GUEST extract goal-use case models from requirement specifications documents?

6.5.1 Evaluation Setup

Table 6-14 presents our formulas to calculate the metrics for each research questions.

To tackle the research question RQ1, we carried out a benchmark validation to compare and contrast our requirements classifier (which is an extended version of the Mallet classifier) with the existing state-of-the-art classifiers: Casamayor et.al. non-functional requirements classifier [18] (we term it as *Casamayor* in this section) and original version of Mallet (we refer to it as *original Mallet* in this section). The reasons for this selection were fourfold. First, Casamayor is among the classifiers developed recently and reportedly obtained high results in its evaluation. Second, Casamayor's experiment steps were described clearly and its data is available, making it possible to reconstruct the exactly same validation. Thirdly, no requirements classifier was available to download at the time when this validation was conducted, and lastly, the original version of Mallet was selected as a benchmark application to verify whether our improvements made to extend it were effective.

In this evaluation, we followed Casamayor's experiment steps to run a validation on the PROMISE dataset that we introduced in Chapter 3. This dataset contains 255 functional requirements and 370 non-functional requirements. However, only the non-functional requirements were used in this evaluation (since Casamayor et al. only employed these requirements). Multiple experiments were run using the cross-validation technique. In each experiment, a k-portion of data (i.e., k=90%) was randomly selected as training data and the rest was used as testing data. Each experiment is run in 10 iterations to obtain the scores for the precision and recall metrics (cf. Table 6-15). 19 experiments were run with k was 5, 10,…, 95 respectively.

Table 6-14. Formulas for Metrics

	Formulas
RQ1	$Precision = \frac{TP}{TP + FP}, Recall = \frac{TP}{TP + FN}$ **TP:** True Positive (number of artifacts are correctly classified) **FP:** False Positive (number of incorrectly classified artifacts)
RQ2	$Precision = \frac{TP}{TP + FP}, Recall = \frac{TP}{TP + FN}, Fmeasure = \frac{2 * P * R}{P + R}$ **TP:** True Positive (number of valid extracted artifacts or relationships) **FP:** False Positive (number of invalid extracted artifacts or relationships) **FN:** False Negative (number of artifacts or relationships not extracted)

To address the research question RQ2, we targeted to evaluate GUEST's performance in different phases of the extraction process. These phases include the extraction of raw artifacts and relationships, the polishing of artifacts, and artifact classification. While all extraction results were considered in the first two phases, only goals, data constraints and non-functional constraints are considered for the last phase (artifact classification) since other types of artifacts (i.e., use case components such as steps, conditions) do not need to be classified during the extraction process (they are identified by a list of component indicators and/or extraction rules).

We chose to use the online publication system (OPS) and split payment system (SPS) industrial case studies in this evaluation. The requirements documents from these case studies follows the IEEE requirements specification template and contains a number of sections for goals and use cases. In each case study, we first manually modeled goals and use cases from the given requirements document. We then ensured the requirements document to meet the formatting requirements of GUEST. For instance, we removed table of contents, replaced slashes (*'/'*) with appropriate words (i.e., *'and,' 'or'* depending on the actual meaning of a slash in a context), and moved important content out of tables (since GUEST ignores all text in tables). We then used GUEST to extract goal-use case model and compared it with our manually extracted model. We gathered and analyzed the comparison results for all extraction phrases.

6.5.2 Evaluation Results

In this section, we discuss the results of our evaluation. Our full experimental data can be found at http://goo.gl/gCUofM.

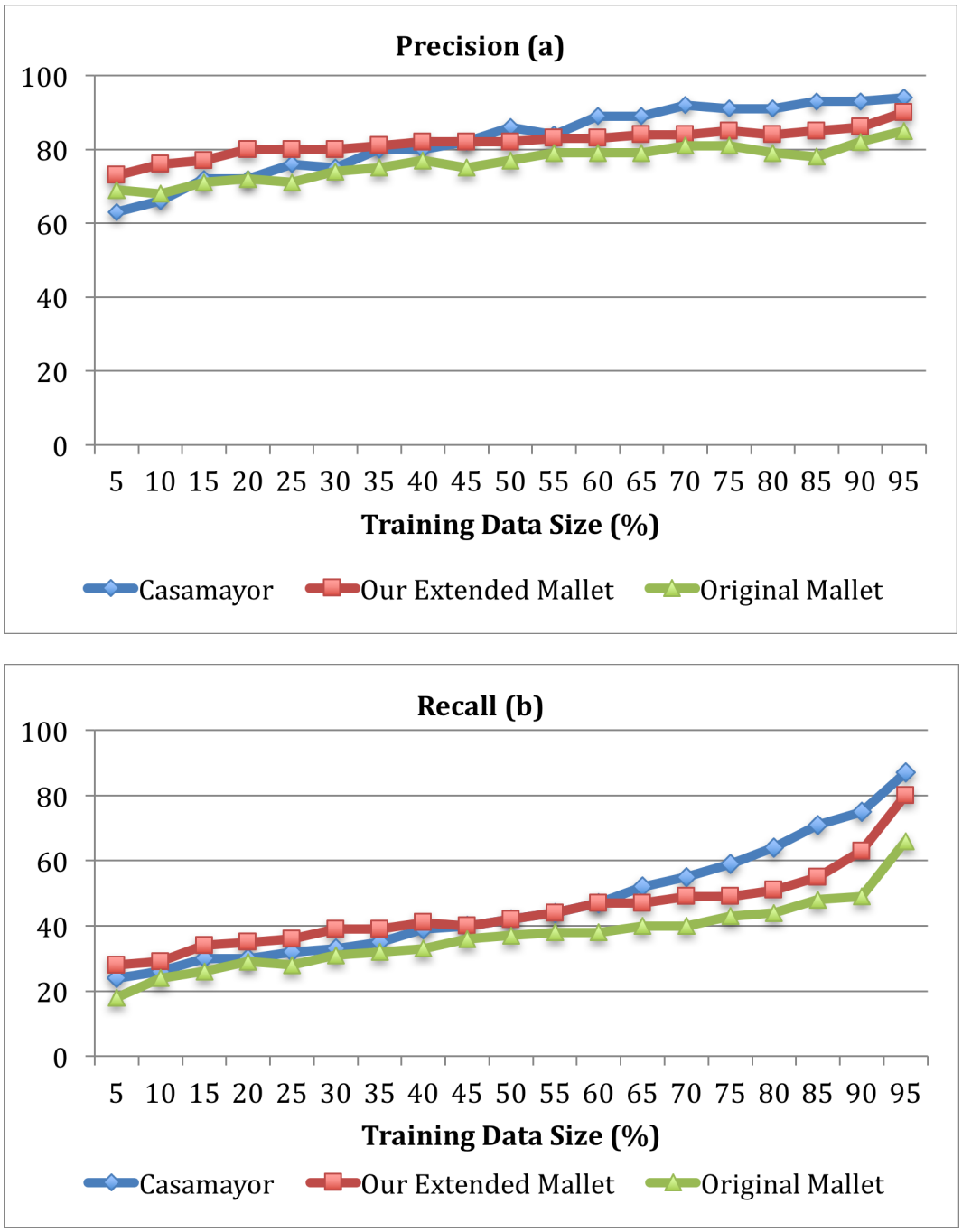


Figure 6-14. Classifier Benchmark Validation Results

RQ1: Artifact Classification

Figure 6-14(a) and (b) present the comparison between three classifiers regarding precision and recall rates. According to the result, all three classifiers obtained higher precision and recall rates when the training set size increased. It can be noticed from this comparison that our classifier produced higher-quality results than others when the training set was small. This was due to the support of our non-functional indicator keywords. As showed in the graphs, when the training set size increased, Casamayor's results raised with highest rates and surpassed our classifier when the training set was over 50% of the entire data (that means the training set is larger than the testing data set). Original Mallet followed a very similar trend as our classifier that is due to the share of algorithms between the two classifiers. However, our classifier outperformed original Mallet with at least 5% difference in most experiments. From these results, it can be seen that the key benefit of our classifier is that it performs relatively well with a small training dataset.

RQ2: Goal-Use Case Model Extraction

The results are showed in table 6-15. We achieved 86% precision and 84% recall rates for the artifact extraction, and 87% precision and 78% recall rate for the relationship extraction. A number of artifacts and relationships were not extracted due to the missing of relevant extraction rules. In addition, some relationships were not detected since identifying such relationships required the understanding of the entire contexts in which the artifacts were specified. For instance, consider these two sentences: *"The system will be easy to use for users. It should ensure articles to be available to readers at any time and allow them to search and download articles quickly and easily."* GUEST could not identify the *'sub-goal – parent-goal'* relationship between *"readers shall be able to search articles easily"* (extracted from the second sentence) and *"The system will be easy to use for users"* (extracted from the first sentence). Such a relationship was not found by GUEST since it is not identifiable in the way the sentences are written and thus, the detection of it must rely on the meaning of the extracted artifacts. This type of relationship can instead by identified by our tool GUITAR (developed for model analysis) that is discussed in Chapter 7.

Table 6-15. Extraction Validation Results

D: total detected FP: False Positive FN: False Negative					
Case Study		**OPS**	**SPS**	**Average**	**F-Measure**
Artifact	**D**	199	167		
	FP	29	22		
	FN	30	28		
	Precision	85%	86%	86%	85%
	Recall	85%	83%	84%	
Relationship	**D**	161	99		
	FP	22	13		
	FN	39	24		
	Precision	86%	87%	87%	82%
	Recall	78%	78%	78%	
Polishing		96%	95%	96%	
Goal Classification		83%	79%	82%	

Moreover, some other relationships between artifacts from different sentences were not detectable due to the use of verbal phrase coreference in such sentences. For instance, consider the sentences: *"The application periodically communicates with the SplitPay server. This allows bills to be uploaded to a central server."* The word 'this' in the second sentence refers to the entire activity in the first sentence (*"The application periodically communicates with the SplitPay server"*). Therefore, apart from the two artifacts *"The application periodically communicates with the SplitPay server"* and *"upload bills to a central server,"* the expected extraction result would need to contain the *'sub goal – parent goal'* relationship between the former and the later artifacts. This type of relationship was not identified by GUEST since our tool currently does not provide support for verbal phrase coreference (in fact, there is no linguistic tool exists that can properly identify such type of coreference currently).

We achieved 96% accuracy rates of the polishing of artifacts. In this evaluation, a polishing result is considered correct if the extracted artifact specification satisfies our defined specification rules (boilerplates). The common errors GUEST had during this evaluation were due to missing relevant rewriting rules. For instance, GUEST could not properly rewrite the goal *"Include support for simultaneous bills"* to the desired form

of *"Support simultaneous bills."* We achieved 81% overall (83% and 79% respectively in OPS and SPS case studies) for artifact classification in this validation. This means about 20% of the extracted goals were not correctly classified into the right classes by our classifiers.

To sum up, from these case studies, it is seen that our artifact specification classifier performed reasonably well. From the benchmark validation, it was found that our classifier's performance, while constantly being higher than that of original version of Mallet, was not too far below the result of Casamayor's classifier, a state-of-the-art requirements classifier. Our classifier even outperformed Casamayor's classifier when the training set is small, due to the support of our classification keywords. Importantly, the quality of our classifier can be further improved by doing additional training, which is supported by GUEST.

In addition, we identified that the false positives and false negatives in artifact and relationship extraction were due to three main reasons: missing extraction rules, parsing errors, and lacking of semantic understanding. In fact, some artifacts and relationships could not be extracted correctly (or extracted while they should be ignored) since no suitable supporting extraction rules exist. On the other hand, in some cases, suitable extraction rules did exist, however a sentence was not parsed properly by the Stanford parser, making the artifacts and/or relationships unrecognized. Similarly, the incorrectness in artifact polishing and classification came from invalid results produced by the linguistic parser, missing modifying rules and artifact classifier respectively. Therefore, extraction results could be improved if our extraction rule and modifying rule sets are extended and the parser and classifier are further trained. Such extension and training are all supported by GUEST.

6.5.3 Threats to Validity

In this section, we discuss the threats to validity from this evaluation.

External Validity: A threat to external validity was the representativeness of the selected case studies and the quality of our parser, classifier and rules. In fact, having good results with the selected case studies may not imply similar results in others. To reduce this threat, we increased the variability of the data by selecting requirements

from different sources while obtaining a large amount of requirements. The parser, classifier and rule collection used were developed prior to the evaluation. They were not re-trained or modified to specifically deal with requirements in these case studies.

Internal Validity: Threats to internal validity include the human factors in determining the correctness of GUEST results in each validation. In RQ2, we manually extracted goals and use cases from the requirements documents and we manually verified the tool's outputs for artifact extraction, polishing and classification. To mitigate this, we reviewed all manual tasks twice to avoid errors. However, having two or more people with relevant knowledge and experience participating in the validation would potentially further improve its reliability.

6.6 Discussion

In this section, we first discuss the benefits and limitation of our approach to extracting goal and use case integrated models from text.

Reduce effort for goal-use cases modeling: with the focus on automating the extraction of goal-use case models from text, our approach can help requirements engineers reduce the effort and time to model goals and use cases from requirements documents, and thus, can potentially be used to quickly gain understanding of natural language requirements documents. Moreover, with a clearly defined syntax, users can add new rules to improve the quality of extraction. In addition, once created, extraction and modifying rules can be used across different projects. Furthermore, since GUEST's underlying techniques for natural language parsing and artifact classification provide support for multiple languages, it is possible to adopt and apply our approach for requirements written in other languages. In this case, a new set of extraction rules that suits the grammars of each such language needs to be developed.

Possible application of our technique in other areas: Our techniques of automated extraction, polishing and classification can be used for any textual sentences. While our set of extraction rules was developed specifically for goals and use cases, its underlining concept can still be applied to support the information extraction in other areas. For instance, extraction rules can be developed in a similar way (i.e., develop rule actions and algorithms to execute these actions) to identify privacy or security policies [174]

from software documents, or development task from software documents [158]. Our rule-based technique can also provide a new approach in ontology learning [17, 24]. Specifically, similar rules can be created to detect ontological concepts, properties and their relationships to extract ontologies from natural language texts. Moreover, our polishing technique using modifying rules can also be applied in the field of automated boilerplate conformance (i.e., [10]) where a text must be ensured to conform to a certain specification rule.

GUEST's extraction accuracy depends on the quality of the Stanford parser and the artifact classifier: a common issue for a statistical machine learning technique is that it may not produce correct results for what it has not been trained for. Therefore, it is possible to have a sentence incorrectly parsed or a specification incorrectly classified in GUEST. Such problems can be resolved by training the parser and classifier with relevant data. GUEST provides the incremental training of both the parser and classifier.

Understanding of grammatical dependency required for rules writing: In GUEST, the extraction and rewriting rules need to be manually written. This requires the rule writers to have knowledge of grammatical dependency and thus some training would be required for end users to be able to extend the rules repository. We plan to overcome this problem by developing an algorithm that semi-automates the generation of a rule from the associations between sentences and lists of desired information to be extracted from them. In addition, a visual rule editor would be developed.

Unidentifiable artifact relationships: a number of relationships between artifacts in different sentences are not detectable since there exists no tool that can automatically detect coreference based on verb phrases in different sentences. In future work, we plan study the possibility of extending the identification techniques for pronoun-based coreference to apply into verb phrase-based coreference problems.

Lack of evaluation of the approach's usefulness: although having promising results in our case study-based evaluation, GUEST has not been validated for a real software project. We thus plan to carry out an evaluation with our industry partners to evaluate the approach's usefulness in requirements engineering.

6.7 Chapter Summary

In this chapter, we presented our techniques for polishing artifact specifications. Such techniques are aimed at ensuring artifact specifications to be grammatically correct and conform to our specification boilerplates. We also discussed the Mallet classifier and how it is extended to support the classification of artifact specifications. GUEST, the tool developed to implement our entire goal-use case integrated model extraction approach, together with some usage examples was also presented. We also discussed our validation conducted to evaluate the performance of our approach in different scenario. The evaluation results are very promising. In two selected case studies, GUEST achieved 86% precision and 84% recall rates for goals and use cases extraction, 87% and 78% of precision and recall rates for relationships extraction. It also obtained 96% accuracy for the automated artifact specification polishing in these case studies. The evaluation showed that our artifact classifier entirely outperformed Mallet and was better than Casamayor's classifier with smaller training datasets. In Chapter 7, we discuss our ontology-based techniques for analyzing goal and use case integrated models.

Chapter 7

Ontology-based Goal-Use Case Model Analysis

In this chapter, we discuss our ontology-based techniques to analyze goal and use case integrated models. The goal of this analysis is to identify incompleteness, inconsistency and incorrectness problems in such models. We also aim to provide explanations for detected problems and suggestions for the resolution of such problems. These techniques are developed to address the requirement R3 and its sub-requirements (discussed in Chapter 3) regarding automated support for goal and use case model analysis. Section 7.1 provides some background about the use of ontology in our analysis approach. Section 7.2 discusses our analysis techniques. Section 7.3 presents GUITAR, the tool developed to implement our entire analysis approach, accompanied with some usage examples. Section 7.4 provides a discussion about the validation that we have conducted to evaluate the performance of GUITAR in analyzing goal and use case integrated models.

7.1 Overview

Figure 7-1 provides an overview of our analysis approach that was briefly discussed in Chapter 3. Our approach has a goal and use case integrated model as input. Such a model can either be extracted from textual requirements documents by GUEST or

manually generated by users. In the first step of our analysis process, the specifications of artifacts in the considered goal-use case model are automatically parameterized based on functional grammar (1). The details as to how functional grammar is used to parameterize artifact specifications were given in Chapter 4. These parameterized specifications are then automatically transformed into Manchester OWL Syntax (MOS) specifications (2) which then are used to identify 3Cs problems (inconsistency, incompleteness and incorrectness) in the model (3). In this step, to support the identification of semantic problems, we developed an ontology-based technique to incorporate domain specific knowledge and semantics into the analysis process. In step 4, the resolutions are generated based on the identified problems (4). In the last steps, the identified problems, together with explanations and resolutions are presented to requirements engineers (5). Based on this, they then need to manually make decisions on how to resolve the detected problems.

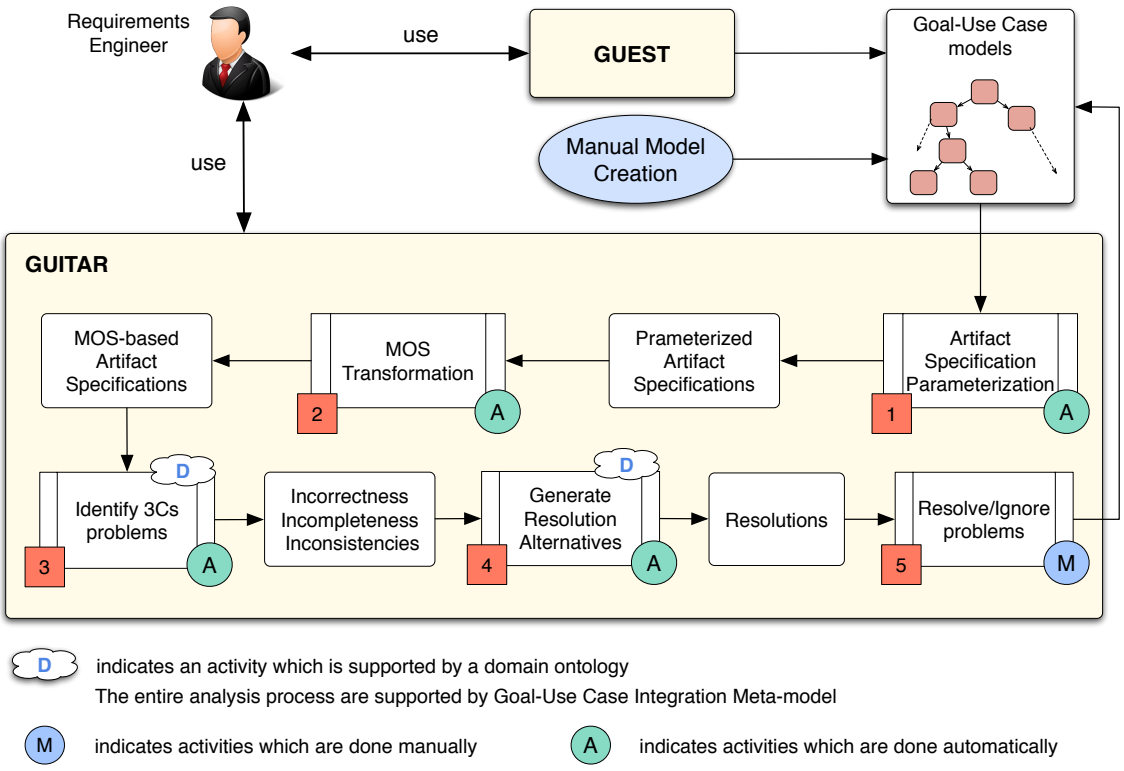


Figure 7-1. Analysis Process

We have developed GUITAR (**G**oal-**U**se Case **I**ntegration **T**ool for **A**nalysis of **R**equirements), a tool that implements our analysis approach. GUITAR automates almost all steps in the analysis process, except the last step in which the resolution decisions are up to users' preferences.

7.2 Ontology

To facilitate the automated analysis of goal-use case models, GUITAR augments artifact specifications with domain semantics and knowledge in the form of ontology. In this section, we provide some background about ontology and its use in our approach.

Ontology Definition and Representation

Ontology is defined as a specification of a conceptualization [56]. In the context of knowledge capturing and sharing, ontology is a description of concepts and their associated relationships within a domain. In our work, ontology is intended to capture the semantics of terms. Using functional grammar, an artifact specification can be automatically parameterized into a set of atomic ontological items whose semantics are known. This provides the basis upon which artifacts can be analyzed. In GUITAR, the existence of domain ontology is a prerequisite of semantic analysis. Figure 7-2 depicts the meta-model of ontology used in our approach. Some key concepts are as follows:

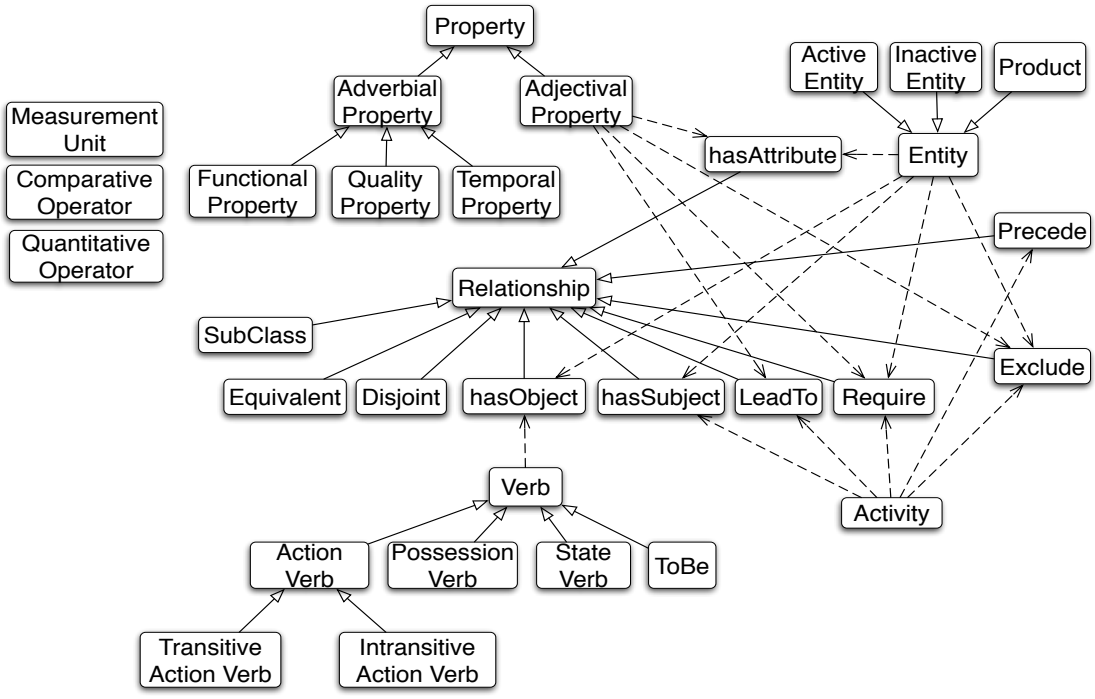


Figure 7-2. Ontology Structure

Verb refers to verbs in the domain that describe actions (e.g., *display*, *show*, *create*), possession (e.g., *have*, *contain*) or statuses (e.g., *to-be*, *arrive*, *come*). Action verbs are

further classified as *transitive action verbs* and *intransitive action verbs*. The former type requires an associated object while the later does not.

Entity refers to core elements in the domain. It has the following sub-types:

- ***Product entities*** refer to the software system/product being developed and its components (e.g., *system*, *product*).
- ***Active entities*** are the rest of the entities that can perform an action (e.g., *user*, *admin*, *librarian*).
- ***Inactive entities*** are entities that do not perform actions; they are instead objects of actions (e.g., *book*, *library*, *password*).

Activity refers to an activity in the domain (e.g., *write reviews*, *update account information*).

Measurement unit refers to measurement units in the domain. They can be classified further (e.g., data storage units like *MB*, or time units like *MHz*).

Comparative operator is used in comparison (e.g., *more than* or *less than*).

Quantitative operator describes cardinality (e.g., *all* or *only*).

Property includes *adjectival properties* (e.g., *high*, *low*) and *adverbial properties*. Adverbial properties are classified into *functional properties* (e.g., *automatically*, *manually*), *qualitative properties* (e.g., *quickly*, *safely*), and *temporal properties* (e.g., *firstly*, *secondly*, *before*).

Some basic *relationships* are used to capture the connections between concepts in a domain:

Equivalent specifies analogous concepts in a particular domain.

Subclass describes a refinement relationship between two concepts.

Disjoint captures non-overlapping pairs of concepts.

Other relationships are mainly used to capture domain knowledge:

hasObject relationship connects verbs with entities to specify the valid application of an action on an entity. For example, the verb '*write*' has a '*hasObject*' relationship with the '*content*' entity, that means '*write*' actions can only be applied to '*content*' entities.

Require/Exclude relationship specifies that a concept requires or contradict to another (i.e., '*banned users* are not allowed to *create reviews'* is captured by a an '*exclude'* relationship between the '*banned user*' entity and the '*create review'* activity).

LeadTo relationship is used between activities or between an activity and an adjectival property to specify that an activity can refine another activity or satisfy a property. For instance, *'Write email' leadTo 'Enable communication'* or *'validate identity' leadTo "secure."*

Precede relationship is used between activities to specify that an activity must be done before another activity can be conducted. For instance, 'create account' precedes 'write review.' The reversed relationship of *'precede'* is a sub-type of *'require'* relationship. For instance, in this case the 'write review' activity requires the 'create account' activity.

Ontology Representation Language

In our work, we chose to use *SROIQ* Description Logic [63] as the formal foundation for ontologies since it has well-defined semantics, known reasoning algorithms, is decidable[9] and importantly, satisfy all our needs regarding defining concepts and relationships in ontologies. In the following, we provide some background about *SROIQ*.

In *SROIQ* logic, the domain of interest is modeled by individuals, concepts and roles. Individuals are the instances that instantiate particular concepts, thus concepts can be viewed as representing unary properties of individuals, while roles consist of binary relations between concepts or individuals. In this section, we use $\mathtt{C}$ and $\mathtt{D}$ to denote concept expressions. A concept inclusion axiom is an expression of the form $\mathtt{C} \sqsubseteq \mathtt{D}$ (which means the concept $\mathtt{C}$ is a specialization of the concept of $\mathtt{D}$). For instance,

[9] Being decidable means that there exists an effective algorithm that always returns results when determining relationships between concepts (instead of looping indefinitely).

`RegisteredUser` ⊑ `User` means that the concept `RegisteredUser` is inclusive in the concept `User`. A *SROIQ TBox* is a set of general concept inclusion axioms. Thus, a TBox can capture a class hierarchy. On the other hand, an *ABox* is a finite set of concept expressions of the form `C(a)` (which means a is an instance of `C`). For instance, to express that `James` is a registered user, we write `RegisteredUser(James)` or, to express that `Mary` is a friend of `James`, we write `friendOf(James, Mary)`. Finally, all assertions concerning roles are gathered in an *RBox*. For instance, to indicate that the relationship `constrain` is the sub-type of the relationship `require`, we can state the following role inclusion axiom in an RBox: `constrain` ⊑ `require`. The *SROIQ* knowledge base (or ontology) is the union of the TBox, ABox and RBox. Table 7-1 illustrates the syntax of *SROIQ* logic.

Table 7-1. Semantics of Concept Constructors in *SROIQ*

Constructor	SROIQ Syntax	Semantics (in first-order logic)
intersectionOf	$C_1 \sqcap \ldots \sqcap C_n$	$C_1(x) \wedge \ldots \wedge C_n(x)$
unionOf	$C_1 \sqcup \ldots \sqcup C_n$	$C_1(x) \vee \ldots \vee C_n(x)$
complementOf	$\neg C$	$\neg C(x)$
inclusion	$C \sqsubseteq D$	$\forall x. C(x) \rightarrow D(x)$
allValuesFrom	$\forall R. C$	$\forall y. R(x,y) \rightarrow C(y)$
someValuesFrom	$\exists R. C$	$\exists y. R(x,y) \rightarrow C(y)$
maxCardinality	$\leq nR. C$	$\exists^{\leq n} y. R(x,y)$
minCardinality	$\geq nR. C$	$\exists^{\geq n} y. R(x,y)$

In our work, we adopted Manchester OWL Syntax (MOS) [62], an OWL2[10] language that is used to represent ontologies. The reasons for choosing MOS are threefold. Firstly, OWL2 language's semantics are compatible to SROIQ. Secondly, MOS provides a natural language-based and easy-to-read syntax for OWL2. Using MOS thus allows users a user-friendly way of reading and manipulating ontologies. Thirdly, although MOS is the primarily used OWL language syntax in our work, GUITAR can flexibly accept ontologies specified using other syntaxes since other OWL2 syntaxes

[10] http://www.w3.org/TR/owl2-overview/

(and older versions of OWL language, i.e., OWL1, OWL1.1) can be converted to MOS format.

Table 7-2. MOS Syntax

Constructor	SROIQ Syntax	MOS Syntax
intersectionOf	$C_1 \sqcap \ldots \sqcap C_n$	$C_1\ AND \ldots AND\ C_n$
unionOf	$C_1 \sqcup \ldots \sqcup C_n$	$C_1\ OR \ldots OR C_n$
complementOf	$\neg C$	$NOT\ C$
inclusion	$C \sqsubseteq D$	$C\ SubClassOf: D$
allValuesFrom	$\forall R.C$	$R\ ONLY\ C$
someValuesFrom	$\exists R.C$	$R\ SOME\ C$
maxCardinality	$\leq nR.C$	$R\ MAX\ C$
minCardinality	$\geq nR.C$	$R\ MIN\ C$

Table 7-3. Examples of Using MOS

Constructor	Example	Meaning
intersectionOf	`UserAccount` **`AND`** `hasAttribute SOME Locked`	A class of locked user accounts
unionOf	`Messaging EquivalentTo: InstantMessaging` **`OR`** `AsynchronousMessaging`	A type of messaging can be either instant or asynchronous
complementOf	`User SubClassOf:` **`NOT`** `Admin`	Users are not admins
inclusion	`ProfilePicture` **`SubClassOf:`** `Photo`	A profile picture is a photo
allValuesFrom	`User AND hasPhoto` **`ONLY`** `LandscapePhoto`	The class of users whose photos are all landscape photos
someValuesFrom	`User AND hasPhoto` **`SOME`** `LandscapePhoto`	The class of users who have at least one landscape photos
maxCardinality	`User AND hasPhoto` **`MAX`** `5 LandscapePhoto`	The class of users who have at most five landscape photos
minCardinality	`User AND hasPhoto` **`MIN`** `2 LandscapePhoto`	The class of users who have at least two landscape photos

In MOS, the term *'class'* corresponds to the term *'concept'* while *'property'* corresponds to *'role'* in *SROIQ*. A property in MOS can be an *object property* that describes a binary relationship between two classes, or a *data property* that describes a relationship between a class and a data value. Using MOS, all concepts (i.e., entity, verb, property) and their sub-concepts described in Figure 7-2 are created as classes while the relationships (except for subclass, equivalent and disjoint that are built-in relationships) are created as object properties. Table 7-2 presents MOS syntax in comparison to SROIQ's syntax. Table 7-3 presents some examples of expressing concepts and relationships using MOS.

Ontological Concept Naming Convention

To ensure the consistency in specifying the name of each concept in ontology, we use the following naming convention rules:

1 A concept name must be in its standard form. For example, *'display'* is used instead of *'displays'* or *'displayed.'*
2 A concept name must be in camel case (e.g., *'Display,' 'RegisteredUser'*).
3 A concept name must have a suffix that indicates the class it belongs to. For instance, an entity concept must have the suffix '_e' while a verb concept must have the suffix of '_v.' Table 7-4 presents the list of suffixes used in our naming convention.

Table 7-4. List of Ontology Class Suffixes

Ontology Class	Suffix
Entity	_e
Verb	_v
Activity	_act
Adjectival Property	_adjp
Quality Property	_qp
Functional Property	_fp
Temporal Property	_tp
Comparative Operator	_co
Measurement Unit	_mu

Incorporating Ontology with Functional Grammar

As discussed in Chapter 4, functional grammar is employed as a means to parameterize artifact textual specifications. Such parameterization enables the understanding of the semantic composition of each specification. Ontology further augments the semantic capturing by providing a way to track the meaning of each individual word used in a specification. This is done by linking each word to the corresponding ontological concept.

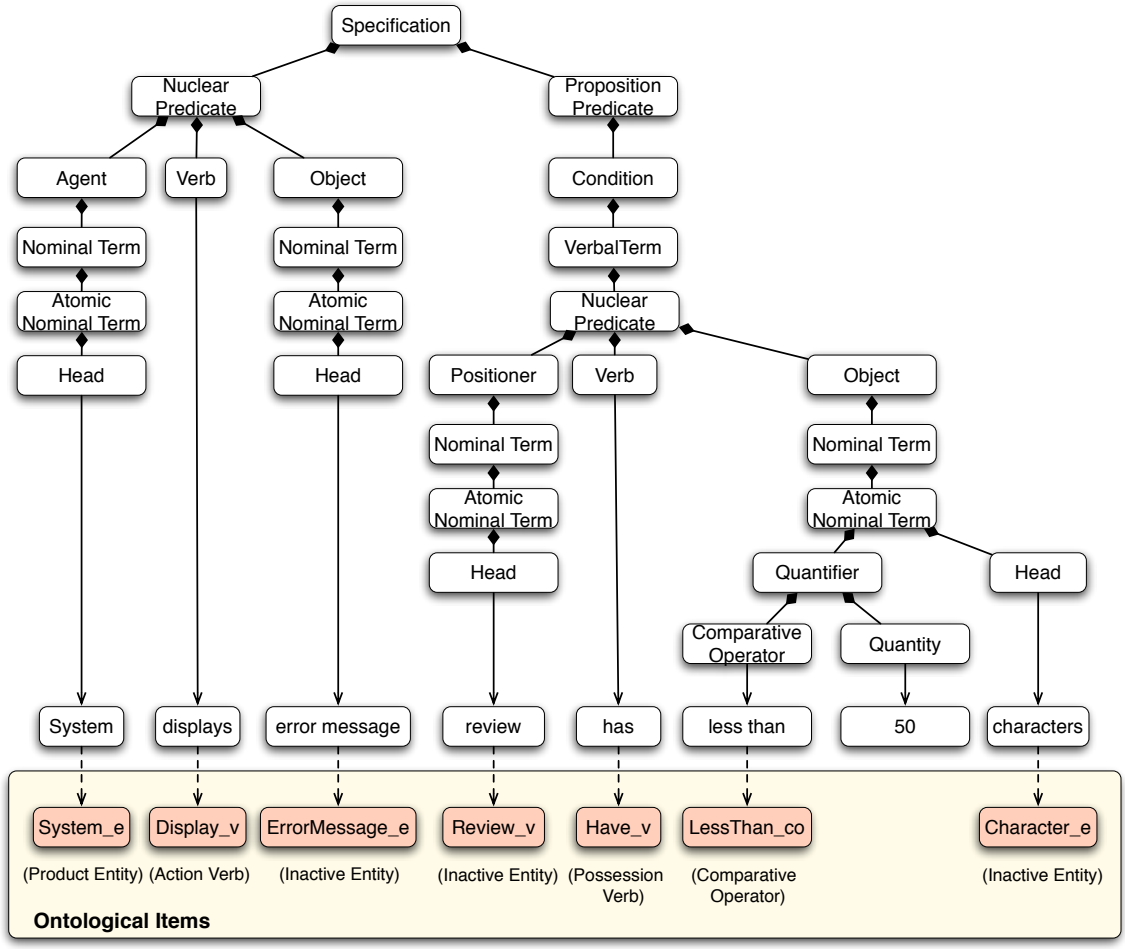


Figure 7-3. An Example of Linking an Artifact Specification to Ontological Items

Figure 7-3 shows an example in which a specification "*if the review has less than 50 characters, the system displays an error message*" is parameterized using functional grammar. Each individual word in this specification is linked to an ontological item (represented as a shaded boxes). For instance, the word *'System'* is linked to the product entity *'System_e'* in the ontology, or the word *'has'* is linked to the possession verb *'Have_v.'* The suffix '_e' or '_v' are used to indicate which class a concept belongs to.

This helps differentiate the use of word that can be both noun and verb (e.g., *review*, *request*). Note that in this example, extended predicates are omitted for simplicity.

How to build and maintain ontology?

GUITAR is able to work independently from domain ontologies. However, the availability of a domain ontology is essential to detect semantic problems that often require the understanding of domain specific knowledge and terminologies. Therefore, obtaining a domain ontology and verifying its quality is an important task in our approach. A detailed discussion regarding this matter is provided in Appendix A7.

7.3 Goal and Use Case Integrated Model Analysis

We discuss our techniques for detecting and resolving incompleteness, inconsistency and incorrectness (3Cs problems) in goal and use case integrated model. We first discuss the categorization of the 3Cs problems handled in our work. We then present the technique used to automate the parameterization of artifact specifications. It will be followed by discussions about our problem detection and resolution techniques.

7.3.1 3Cs Problem Classification

From the definition, categorization and rules of artifacts and relationships in GUIMeta, we define and classify the possible 3Cs problems that may exist in goal-use case models. Such classification provides a foundation on which artifacts in a goal-use case integration model can be analyzed for inconsistency, incompleteness and incorrectness.

In GUIMeta, problems in each 3Cs group (i.e., inconsistency, incompleteness) are categorized based on how they can be detected. Specifically, they are divided into *syntactic* and *semantic problem* classes. *Syntactic problems* are concerned about the conformance of a specification to GUIMeta's defined rules regarding artifact or relationship specifications (i.e., if an artifact specification violates a specification rule, then it is *incorrect*. If a functional service goal does not have any use case that operationalizes it, then it is *incomplete*). Therefore, syntactic problems can be detected based on rule-matching techniques. *Semantic problems* refer to the issues involving one or more specifications that can be only detected if the meanings of such specifications are known. In the following sub-sections, we provide details on the classification.

7.3.1.1 Incompleteness

7.3.1.1.1 Syntactic Incompleteness

Syntactically incomplete artifact specification: an artifact specification is considered syntactically incomplete if its parameterization does not contain all the compulsory semantic functions (specified by specification rules). For instance, the functional service goal "*Users shall be able to create*" is incomplete since there is a missing *object* of the action '*create.*'

Syntactically missing artifact: in our framework, artifacts must be refined from the business level until the use case level is reached. That means every artifact (except for use case internal components) in a model must be part of a tree (called a *refinement tree*) with the root is a business goal and the leaves are use cases (which operationalize functional service goals) or non-functional use case constraints (which refine non-functional service goals). Moreover, in a use case, some certain compulsory components must be specified. We categorize the *"missing artifact"* syntactic incompleteness as follows:

- ***Missing goal or constraint:*** refers to the cases in which a goal or constraint does not have its required parent-artifact and/or sub-artifact to form a *refinement tree*. For instance, a business goal does not have any sub-goal, a functional service goal is not operationalized by any use case or does not have a parent-goal, a non-functional use case constraint does not have a parent goal (which is a non-functional service goal).
- ***Missing use case component:*** refers to the cases in which a use case does not have all of its required components specified, including: pre-condition, post-condition, actor and use case step. A use case specification is also incomplete one of its extensions does not have any condition or starting step specified.

7.3.1.1.2 Semantic incompleteness

Semantically missing artifact: there are cases in which an artifact, that should be specified in the model, is not specified. There are many possible reasons for this situation. For instance, an important requirement may not be provided by the stakeholders of the system, or a requirement may be overlooked when requirements

engineers create the model. The missing requirement is possible about a function that the system should perform, a constraint that it should satisfy, a condition or a step in a use case. In order to identify such incompleteness, knowledge in the domain of interest is normally required. Based on our definition and classification of artifacts and relationships in the artifact layer, we categorize *"semantically missing artifacts"* as follows:

- ***Missing artifact with relationship:*** this category covers the cases in which an artifact that has a certain relationship with an existing artifact in the model has not been specified. The possible situations are: the missing artifact is required by an existing artifact, the missing artifact is a parent or sub artifact of an existing artifact, the missing artifact should provide a certain constraint on an existing one. For instance, the functional service goal FSG8 "*Users shall be able to edit reviews*" is specified while another goal "*users shall be able to write reviews*" has not. Assume that it is a general knowledge in this domain that if a content can be edited, it is required that such content has been created somewhere. In this case, there is an incompleteness which can be resolved by adding the missing goal and a require relationship between FSG8 and such goal.
- ***Missing use case components:*** this category is concerned with the missing internal components of a use case specification. The difference between this category and the *"syntactically missing use case components"* category is that, syntactic problems can only be detected if a certain component is completely missing in a use case (i.e., a use case has no precondition) while in semantic problems, a component is considered missing even if one or more components of its type have been specified (i.e., if a use case has a precondition, there may be another precondition that has not be defined). This type of semantic incompleteness contains the following sub-categories:
    - ***Missing use case step:*** includes the cases in which one or more steps is missing in a use case's main success scenario or extension.
    - ***Missing post-condition:*** one or more post-conditions of a use case are missing. Such a situation happens when a use case has more intended outcome than what have been specified in the existing post-conditions.

- ***Missing exceptional handling:*** Such a situation happens when the existing preconditions and extensions of a use case do not sufficiently exclude exceptional cases that may cause failure to that use case's main success scenario. For instance, consider the use case UC1 "a *user creates a review*." Assume that in this domain, banned users must not be allowed to create reviews. If there is no existing precondition or extension that handles this case, then the use case specification is incomplete. This can be resolved by adding either a precondition or an extension to deal with the exception.

Semantically missing relationship: refers to cases when a relationship is missing between two already specified artifacts while it needs to be specified. For instance, given FSG7 "*Users shall be able to edit reviews"* and FSG1 "*Users shall be able to write reviews.*" There should be a '*require'* relationship between FSG7 and FSG1 given the domain knowledge that if a content can be edited, it is required that such content has been created somewhere. In addition, if there is a use case UC3 with the description of "*a user edits a review*" and it is not linked to FSG7, then an *operationalize* relationship (UC3 operationalizes FSG7) is missing.

7.3.1.2 Incorrectness

7.3.1.2.1 Syntactic Incorrectness

Syntactic incorrect artifact specification: refers to cases in which a specification of an artifact contains one or more semantic functions that are not allowed according to the specification rule of that artifact's type. For instance, consider a use case step that specifies "*User enters article subject quickly.*" It is incorrect since the quality attribute ("*quickly*") should not be used in a use case step specification according to the specification rule for use case steps.

Syntactic incorrect relationship: refers to cases in which a relationship is defined between two artifacts while it should not be. Such situation happens when the declaration of the relationship violates our defined rules regarding relationships in the artifact layer. For instance, a use case is specified to *operationalize* a functional feature goal. This is incorrect according to our rules as functional feature goals are normally

very abstract and not operationalizable. Similarly, it is incorrect if a functional service goal is specified to *refine* a business goal.

7.3.1.2.2 Semantic Incorrectness

Semantic incorrect artifact specification: refers to cases when an individual specification does not have a proper meaning, or is not semantically suitable for its type. For example, it is incorrect if a goal is specified as "*Users shall be able to write hotels*," obviously because *'hotels'* cannot be *'written.'* In another example, consider a functional service goal is defined as *"Users shall be able to configure their settings."* This specification is correct syntactically since it does not contain any disallowed semantic functions. However, it is an incorrect specification considering its semantic. Since "configuring settings" is a very abstract activity and cannot be directly operationalized into a use case, the artifact should instead be defined as a functional feature goal and refined by more specific (functional service) goals, for instance, *"Users shall be able to change their profile privacy settings."*

Semantic incorrect use case structure: refers to cases when two or more components in a use case violate each other. Such situations happen when use case's rules (discussed in section 1.2.1.2) are broken. For instance, a use case has preconditions cover its post-conditions (that means the goal of the use has already been achieved before it is started), or a use case includes another use case while its post-conditions is a subset of that included use case's preconditions, or a use case has two extensions that start from the same step while having semantically equivalent conditions. Consider, as an example, a use case has a precondition "*User has been logged in*" and a post-condition "*Traveler has been signed in.*" This is invalid since the precondition is semantically equivalent to the post-condition, given that *user* is equivalent to *traveler* and *"logged in"* and *"signed in"* are synonyms in the domain of interest.

Semantic incorrect relationship: refers to cases when a relationship is created between two artifacts that should not have such relationship, considering their meanings. For instance, a *require relationship* is defined between two artifacts that do not *require* each other, or an extension is specified to start from a use case step when it is actually invoked from another step.

7.3.1.3 Inconsistency

7.3.1.3.1 Syntactic inconsistency

Inconsistent relationships: refers to cases when inconsistent relationships are defined between artifacts. For example, consider the goals: FSG1 "U*sers shall be able to create reviews*," NSG2 "*Ensure the reliability of reviews*," and FSG3 "*Admins shall be able to create reviews*." If it is specified that FSG2 *requires* NSG2, NSG2 *requires* FSG3, FSG3 *excludes* FSG1, there is an inconsistency because it can be inferred that FSG1 requires FSG3 while FSG3 excludes FSG1.

7.3.1.3.2 Semantic inconsistency

Goal-use case mismatched: refers to cases when a use case is defined to operationalize a goal while its defined system-user interactions are not relevant to achieving such goal. For instance, consider a goal "*Users shall be able to create travel articles*" which is operationalized by a use case that describe steps for a user to edit a review. This is an inconsistency since the goal and its associated use case are irrelevant.

Inconsistency involved two artifacts: refers to cases when two artifacts in a model have their meaning conflicting with each other. For instance, given the goals "*Users shall be able to write reviews* ***for tours***" and "*Travelers shall be able to create reviews* ***for only places***." They are inconsistent given *user* and *traveler* are equivalent while *tour* and *place* are disjoint concepts in the domain of interest (traveler social networking domain).

Inconsistency involved more than two artifacts: refers to cases when a set of three or more artifacts in a model have their meaning conflicting with each other, while individual pairs of artifacts in such set are not insistent. For instance, consider three goals: "*If a user account is locked, system sends an email notification to the user*" (FSG10), "*If a user account is locked, system sends a SMS notification to the user*" (FSG11) and "*If system send a user an email notification, it will not send any SMS notification to that user*" (FSG12). They are inconsistent because it can be deducted from FSG10 and FSG11 that both email and SMS notification will be sent in case an account is locked, which is conflicting with FSG12.

Table 7-5 summaries the discussed 3Cs problem categories.

Table 7-5. Summary of 3Cs Problem Categories

		Problem sub-type
Incompleteness	**Syntactic**	P1. Syntactically incomplete artifact specification
		P2. Syntactically missing artifact • P2.1. Missing goal or constraint • P2.2. Missing use case component
	Semantic	P3. Semantically missing artifact • P3.1. Missing artifact with relationship • P3.2. Missing use case component o P3.2.1. Missing use case step o P3.2.2. Missing post-condition o P3.2.3. Missing exceptional handling
		P4. Semantically missing relationship
Incorrectness	**Syntactic**	P5. Syntactic incorrect artifact specification
		P6. Syntactic incorrect relationship
	Semantic	P7. Semantic incorrect artifact specification
		P8. Semantic incorrect use case structure
		P9. Semantic incorrect relationship
Inconsistency	**Syntactic**	P10. Relationship inconsistency
	Semantic	P11. Goal-Use Case mismatched
		P12. Inconsistency involved two artifacts
		P13. Inconsistency involved more than two artifacts

7.3.2 Artifact Specification Parameterization

The functional grammar-based parameterization of artifact specification plays an important role in the detection of 3Cs problems. Firstly, since the specification rules are defined based on functional grammar, we need to have artifact specifications being parameterized in functional grammar to verify their syntactical correctness. Secondly, the functional grammar-based parameterization of artifacts enables the incorporation of ontology into artifact specifications, which allows for the semantic analysis of artifacts.

Artifact parameterization is carried out when analysis is needed for an entire goal and use case integrated model or an individual artifact (i.e., verify if an artifact is correctly specified). Therefore, parameterization is done after a model has been extracted by GUEST or when a new artifact is manually entered into a model by GUITAR. In case an artifact in the model was extracted using GUEST, its textual specification has been polished so that it has the following properties: being grammatically correct, written in active voice and written using an action (or to-be) verb (discussed in Chapter 6). In case an artifact is manually entered into the model using GUITAR, it would also be automatically verified and ensured to have such properties using the same polishing technique. Therefore, we assume the input into this parameterization steps are specifications that satisfy the above properties.

To parameterize an artifact specification, we need to identify the semantic role (i.e., agent, object, beneficiary) that each word or group of words in such a specification plays. In addition, to enable the semantic problem identification, we need to link each word in a parameterized specification with a concept in domain ontology. In the followings, we discuss these steps in details.

Identify semantic roles

To identify the semantic role of each word or group of words in such a specification, we rely on the dependencies between words in a specification, which can be obtained from a dependency parse tree. The example below illustrates how the parameterization is supported by a dependency parse tree.

Example 7-1: Assume that we need to parameterize the specification *"if the review has less than 50 characters, the system displays an error message."* Its dependency tree is presented in Figure 7-4. In order to parameterize this specification, we first consider the root of this dependency tree (which is `displays-12`). Since it is an action verb, we obtain the *verb* semantic function and can identify other semantic functions based on the dependencies between it and other nodes. For instance, the dependency `nsubj(displays-12, system-11)` indicates that the sub-tree rooted at `system-11` contains the value of an `agent` semantic function. In other words, *'system'* is the agent in this specification (although the entire phrase is *'the system,'* since determiners, i.e., *the*, *a*, *an*, are not semantically important in our work, they are always ignored).

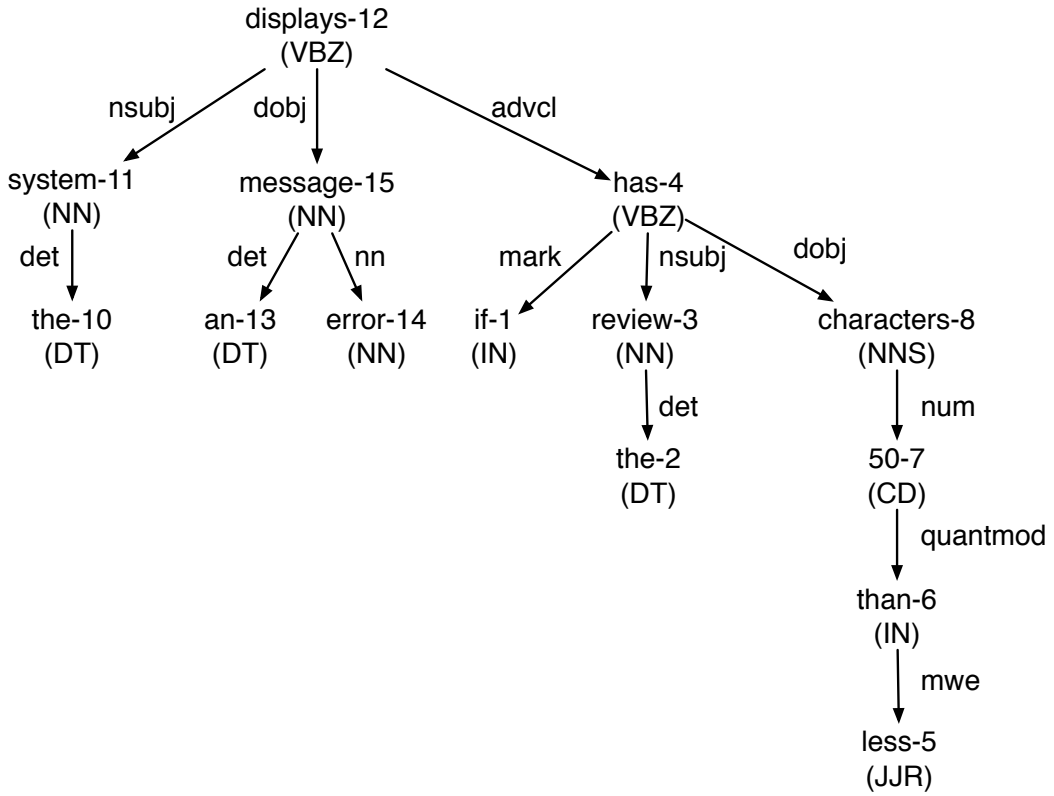


Figure 7-4. Dependency Tree for Example 7-1

Similarly, the dependency `dobj(displays-12, message-15)` indicates that *'error message'* is the object in this specification. In addition, according to our observation, the following dependencies always together indicate a condition specified in a sentence: `advcl(X, Y)` and `mark(Y, "if")` in which X is the root of the entire dependency tree and Y is a node in such a tree. If these two dependencies exist, then Y is the root of a tree that makes up the condition semantic function. Therefore, based on the existence of these dependencies in this example, *"review has less than 50 characters"* is the condition of this specification.

The parameterization of the specification at this stage is*: "Verb(displays) + Agent(system) + Object(error message) + Condition(review has less than 50 characters)."* We now consider the internal structure of the condition, which should be parameterized as a verbal term. Similar to the above discussion, since the root of this sub-tree is a possession verb (`has-4`) and it has the dependencies `nsubj(has-4, review-3)` and `dobj(has-4, characters-8)`, we can identify the positioner in this condition is *'review'* (the subject of a possession verb is a positioner instead an agent) and the object is *"less than 50 characters."* The internal structure of this object

semantic function is then also identified based on its internal dependencies. Firstly, since its root is a noun (`characters-8`), it should be parameterized as a nominal term. In addition, the `num(characters-8, 50-7)` dependency implies that this nominal term has a quantifier whose quantity is 50. The `quantmod(50-7, than-6)` and `mwe(than-6, less-5)` suggest that there is a comparative operator (less than) in this quantifier (`quantmod` is quantity modifier, and `mwe` is multiple word expression).

Moreover, the POS tag of the root node is used to determine the value of the tense semantic function. It is VBZ in this case. This means the tense is *'present.'* Furthermore, the value of the negation semantic function is determined by the existence of a neg dependency whose governor is the root node. Since there is no such a dependency in this case, the value of negation is *'false.'* Therefore, based on the investigation of the dependencies in this specification, we are able to parameterize it as `Agent(system) + Verb(displays) + Object(error message) + Condition(Positioner(review) + Verb(has) + Object(Head( characters) + Quantifier(Quantity(50) + Comparative_Operator( less than) + Tense(Present) + Negation(false)) + Tense(Present) + Negation(false).`

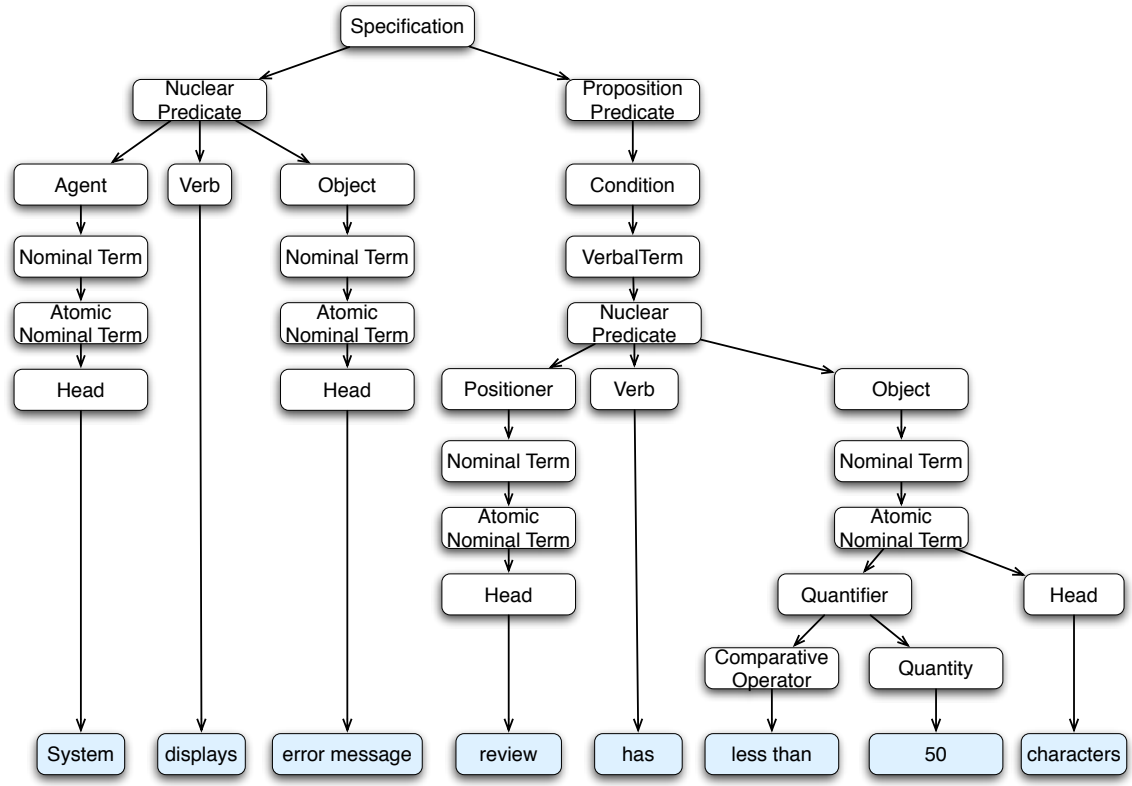


Figure 7-5. Tree View of a Parameterized Specification

A parameterized specification can be viewed as a tree in which each semantic function is a child node of a predicate that it belongs to (according to functional grammar). Each semantic function is then refined by the functional grammar term describing it while each term is refined by its internal components. The leaf nodes are the values of the components that cannot be refined further. In this session, we refer to such values as *parameterization atomic values*. Figure 7-5 presents the tree view of the discussed parameterized specification.

Table 7-6. Indicating Dependencies for Semantic Functions

Semantic Function	**Indicating Dependencies**
Verb	• When Y = X and Y is a verb • `cop(X, Y)` (when root is not a verb)
Agent	`nsubj(X, Y)` (when root is an action verb)
Positioner	`nsubj(X, Y)` (when root is not an action verb)
Object	`dobj(X, Y)`
Quality	When Y = X and Y is not a verb
Location	`prep_in(X, Y)`, or `prep_on(X, Y)`, or `prep_at(X, Y)`
Destination	`prep_to(X, Y)`
Source	`prep_from(X, Y)`
Beneficiary	• `prep_for(X, Y)`, or `prep_against(X, Y)` • Require semantic labeling rules
Company	Require semantic labeling rules
Purpose	Require semantic labeling rules
Reference	`prep_about(X, Y)` Require semantic labeling rules
Manner	`advmod(X, Y)` & Y's POS tag is RB and Y is not a time manner (i.e., monthly, daily)
Means	`prep_with`
Frequency	• `tmod(X, Y)` • `advmod(X, Y)` & Y's POS tag is RB and Y is a time manner
Duration	`prep_in(X, Y)` & `num(Y, Z)` & Y is a time unit (i.e., second, minute) & Z is a number
Condition	`advcl(X, Y)` & `mark(Y, Z)` & Z is "if"
Event	`advcl(X, Y)` & `advmod(Y, Z)` & Z is "when" or "as"
Tense	• If the root is a verb, if its POS tag is VBN then the tense is "past", otherwise it is "present" • If the root is not a verb, then there must be a dependency `cop(X, Y)`. If Y's POS tag is VBN, then the tense is "past", otherwise it is "present"
Negation	`neg(X, Y)`

Table 7-6 presents the association between each semantic function and the dependencies that indicate the existence of such a semantic function in a specification. In this table, X denotes the root of the currently considered tree (this tree can be a sub-tree of the entire dependency tree of a specification to be parameterized). In each row, the left column presents a semantic function that may exist in a specification represented by this tree. The right column presents the dependencies that can indicate the existence of such a semantic function in this tree. In this table, we use Y to denote the root of the sub-tree that contains the value of such a semantic function. Take the sub-tree representing the condition in Example 7-1 as an example. In this case, X is `has-4`, and `characters-8` (Y) is the root of the sub-tree that contains the value of the object semantic function of this condition.

Similarly, Table 7-7 shows the dependencies that can indicate the use of internal structure of functional grammar terms.

Table 7-7. Indicating Dependencies for Term's Internal Structure

Term's internal structure	Indicating Dependencies
Quantity	`num(X, Y)`
Comparative Operator	• `quantmod(X, Y)` & `mwe(Y, Z)` in which Y and Z make up the word "less than" or "more than" • `amod(X, Y)` & `prep(Y, Z)` & `conj_or(Y, W)` & `prep_to(W, T)` in which Y, Z, W, and T make up the word *"less than or equal to"* or *"more than or equal to"*
Attribute	`amod(X, Y)` and Y's POS tag is JJ, JJR, or JJS
Verbal Restrictor	`vmod(X, Y)`, or `vcmod(X, Y)`, or `amod(X, Y)`, or `advmod(X, Y)`, In which Y is a verb
Possessor	`poss(X, Y)`

Due to the complexity of English, to identify some semantic functions (i.e., company, means, purpose), considering only dependencies is not sufficient. For instance, consider the specification *"system notifies users of the changes"* (its textual dependency tree is given in Table 7-8), although `notifies-2` is the root and there exists the relationship `dobj(notifies-2, users-3)`, `users-3` is not the *object*, but instead the *beneficiary* in the specification. In addition, the same preposition dependency (i.e.,

prep_with, prep_of) may indicate different semantic function in different specifications. Consider the specification *"editors can login with their accounts."* Its dependency tree has the prep_with(login-3, accounts-6) dependency and *"their accounts"* makes up a *means* semantic function. However, in the specification *"editors can communicate with reviewers," 'reviewers'* has the *company* semantic role although the dependency prep_with(communicate-3, reviewers-5) exists.

Table 7-8. Dependency Tree

nsubj(notifies-2, system-1) root(ROOT-0, notifies-2) dobj(notifies-2, users-3)	det(changes-6, the-5) prep_about(users-3, changes-6)

Table 7-9. Semantic Labeling Rule Syntax

Syntax	Explanation
X-> <semantic function>	**Meaning:** if a dobj(X, Y) dependency exists, then the sub-tree rooted at Y makes up the value of the specified semantic function **Example:** Alert->BENEFICIARY
X(%)?, p-> <semantic function>	**Meaning:** if a prep_p(X, Y) dependency exists, then the sub-tree rooted at Y makes up the value of the specified semantic function. If '%' is used, that means a dobj(X, Z) needs to also exist. **Examples:** discuss, on->REFERENCE align%, to->REFERENCE
X%, p-> A1:<semantic function 1>, <semantic function 2>	**Meaning:** is a prep_p(X, Y) dependency exists, and a dobj(X, Z) dependency exists, then the sub-tree rooted at Z makes up the value of the semantic function 1, while the sub-tree rooted at Y makes up the value of the semantic function 2. **Example:** equip%, with->A1:BENEFICIARY, OBJECT
Notes: X refers to a verb or an adjective p refers to a preposition (%)? means % is optional A1 means argument 1 (in the field of semantic parsing), it refers to object.	

To overcome this problem, we developed a set of *semantic labeling rules* based on an investigation on the common English verb, adjective + preposition combinations. In Table 7-6, the semantic functions that may need to be identified by semantic labeling rules are associated to the text *"require semantic labeling rules"*). The syntax of semantic labeling rules is given in Table 7-9. Currently, we have developed 168 semantic labeling rules based on the commonly used verb-preposition combination. This set of rules can be extended using this syntax.

Table 7-10 presents the labeling rules for the discussed examples. The first rule means that if a `prep_with(communicate, Y)` dependency exists, then the Y sub-tree has the *means* semantic role. From this it can be determined that *'reviewers'* is the value of the company semantic function in the specification *"editors can communicate with reviewers."* The second rule means that if `prep_of(notify, Y)` and `dobj(notify, Z)` exist, then sub-tree rooted at Y is the *reference* semantic function and the sub-tree rooted at Z is the *beneficiary* semantic function. From this it can be seen that *'users'* is the beneficiary and *'the changes'* is the reference semantic function in the specification *"system notifies users of the changes."*

Table 7-10. Sample Semantic Labeling Rules

```
1. communicate, with->COMPANY
2. notify%, of-> A1:BENEFICIARY, REFERENCE
```

Link to ontology

In order to enable the detection of semantic problems, each word in a parameterized specification is linked to a concept in a domain ontology. To do that, we transform each parameterization atomic value to the required format for ontological items (i.e., values must be in their standard form, camel format and followed by suffixes that indicate their ontological classes). For instance, the value *'error message'* would become *'ErrorMessage_e.'* This is because *'error message'* is a composite noun (determined by the POS tag of *'error'* and *'message'* and the `nn` – noun compound modifier dependency between them) and thus it is classified as an entity.

In similar examples, *'displays'* is transformed to *'Display_v,' 'less than'* is transformed to *'LessThan_co,'* and *'reviews'* is transformed to *'Review_e.'* In case these new words

are concepts in ontology, then they would be used by GUITAR to identify semantic problems. In case one or more of these words do not exist in ontology, GUITAR requests users to consider adding them properly into ontology. GUITAR can still work if a specification has a word that does not exist in ontology. However, in this case its meaning would be ignored during the semantic analysis.

7.3.3 3Cs Problem Detection

We discuss the main techniques used for identifying the problems of incompleteness, incorrectness and inconsistency defined in section 7.2. The techniques for detecting syntactic problems are generally different from those for semantic problems. In the following sub-sections, we present these two sets of techniques in details.

7.3.3.1 Syntactic Problems Detection

The techniques for syntactic problem detection focus on checking the conformance of goal-use case model specifications to the rules defined in GUIMeta.

T1: Meta-model Matching Technique. This technique is used to identify syntactical incompleteness and incorrectness (problems P1, P2, P5 and P6 in Table 7-5). The key idea is to use GUIMeta (discussed in Chapter 4) to validate an artifact specification or relationship. Specifically, the definitions and dependencies of artifacts and relationships specified in the artifact layer are used to validate artifacts, relationships and identify missing artifacts. The specification rules defined in the specification layer are used in conjunction with the functional grammar-based specification parameterization to identify missing or invalid parts in a specification. The following examples illustrate the use of this technique in detecting problems. Table 7-11 summaries how this technique to identify problems.

Example 7-2: If a functional feature goal is not refined by any functional service goal, then it is incompleteness since according to our rules, functional feature goals are on high-levels and must be further refined by service goals.

Example 7-3: Given a goal specification "*Users shall be able to create*." It is incomplete since an object of the action '*create*' is missing. The parameterization of this specification is `"Agent(users) + Verb(create)."` Since our rules specify that an

action verb needs to be associated to an *object*, this incompleteness can be identified (because there is no object semantic function in the parameterization).

Example 7-4: A use case step specification "*User enters article subject quickly*" is incorrect, as the quality attribute ('*quickly*') should not be used in a use case step specification according to our specification rules.

Example 7-5: Given a use case operationalizes a functional feature goal. This is incorrect because functional feature goals are normally abstract and cannot be operationalizable.

Table 7-11. Problem Detection Rules Used in Meta-Model Matching Technique

<table>
<tr><th>'Invalid Artifact' Detection</th><th>'Invalid Relationship' Rule Detection</th></tr>
<tr><td>specifiedBy(a, sfs),
elementOf(sf, sfs),
¬ allow(Type(a), sf)
=> invalid(a)</td><td>link(a1, a2, rel),
¬ allow(type(a1), type(a2),rel)
=> invalid(rel)</td></tr>
<tr><td rowspan="2">specifiedBy(a, sfs),
elementOf(sf1, sfs),
elementOf(sf2, sfs),
incompatible(sf1, sf2)
=> invalid(a)</td><td>'Missing Artifact' Detection</td></tr>
<tr><td>require(type(a), art_type, rel),
¬ exist(a, art_type, rel)
=> missing(a, art_type, rel)</td></tr>
<tr><td colspan="2">a, a1, a2 are artifacts. rel is a relationship
specifiedBy(x, y): x is a list of semantic functions to specify artifact y
allow(x, y): semantic function y is allowed to specify an artifact of type x
incompatible(x, y): semantic functions x and y are not allowed to be concurrently used to specify an artifact
link(x, y, z): artifacts x and y are connected by relationship z
allow(x, y, z): relationship z is allowed to connect an artifact of type x and an artifact of type x according to meta-model
exist(x, y, z): there exists an artifact of type y that has a z relationship with artifact x
missing(x, y, z): there is a missing artifact of type y to be connected to the artifact x by the relationship z</td></tr>
</table>

T2: Rule-based Inference. This technique is used to identify inconsistent relationships specified between artifacts (problem P10 in Table 7-5). It can also infer new relationships based on the existing ones. The key idea is to use Semantic Web Rule Language (SWRL) [64] to capture relationships and their rules (these rules have been

described in Chapter 4). Based on that, new relationships can be automatically deduced using Pellet reasoner [150].

For example, given the following goal specifications: G1 "U*sers create reviews*," G2 "*Ensure the reliability of reviews*," and G3 "*Admins create reviews*" and the relationships: G1 *requires* G2, G2 *requires* G3, G3 *excludes* G1. These relationships can be represented in SWRL as *require(G1, G2), require(G2, G3) and exclude(G3, G1)*. Using this technique, the *require(G1, G3)* relationship can be inferred. However, since '*require'* and '*exclude'* relationships cannot be defined between the same pair of artifacts, the inconsistency is identified.

7.3.3.2 Semantic Problems Detection

The detection of semantic problems is supported by the use of ontology, which provides domain-specific semantic and knowledge. In order to automate such detection, we use Pellet reasoner that is able to infer the relationships (including inconsistency, disjointness, sub-class, and instance-of) between ontological concepts and provide explanations for the inferred relationships. Since Pellet can only reason about concepts in a single ontology specified in an OWL language, there must be a way to represent artifact specifications as concepts using an OWL language and incorporate such concepts into ontology to do analysis. In our work, we adopted Manchester OWL Syntax (MOS) as a means to incorporate artifact specifications and ontology. In the following sub-sections, we present how artifact specifications can be represented with MOS before discuss in details the semantic analysis techniques.

Represent Artifact Specifications with MOS

The representation of artifact specifications with MOS is based on the parameterization of such specifications. The following examples illustrate how MOS presentations are derived from parameterized specifications.

Example 7-6: Given the specification *"Only registered users shall be able to create reviews."* Its parameterization (with links to ontological concepts) is `"Agent (QuantitativeOperator(Only) + RegisteredUser_e) + Verb(Create_v) + Object(Review_e)."` Based on this, its MOS representation is `"Specification`

`AND hasAgent ONLY RegisteredUser_e AND hasVerb SOME Create_v AND hasObject SOME Review_e."`

Example 7-7: Given the specification *"System displays a prompt for amount to the user."* Its parameterization is `"Agent(System_e) + Verb(Display_v) + Object(Prompt_e + Qualifier(AdpositionalRestrictor(Purpose (Amount_e)))) + Beneficiary(User_e)."` Its MOS representation is `"Specification AND hasAgent SOME System_e AND hasVerb SOME Display_v AND hasObject SOME (Prompt AND hasPurpose SOME Amount_e) AND hasBeneficiary SOME User_e."`

Example 7-8: Given the specification *"Users who do not have technical background shall be able to create reviews easily."* It parameterization is `"Agent(User_e + Qualifier(VerbalRestrictor(Verb(Have_v) + Object(Background_e + Attribute(Technical_adjp)) + Negation(true)))) + Verb(Create_v) + Object(Review_e) + Manner(Easily_qp)."` Its MOS representation is `"Specification AND hasAgent SOME (User_e AND hasVerbalRestrictor SOME (VerbalRestrictor AND hasVerb SOME Have_v AND hasObject SOME (Background AND hasAttribute SOME Technical_adjp))) AND hasVerb SOME Create_v AND hasObject SOME Review_e AND hasManner SOME Easily_qp."`

As can be seen from the above examples, each artifact specification is represented in MOS by a sub-class of the class `Specification`. This sub-class is the intersection of `Specification` and a number of classes created based on the semantic functions exist in this specification's parameterization. For instance, if the parameterization of the specification has an object semantic function, then we add into such intersection a class specified as `hasObject` followed by the MOS specification of the internal structure of the object.

However, as shown in the discussion below about the semantic problem detection techniques, there is no case in which an entire MOS representation of an artifact specification is used in semantic analysis. Instead, only parts of it are used to compare to the corresponding parts in the MOS representation of another artifact. For instance, the MOS representation of an object of an artifact is compared to that of an object of another artifact to identify problems.

Semantic Problem Detection Techniques

In this section, we present a high-level discussion on our techniques for detecting semantic 3Cs problems. Details regarding the algorithms behind these techniques are provided in Appendix A8.

T3: Relationship Checking/Inferring. This technique is used to identify semantic incompleteness or incorrectness (problems P3, P4 and P7 in Table 7-5). Its key idea is to extract an activity from a parameterized specification and use ontologies to identify its relevant relationships. For example, consider the "*user creates a review*" use case and assume that although it is known in this domain that a *banned user* is not allowed to create reviews, there is no pre-condition or extension in this use case to handle this case. Using this technique such a problem can be identified by first extracting the activity in the use case's description (i.e., `Verb(Create_v) + Object(Head(Review_e))`), then transform it into a MOS description (i.e., `hasVerb SOME Create_v AND hasObject SOME Review_e`).

The next step is to check the ontology for an `exclude` relationship between that *activity* and an *entity* (i.e., `"create review" excludes "banned user"`). If the entity is a sub-class of the *agent* in the use case's description (i.e. *User*) and if there is no pre-condition or extension handling this exception, then there is incompleteness. Note that it does not need to be an explicit relationship between `"create review"` and `"banned user"` in the ontology. Such relationship can be inferred in by GUITAR based on the semantics of concepts in the ontology.

T4: Parameter Matching. This technique is used to detect the equivalence, inconsistency (problems P11 and P12 in Table 7-5) and overlap between two specifications. The key idea is to compare the corresponding parameters in the parameterizations of both specifications. For example, consider the goals G1 "*Users shall be able to write reviews for tours*" and G2 "*Travelers shall be able to create reviews for only places.*" The parameterization of G1 is `"Agent(User_e) + Verb(Write_v) + Object(Review_e + Object(Tour_e)) + Negation( true)"` and G2's parameterization is `"Agent(Traveler_e) + Verb(Create_v) + Object(Review_e + Object(QuantitativeOperator(Only) + Place_e )) + Negation(true)."` The inconsistency between these goals is detected by

matching the corresponding parameters in these parameterizations by their MOS representations (i.e., the agent or object of G1 is compared to the agent or object of G2, respectively). To determine if two goals are inconsistent, the following conditions must be met:

- **C1:** There are no parameters `X` and `Y` such that G1's parameterization has `X` and G2's parameterization does not have `X` while G2's parameterization has `Y` and G1's parameterization does not have `Y`.
- **C2:** If there are verb and object in both parameterizations, and if the verbs in G1 and G2 are different (not equal semantically), then the goals' activity must belong to the same ontological class (activity is a combination of a verb and an object).
- **C3:** There is only one pair of corresponding parameters of these goals are conflicting
- **C4:** For any parameter `X` that exists in both G1 and G2 parameterizations, G1's `X` value and G2's `X` value must be overlapped, which means G1's `X` value is equivalent to, or a sub-class, or a super-class of G2's `X` value.

It can be seen that G1 and G2 meets condition C1 since they have the same set of parameters (*agent*, *verb*, *object,* and *negation*). Although the verbs in G1 and G2 are different (i.e., *'write'* and *'create'*), they share the same activity (i.e., *'write review'* and *'create review'*). The overlap between these activities are identified based on their MOS representations `"hasVerb SOME Write_v AND hasObject SOME (Review_e AND hasObject SOME Tour_e)"` and `"hasVerb SOME Create_v AND hasObject SOME (Review_v AND hasObject ONLY Place_e)."` Assume that there are two equivalent classes `'WriteContent_act'` and `'CreateContent_act'` defined in the ontology as `"hasVerb SOME Write_v AND hasObject SOME Content_e"` and `"hasVerb SOME Create_v AND hasObject SOME Content_e"` respectively and the class `'Review_e'` is defined as a sub-class of `'Content_e.'` Then GUITAR is able to identify that G1 and G2 describe the same class of activities (that belong to the class of 'create content'). Therefore, the condition C2 is met.

The condition C3 and C4 are also satisfied since the agents of G1 (`'User_e'`) and G2 (`'Traveler_e'`) are equivalent (assume `'User_e'` and `'Traveler_e'` are equivalent concepts in this domain). In addition, the objects in G1 (`"Review_e AND hasObject SOME Tour_e"`) and G2 are conflicting since `'Tour_e'` and `'Place_e'` are disjoint concepts (assume this has been defined in the ontology).

In case all conditions, except for C3, are met, then it can be concluded that the specifications are overlapping with each other. If each pair of corresponding parameters in both specifications has equivalent values, then it can be concluded that the specifications are equivalent.

T5: Relationship Extraction. This technique is used to detect inconsistency between more than two artifacts that usually forms an inconsistent cyclic relationship between artifacts specifications (problem P13 in Table 7-5). This type of problems normally occurs between goals. For example, consider three goals: "*if a user account is locked, system sends an email notification to the user*" (G1), "*if a user account is locked, system sends a SMS notification to the user*" (G2) and "*if system send a user an email notification, it shall not send any SMS notification to that user*" (G3). They are inconsistent because it can be deducted from G1 and G2 that both email and SMS notification will be sent in case an account is locked, which is conflicting with G3.

This inconsistency can be detected by separating the main description and condition (or event) in a specification into different statements with `'require'` relationships. For instance, the first specification is converted into "*a user account is locked"* (G1.1) `requires` *"system sends an email notification to the user*" (G1.2). The second specification is converted into *"a user account is locked"* (G2.1) `requires` *"system sends a SMS notification to the user"* (G2.2). The third specification is converted into *"system sends a user an email notification"* (G3.1) `requires` *"system shall not send SMS notification to that user"* (G3.2). Doing so would transform this problem into the relationship inconsistency problem in which rule-based inference technique (T2) can be applied.

Using the technique T2, it is possible to detect that G1.1 is equivalent to G2.1, G1.2 is equivalent to G3.1 and G2.2 is opposite to G3.2. Based on these relationships, together

with the specified 'require' relationships, the inconsistency can be identified. Interested readers can find the algorithm to extract condition (or event) and main description of an artifact for the purpose of identifying cyclic inconsistent relationship in Appendix A8.

7.3.4 3Cs Problem Resolution

Incompleteness Resolutions

Incompleteness resolutions come directly from the reasons of the identified problems. For instance, if a specification is missing an object (i.e., *"Users shall be able to create"*), then "adding an object" (i.e., an object of the verb *'create'* in this example) is a resolution. The decision as to which object to be added belongs to users. If a relationship is missing between two artifacts, then its addition will be suggested.

Incorrectness Resolutions

Resolutions for incorrectness are mainly removal or change requests. For instance, if an invalid semantic function is used, the resolution is to remove it. If a use case has its post-condition equivalent to its pre-condition, then either the involved pre-condition or post-condition is requested to be changed.

Inconsistency Resolutions

Inconsistency resolutions are based on three strategies: *removal*, *change* and *restriction weakening*. The *Removal* strategy removes one of the conflicting artifacts. The *Change* strategy requires the modifications of the parts of specifications that cause inconsistency. The *Weakening* strategy is used when conflict arises from restrictions in one or more specifications. For instance, *G1: "Users shall be able to write reviews for tours"* and *G2: "Users shall be able to create reviews for* ***only*** *places."* This strategy suggests the removal of the '*only*' restriction in G2.

Resolution Ranking

A resolution of a problem may entail another problem to be created. Therefore, it is helpful to provide some effect analysis of each resolution presented to users. In this section, we define a *concrete resolution* is one that suggests specific modification to the model (i.e., add a specific missing relationship, remove an artifact, remove an 'only'

restriction in an artifact specification) rather than the one that requires user input (i.e., add an object into the specification without knowing which value the object has). For each concrete resolution presented to users, GUITAR analysis the effect of it by taking the following steps.

First, GUITAR generates a new model based on the modification indicated in such a resolution. Second, it runs a full analysis (including incompleteness, inconsistency and incorrectness detection) on the new model and identifies problems. Third, GUITAR identifies the new problems that occur due to the modification suggested in such a resolution (by comparing to the set of problems detected in the original version of the model). If there is a new problem occurring due to the resolution, then such a resolution is considered as an *unsafe resolution*. Otherwise it is a *safe resolution*. The resolutions identified for a single problem are ranked according to the number of problems generated if each resolution is executed. Therefore, safe resolutions are always ranked higher than unsafe resolutions. GUITAR also allows highlight unsafe resolutions and allow users to view the effect resulted (i.e., the problems generated) by running such resolutions.

7.4 GUITAR: Goal-Use Case Integration Tool for Analysis of Requirements

7.4.1 Tool Architecture

GUITAR was developed and integrated into GUEST to provide seamless support for the modeling and analysis of goal and use case integrated models. The entire tool is called GUITARiST, which is a combination of the word **GUITAR** and GUE**ST**. Figure 7-6 shows an overview of GUITAR's architecture. The tool contains three main inter-connected components: *artifact modeling module*, *knowledge module* and *analysis module*.

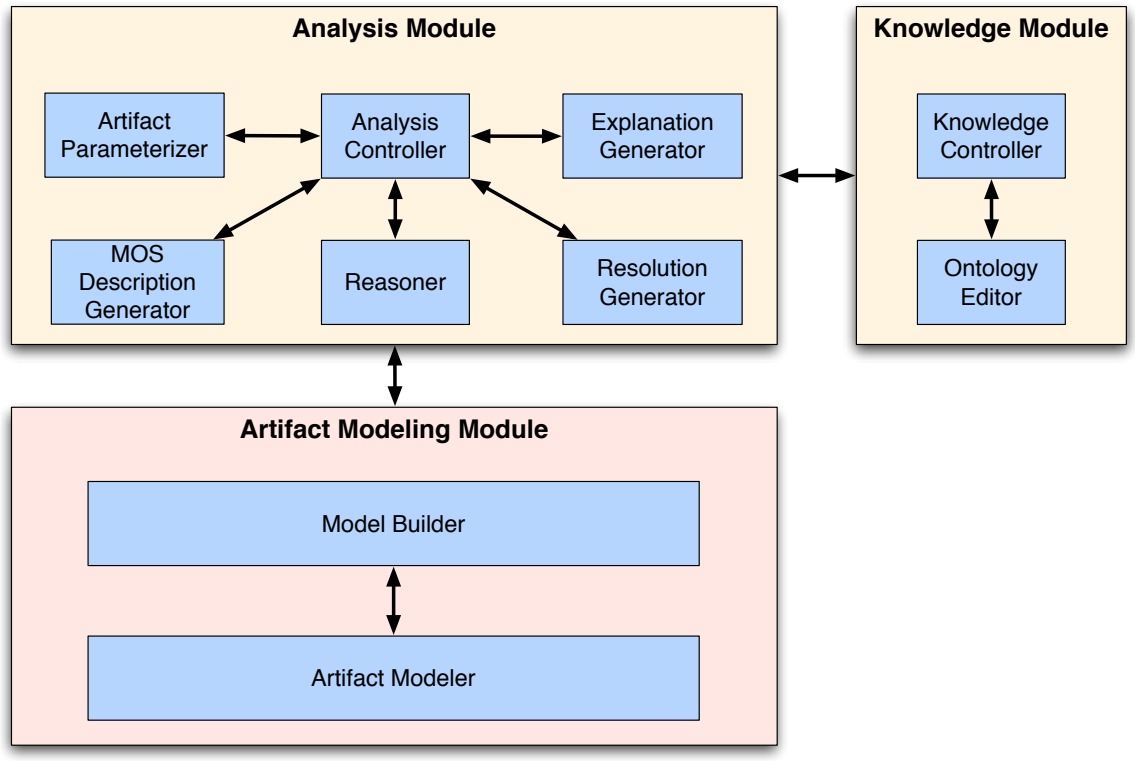


Figure 7-6. GUITAR's Architecture

The *artifact modeling module* is shared between GUEST and GUITAR. As discussed in Chapter 6, this module is responsible for creating and editing goal-use case models. The *knowledge module* contains an ontology editor that allows the edition and analysis of domain ontologies. Such edition and analysis are supported by OWL API [61] and the Pellet reasoner . The *knowledge controller* manages the retrieval and updates of domain ontologies during the analysis process.

GUITAR's *analysis module* is responsible for the detection and resolution of the incompleteness, incorrectness and inconsistency of artifacts. It contains five main components: *analysis controller*, *artifact parameterizer*, *MOS description generator*, *explanation generator* and *resolution generator*. The *analysis controller* manages the entire analysis process. It takes a goal-use case model as input and returns the detected problems with explanations and resolution alternatives. The *artifact parameterizer* is used to automatically generate the parameterizations of artifact specifications.

The *MOS description generator* produces MOS representations of parameters from parameterized specifications to support the semantic analysis. The *reasoner* is built on top of Pellet reasoner to facilitate the automated analysis of artifacts for 3Cs problems.

The *explanation generator* takes the analysis outcomes from the *reasoner* and produces explanations for detected problems. The *resolution generator* analyzes the explanations of problems and generates resolution alternatives.

7.4.2 Usage Example

In this section, we provide some examples to demonstrate the use of GUITAR in analyzing goal and use case integrated models for incompleteness, inconsistency and incorrectness. These examples are based on a scenario in the domain of traveler social networking system.

Ontology Update

Although GUITAR requires domain ontology to be available as a pre-requisite for the semantic problem detection, it does incorporate an ontology editor that allows users to quickly maintain and extend ontologies on the fly. Let us assume the requirements engineers have obtained ontology for the traveler social network domain. However, they now want to update it with an *equivalent* relationship between the activities '*create content*" and "*write content*." To do this, they invoke the ontology editor, browse to `CreateContent_act` activity and choose to add a new *equivalent class*. Figure 7-7(a) shows that "*write content*" is being defined. The description of this class is written in MOS as `"hasVerb SOME Write AND hasObject SOME Content."` GUITAR shows a light red background if the input is invalid or incomplete; otherwise it is turned to white (Figure 7-7(b)).

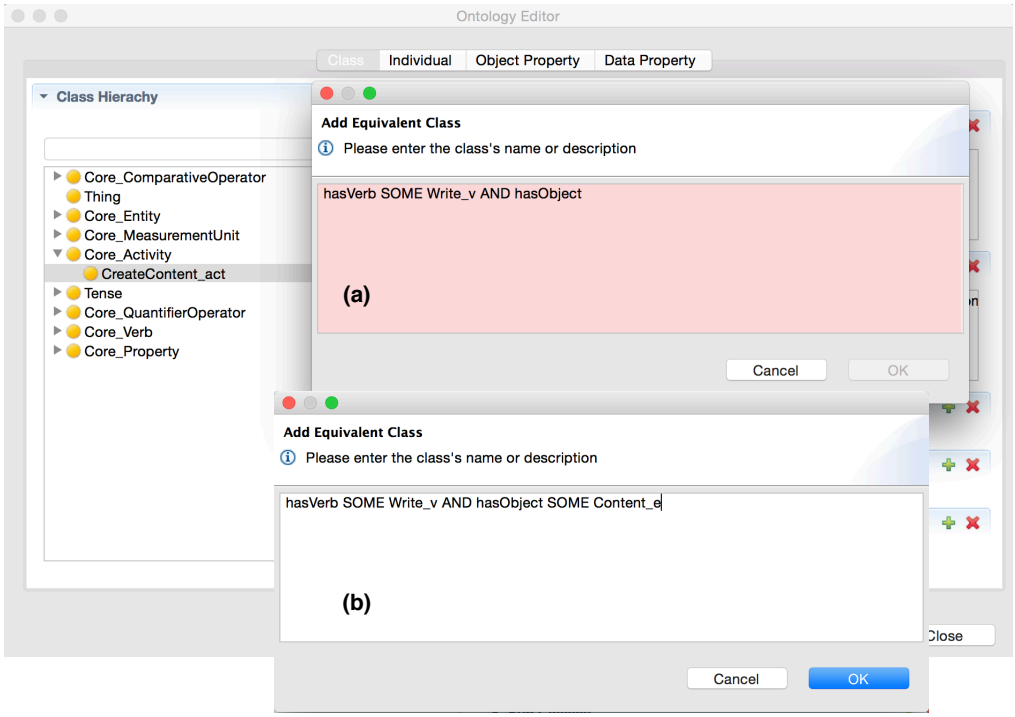


Figure 7-7. Updating the Domain Ontology with Ontology Editor

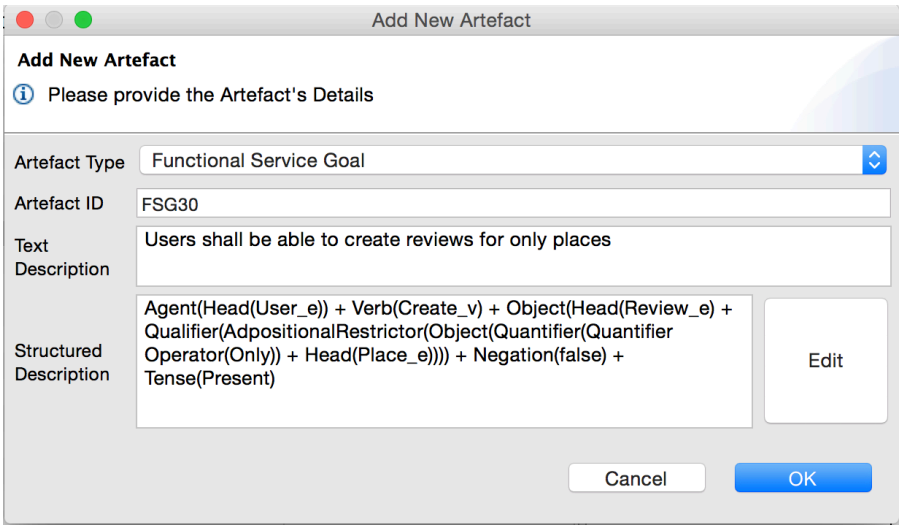


Figure 7-8. Adding an artifact

Adding Artifacts

GUITAR enables users to enter artifacts' specifications in natural language. To facilitate automated analysis. The parameterization of a specification (also called *structured description*) is optional. It can automatically be generated by GUITAR at this stage. However, users can postpone the parameterization until a semantic analysis is conducted in the model (GUITAR automatically parameterize each artifact specification at the beginning of the semantic analysis if its parameterization has not been generated).

Assume that the requirements engineers are adding a new functional service goal: "*FSG30 – users shall be able to create reviews for only places*." To do this, they open a *"new artifact"* dialog, selects to create a functional service goal. GUITAR then generates the goal's ID based on the previously used IDs. In case the requirements engineers want to parameterize this specification, they can click on the "structure description" text area and GUITAR would then generate the parameterization of this goal specification.

Figure 7-8 shows that the parameterized specification `"Agent(Head(User_e)) + Verb(Create_v) + Object(Head(Review_e) + Qualifier(Adpositional Restrictor(Object(Quantifier(QuantifierOperator(Only)) + Head( Place_e)))) + Negation(false) + Tense(Present)"` has been generated for the specification *"users shall be able to create reviews for only places."* GUITAR is also linked to the artifact polishing module in GUEST (discussed in Chapter 6) to ensure artifact specifications to conform to our specification boilerplates. For instance, if the specification "users shall be capable of creating reviews for only places." GUITAR would rewrite it as "users shall be able to create reviews for only places."

Inconsistency Detection

The requirements engineers now try to validate the set of artifacts for inconsistencies. They start the inconsistency validator. An inconsistency has been identified between *FSG30 – "users shall be able to create reviews for only places"* and *FSG6 – "users shall be able to write reviews for places and tours."* Figure 7-9(a) shows that the problem is described and the involved artifacts are highlighted in both artifact tree view and graphical view. The explanations (Figure 7-9(b)) show that they are inconsistent as '*create content*' is equivalent to '*write content*' while '*review*' is a sub-class of '*content*,' and '*place*' is disjoint with '*tour.*'

Inconsistency resolution

The requirements engineers now want to resolve the detected inconsistency. They click on the "*Quick Fix*" icon and GUITAR provides a dialog showing a number of possible resolution options (Figure 7-10). They include deleting either FSG30 or FSG6, remove the 'only' restriction in FSG30 (weakening strategy), make other modifications to either FSG30 or FSG6, or ignore the problem for now and come back later.

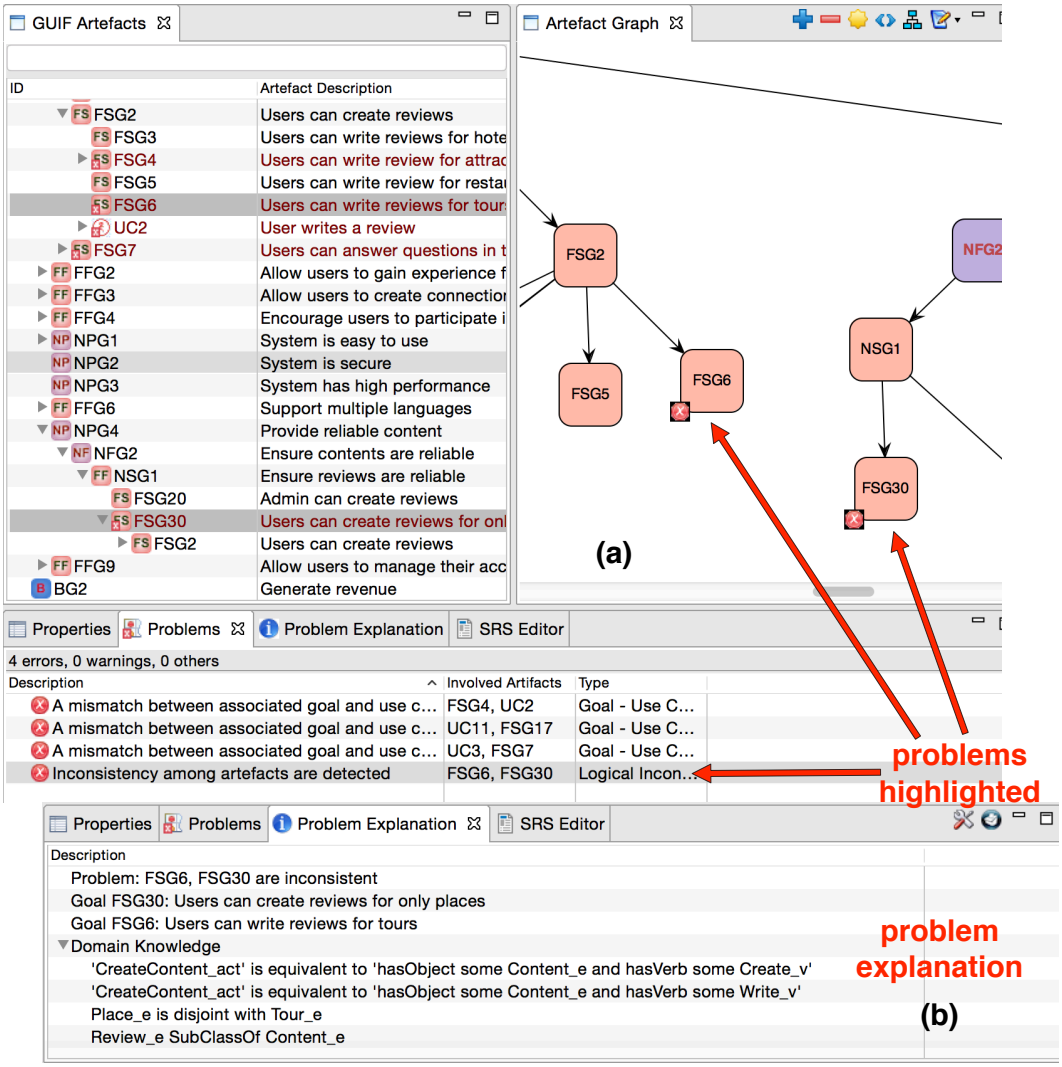


Figure 7-9. Inconsistency Detected

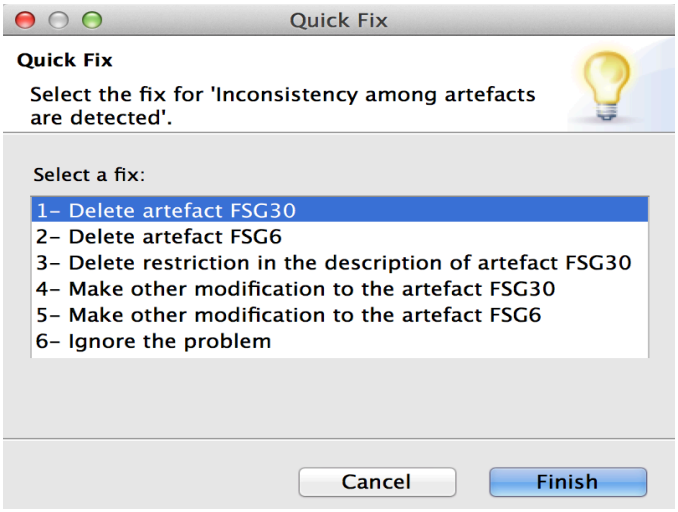


Figure 7-10. Inconsistency Resolution

7.5 Evaluation

In this section, we present some evaluations conducted to evaluate the effectiveness of GUITAR in automatically analyze goal and use case integrated models specified in natural language. The key question we seek to answer in this evaluation is:

> *"How accurately does GUITAR analyze natural language-based goal-use case models for 3Cs problems"*

Specifically, we aimed to address the following research questions:

- **RQ1:** How accurately does GUITAR parameterize natural language-based requirement specifications?
- **RQ2:** How accurately does GUITAR analyze goal-use case integrated models for inconsistency, incompleteness and incorrectness?
- **RQ3:** How is GUITAR's performance in comparison with other industrial and academic tools regarding analyzing textual requirements for inconsistency, incompleteness and incorrectness?

To address these research questions, we performed two types of evaluation. Firstly, for RQ1 and RQ2, we used a number of industrial and academic case studies and evaluated GUITAR's performance on these case studies. Secondly, for RQ3, we conducted a benchmark validation in which a number of comparable industrial and academic tools were selected and compared with GUITAR in regards to their analysis performance in a number of chosen case studies. In the following sub-sections, we discuss these evaluations in details. Our evaluation data and results can be found at http://goo.gl/gCUofM.

7.5.1 RQ1: Specification Parameterization Evaluation

We used the PROMISE requirements dataset introduced in Chapter 3 in this evaluation. The PROMISE data was pre-processed before being used in this validation to ensure they appropriately represented typical artifact specifications in goal-use case models. For instance, we split a requirement into multiple ones if it contains more than one sentence. In addition, combined words such as *'his/her,' 'himself/herself'* were changed to single words (i.e., *his*). Moreover, each requirement specification with a coordinating

conjunction (i.e., *'and'*) is split into two requirements. For instance, the requirement "readers can search and download articles" is split into two requirements which are *"readers can search articles"* and *"readers can download articles."* However, if the obtained requirements have identical structure (i.e., *"readers can search articles"* and *"readers can download articles"*), we only keep one of them to maintain the structural differences between requirements. In this evaluation, we randomly selected 200 requirements from all 15 PROMISE projects. To further diversify the set of sample requirements, we also included 110 requirements collected from different sources in the literature.

Each requirement was automatically parameterized by GUITAR and the results were manually checked by us to determine the accuracy rates. A parameterization result was considered accurate if: (1) a requirement was completely and correctly parameterized by GUITAR, or (2) part of a requirement was not possible to be parameterized since it was not supported by our meta-model (GUIMeta) and that GUITAR pointed this out correctly. For instance, consider the requirement *"The product shall continue to assign turns until the game is ended."* If GUITAR correctly parameterizes the phrase *"The product shall continue to assign turns"* while indicating that the phrase *"until the game is ended"* is not parameterizable, then this is correct since a temporal phrase such as *"until…"* is not supported in our work.

A two round-validation was conducted. Firstly, we parameterized the requirements based on our existing collection of modifying and semantic labeling rules to verify GUITAR's existing parameterization capability. We then identified the reason for errors found in the results. If the reason was incorrect parsing or missing supporting rules, we then trained the parser with correct parse trees or attempted to write new rules using our defined syntax. In the second round, we attempted to parameterize the requirements that GUITAR failed to parameterize in round 1 with the newly trained parser and/or new collection of rules. The result of this round indicated the best achievable capability of GUITAR in this validation.

Table 7-12. Parameterization Validation Results

Round 1 (existing capability)	274 over 310 (88%)
Round 2 (best achievable capability)	304 over 310 (98%)

Table 7-16 presents our evaluation results. We obtained 88% and 98% of parameterization accuracy in round 1 and 2 respectively. In the second round, there were a few number of requirements that GUITAR failed to parameterize since their grammatical structures confused GUITAR in determining the semantic functions. For instance, consider the requirement specification of *"The product shall determine when the road sections will freeze."* GUITAR mistakenly identified the phrase *"when the road sections will freeze"* as an event of the specification (while it was actually not) due to the existence of the following dependencies: `root(ROOT-0, determine-4)`, `advmod(freeze-10, when-5)`, `advcl(determine-4, freeze-10)`. In another example, GUITAR incorrectly labeled *"90 minutes"* as the value of a duration semantic function in the specification *"90% of new users shall be able to start the display of events within 90 minutes of using the product."* The correct parameterization should instead indicate that the phrase "*within 90 minutes of using the product"* is not supported by our meta-model. We plan to overcome these problems by extending the syntax of our semantic labeling rules to allow the specification of situation in which a phrase starting with *'when'* while having `advmod` and `advcl` dependencies should and should not be considered as an event semantic function. Similarly, this technique should also tackle the confusion over duration semantic functions in the second example.

Overall, the results from this evaluation were very promising. It implied that our parameterization technique can be used to well parameterize various types of requirements specifications. Especially, the very high accuracy rate obtained in the second round of the evaluation showed that GUITAR is able to correctly parameterize most requirement specifications given the linguistic parser is well trained and the modifying and semantic labeling rules are well developed. Since such parser training and rule development are supported by our tool, our parameterization quality can always be improved.

7.5.2 RQ2: Goal-Use Case Model Analysis Evaluation

This evaluation was aimed to assess the effectiveness of our goal-use case model analysis approach. We use precision rate and recall rate to measure the soundness and completeness of the analysis results. The precision metric is used to evaluate the approach's soundness. A higher precision means the approach returns more valid results

(true positive – TP) and less invalid results (false positive – FP). The recall metric is used to assess the completeness of the approach. A higher recall means the approach returns more valid results (true positive) and has less missed valid results (false negative – FN). The maximum of both precision and recall are 1 (equation 1, 2).

$$Precision = \frac{Valid\ Detected\ Problems\ (TP)}{Total\ Detected\ Problems\ (TP+FP)} \quad \text{Equation 1}$$

$$Recall = \frac{Valid\ Detected\ Problems\ (TP)}{Total\ Problems\ in\ \text{the artifacts}\ (TP+FN)} \quad \text{Equation 2}$$

We carried out two rounds of evaluation as follows.

Evaluation with PROMISE Data. We started with an evaluation with PROMISE projects' requirements. We have randomly three projects in this dataset. They offered requirements in the domains of Master Scenario Events List Management (MSEL), Real Estate (REs) and Nursing Training Program Administration (NTPA). These projects contain various types of requirements. For each project, we built an ontology for storing knowledge and semantics in its domain. The ontologies were built by our investigation in the relevant requirements and study of the domains. The ontologies were checked for consistency using GUITAR's embedded ontology editor tool before the evaluation. In this evaluation, if a requirement could be classified as a GUI-F artifact, it was then entered into GUITAR to build a goal-use case model. Since PROMISE data mainly contains a list of requirements without explicitly specifying their connections, artifact relationship validation was ignored in this experiment.

Table 7-13. Approach's Effectiveness Evaluation Results (with PROMISE Data)

Problem Types		MSEL	REs	MTPA	Precision	Recall
Incorrectness	D	6	5	20	100%	100%
	FP	0	0	0		
	FN	0	0	0		
Incompleteness	D	11	7	11	86%	83%
	FP	1	1	2		
	FN	2	1	2		
Inconsistency	D	0	0	0	-	-
	FP	0	0	0		
	FN	0	0	0		
D: Detected Problems FP: False Positive FN: False Negative						

The evaluation was done by first using GUITAR to identify the problems and the result was then manually verified by us. Table 7-17 shows the result of this evaluation. The main problems found in these case studies were incorrect goal specifications. For instance, *"The product shall synchronize with the office system every hour"* is defined as a non-functional goal instead of a functional goal. Both precision and recall for incorrectness detection in this evaluation are 100%. We achieved 86% and 83% for incompleteness evaluation's precision and recall respectively. That was due to GUITAR's suggestion about a missing non-functional goal of a specific type (i.e., *law compliance*) when it may not be needed. We noticed that it is not feasible to thoroughly verify the recall rate of incompleteness detection since it is not possible to find all potential missing requirements for a project. Our verification was done by determining missing requirements based on the domain knowledge that we collected and stored in the relevant ontologies. Although no inconsistency existed in the case studies, there was a positive result that GUITAR did not identify any "problem" which actually was not a problem (false positive is zero).

Evaluation with other industrial case studies. Due to the limitation of the range of problems existing in the PROMISE data; we carried out another round of evaluation with some industrial case studies. We employed three industrial case studies in the domains of traveler social network (TSN), online publication system (OPS) and split payment system (SPS) that were introduced in Chapter 3. In this round of evaluation, we first obtained a goal-use case model from each case study and manually analyzed it for 3Cs problems. However, since the number of problems was not significant, we took another step to manually seed incompleteness, inconsistencies and incorrectness into the models. Doing so enabled us to have the full control over the problems existing in the models and thus was able to determine if there was a problem that could be correctly identified by our tool (true positive), could not be detected by our tool (false negative), or an identified problem was not actually a problem (false positive).

The seeding process was done by first referencing example 3Cs problems from the literature, categorizing them into different sub-types, each sub-type was then classified into different difficulty levels (called *sub-type/difficulty level categories*) by investigating how sophisticated the relevant detection techniques are. For instance, under the *"use case specification incorrectness"* sub-type, *"invalid use case step*

specification" is categorized as a *simple* problem as most of them can be detected by meta-model matching technique (and some simple ontology inference), while problems of *"Different use case exceptions whose conditions are same/subsuming each other"* are considered *difficult* since the inference over multiple ontology concepts and relationships may be needed for identifying these problems (these types of problems were extracted from [29]). The last step of this process was to introduce similar problems into the case studies. For each type of 3Cs problems (i.e., incorrectness), we maintained the balance between the numbers of problems belonging to different sub-type/difficulty level categories. Specifically, 66 incompleteness, 32 incorrectness and 24 inconsistency problems were introduced across three case studies (there were 10, 8 and 8 sub-type/difficulty level categories defined under incompleteness, incorrectness and inconsistency respectively). When generating problems, we considered only those involving artifacts that can be specified with GUITAR (i.e., without temporal properties in their specifications).

Table 7-14. Approach's Effectiveness Evaluation Results (with Other Case Studies)

Problem Types		**TSN**	**OPS**	**SPS**	**Precision**	**Recall**
Incorrectness	D	10	12	7	100%	90%
	FP	0	0	0		
	FN	1	1	1		
Incompleteness	D	23	22	22	89%	90%
	FP	2	2	3		
	FN	2	2	2		
Inconsistency	D	6	8	5	100%	79%
	FP	0	0	0		
	FN	3	1	1		
D: Detected Problems FP: False Positive FN: False Negative						

Table 7-18 shows our evaluation results. GUITAR achieved 100% for precision and 90% for recall in incorrectness detection. The main kinds of incorrectness detected were mismatches between artifacts descriptions and their categories. GUITAR was unable to reveal some artifacts that were incorrectly placed on invalid levels of abstraction. For instance, in the context of a traveler social network system, if a *functional service goal* is defined as *"Users shall be able to configure their settings,"* then it has an incorrect

specification since *"configuring settings"* is a very abstract activity and cannot be directly operationalized into a use case. It instead should be defined as a functional feature goal and refined by more specific (functional service) goals, for instance, *"Users shall be able to configure their profile privacy settings."* GUITAR could not identify this problem since it was unable to recognize abstract activities from more specific ones. A possible way to overcome this issue is to incorporate our artifact classifier (introduced as part of GUEST in Chapter 6) to determine the abstraction level of artifacts.

We have achieved 89% and 90% for precision and recall respectively in incompleteness identification. That was due to our use of non-functional constraint categories in identifying missing non-functional artifacts. For instance, GUITAR matches our non-functional constraint categories against the set of modeled non-functional artifacts, if there is any category (i.e., *security*) which does not have corresponding modeled artifacts, then the tool indicates that there could be a missing non-functional artifact in such category (i.e. a security goal is missing). Although this technique is able to identify a number of missing non-functional artifacts, it was the reason why the precision and recall values decreased.

Firstly, not all non-functional constraint categories are applicable to a certain domain. For instance, the *"software interoperability"* category is not applicable to the traveler social network system used. This produces invalid detection result (reduced precision). Secondly, if there were already a security artifact of a certain class (i.e., security non-functional service goal) being modeled, GUITAR would not indicate that a security artifact of such class is missing (if it does not fall into other types of incompleteness we support). This may cause the situation that one or more security artifacts are missing but not detected (reduced in recall). We plan to overcome this issue by doing an investigation in categories of non-functional constraints that are normally used in a number of common domains. Doing so will help us to minimize false positives and thus increase the precision rate.

We have achieved 100% precision in inconsistency detection. However, the recall rate is 79%. This was because GUITAR currently does not support the identification of related words used in different forms. For instance, GUITAR was not able to detect an

inconsistency between two constraints "up*loadable images must be less than 4mb*" and "*users shall be able to upload images up to 5mb*." In this example, it could not find the relationships between the term "*uploadable images*" and the activity "*upload images*." We plan to overcome this issue by incorporating a natural language technique to identify the relationships between different forms of a word (i.e., *uploadable* and *upload*) and based on that infer the relationships between different specifications.

7.5.3 RQ3: Comparing GUITAR with Other Tools

We have conducted a benchmark validation with a number of selected requirements analysis approaches in order to further evaluate GUITAR's capabilities in analyzing 3Cs problems. In this validation, we performed two types of evaluations. The first type of evaluation was used for approaches that have tools available (for us to download and validate). In this evaluation, we compared them with GUITAR by using their tools to analyze requirements in our case studies. The second type was for approaches that do not have their tools available. There are 2 criteria for considering an approach for this type. Firstly, the approach must be knowledge-based (ontology-based or dictionary-based). The rationale for this limitation is that approaches without knowledge bases are generally disadvantaged when compared with GUITAR on the same set of requirements. Secondly, the data used to demonstrate or validate the approach must be available so that we could re-use to validate GUITAR. Such data include the ontology or knowledge base used, the original requirement set to analyze and the analysis results. In this type of evaluation, the aim is to verify which types of problems detectable in the benchmark approaches were also identifiable or unidentifiable with GUITAR.

7.5.3.1 Experiment Setup

The selection criteria for benchmark approaches were as follows:

- **Compulsory criteria**
  - (1) The approach is capable of automatically or semi-automatically analyzing textual requirements for at least one of the 3Cs problems.
  - (2) The approach's tool can be from either research or industry
  - (3) The approach's tool is available to download and installable on common operating systems (Windows, Mac OS and Linux), OR

The approach's tool is not available, however the approach is knowledge-based and the data used to demonstrate or validate the approach is available for re-use with GUITAR

- **Desirable criteria**
  - (4) The approach provides support for goal and/or use case modeling and analysis
  - (5) The approach employs domain ontologies for semantic analysis

The selection process resulted in seven approaches in which five with tools and two without tools. In the list of approaches below, the tool name is used if it was available; otherwise the name of the first author of the contribution is used.

Requirement Quality Analyzer (RQA): RQA[11] is an industrial tool developed by the REUSE Company. The tool uses a wide set of metrics to assess the quality of a requirement specification, mainly correctness, consistency and completeness. RQA is based on natural language processing, ontologies and semantic techniques to allow a comparison of the meaning of the requirements.

Requirements Assistant (RA): Requirements Assistant[12] is an industrial tool that analyzes requirements written in a natural language. It detects incompleteness, inconsistency, vagueness, testability issues, and ambiguity in a set of requirements.

Innoslate: Innoslate[13] is an industrial tool that supports the full lifecycle from requirements definition and management to operations and support. The tool provides an automated support to evaluate the clearness, completeness, design-orientation (i.e., whether a requirement describes design option) and verifiability.

Requirements-Driven Software Development System (ReDSeeDS): ReDSeeDS [118] is a research tool which offers a full Model-Driven Engineering lifecycle, from modeling to analyzing requirements and converting requirements into JAVA code. ReDSeeDS is the only tool in this evaluation that supports the modeling of both goals and use cases.

[11] http://www.reusecompany.com/requirements-quality-analyzer
[12] http://www.sirius-requirements.com/product/
[13] https://www.innoslate.com/

Requirements Processing Tool (ReProTool): ReProTool [41] is a research tool that allows users to bind requirements with the code of a developed application. Besides general requirements, the tool is capable of semi-automatically analyzing use cases written in natural language.

Kaiya: [71] developed a technique to verify completeness, correctness and consistency of textual requirements based on ontologies of knowledge and semantics. In this approach, terms used in each requirement need to be manually mapped into an ontological item. Based on the mapping and the relationships between ontological items, 3Cs problems can be identified. A tool was reportedly developed for this approach. However, it could not be obtained at the time this evaluation was done.

Dzung: [44] proposed a similar technique to Kaiya. However, it focuses on providing suggestion to improve the completeness of requirements based on domain ontologies. In this work, terms in a requirement are also manually linked to ontological terms. A set of rules was proposed to generate suggestions regarding new requirements. The tool from this approach was not obtainable.

We divided these benchmark approaches into 2 groups. The first one contains the 5 approaches with tools and the other 2 belong to the second group.

For the first group, we re-used the three industrial case studies used in the research question RQ2: TSN, OPS, and SPS. For each benchmark application, we import requirements from the case studies in the format that the application supports. For instance, relationships between requirements were only imported into ReDSeeDS since it was the only tool that supports requirements relationship modeling. In addition, use cases were only entered into ReDSeeDS and ReProTool since they were not supported by other tools. If a tool did not allow use case specifications, use case steps were entered as normal requirements. Moreover, in RQA that supported the use of ontologies, we entered our ontologies developed in the previous validation into the tool.

For each approach in the second group, we recreated an ontology in our format based on the ontology they provided. For each provided requirement, we determined the artifact type it should belong to (i.e., *"Users can retrieve books via the Internet"* was classified as a functional service goal) and entered it into GUITAR.

Table 7-15. Benchmark Validation Result for Group 1

			Detection result by each tool (True Positive/False Positive/False Negative)					
Case Study	**Problem Type**	**No. of problems**	**GUITAR**	**RQA**	**RA**	**Innoslate**	**ReDSeeDS**	**ReProTool**
TSN	**Incorrectness**	11	10/0/1	N/A	N/A	0/30/11	0/0/11	0/0/11
	Incompleteness	23	21/2/2	N/A	0/6/23	0/4/23	2/0/21	3/0/20
	Inconsistency	9	6/0/3	2/3267/7	0/4/9	N/A	N/A	N/A
	Others	N/A	N/A	Other quality problems	Testability & ambiguity problems	Ambiguity problems	N/A	N/A
OPS	**Incorrectness**	13	12/0/1	N/A	N/A	0/18/13	0/0/13	0/0/13
	Incompleteness	22	20/2/2	N/A	0/12/22	0/0/22	3/0/19	2/0/20
	Inconsistency	9	8/0/1	2/3010/7	0/2/9	N/A	N/A	N/A
	Others	N/A	N/A	Other quality problems	Testability problems	Ambiguity problems	N/A	N/A
SPS	**Incorrectness**	8	7/0/1	N/A	N/A	0/11/8	0/0/8	0/0/8
	Incompleteness	21	19/3/2	N/A	0/18/21	0/0/21	2/0/19	2/0/19
	Inconsistency	6	5/0/1	2/3537/4	0/3/6	N/A	N/A	N/A
	Others	N/A	N/A	Other quality problems	Testability & ambiguity problems	Ambiguity problems	N/A	N/A

7.5.3.2 Experiment Results

First-group: benchmark approaches with tools

Table 7-19 depicts the results achieved in this validation. In the following subsections, we discuss about the results obtained by each benchmark application in comparison with the results produced by GUITAR.

RQA

Inconsistency. Inconsistency in RQA is defined as the mismatch between measurement units used in requirements or the overlapping between requirements. The overlapping between two requirements is calculated based on the number of words with close or same meanings found in both requirements. This comparison is supported by the use of domain ontologies in the tool. The problem is that finding requirements that are overlapping is not sufficient to identify real inconsistency between them because having something in common does not necessarily means two requirements are contradict to each other. In this validation, RQA produces over 3000 overlapping pairs of requirements in each of the case studies. However, only a very small number of overlaps were inconsistencies, meaning that the number of false positives is very high. For instance, RQA identified an overlap between two requirements *"System shall be secure"* and *"System shall be easy to use."* However, they are not inconsistent with each other. In addition, there were many cases detected overlapping requirements do not describe a common feature or functionality. For example, *"members shall be able to disband a group"* and *"group members shall be able to simulate transfers of debt."* Furthermore, there are a number of inconsistencies that were detectable by GUITAR however remained unknown in RQA. An example of such cases is two requirements *"If a new bill is entered into the group, all group members will be notified by email," "If a new bill is entered into the group, all group members will be notified by SMS"* and *"If a member is notified by email, the member will not be notified by SMS."*

Incorrectness. RQA utilizes a set of metrics to evaluate the correctness of requirements. Those metrics cover a number of well-known requirement qualities such as ambiguity, measurability, verifiability and vagueness. In this validation, the requirements marked as having low quality by RQA were those contain the word/phrase *"shall," "shall be able to"* or *"all"* and those with abstract concepts such as *"easy to*

use," "quickly," "manage." However, since GUITAR is designed to deal with goals on different level of abstractions and goals are usually described in the format of *"shall be able to"* sentences, we accept these vagueness characteristics. Therefore, it was not applicable to compare GUITAR and RQA regarding incorrectness detection.

Incompleteness. RQA evaluates the completeness of requirements by matching a requirement description with its list of boilerplates. Since the boilerplates used in RQA are different from the boilerplates used in GUITAR (which are designed for goal and use case specifications), a comparison between the tools was not suitable.

Requirements Assistant (RA)

RA identified 4, 2 and 3 inconsistencies in TSN, OPS and SPS case studies respectively. However, they were all false positives. For instance, two requirements *"System shall store the list of articles"* and *"System shall store the list of reviewers"* were considered inconsistent by the tool why they obviously are not.

While having no support for correctness evaluation, RA validated the completeness of requirements based on a set of *"absolute"* keywords such as *"all," "any," "anything."* For instance, the requirement *"When a new bill is entered into the group,* ***all*** *group members will be notified by email"* was flagged as incomplete with a suggestion of reviewing for exceptions by the tool. We consider such cases are better classified as general warnings rather than actual completeness problem.

Innoslate

Innoslate supports the automated detection of completeness, clearness, design-orientation and verifiability problems. Design-orientation is a characteristic of a requirement regarding whether it describes how a certain functionality is implemented rather than what the system should support. Since our notion of correctness refers to the appropriateness of an artifact specification (goal or use case component) in the consideration of their type, design-orientation can be classified as a sub-type of correctness according to our definition. In this validation, Innoslate identified 30, 18 and 11 incorrectness (design-orientated) cases in the TSN, OPS and SPS case studies respectively. However, they were all false positives according to our judgment. For instance, *"Members shall be able to view their debts"* was flagged as design-oriented

statement while it is really a requirement describing a functionality of the system. Moreover, there were 4 incompleteness cases identified in the TSN case study. However, they were all false positives since the tool incorrectly indicated that the involved requirements contained no verb phrase while they actually had. No incompleteness was detected in the OPS and SPS case studies.

ReDSeeDS

ReDSeeDS enables the modeling of requirements and use cases together. Various types of relationships between these artifacts can be specified in the tool. In the validation, ReDSeeDS was able to identify ill-formed use case specifications. For instance, it revealed use cases with missing pre-conditions or a use case extension in which no *"resume"* step or *"failure"* step defined. ReDSeeDS, however, was unable to identify a majority of problems in the case studies. For instance, invalid relationships (i.e., *"operationalizing"* and *"conflict"* relationships defined between a pair of requirements) or a use case with the same pre-condition and post-condition.

ReProTool

ReProTool allows use cases to be modeled and analyzed in a semi-automatically way. Users are required to manually classify use case steps and based on that they can be analyzed by the tool. Due to the lack of semantic support (i.e., ontology or pre-defined language resources), ReProTool was not able to identify semantic problems. In this validation, only simple syntactic problems were identified by ReProTool. For instance, it was able to detect that a use case step *"the information"* is incomplete due to the missing of an actor, action and object. More complicated problem, i.e., a lack of an extension to handle the case when the article file does not exist in the use case step *"The editor attaches the article file to the form"* was not identifiable.

Second group: approaches without tools

Table 7-20 presents the results obtained for this group. For Kaiya and Dzung, the value in each cell depicts the number of problems of the corresponding types detectable by their approaches (according to their papers). For GUITAR, it shows the number of problems detected by our tool.

Table 7-16. Benchmark Validation Result for Group 2

Problem Type	Tools			
	Kaiya	GUITAR	Dzung	GUITAR
Incorrectness	2	0	0	0
Incompleteness	2	2	11	8
Inconsistency	1	1	0	0

GUITAR could identify all 3Cs problems by Kaiya except 2 incorrectness cases. In Kaiya, if there is a requirement containing at least a word which is not mapped to an ontological item, then it is considered incorrect. For example, the requirement *"User can play a music in any speed"* is incorrect since its words are not fully mapped to ontological items. This requirement, however, was not considered incorrect by GUITAR since it is not malformed by itself. In GUITAR, the mapping between words and ontological concepts is done automatically by through our automated support for artifact parameterization.

In the comparison between Dzung and GUITAR, our tool identified 8 over 11 incompleteness cases raised by Dzung. The other 3 cases were suggested by Dzung to enhance the requirement completeness because there exist the corresponding functions in the ontology. For example, a requirement about adding a user into the system is suggested since *"add user"* function exists in the ontology. GUITAR did not raise an attention regarding this function because there were no existing requirements that describe relevant functions to *"add user."* Since GUITAR was designed to provide the most precise information to users (the key is to minimize false positives), it generally does not consider a function defined in the ontology that has no corresponding requirement specified (we call it as a *"missing ontological function case"*) as incompleteness. The rationale is that a domain ontology may be very large and shared across projects in the same domain. A single function may be required in a project but may not be needed in another. Therefore, considering *"missing ontological function cases"* as incompleteness could overwhelm users with a large number of false positives. In our work, *"missing ontological function cases"* are only considered as possible incompleteness if there is an existing requirement that describes a closely related function to it. For example, if there exists a requirement specifying *"admins shall be able to remove users from the system,"* then GUITAR would report an incompleteness

concerning *"Add user."* In GUITAR, although *"missing ontological function cases"* are not considered incompleteness, it does allow users to view all functions captured in the ontology (referred to as *activities* in our terminologies) using GUITAR's *Ontology Editor*.

Thus, from the results of this validation, it was shown that GUITAR was able to identify at least all representative problems detectable by Kaiya and Dzung.

7.5.4 Threats to Validity

External Threats

Firstly, since all evaluations that we carried out involved the use of a number of case studies and/or requirement data, there is an external threat regarding the representativeness of such requirements. In the research question RQ1 regarding the parameterization of requirements specifications, the accuracy of GUITAR may vary depending on the requirements selected, and the quality of our linguistic parser and collection of modifying and semantic labeling rules.

To reduce this threat, we maintained a high variability of the data by selecting requirements from different sources and ensured some difference exist between requirements (i.e., we avoided having requirements with the same grammatical structure). In addition, apart from the evaluation with the existing parser and rule collection, we carried the second round of validation to evaluate the tool in case the parser and rules were perfect for the given set of requirements. This allowed us to gain the understanding about the standard performance of the tool (when the parser and rules were not specifically trained and developed for the experiment data) and its highest capability in the experiment settings (when the parser and rules are perfect for the experiment data). To reduce the representativeness threat in RQ2, we selected six case studies in different domains with different types of requirements to diversify the experiment data. Importantly, we also ensured problems in those case studies covered a wide range of 3Cs problems.

Secondly, in our benchmark validation, having good results with the selected approaches may not imply similar results with others. However, the best attempt has been made to get all relevant tools that were available to download and install at the

time the validation was carried out. In addition, to alleviate this threat, we have broadened our selection criteria to consider approaches with no available tool but having their demonstrative data available for use to validate GUITAR.

Internal Threats

Firstly, for RQ2, the types of requirements problems exist (for evaluation on PROMISE data) or being seeded in each case study may vary and affect the precision and recall rate of our tool. For instance, if there are more problems of the types the tool cannot detect and less problems of the types it can identify, then the precision will be reduced, and vice versa. To alleviate this, we tried to generate situations in which the problems are realistic and the balance between different types of problems is maintained. In the evaluation with seeded problems, we referenced example 3Cs problems in the literature to ensure the seeded errors were realistic. In addition, we also maintained the balanced numbers of errors across categories and difficulty levels. In future work, we plan to carry out another evaluation with industry partners' real requirements and errors.

Secondly, the quality of domain ontologies might affect the validity of our analysis support evaluation (RQ2). However, since this validation was designed to validate the soundness and completeness of our technique based on what it "knew" (what was stored in the ontology) and what it "did not know" (what was not stored in the ontology), the quality of ontology is less important. In fact, in this validation, we did not judge the tool as unsatisfied for not detecting a problem whose relevant semantic and knowledge are not known to it. Therefore, the cases that mattered were those in which GUITAR failed to identify a problem while it had sufficient knowledge to detect such problem.

Following this principle, we have minimized the effect of ontology quality by using an iterative process in which we ran the validations, then checked the results with the set of the generated problems, then verified if there were any problems that had not been identified because of missing concepts or relationships in the ontologies. If yes, then updated the ontologies with those missing parts and ran the validations again. If no, we recorded that our tool had failed to identify the problems and diagnosed the reasons. The result from the last iteration was the most important one because in that iteration we could eliminate all cases in which a problem was undetected when the tool had no

relevant knowledge about it. The documented result in the paper was the one after the last iteration.

Threats to internal validity also include the human factors in our evaluation. In RQ1, the parameterization results were manually verified to determine their correctness. In RQ2, the 3Cs problem detection outcomes were manually checked to determine their accuracy. In RQ3, the tasks of entering requirements from the case studies into the benchmark applications and verifying their analysis results were done manually. Therefore, there were chances that errors could be generated due to the manual effort. To reduce this threat, we carried out the tasks very carefully and reviewed every steps twice during the evaluation.

7.6 Discussion

We have presented our ontology-based technique to analyze goal and use case models for 3Cs problems. Our technique was built on top of our goal-use case integration meta-model (GUIMeta) introduced in Chapter 4, which provides a fundamental foundation to model, classify, parameterize and analyze artifacts in a goal and use case integrated model. The functional grammar-based parameterization allows the semantics of artifacts to be examined in a consistent way. Such parameterization of artifacts allows artifacts' textual specifications to be automatically transformed into formal descriptions in MOS that then promotes the automated analysis of the artifacts. Importantly, our technique is designed to encapsulate "intelligence" so that it can deal with problems that require domain specific knowledge to analyze. Such "intelligence" includes knowledge and semantics in certain domains that are to be captured and represented in the form of ontologies. The principle behind this feature is that the more GUITAR knows, the higher analysis quality it can perform. This allows GUI-F to be further augmented by enriching the domain ontologies.

Our experiments to date showed some promising results in the capability of GUITAR in analyzing 3Cs problems. Our benchmark validation indicated that GUITAR could identify a number of 3Cs problems that were not detectable by other tools. The key reasons are the use of domain ontologies and the technique for parameterizing requirements consistently. This second point explains for the better outcome GUITAR achieved in the validation compared to other ontology-based tools like RQA. In RQA,

false positive values were very high since requirements were compared to each other based on the meanings of the words they contained without considering which roles played by those words (i.e., object, beneficiary). Regarding user experience, while Kaiya and Dzung tools require manual matching of requirements and ontological concepts, it can be done automatically in GUITAR through our parameterization process.

In the following subsections, we discuss about the key limitation and challenges of our approach to analyzing goal and use case integrated models specified in natural language.

The effort of building domain ontologies. Ontology building is a time and labor-intensive exercise. It is an iterative and on-going process rather than a one-time task. In order to create a domain ontology, a domain study needs to be performed. In such study, domain concepts and relationships between them are extracted. In addition, knowledge and constraints needs also to be captured. There are a number of ontology learning tools that support the automated or semi-automated extraction of ontology from natural language documents. Although some of them are helpful in the ontology creation (i.e., Text2Onto [24], OntoLT [17]), they are far from entirely replacing human effort in this process.

However, in our view, the effort required for creating ontologies is a challenge rather than a limitation of the approach. It can be considered as a trade-off between the ability of identifying semantic problems, that are not detectable without a collection of domain knowledge and semantic (such as an ontology), and the effort spent to create such ontology. In fact, ontology-based approaches have been used extensively in requirements engineering (RE) and software engineering. According to a recent systematic literature review on the applications of ontologies in RE [39], several ontology-based techniques have been proposed and proven the effectiveness of using ontologies in solving different problems in RE, from elicitation to analysis and to validation. As shown in the benchmark validation, our approach was able to identify problems that are not detectable by other tools, partly because the use of domain ontologies. In addition, according to the same study, 34% of the reviewed approaches reported their success in reusing ontologies in their contributions. In our view,

reusability is one of the key benefits of ontologies. Creating ontologies can be costly, however it can be reused across projects in the same domain. Therefore, in our view, the benefits of building ontologies outweigh the effort to create them.

Nevertheless, we plan to build an "initial ontology" which contains pre-populated concepts that are commonly shared across domains, as an extension of our developed collection and categorization of commonly used verbs. It is intended to be done by consulting lexical resources like Wordnet [46], Verbnet [143] and other dictionaries. Such "initial ontology" would potentially help reduce the time and effort since only very domain specific semantics and knowledge would need to be captured in the ontology creation process.

The lack of support for analyzing requirements with temporal properties. Our approach currently does not support the specification of temporal properties such as *'until,' 'unless,' 'after x seconds'* due to two reasons. Firstly, as discuss in Chapter 4, these properties are not supported by the functional grammar which our technique is based on and thus, such requirements cannot be parameterized by GUITAR. In addition, temporal properties are not specifiable by MOS, the language used for formal specifications of artifacts in our work. There are more formal languages with more expressiveness power compared to MOS, however the reason for the choice of MOS was twofold. Firstly, it can be closely integrated with ontologies. Secondly, MOS is based on Description Logics and thus is computationally decidable, as opposed to more expressive logics like first-order logic or linear temporal logic that are undecidable. To overcome this problem, we plan to consider extending the functional grammar with additional semantic functions to accommodate temporal properties. Each property would be associated to a unique semantic function so that its semantic role in a specification can be fully differentiated. In addition, we plan to investigate the possibility of enhancing MOS with a set of inference rules concerning temporal properties in our future work.

The lack of support for analyzing other quality problems. Currently, our approach mainly focuses on analysis the consistency, completeness and correctness of goal-use case models. In the benchmark validation, there were also a number of quality problems (other than 3Cs) were detectable by other tools but was not with GUITAR. For instance,

RA was able to identify a testability issue with the requirement *"Communication forms are consistent throughout the system"* (although at the level of keyword-based detection with no detailed and meaningful explanation or suggestion). This requirement is hard to test since the quality of being *"consistent"* is unclear. It is planned to consider extending GUITAR with the support for other quality problems in our future work. This extension would be strongly supported by our functional grammar-based parameterization that helps in deeply analyzing the semantics of requirements, as opposed to keyword-based approaches.

7.7 Chapter Summary

In this chapter, we presented our ontology-based approach to analyzing goal and use case integrated models for incompleteness, inconsistency and incorrectness (3Cs problems). We started the chapter with a discussion on ontology, and its structure and presentation in our work. We then discussed the techniques for creating and assessing the quality of domain ontologies. The techniques for analyzing 3Cs problems were described in details. These include the classification of 3Cs problems based on GUIMeta – our meta-model for goal and use case integration, the parameterization of artifact specification, and the identification and resolution of 3Cs problems. We have also discussed our validation carried out to evaluate GUITAR's performance in analyzing goal-use case models for incompleteness, inconsistency and incorrectness. We achieved 88% and 98% accuracy two rounds of the parameterization validation. In addition, we achieved high precision (95% on average) and recall rates (88% on average) in six cases studies that indicate the high effectiveness of our approach in 3Cs problems detection. Moreover, the benchmark validation result showed that our tool could detect a wide range of 3Cs problems that were not identifiable in other approaches. In Chapter 8, we summarize our contributions in this research and discuss our future work.

Chapter 8

Conclusion and Future Work

In this chapter, we summarize the key problems we addressed in this research, and the key contributions that we have made. We also discuss the key limitations of our techniques developed during this research and describe the future extensions that we plan to make to improve our techniques.

8.1 Key Problems Addressed in Goal and Use Case Integrated Modeling

In this section, we discuss how the research objectives discussed in Chapter 1 have been addressed in our work.

RO1: *Develop a conceptual foundation for goal and use case integrated modeling.* This objective is concerned with the lack of a consensus among goal and use case integrated modeling approaches regarding what concepts and relationships to capture, and how goal-use case model can be systematically analyzed. To address this problem, we have developed a novel meta-model for goal and use case integrated modeling called GUIMeta to serve as a conceptual foundation for modeling goals and use cases together. GUIMeta was proposed to address the problem that, there is no consensus among goal and use case integrated modeling approaches regarding what concepts and relationships to capture, and how goal-use case model can be systematically analyzed. GUIMeta contains two layers. The artifact layer provides a comprehensive classification of

artifacts in a goal-use case model and defines relationships between them. The specification layer provides the specification rules for the defined artifacts based on the functional grammar-styled parameterization. Such rules offer the guidelines as to what should and should not be included in specifications of each type of artifacts. The combination of such artifact classification, functional grammar-styled parameterization, and specification rules provides a framework that governs specification and analysis of goal and use case integrated models. Moreover, we have also established the correspondence between GUIMeta's artifacts and those in other GUIM approaches to enable the unification of models specified in different approaches.

RO2: *Develop a technique to automatically extract goal and use case integrated models from textual requirements documents.* To address this objective, we have developed a novel rule-based technique to semi-automatically extract goal and use case models from textual requirements documents. The key novelty of our approach is that, it is the first technique, to the best of our knowledge, to combine the syntactic and semantic aspects of text in specifying extraction rules to identify necessary details while ignoring unneeded content from uncontrolled natural language text. The fact that our technique can work directly with uncontrolled natural language text potentially enhances its applicability, compared to a number of existing information extraction approaches in requirements engineering that rely on constrained input text.

Moreover, our technique is able to ensure the extracted artifacts to be well-formed according to the specification rules defined in our meta-model GUIMeta. Moreover, such extracted artifacts can also be automatically classified, allowing the entire extraction process to be mostly automated (only some manual pre-processing is needed). Furthermore, although we targeted only English text in our research, our extraction technique can be extended to support other languages (i.e., French, Germany) with little modifications (i.e., the main needed modification is to adapt to the grammars of the new languages).

RO3: *Provide a technique to automatically analyze goal and use case integrated models for inconsistency, incompleteness, and incorrectness.* To address this objective, we introduced a novel ontology-based technique to semi-automatically analyze natural language-based goal and use case integrated models for 3Cs problems. Our techniques

provide fully automate most tasks in the analysis process, except for the resolution of problems that need users to manually make selections. Our approach goes beyond the existing ontology-based requirements analysis techniques by offering a method to incorporate sophisticated domain knowledge into the analysis process (not just domain-specific terms and their dependencies).

Moreover, we have applied functional grammar to semantically parameterize textual goal and use case specifications. Such parameterization enables the semantics of specifications to be captured, represented, and interpreted during the analysis process. Last but not least, our developed technique to automatically parameterize textual artifact specifications and transform such parameterized specifications into Manchester OWL Syntax statements enables requirements engineers to work directly with natural language-based specifications with minimal required expertise of functional grammar and Manchester OWL Syntax. The benchmark evaluation (discussed in Chapter 7) proved the outstanding performance of our technique in comparison with some industrial and academic requirements analysis tools in some selected case studies.

8.2 Key Contributions

In this section, we discuss our key contributions in the related research areas, including goal and use case modeling, information extraction in requirements engineering, and requirements analysis.

Rule-based extraction technique: We have invented a novel rule-based technique to extract goals, use cases, and their dependencies from text. Although the technique was originally designed to be used in the goal-use case context, its key underlying concepts are independent and thus could be applied to other information extraction tasks.

The key value of this technique is its ability to overcome two main challenges in information extraction (discussed in Chapter 2): the difficulty in dealing with uncontrolled input text, and the lack of semantic consideration when writing extraction rules. We solve these problems by developing a language for specifying extraction rules based on the dependency grammar. The rules can also incorporate semantic information to specify matching conditions. For instance, we can define matching conditions by specifying which words or phrases should or should not exist in a sentence, and what

dependencies they should or should not have. This allows our technique to deal with uncontrolled text to identify what should and should not be extracted from such text. Importantly, our technique is accompanied with a clearly defined rule syntax and parser and thus, makes it possible for users to modify and extend the extraction rule collection.

There are various areas in which our extraction technique could be applied. For instance, extraction rules can be developed in a similar way (i.e., develop rule actions and algorithms to execute these actions) to identify privacy or security policies [174] from software documents, or development task from software documents [158]. Our rule-based technique can also provide a new approach in ontology learning [17, 24]. Specifically, rules can be created to detect ontological concepts, properties and their relationships to extract ontologies from natural language texts.

Automated text modification technique: we have developed a rule-based technique to automatically modify text that is written in an undesired format. This technique shares the same principle as the extraction technique discussed above. Specifically, a modifying rule contains a matching condition and a list of modification actions. A matching condition is specified based on the existence of and dependencies between certain words or phrases in a sentence, and can be used to identify undesired formats. Modification actions are used to automatically modify the text that satisfies the corresponding matching condition.

This technique could be adopted for text correction. For instance, it could be used to verify if a certain piece of text matches a required template and if not, the text could be automatically corrected.

Functional grammar-based parameterization: We have developed a method to use functional grammar as the fundamental theory to semantically parameterize textual requirements. Using this parameterization technique, the semantics of requirements can be captured in a systematic and consistent way. In addition, we have developed algorithms to automate such parameterization process so that textual specifications can be automatically parameterized. This thus opens an opportunity to automate the transformation from textual informal requirements into formal specifications via parameterized specifications since the parameterization provides details regarding the syntactic and semantic decomposition of textual specifications.

In addition, since the functional grammar is a general theory, our parameterization technique is not limited to requirements specifications. Therefore, its application could be extended to other areas that involve the need of capturing the semantics of text.

Specification boilerplates: We have developed a set of boilerplates for goals and use case components specifications. Although these boilerplates were originally designed for integrated models of goals and use cases, they can be used with individual use case models or goal models provided that goals are classified according to our classification (i.e., business goals, functional feature goals, and so on). Importantly, our automated parameterization technique can also be used in conjunction with the boilerplates to verify if a specification properly conforms to one of its corresponding boilerplates. Moreover, our automated text modification technique can be applied to automatically rewrite such a specification to ensure its boilerplate conformance.

Ontology structure: We have developed an ontology structure that offers a way to comprehensively capture and represent various types of domain knowledge, including domain-specific terms' semantics, domain-specific activities and their dependencies, and activity-actor assignment. Moreover, through the use of the functional grammar-based specification parameterization, we provide a technique to automate the matching of terms used in requirement specifications to the concepts in ontologies. Our technique thus offers a significant improvement in the area of ontology-based requirements engineering since most efforts in this area require the manually mapping between words in textual requirements into ontological concepts to perform analysis.

Requirements analysis technique: Although our analysis technique was originally designed for goal and use case integrated models, except a number of checking specifically developed for goals and use cases, several types of checking, such as semantic inconsistency or incompleteness detection, can be applied into general textual requirements. The key advantages of our techniques in comparison to the existing requirements analysis efforts include: (1) The ability to accept textual requirements as input, (2) The ability to incorporate domain knowledge and semantics to support the detection of semantic problems in a more comprehensive way compared to existing ontology-based approaches (we allow more types of knowledge and semantics to be

captured), (3) Little manual effort is needed as the entire process of transforming textual requirements into MOS statements and analyzing such statements is fully automated.

8.3 Key Limitations and Future Work

In this section, we discuss the limitations of our techniques in supporting the extraction and analysis of goal-use case integrated models. Based on these limitations, where appropriate, we discuss our plan for future work.

The accuracy of our extraction technique depends on the quality of the integrated linguistic tools: These include the Stanford parser for sentence parsing, the Stanford Coreference Resolution System for resolving coreference in sentences, and our extended version of Mallet classifier for classifying artifact specifications. The key reason is that, these linguistic tools are developed based on statistical machine learning techniques that require proper and sufficient training data to produce correct results. Therefore, it is possible to have a sentence incorrectly parsed or a specification incorrectly classified, especially in the domains that the linguistic tools have not been trained for. To simplify the training process, we have implemented the training feature that allows these linguistic tools to be trained via our GUITARiST tool.

Understanding of grammatical dependency required for rules writing: Our extraction technique requires the extraction and modifying rules to be manually written. This requires the rule writers to have knowledge of grammatical dependency and thus some training would be required for end users to be able to extend the rules collection. To alleviate this problem, in future work, we plan to develop algorithms that semi-automate the generation of a rule from input of users. Such input may contain the associations between sentences and lists of desired information to be extracted from them. Based on that, the algorithms are expected to identify the matching conditions and extraction actions, and suggest possible rules that fulfill the users' needs. In addition, we also plan to develop a visual rule editor to further support rule creation.

Unidentifiable artifact relationships: a number of relationships between artifacts in different sentences are not detectable since there exists no tool that can automatically detect coreference based on verb phrases in different sentences with a sufficiently high level of accuracy. For instance, such type of coreference can be found in the sentences:

"The application periodically communicates with the SplitPay server. This allows bills to be uploaded to a central server." In this example, the word *'this'* refers to the entire activity described in the first sentence. In future work, we plan study the possibility of improving the current identification techniques for this type of coreference problems.

Lack of support for analyzing requirements with temporal properties. Our approach currently does not support the specification of temporal properties such as *'until,' 'unless,' 'after x seconds'* due to two reasons. Firstly, these properties are not supported by the functional grammar which our technique is based on and thus, such specifications with temporal properties cannot be parameterized in our work.

In addition, temporal properties are also not specifiable in Manchester OWL Syntax (MOS), the language used for formally specifying artifacts in our work. There are more formal languages with more expressive power (e.g., LTL [166]) compared to MOS, however the reason for the choice of MOS was twofold. Firstly, it can be closely integrated with ontologies. Secondly, MOS is based on Description Logics and thus is computationally decidable, as opposed to more expressive logics like first-order logic or linear temporal logic that are undecidable.

To overcome this problem, we plan to extend functional grammar with additional semantic functions to accommodate temporal properties. Each property would be associated to a unique semantic function so that its semantic role in a specification can be fully differentiated. In addition, we plan to investigate the possibility of enhancing MOS with a set of inference rules concerning temporal properties in our future work.

The lack of analysis support for other quality problems. Currently, our approach mainly focuses on the consistency, completeness and correctness of goal-use case models. Other quality problems such as ambiguity, verifiability remained out of scope. Therefore, we plan to investigate the possibility of extending our analysis technique to detect these problems. By using our functional grammar-based parameterization technique that helps in deeply capturing and analyzing the semantics of requirements, our technique has a strong potential to achieve high detection accuracy, compared to many existing analysis approaches that rely on only keyword-matching techniques.

8.4 Final Words

In this thesis, we presented a number of challenges in using goals and use cases to model software requirements. Specifically, we focused on three important problems regarding the elaboration and analysis of such models: (1) a lack of unified conceptual meta-model for goal and use case integration, (2) a lack of technique for acquiring goal-use case models from textual requirements documents, and (3) a lack of technique for analyzing such models for incompleteness, inconsistency, and incorrectness. Our developed framework and tool support, which are backed by a set of linguistic techniques (functional grammar, natural language processing, text classification, etc.), have been proved (through a number of experiments) to be beneficial to addressing these challenges.

One of the important contributions of this thesis is that, our techniques are not limited to the field of Requirements Engineering. In fact, most of the presented techniques could also be applied into various research areas. For example, a customization or an extension of our rule-based goal and use case extraction techniques could be a solution to various information retrieval problems. Similarly, an extension of our automated text parameterization technique could become an underlying component for various text analysis systems.

However, there is a number of issues that have not been addressed within the scope of this thesis. They include the lack of automated support for identifying artifact relationships implied in verb phrase coreferences, the lack of support for analyzing requirements with temporal properties, and the dependency of our extraction techniques on users' ability of writing extraction rules. These issues could be a great input into future research and the solution to them could result in great improvement in the quality of goal-use case integrated modelling.

Appendix A1 Case Studies

This appendix provides 3 case studies used in our research. Due to space limitation, unnecessary details (e.g., unused pictures, tables, design information) have been removed.

Online Publishing System[14]

1. Introduction

1.1. Purpose

The purpose of this document is to present a detailed description of the Web Publishing System. It will explain the purpose and features of the system, the interfaces of the system, what the system will do, the constraints under which it must operate and how the system will react to external stimuli. This document is intended for both the stakeholders and the developers of the system and will be proposed to the Regional Historical Society for its approval.

1.2. Scope of Project

This software system will be a Web Publishing System for a local editor of a regional historical society. This system will be designed to maximize the editor's productivity by providing tools to assist in automating the article review and publishing processes. This helps reduce the manual effort in these processes. By maximizing the editor's work efficiency and production the system will meet the editor's needs while remaining easy to understand and use.

More specifically, this system is designed to allow an editor to manage and communicate with a group of reviewers and authors to publish articles to a public website. The software will facilitate communication between authors, reviewers, and the editor via E-Mail. Preformatted reply forms are used in every stage of the articles' progress through the system to provide a uniform review process; the location of these

[14] The original version of this case study can be found at http://goo.gl/34EVX4

forms is configurable via the application's maintenance options. The system also contains a relational database containing a list of Authors, Reviewers, and Articles.

1.3. Glossary

Active Article: The document that is tracked by the system; it is a narrative that is planned to be posted to the public website.

Author: Person submitting an article to be reviewed. In case of multiple authors, this term refers to the principal author, with whom all communication is made.

Database: Collection of all the information monitored by this system.

Editor: Person who receives articles, sends articles for review, and makes final judgments for publications.

Field: A cell within a form.

Historical Society Database: The existing membership database (also HS database).

Member: A member of the Historical Society listed in the HS database.

Reader: Anyone visiting the site to read articles.

Review: A written recommendation about the appropriateness of an article for publication; may include suggestions for improvement.

Reviewer: A person that examines an article and has the ability to recommend approval of the article for publication or to request that changes be made in the article.

Software Requirements Specification: A document that completely describes all of the functions of a proposed system and the constraints under which it must operate. For example, this document.

Stakeholder: Any person with an interest in the project who is not a developer.

User: Reviewer or Author.

1.4. References

IEEE. IEEE Std 830-1998 IEEE Recommended Practice for Software Requirements Specifications. IEEE Computer Society, 1998.

1.5. Overview of Document

The next chapter, the Overall Description section, of this document gives an overview of the functionality of the product. It describes the informal requirements and is used to establish a context for the technical requirements specification in the next chapter.

The third chapter, Requirements Specification section, of this document is written primarily for the developers and describes in technical terms the details of the functionality of the product.

Both sections of the document describe the same software product in its entirety, but are intended for different audiences and thus use different language.

2 Overall Description

2.1 System Environment

The Web Publishing System has four active actors and one cooperating system.

The Author, Reader, or Reviewer accesses the Online Journal through the Internet. Any Author or Reviewer communication with the system is through email. The Editor accesses the entire system directly. There is a link to the (existing) Historical Society.

2.2 Functional Requirements Specification

This section outlines the use cases for each of the active readers separately. The reader, the author and the reviewer have only one use case apiece while the editor is main actor in this system.

2.2.1 Reader Use Case

2.2.1.1 Use case: Search Article

Brief Description: The Reader searches for an article

Pre-condition: Before this use case can be initiated, the Reader has already accessed the Online Journal Website.

Initial Step-By-Step Description

1. The Reader chooses how to search the Web site. The choices are author, category, and keyword.
2. If the search is by author, the system creates and presents an alphabetical list of all authors in the database. In the case of an article with multiple authors, each is contained in the list.
3. The Reader selects an author.
4. The system creates and presents a list of all articles by that author in the database.
5. The Reader selects an article. The selected article is displayed clearly.
6. The system displays the Abstract for the article. The abstract should appear within half a second after the reader selects the article
7. The Reader selects to download the article or to return to the article list or to the previous list.

Extension

In step 2, if the Reader selects to search by category, the system creates and presents a list of all categories in the database.

3. The Reader selects a category.

4. The system creates and presents a list of all articles in that category in the database. Return to step 5.

Extension

In step 2, if the Reader selects to search by keyword, the system presents a dialog box to enter the keyword or phrase.

3. The Reader enters a keyword or phrase.

4. The system searches the Abstracts for all articles with that keyword or phrase and creates and presents a list of all such articles in the database. Return to step 5.

Post-condition: The selected article is downloaded to the client machine.

2.2.2 Author Use Case

2.2.2.1 Use case: Submit Article

Brief Description: The author either submits an original article or resubmits an edited article.

Pre-condition: Before this use case can be initiated, the Author has already connected to the Online Journal Website.

Initial Step-By-Step Description

1. The Author chooses the Email Editor button.

2. The System uses the sendto HTML tag to bring up the user's email system.

3. The Author fills in the Subject line and attaches the files as directed and emails them.

4. The System generates and sends an email acknowledgement.

2.2.3 Editor Use Case

2.2.3.1 Use Case: Add author

Brief Description: The Editor adds a new author to the database.

Precondition: The Editor has accessed the Article Manager main screen.

Basic Path

1. The system presents a blank grid to enter the author information. The blank grid should be familiar to most Internet Users.

2. The Editor enters the information and submits the form. The author information includes name and email.

3. The system checks that the name and email address fields are not blank and updates the database.

Extension

If in step 2, either field is blank, the Editor is instructed to add an entry. No validation for correctness is made.

Post-condition: The Author has been added to the database.

2.2.3.2 Use Case: Add Reviewer

Brief Description: The Editor adds a new reviewer to the database

Pre-condition: The Editor has accessed the Article Manager main screen.

Basic Path

1. The system accesses the Historical Society (HS) database and presents an alphabetical list of the society members.

2. The Editor selects a person.

3. The system transfers the member information from the HS database to the Article Manager (AM) database. If there is no email address in the HS database, the editor is prompted for an entry in that field.

4. The information is entered into the AM database.

Extension

In step 3, if there is no entry for the email address in the HS database or on this grid, the Editor will be reprompted for an entry. No validation for correctness is made.

Post-condition

The Reviewer has been added to the database.

2.2.3.3 Use Case: Update Person

Brief Description: The Editor selects to update a person

Pre-condition: The Editor has accessed the Article Manager main screen.

Basic Path

1. The Editor selects Author or Reviewer.

2. The system creates and presents an alphabetical list of people in the category. the list of people should appear after 2 seconds

3. The Editor selects a person to update. System should clearly highlight the selected person.

4. The system presents the database information in grid form for modification.

5. The Editor updates the information and submits the form.

6. The system checks that required fields are not blank.

Extension

In step 5, if any required field is blank, the Editor is instructed to add an entry. No validation for correctness is made.

Post-condition: The database has been updated.

2.2.3.4 Use Case: Update Article Status

Brief Description: The Editor selects to update the status of an article in the database

Pre-condition: The Editor has accessed the Article Manager main screen and the article is already in the database.

Basic Path

1. The system creates and presents an alphabetical list of all active articles.

2. The Editor selects the article to update.

3. The system presents the information about the article in grid format.

4. The Editor updates the information and resubmits the form.

Extension

In step 4, the use case Enter Communication may be invoked.

Post-condition: The database has been updated.

2.2.3.5 Use case: Receive Article

Brief Description: The Editor enters a new or revised article into the system.

Pre-condition: Before this use case can be initiated, the Editor has already accessed the main page of the Article Manager and has a file containing the article available.

Initial Step-By-Step Description

1. The Editor selects to Receive Article.

2. The system presents a choice of entering a new article or updating an existing article.

3. The Editor chooses to add or to update.

4. If the Editor is updating an article, the system presents a list of articles to choose from and presents a grid for filling with the information; else the system presents a blank grid.

5. The Editor fills in the information and submits the form.

6. The system verifies the information and returns the Editor to the Article Manager main page.

2.2.3.6 Use case: Update Reviewer

Brief Description: The Editor enters a new Reviewer or updates information about a current Reviewer.

Pre-condition: Before this use case can be initiated, the Editor has already accessed the main page of the Article Manager.

Initial Step-By-Step Description

1. The Editor selects to manage Reviewer.

2. The system presents a choice of adding or updating.

3. The Editor chooses to add or to update.

4. The system links to the Historical Society Database.

5. If the Editor is updating a Reviewer, the system and presents a grid with the information about the Reviewer; else the system presents list of members for the editor to select a Reviewer and presents a grid for the person selected. The system displays at most 10 members at a time.

6. The Editor fills in the information and submits the form.

7. The system verifies the information and returns the Editor to the Article Manager main page.

2.2.3.7. Use Case: Check Status

Brief Description: The Editor has selected to check status of all active articles

Pre-condition: The Editor has accessed the Article Manager main screen.

Basic Path

1. The system creates and presents a list of all active articles organized by their status. Article statuses include received, reviewer assigned, revision awaited, and finished

2. The Editor may request to see the full information about an article.

Post-condition: The requested information has been displayed.

2.2.3.8. Use case: Send Response

This use case extends the Update Article use case.

Brief Description: The Editor sends a response to an Author.

Pre-condition: Before this use case can be initiated, the Editor has already accessed the article using the Update Article use case.

Initial Step-By-Step Description

1. The Editor selects to Send Response.

2. The system calls the email system and puts the Author's email address in the Recipient line and the name of the article on the subject line.

3. The Editor fills out the email text and sends the message.

4. The system returns the Editor to the Article Manager main page.

2.2.3.9. Use Case: Publish Article

Brief Description: The Editor selects to transfer an approved article to the Online Journal

Pre-condition: The Editor has accessed the Article Manager main screen.

Basic Path

1. The system creates and presents an alphabetical list of the active articles that are flagged as having their copyright form returned.

2. The Editor selects an article to publish.

3. The system accesses the Online Database and transfers the article and its accompanying information to the Online Journal database. Published article information include name, author, abstract, category and content. The transfer must not take more than 2 seconds

4. The article is removed from the active article database.

Post-condition: The article is properly transferred.

2.2.3.10. Use Case: Remove Article

Brief Description: The Editor selects to remove an article from the active article database

Pre-condition: The Editor has accessed the Article Manager main screen.

Basic Path

1. The system provides an alphabetized list of all active articles. The list of articles should be easy to understand.

2. The editor selects an article. The selected article should be clearly highlighted

3. The system displays the information about the article and requires that the Editor confirm the deletion.

4. The Editor confirms the deletion.

Post-condition: The article is removed from the database.

2.3 User Characteristics

The Reader is expected to be Internet literate and be able to use a search engine. The main screen of the Online Journal Website will have the search function and a link to "Author/Reviewer Information."

The Author and Reviewer are expected to be Internet literate and to be able to use email with attachments.

The Editor is expected to be Windows literate and to be able to use button, pull-down menus, and similar tools.

The detailed look of these pages is discussed in section 3.2 below.

2.4 Non-Functional Requirements

The Online Journal will be on a server with high speed Internet capability. The physical machine to be used will be determined by the Historical Society. The software developed here assumes the use of a tool such as Tomcat for connection between the

Web pages and the database. The speed of the Reader's connection will depend on the hardware used rather than characteristics of this system.

The Article Manager will run on the editor's PC and will contain an Access database.

3. Requirements Specification

3.1 External Interface Requirements

The only link to an external system is the link to the Historical Society (HS) Database to verify the membership of a Reviewer. The Editor believes that a society member is much more likely to be an effective reviewer and has imposed a membership requirement for a Reviewer. The HS Database fields of interest to the Web Publishing Systems are member's name, membership (ID) number, and email address (an optional field for the HS Database).

The Assign Reviewer use case sends the Reviewer ID to the HS Database and a Boolean is returned denoting membership status. The Update Reviewer use case requests a list of member names, membership numbers and (optional) email addresses when adding a new Reviewer. It returns a Boolean for membership status when updating a Reviewer.

3.2 Detailed Non-Functional Requirements

3.2.1 Logical Structure of the Data

The logical structure of the data to be stored in the internal Article Manager database is given below <table removed>.

3.2.2 Usability

The system will be easy to use for users. It should ensure articles to be available to readers at any time and allow them to search and download articles quickly and easily. Editors should find it easy to enter or update authors, reviewers, articles when first using the system. The form configuration must be simple for typical internet users.

3.2.3 Security

The server on which the Online Journal resides will have its own security to prevent unauthorized write and delete access. There is no restriction on read access. The use of email by an Author or Reviewer is on the client systems and thus is external to the system.

The PC on which the Article Manager resides will have its own security. Only the Editor will have physical access to the machine and the program on it. There is no special protection built into this system other than to provide the editor with write access to the Online Journal to publish an article.

SplitPay[15]

1. INTRODUCTION

1.1 PURPOSE

The purpose of this Software Requirements Specification (SRS) document is to provide a detailed description of the functionalities of the SplitPay system. This document will cover each of the system's intended features, as well as offer a preliminary glimpse of the software application's User Interface (UI). The document will also cover hardware, software, and various other technical dependencies.

1.2 DOCUMENT CONVENTIONS

This document features some terminology which readers may be unfamiliar with. See Appendix A (Glossary) for a list of these terms and their definitions.

1.3 INTENDED AUDIENCE AND READING SUGGESTIONS

This document is intended for all individuals participating in and supervising the SplitPay project. Readers interested in a brief overview of the product should focus on the rest of Part 1 (Introduction), as well as Part 2 of the document (Overall Description), which provide a brief overview of each aspect of the project as a whole. These readers may also be interested in Part 6 (Key Milestones) which lays out a concise timeline of the project.

---

[15] The original version of this case study can be found at http://goo.gl/LbN0Po

Readers who wish to explore the features of SplitPay in more detail should read on to Part 3 (System Features), which expands upon the information laid out in the main overview. Part 4 (External Interface Requirements) offers further technical details, including information on the user interface as well as the hardware and software platforms on which the application will run.

Readers interested in the non-technical aspects of the project should read Part 5, which covers performance, safety, security, and various other attributes that will be important to users. Readers who have not found the information they are looking for should check Part 8 (Other Requirements), which includes any additional information which does not fit logically into the other sections.

1.4 PROJECT SCOPE

The SplitPay system is composed of two main components: a client-side application which will run on Android handsets, and a server-side application which will support and interact with various client-side features. The system is designed to facilitate the process of tracking and settling shared expenses. Potential scenarios include paying rent, splitting a check at dinner, sharing travel expenses, etc.

For more information about the project and its goals, see Appendix B (Project Proposal).

2. OVERALL DESCRIPTION

2.1 PRODUCT PERSPECTIVE

The SplitPay project is a new, self-contained product intended for use on the Android platform. While the SplitPay mobile application is the main focus of the project, there is also a server-side component which will be responsible for database and synchronization services. The scope of the project encompasses both server-side and client-side functionalities, so both aspects are covered in detail within this document.

2.2 PRODUCT FEATURES

The following list offers a brief outline and description of the main features and functionalities of the SplitPay system. The features are split into two major categories: core features and additional features. Core features are essential to the application's operation, whereas additional features simply add new functionalities. The latter features will only be implemented as time permits.

2.2.1 CORE FEATURES

2.2.1.1. USER REGISTRATION AND WELCOME

- Only appears once (the first time the application is run)
- Allows the user to register with the SplitPay server
- Enables the user to customize his account settings and preferences

2.2.1.2. GROUP CREATION & MANAGEMENT

- Streamlines the process of creating and organizing groups
- Provides support for multiple groups
- Allows the user to add group members manually or from contacts list

2.2.1.3. POSTING A BILL

- Stores and monitors the bill amount, the individuals involved, etc.
- Includes support for multiple simultaneous bills
- Efficiently distributes debt amongst the individuals responsible for the bill

2.2.1.4. MEMBER-TO-MEMBER TRANSACTIONS

- Enables group members to simulate transfers of debt, payments made, etc.
- Adjusts member balances accordingly
- Records relevant information (amount paid, members involved, etc.)

2.2.1.5. FINAL DEBT RESOLUTION

- Calculates the most efficient method of sorting out debts
- Notifies group members of unresolved debts, credits, etc.
- Offers the option to disband a group once all payments are made

2.2.1.6. GROUP HISTORY

- Automatically records all transactions and bills posted to each group
- Provides users with access to a detailed history of transactions o Supports sorting transactions by date, amount, payer, etc.

2.2.1.7. SHOW ALL DEBTS

- Enumerates all of a user's unresolved debts across each group he is a part of
- Provides easy access to relevant information (past transactions, group info, etc.)

- Offers the option to resolve a debt (or debts) immediately

2.2.1.8. SETTINGS MENU

- Allows the user to customize his preferences
- Enables the user to modify certain features and functionalities
- Can be accessed at any time using the built-in Settings button on Android phones

2.2.1.9. HELP MENU

- Displays a list of topics covering the different components of SplitPay
- Offers detailed information on each feature, menu, etc.
- Can be accessed at any time via the Settings menu

2.2.1.10. PUSH NOTIFICATIONS

- Appear after any significant event occurs in a group
- Alert group members of newly incurred expenses o Remind users of unresolved debts

2.2.2. ADDITIONAL FEATURES

2.2.2.1 MEMBER DEBT VISUALIZATION

- Presents a visual representation of current member balances
- Allows users to navigate through financial information in a more intuitive fashion
- Automatically updates as users post expenses and make transactions

2.2.2.2. PAYPAL INTEGRATION

- Incorporates a mechanism for initiating real transactions
- Facilitates secure, hassle-free transactions between members
- Automatically updates member balances as transactions occur

2.2.2.3. GPS TRACKING

- Stores location data associated with certain events
- Utilizes Google Maps to display transaction locations
- Creates an expense map which can be viewed by all members of a group

2.2.2.4. RECEIPT IMAGING

- Utilizes the camera built into Android handsets
- Records and stores a snapshot of receipts associated with different expenses
- Provides a method of checking and verifying expenses posted to a group

2.2.2.5. E-MAIL and SMS NOTIFICATIONS

- Extends the standard notifications service built into SplitPay
- Automatically delivers notifications via e-mail and text message
- Enables individuals without SplitPay to receive group notifications

2.2.2.6. SPLITPAY TUTORIAL

- Provides an abridged version of the Help menu for first-time users
- Offers a step-by-step run through of each feature, menu, etc.
- Enables any user to quickly and easily take advantage of all of SplitPay's functionalities

A major functionality present in several of these features is automatic synchronization. Using Android's internet capabilities, the application periodically communicates with the SplitPay server. This allows bills, transactions, groups, and group histories to be uploaded to a central server where the data can be shared with all other Android users in the group. This process of exchanging data between the server and the phone(s) is referred to as syncing.

2.3 USER CLASSES AND CHARACTERISTICS

The SplitPay project is meant to offer a shared expenses solution that is faster, easier, and more convenient than manually calculating and handling debts. Consequently, the application will have little or no learning curve, and the user interface will be as intuitive as possible. Thus, technical expertise and Android experience should not be an issue. Instead, anticipated users can be defined by how they will use the product in a particular situation. The following list categorizes the scenarios in which SplitPay is expected to be utilized:

1. Long-term recurring expenses (e.g. rent, groceries, utilities)

- Keep track of expenses
- Notify users when debts are incurred

- Record who has paid and who still owes

2. Short-term recurring expenses (e.g. travel costs? gas, food, hotel)

- Add new expenses (quickly and easily)
- Record who is paying and what he is paying for
- Update member balances on the fly

3. Single expense (e.g., splitting a bill at dinner)

- Create a group (quickly and easily)
- Add non-registered individuals to the group
- Quickly calculate each member's balance

These groups are not meant to separate or categorize users, just the different situations in which SplitPay is likely to be used. In fact, a user may utilize the application for all of these scenarios simultaneously. This is another defining feature of the SplitPay system: support for multiple groups. This functionality allows a user to track expenses pertaining to several unrelated groups at the same time.

It is crucial that each of these situations be fully supported in the final product so as to maximize the overall value of the product. It is also important that the application be as user-friendly as possible, otherwise it will not be a viable alternative to handling shared expenses manually. Most importantly, the application must be reliable. Regardless of the situation, the application must accurately distribute costs. There is zero tolerance for error when dealing with financial transactions.

2.4 OPERATING ENVIRONMENT

The main component of the SplitPay project is the software application, which will be limited to the Android operating system (specifically Android 2.2 and above). The application is not resource- or graphics-intensive, so there are no practical hardware constraints. The app will rely on several functionalities built into Android's Application Programming Interface (API), so ensuring appropriate usage of the API will be a major concern. Beyond that, the application is a self-contained unit and will not rely on any other Android-related software components.

The application will, however, frequently interact with the SplitPay server, a virtual dedicated server hosted by GoDaddy.com. The server operates on a Linux CentOS

platform with 1GB of RAM and 15GB of allocated storage space. The SplitPay database will be stored on the server using MySQL and will be interfaced with a wrapper written in PHP 5.

2.5 DESIGN AND IMPLEMENTATION CONSTRAINTS

The primary design constraint is the mobile platform. Since the application is designated for mobile handsets, limited screen size and resolution will be a major design consideration. Creating a user interface which is both effective and easily navigable will pose a difficult challenge. Other constraints such as limited memory and processing power are also worth considering. SplitPay is meant to be quick and responsive, even when dealing with large groups and transactions, so each feature must be designed and implemented with efficiency in mind.

2.6 USER DOCUMENTATION

The primary goal of SplitPay is to facilitate the process of managing shared expenses. Consequently, the application will be designed to be as simple to use as possible. Nonetheless, users may still require some supplementary information about each component of the SplitPay system. The application will contain two features that offer this: The SplitPay Tutorial and the Help menu.

The Help menu is a collection of topics covering each of the application's menus, features, etc. At any time, the user can navigate to the Help menu and select any of these topics to obtain more information. Details about the Help menu can be found in section 3.9 of this document.

The SplitPay Tutorial takes all of these topics and condenses them into a single, step-by-step demonstration that the user can access immediately after installing the application. This tutorial is meant to quickly and effectively teach new users the "ins and outs" of the application. Section 3.16 covers the tutorial in further detail.

2.7 ASSUMPTIONS AND DEPENDENCIES

TIME DEPENDENCIES

As mentioned previously, the features of SplitPay are divided into two groups: core features and additional features. Core features are crucial to the basic functionality of the SplitPay application. These features must all be implemented in order for the application to be useful.

Optional features, however, are not critical to the function of the application. They are usability improvements and convenience enhancements that may be added after the application has been developed. Thus, the implementation of these features is entirely dependent upon the time spent designing and implementing the core features. The final decision on whether or not to implement these features will be made during the later stages of the design phase.

HARDWARE DEPENDENCIES

Some of the additional features rely on hardware components present in Android handsets. For instance, the camera will be used to record images of receipts for digital storage. Consequently, this feature is entirely reliant upon the ability to access the camera's functionalities. In addition, the application will use the handset's location sensors (GPS) to record the location of a specific bill or transaction. Both the camera and the GPS functionalities will be achieved using the API provided by the Android operating system.

EXTERNAL DEPENDENCIES

Several of the features presented in this document rely on the existence and maintained operation of several APIs. A non-exhaustive list follows.

EMAIL NOTIFICATIONS

The Android platform is not suited for sending mass emails. Thus, the central server will be responsible for this feature of the application. The smartphone client will notify the server when messages need to be sent using a custom API that is to be created. This API will use standard HTTP messaging to facilitate client-server communications. The API will be implemented using PHP.

SMS NOTIFICATIONS

This feasibility of this feature is yet to be determined. If implemented, this feature would allow offline users without an Android smartphone to receive notifications of outstanding debt and other information via text messages. A suitable and free text messaging API that can be called from the server has yet to be found. The possibility of sending text messages from the Android smartphone client itself is also being reviewed.

PAY-PAL WEB API

We will use the PayPal API in order to facilitate payment of debts that users may have incurred and wish to pay using the software.

GOOGLE PLACES API

The software application may integrate the ability to interact with Google Places for marking the location of a bill or purchase. Note that this API is not guaranteed to be perfectly functional, as it is currently in the beta phase of its lifecycle and is provided by an experimental branch of its host company, Google Labs. Using this API may require the use of the GPS hardware on the Android platform, where it is available.

3. SYSTEM FEATURES

SplitPay's system features are divided into two main categories: core features and additional features. Core features form the body of the application and include any features that are essential to the functionality of the SplitPay system. These features must be implemented in order to have a fully functioning application. Additional features, however, are not required for the app to function. They include any features which, if time permits, will be added to the application in order to provide extra functionality.

CORE FEATURES

3.1 USER REGISTRATION AND WELCOME

When the application is installed and run for the very first time, the user is presented with an initial registration and welcome screen. This screen prompts the user to create an account on the SplitPay server using the email address associated with his Google account. The user also enters a "Display Name," which will be the name that is shown as their handle within the groups. Completing this process will create and store an account for the user on the SplitPay server, enabling all of the app location's synchronization capabilities.

3.1.1 STIMULUS/RESPONSE SEQUENCES

Step 1 SplitPay application launched from the Android home screen

Step 2 The user is prompted to enter an email address and a display name

- The user's Google account info is entered by default

Step 3 This information is sent to server and stored in the database

Step 4 Registration is completed and user is taken to main screen

3.2 GROUP CREATION AND MANAGEMENT

The "Groups" screen will be the main screen of the application. From this screen, users will be able to view and manage existing groups. Groups may be created by adding members from the user's contacts or by manually entering an email address and name. The creator of a group is designated as the "Leader" of that group. The Leader is responsible for confirming transactions submitted by other members of the group and will also have the ability to stop or disband the group.

3.2.1 STIMULUS/RESPONSE SEQUENCES: GROUP CREATION

Step 1 The user selects "Create a New Group" from the main Groups screen

Step 2 The user must choose a unique name to be used as the group's identifier (e.g., "D.C. Road Trip"). The user can add members to the group from contacts stored on the phone or via manual entry (email address and display name are required).

- Users who are not registered with the SplitPay server are designated on the server as "offline users"
- These users cannot participate in transactions directly, so the leader is responsible for reconciling their balances
- At any time, these offline users may register a SplitPay account (using the same information stored on the server) and take control of their transactions

Step 3 The user finishes selecting members and confirms that the group is complete

Step 4 The user is then designated as the leader of the group

3.3 POSTING A BILL

On several screens, users are given the option to create a new expense, or "Bill," and post it to a particular group. Bills created within the application are meant to represent expenses incurred by the group in real life. Users must input a display name (e.g., "Groceries"), the associated cost, and select the individuals responsible for the expense. Once the Bill is confirmed, it is synced to the server, enabling all other SplitPay users in the group to view it.

3.3.1 STIMULUS/RESPONSE SEQUENCES: POSTING A BILL TO THE GROUP

Step 1 The user is presented with a screen where they enter the following information:

- Expense title
- Bill amount
- Optional information includes
- Location
- Type of expense (e.g., Travel, Groceries, Food, etc.)
- Comments (i.e., any other information the user finds relevant)

Step 2 The user then selects the members of the group involved in the Bill

- User can choose "Select All" or pick members individually

Step 3 An algorithm calculates the distribution of debt amongst each member involved in the bill

Step 4 The Bill is pushed to the server, which notifies other group members of the Bill and also updates their running balances

3.4 MEMBER-TO-MEMBER TRANSACTIONS

This feature represents a real-world transaction between two or more m embers of a group. A common scenario would involve the user resolving his debt by making payments to other group members. The resulting changes in balances are calculated automatically and displayed for the users involved.

3.4.1 STIMULUS/RESPONSE SEQUENCES

Step 1 The user selects Member-to-Member Transaction and inputs the following data:

- Transaction Description (optional)
- Member(s) to be paid
- Amount to be paid (to each member if multiple)

Step 2 The user confirms the transaction

Step 3 Each involved member has his balance adjusted automatically

Step 4 The information is sent to the server for approval by the Leader (unless the Leader created the transaction)

3.5 FINAL DEBT RESOLUT ION

This feature is utilized when the Leader of a group wants to end the group and resolve all debts. An algorithm takes into account the balances of all members and determines the most efficient method of resolving debts, minimizing the number of transactions between members.

3.5.1 STIMULUS/RESPONSE SEQUENCES

Step 1 The group's Leader chooses to end the group and resolve all debts

Step 2 The algorithm automatically determines the most efficient way to resolve all debts

Step 3 E-mails and notifications are sent out containing the following information:

- The group's name, members, and the user's overall balance in that group
- A list of group members who the user owes and how much he owes to each person
- A list of group members who owe the user and how much each person owes him

Step 4 Each member can then use Member-to-Member transactions to resolve his debts.

Step 5 Once all debts are resolved, all members are notified and the group may then be removed from the main Groups page. Bill and Transaction Histories are still available, however.

3.6 GROUP HISTORY

This screen provides a view of all transactions and bills that occurred within a group. This list will be presented in chronological order by default, but can also be sorted by payer, amount, etc. It will show the names of members involved in each transaction and the amounts paid, and the user has the option of viewing each item in more detail by selecting it. The detailed view will display all members involved in the bill, any comments about it, and any additional information that was included when it was created (Location, Type, etc.).

3.6.1 STIMULUS/RESPONSE SEQUENCES

Step 1 The user is presented with a list of all transactions posted to the current group

- The user may sort these items by date, amount, payer, etc.

Step 2 The user may select any one of the transactions for detailed viewing

- Upon selection, a dialog is presented with the details o f the transaction

3.7 SHOW ALL DEBTS

This is a global feature that will enumerate an individual's debt across all groups of which he is a member. Visual cues will be used to provide a distinction between positive, negative, and even balances within each group. There will also be an option to reconcile all debts from this screen. Selecting this option will show the user a list of transactions which are required before his debts can be resolved. The user will be able to initiate these transactions within the app from this screen.

3.7.1 STIMULUS/RESPONSE SEQUENCES

Step 1 The user enters this screen from the popup context menu from any other screen in the app

Step 2 The user can see his status in each of the groups he is a member of

- Selecting a specific group from this list displays that group's History page

Step 3 The user has the option to resolve his debts with a transaction for each group

3.8 SETTINGS MENU

This menu allows the user to modify more advanced settings within the application. The menu is accessed from any screen by the Preferences button, a hardware button built into all Android handsets.

3.8.1 STIMULUS/RESPONSE SEQUENCES

Step 1 From any menu, the user may press the Settings button

- This is a physical button featured on all Android phones

Step 2 From a pop-up menu, the user may select what he wishes to modify

Step 3 After making the desired changes, the user may save his preferences and exit the menu, or simply exit without saving

Step 4 The user is then returned to the screen he was on before accessing the Settings menu

3.9 HELP MENU

The Help menu is meant to answer any questions the user may have while using SplitPay. The menu displays a list of topics related to features, menus, and the app in general. The user can select any of these topics to access further information and explanations.

3.9.1 STIMULUS/RESPONSE SEQUENCES

Step 1 From any screen, the user may press the Settings button built into the phone

Step 2 This displays the Settings menu, where he can then select the Help menu

Step 3 The user is presented with a list of help topics which he can scroll through

Step 4 Once the user selects a topic, more information on that topic i s displayed

Step 5 The user can navigate back to the Help screen or exit the Help menu altogether

3.10 PUSH NOTIFICATIONS

Push notifications is an added mechanism to provide updates and alerts from the application server to the user's android device. Whenever a new expense is posted in a group, the server will broadcast notifications to all group members' phones of the new bill and each user's respective balances. Members are also notified when all debts in a group are resolved or when the leader chooses to notify all members of the group's balance.

3.10.1 STIMULUS/RESPONSE SEQUENCES: GROUP NOTIFICATION

Step 1 The leader selects to notify all members of the group's current balance.

Step 2 Server pushes a notification to all members' devices with a pop-up.

- e.g., "Current Balance of the Group 'Trip to DC' is $100."

ADDITIONAL FEATURES

3.11 MEMBER DEBT VISUALIZATION

This feature allows users to access group balances as bar graphs, rather than the default text-only form. These visuals can be obtained at any time by selecting the Member Debt Visualization option from any group. The visuals are updated any time a group member's balance changes.

3.11.1 STIMULUS/RESPONSE SEQUENCES

Step 1 The user activates this visualization via a button on a particular group page

Step 2 From here, the user can view details about individual member s by selecting them

3.12 PAYPAL

This feature will offer users the option to resolve their debts through using PayPal. This will enable users to make real financial transactions through the app, which will then update all balances accordingly.

3.12.1 STIMULUS/RESPONSE SEQUENCES

Step 1 The user accesses this feature from any transactions-related screen

Step 2 The user enters his PayPal account information

- The user has the option of being automatically logged in next time he uses this feature

Step 3 The user is taken to a screen where he must fill out all of the necessary details

of the transaction

- If this information was already entered in the transactions screen, it will be filled in automatically

Step 4 The user confirms the payment, and the transaction is completed

3.13 GPS TRACKING

When a Bill is posted or a transaction is made, this feature records the GPS location of the phone(s) involved. At any point, a user may access this information in the form of a map with each saved location displayed. This feature provides a visual record of where the group has been, what payments were made at specific locations, and what path on the map the group has taken. The application will use Google Places as its maps tool.

3.13.1 STIMULUS/RESPONSE SEQUENCES

Step 1 When posting a bill or transaction, the user has an option to add the GPS location

Step 2 If this option is selected, the phone records the GPS location and also sends it to the server, where it is synced to all other SplitPay-enabled phones

Step 3 The user may also manually enter in an address if the GPS location is not desired

Step 4 From any group screen, the user may access the group's associated map

Step 5 Each address and GPS location is pinned to the map, with a line tracing the path between each pin

3.14 RECEIPT IMAGING

Receipt imaging allows users to store the receipt associated with a Bill for later viewing. Whenever a user posts a Bill, he will have the option to take a pho to of the receipt using the phone's built-in camera. This image is then sent to the server and synced with all SplitPay users in the group.

3.14.1 STIMULUS/RESPONSE SEQUENCES

Step 1 While creating a new Bill, the user may select the Receipt Imaging feature.

Step 2 This activates the camera and creates a window on the screen to act as a viewfinder.

Step 3 The user then take s a picture of the receipt by pressing a button on the app.

Step 4 The user has the option to retake the picture until he is satisfied with the result.

Step 5 The picture of the receipt is then saved and becomes accessible by viewing the Bill.

Step 6 After the new bill is created, the data will be pushed to the server and synced with other phones

3.15 E-MAIL and SMS NOTIFIC ATIONS

Since groups may involve individuals without SplitPay (i.e., offline users), there must be an alternative method of notifying these individuals. The simplest way to achieve this is via e-mail or text message (SMS). This will allow these individuals to enjoy some of the same functionalities as SplitPay users.

3.15.1 STIMULUS/RESPONSE SEQUENCES

Step 1 When a Bill is posted or a transaction is made, the server is updated and any user involved is sent a notification

- The group leader may modify offline users? notification preferences, which will determine whether the user receives an e-mail, a text message, or no notification at all

Step 2 SplitPay users are notified within the application, but offline users are sent an e-mail or text message containing information about the update.

3.16 SPLITPAY TUTORIAL

The SplitPay Tutorial is a structured, condensed version of the Help menu. This feature will allow the user to view brief explanations of each menu and feature. These explanations will be kept brief, as the user interface will be designed to be as intuitive as possible. The tutorial will be designed to answer any questions a first-time user may have about the application.

3.16.1 STIMULUS/RESPONSE SEQUENCES

Step 1 At the initial registration screen, the user will have the option to view the tutorial

Step 2 The tutorial will then direct the user to each of the menus within the app, explaining each feature in some detail. The user will not be able to do anything on these screens besides navigating through the steps of the tutorial.

Step 3 Once each menu has been explained, the tutorial will return the user to the welcome screen and allow him to complete the registration process.

Alternatively, the user may quit at any point in order to return to the welcome screen

4. OTHER NONFUNCTIONAL REQUIREMENTS

4.1 PERFORMANCE REQUIREMENTS

Performance should not be an issue because all of our server queries involve small pieces of data. Changing screens will require very little computation and thus will occur very quickly. Server updates should only take a few seconds as long as the phone can maintain a steady signal. The cost-division algorithms used by in application will be highly efficient, taking only a fraction of a second to compute.

4.2 SAFETY REQUIREMENTS

SplitPay will not affect data stored outside of its servers nor will it affect any other applications installed on the user's phone. It cannot cause any damage to the phone or its internal components. The only potential safety concern associated with this application applies to virtually all handset apps: SplitPay should not be used while operating a vehicle or in any other situation where the user's attention must be focused elsewhere.

4.3 SECURITY REQUIREMENTS

This application assumes that only the user or whoever he allows will have access to his Android handset. With that being said, only a Google email address is required to verify the identity of the user upon opening the app. Since it is not password protected, there is no method to authenticate the user's identity. This could only pose a thread if a user has set up PayPal functionality, however any transaction involving real currency must be authorized and confirmed before becoming final. The PayPal API provides all of the security checks needed to ensure that no fraudulent transactions occur.

4.4 SOFTWARE QUALITY ATTRIBUTES

The graphical user interface of SplitPay is to be designed with usability as the first priority. The app will be presented and organized in a manner that is both visually appealing and easy for the user to navigate. There will be feedbacks and visual cues such as notifications to inform users of updates and pop-ups to provide users with instructions.

To ensure reliability and correctness, there will be zero tolerance for errors in the algorithm that computes and splits expenses between group members. To maintain flexibility and adaptability, the app will take into account situations in which a user loses internet connection or for whatever reason cannot establish a connection with the server. These users will still be able to use the application, but any Bills, transactions, etc. posted while disconnected will be cached until the connection is restored.

Furthermore, the group leader also has the option to add members who do not own an Android phone and add transactions on their behalf. With SplitPay being ported solely for the Android platform, this software application has the advantage of being portable and convenient to use whenever and wherever. Overall, the app balances both the ease of use and the ease of learning. The layout and UI of the app will be simple enough that users will take no time to learn its features and navigate through it with little difficulty.

Traveler Social Networking System

1.1 Document Purpose

This document provides information about requirements and qualities of HWG, a new social networking system for travelers.

Due to some confidentiality restriction, this document only contains part of the entire original requirements document.

1.2 Product Scope

This system is intended to connect travelers around the World and improve the quality of travel planning. In addition, it should allow travelers to gain and share travel experiences by enabling them to write reviews for places and services.

Travelers can also write articles to tell friends about the interesting stories during their trips. Moreover, travelers can gain travel experiences by asking questions in a forum and obtaining information from reviews and articles. Furthermore, to enable travelers to share and gain travel experience, the system supports the communication between them.

An important requirement is that the system must be easy to use and understand for travelers. In addition, the system must be secure and can prevent attacks.

2 Overall Description

2.1 Product Perspective

The key objective of the system is encouraging experience gaining and sharing. This system shall be a stand-alone web-based application and accessible via the Internet at any time. In the future, a mobile application shall be developed to allow users to access the system via smartphones.

2.2 Product Functionality

2.2.1 User management

The system should provide facilities for users to register for accounts. They can register for accounts using social media accounts (i.e., Facebook, Google or Twitter). If users register using these social media accounts, their profile pictures will be imported by the system from these accounts.

The system needs also to provide administrators with the ability to manage user accounts. Specifically, they should be able to ban and remove user accounts when necessary. Moreover, administrators can also reset accounts' passwords.

System shall also support privilege assignment and management. Administrators are enabled to specify the part of the systems to which a user can access. For example, particular users are only allowed to access forum settings/modify forum posts and other

users are allowed only to change the frequency of emails sent to particular groups of users. Those privileges of users can easily be changed by the administrators.

2.2.2 Connection

2.2.2.1 Making friends and following

Users can make friends with each other. Specifically, a user can send a friend request to another user and this user can either accept or deny the request. In addition, users can also choose to follow another user instead of sending a friend request to that user. If a user follows another user, he/she would receive updates about the posts such a user makes. Users can also follow a forum threads to receive updates about new posts in such threads. They can also follow a certain topic to receive updates about new contents in such a topic.

2.2.2.2 Profile page

Each user has a profile page. A profile page contains the summary of the user's activities, status and wall comments. Profile pages should be organized in a clear way. The system displays the activities of users on their profile pages. Users can set their account privacy to prevent some certain activities to be displayed on their profile pages. A user can write comments of their friends' walls. They can later edit or remove their comments. Only administrators can delete comments from users' walls.

2.2.2.3 Group page

Users can also create groups. Group owners can set the visibility of the group which is either public or private. If the group is private, only group members can see posts in that group. If a group is public, every user can access the content of that group. Content of a group includes the list of activities in the group, the wall comments of its members and the files shared between members in the group. Group members can invite their friends to join a group.

2.2.2.4 Newsfeed

System displays the updates of the entities a user follows on the user's news feed. Such entities include other users, forum and topics. The newsfeed in this system should be similar to the newsfeed in Facebook. The system should support update filtering. More specifically, users shall be able to filter the content to be displayed on their newsfeed.

2.2.2.5 Wall comments

Users are able to post comments on their walls or their friends' walls. Users can also post a reply to an existing comment on a wall. To support users to control their walls, the system allows them to set the users who can post comments on their walls. In addition, users can remove any comments on their walls. However, they cannot edit other users' comments. They can edit only their comments.

2.2.3 Forum

The system shall allow users to post questions and answers in a forum. To enable users to use the forum conveniently, the system allows them to post questions to forum from their profile pages. To answer a question in the forum, users can reply to a question or reply to an existing answer. Users can follow a question to receive updates whenever new answers to that question are posted. Users can choose to receive notifications by email or internal messages. The organization of the forum is similar to the forum's organization in tripadvisor.

2.2.4 Communication

The system supports the communication between users. Specifically, the system supports both one-to-one and group instant messaging. Users can send messages to their friends. They can also attach files to messages.

2.2.5 Article writing

The system allows users to create travel articles. They can also add media files (videos, photos) into an article. An article with photos or videos is marked with a media file logo to differentiate it with text-only articles. The system supports typographic features to enable users to create appealing articles.

To upload a video to an article, a users need to upload it to a third-party website, obtain the video's link and embed the link into the articles. User can remove a video from an article by removing its link from the article.

Users can upload images directly to the article. The images shall be stored in the system. The size of an uploaded image must not be greater than 1Mb. If a user uploads an image that is greater than 1Mb, the system shall automatically resize the image. The system allows users to position the images in an article. Obviously, users can remove images from an article.

2.2.6 Review

Users are allowed to write reviews for tours and places. Obviously, this requires tours and places' information to be previously added by administrators into the system. Users can also send an entity-adding request to the administrators to request them to add a new entity. A review contains a review subject, review description and a rating. When a user views a tour or a place, the system displays its reviews. The system also shows an entity's average review score on its page. Users can vote for a review. If a review receives a vote, some scores shall be added into the review owner's account. The score calculation should be accurate. To ensure the reliability of reviews, the system should be able to prevent fake reviews to be created. It does so by monitoring the review activities of users.

2.2.7 Trip planning

The system allows users to plan their travels. Users can create a trip on their profile page and send invitation to their friends to invite them to join the trip. When accepting an invitation to a trip, a user can access and edit the trip information. In order to enable users to plan their travels effectively, the system allows them to ask questions related to their trips on the trip's page. These questions will then be displayed on the pages of locations relevant to the trip. Users who follow these locations shall be notified about the new questions.

2.2.8 Travel expert

Users can register to be travel experts for some certain destinations. Whenever a question about a certain destination is posted in the system, that destination's travel experts shall be notified. The administrators can monitor user activities. If the administrators identify a user who has substantial knowledge about a destination, they will send an expert invitation to that user.

2.3 Users and Characteristics

The intended users of the system are travelers or people who are interested in travelling around the World. Typically, they have regular access to the Internet via computers or smartphones. The technical expertise, educational level and computer experience of users may vary. Therefore, the system should be designed to be easy to use by typical Internet users. The categories of the anticipated users are as follows:

- People who travel frequently. These users would frequently ask questions, make connections with other travelers and share their experiences. This is the main intended user group of the system.
- People who do not frequently travel, however having interest in travel. These users would frequently read the content posted in the system. They would frequently follow popular travelers in the system to receive updates from them.
- People who have a lot of travel experience. They have been to many cities and countries. They are usually experts in one or more destinations. This type of users frequently offers help, guidance to other users. They would frequently answers questions in the forum or trip plans.
- Business users: these users come to the system not for gaining or sharing experiences. Their purpose is to advertise for some certain hotels, restaurants, events or tours. They are typically business owners. For instance, the owner of a restaurant may come to suggest an addition of his/her restaurant into the system if it has not been added so that travelers shall be able to get more information about the restaurant and/or provide reviews for the restaurant.

2.4 Operating Environment

Since HWG is a web-based system, there is no specific requirement regarding the client's hardware or operating system. Users only need a computer/smartphone with a browser and Internet connection to access web applications in order to use the system.

The system will be hosted on a GoDaddy.com's virtual dedicated server. The server is required to run on a Linux operating system with at least 2GB RAM and 30GB storage space for the first release. The server hardware requirements may increase when the number of users increases. MySQL is required for storing data in the system.

The system shall frequently interact with Facebook, Twitter and Google via their APIs to allow users to link their social media accounts to HWG. In addition, that also enables users to share HWG contents on the social media that they are using.

2.5 Design and Implementation Constraints

The system must be able to respond quickly to client requests. In the first release, the system should be able to handle 500 concurrent users. The system's server should have

at least 2GB RAM and 30GB storage space. The system also needs to be available at any time. Backups are conducted every day.

2.6 User Documentation

HWG is intended to be easy to use and understand by users. The system also provides online help to users. In addition, a number of tutorials showing the instructions of using some features are provided to users after their registrations. Users obviously can view these instructions later at any time.

3 Specific Requirements

This section presents the detailed the use cases that describes the above feature of the system.

3.1.1 User management

3.1.1.1 Use case: Register membership

Description: a user registers for an account

Pre-condition: the user has accessed the registration page

Post-condition: a new account is created

Main Success scenario:

Step 1. The user selects a registration option. The registration options are with_social_media_account and without_social_media_account.

Step 2. If the users select to create an account without a social media account, the system displays the registration form to the user

Step 3. The user enters their details into the registration form and submits it

Step 4. The system verifies the details. The system should verify the details in less than 1 second.

Step 5. If the details are valid, the system creates a new account

Step 6. The system displays the confirmation message to the user

Extension:

In step 2, if the user selects to create an account with a social media account, then:

Step 2.1. The system displays a list of social media account types to the user. The list of supported account types includes Facebook, Twitter and Google.

Step 2.2. The user selects an account type

Step 2.3. The system requests the user to enter the account details

Step 2.4. The user enters the account details

Step 2.5. The system connects to the selected social media website

Step 2.6. The system displays the registration form with the user's details from the social media website

The extension resumes at step 3

Extension:

In step 5, if the details are not valid, then the system displays an error message and the user repeats step 3.

3.1.1.2 Use case: Reset password

Description: A user resets his password

Pre-condition: User is at the login page

Post-condition: an account resetting email is sent to the user

Main Success scenario:

Step 1. The user requests to reset their password

Step 2. The system requests the user to provide their email address

Step 3. The user enters their email address

Step 4. The system verifies the email address

Step 5. If the email address is valid, the system sends an account reset email to the user. The user should be able to receive the email quickly.

Extensions:

Extension:

In step 5, if the email address is not valid, the system displays an error message and the user repeats step 3.

3.1.2 Connection

3.1.2.1 Use Case: Create group

Description: A user creates a group

Pre-condition: a user is already logged in

Post-condition: a group is created and the user is assigned as the group's owner

Main Success Scenario:

Step 1. The user requests to create a group

Step 2. The system displays a group creation form to the user. The group creation form should be intuitive for the user.

Step 3. The user enters the group details into the form. The group details include group name, group description and group visibility.

Step 4. The system creates the group

Step 5. The system requests the user to invite members to the group

Step 6. The user invites members to the group. This step is done by the 'invite group member' use case.

3.1.2.2 Use case: invite group member

Description: a group owner invites users to the group

Pre-condition: the group owner has accessed the group's page

Post-condition: a user is invited to the group

Main Success Scenario:

Step 1. The group owner requests to invite a user to the group

Step 2. The system displays the member invitation form to the group owner. The member invitation form should be intuitive for the user.

Step 3. The group owner enters a list of usernames of users to be invited. The group owners should be able to enter the usernames efficiently. The system should supports AJAX automated fill-in.

Step 4. If the usernames are correct, the system sends invitation emails to the users; else the system displays an error and the group owner repeats step 3.

3.1.2.3 Use case: Add friend

Description: a user adds a friend

Pre-condition: the user has accessed the other user's profile page

Post-condition: a friend invitation is sent to the other user

Main Success Scenario:

Step 1. The user requests add the other user as a friend

Step 2. The system verifies the user's eligibility to the other user as a friend

Step 3. If the user is eligible to add friends, then the system sends a friend request to the other user; else the system displays an error message to the user.

3.1.2.4 Use case: Remove friend

Description: a user removes a friend

Pre-condition: the user has accessed his profile page

Post-condition: a friend is removed from the user's friend list

Main Success Scenario:

Step 1. The user requests to display his list of friends

Step 2. The system displays the user's list of friends

Step 3. The user selects the friends to be removed. He/she should be able to select multiple friends in an efficient way.

Step 4. The system displays a confirmation message to the user

Step 5. The user confirms the deletion

Step 6. The system removes the friends from the list

Extension:

In step 6, if the user cancels the deletion, the use case is terminated.

3.1.2.5 Use case: Post comment

Description: a user posts a comment on a wall

Pre-condition: the user has logged in

Post-condition: a comment is posted on the wall

Main Success Scenario:

Step 1. If the user wants to post a new comment on the wall, then he requests to post a new comment

Step 2. The system displays a commenting space

Step 3. The user enters the comment

Step 4. The user submits the comment

Step 5. The system stores the comment

Extension:

In step 1, if the user wants to post a reply to an existing comment, he requests to post a reply comment and the use case resumes at step 2.

3.1.2.6 Use case: Remove comment

Description: a user removes a comment

Pre-condition: the user has logged in and the user is permitted to remove the comment

Post-condition: the comment is removed

Main Success Scenario:

Step 1. The user requests to remove the comment

Step 2. The system displays a confirmation message

Step 3. The user confirms the deletion

Step 4. The system removes the comment from the wall

3.1.2.7 Use case: Update status

Description: a user updates his status

Pre-condition: the user has logged in

Post-condition: a new status is added

Main Success Scenario:

Step 1. The user requests to update the status

Step 2. The system displays a status space

Step 3. The user enters the status

Step 4. The user submits the status

Step 5. The system stores the status

3.1.2.8 Use case: Filter newsfeed

Description: a user filters his newsfeed

Pre-condition: the user has accessed his newsfeed

Post-condition: the newsfeed's content is filtered

Main Success Scenario:

Step 1. The user requests to filter the newsfeed

Step 2. The system provides the user with the filtering options

Step 3. The user selects filtering options. The filtering options should be easy to understand for typical Internet users.

Step 4. The system displays a confirmation message

Step 5. The user confirms the selection

Step 6. The system updates the newsfeed settings.

Extension:

In step 5, if the user cancels the selection, step 3 is repeated.

3.1.2.9 Use case: Change privacy settings

Description: a user filters his privacy settings

Pre-condition: The user has accessed his profile page

Post-condition: the privacy settings are changed

Main Success Scenario:

Step 1. The user requests to change the privacy settings

Step 2. The system provides the user with the list of privacy setting options. The setting options should be easy to understand for typical Internet users.

Step 3. The user selects privacy setting options

Step 4. The system displays a confirmation message

Step 5. The user confirms the selection

Step 6. The system updates the privacy settings.

Extension:

In step 5, if the user cancels the selection, step 3 is repeated.

3.1.2.10 Use case: Follow entity

Description: a user follows an entity

Pre-condition: The user has logged in

Post-condition: the user follows the entity

Main Success Scenario:

Step 1. If the user is at his profile page, he requests to search for an entity. The followable entities include users, questions and topics.

Step 2. The system displays a search form. The search form should be easy to use.

Step 3. The user enters keywords

Step 4. The system displays the list of matching entities

Step 5. The user selects an entity

Step 6. The user requests to follow the entity

Step 7. The system stores the following

Extension:

In step 1, if the user is at the entity's page, the use case continues at step 4.

3.1.3 Forum

3.1.3.1 Use case: post question

Description: a user posts a question

Pre-condition: the user has logged in

Post-condition: a question is posted

Main Success Scenario:

Step 1. The user requests to post a new question

Step 2. The system displays a question form

Step 3. The user enters the question

Step 4. The user submits the question

Step 5. The system stores the question

3.1.3.2 Use case: post answer

Description: a user posts an answer

Pre-condition: the user has logged in and the user is at a question's page

Post-condition: an answer is posted

Main Success Scenario:

Step 1. The user requests to post an answer for the question

Step 2. The system displays an answer form

Step 3. The user enters the answer

Step 4. The user submits the answer

Step 5. The system stores the answer

3.1.4 Communication

3.1.4.1 Use case: send message

Description: a user sends a message to other users

Pre-condition: The user has logged in

Post-condition: a message is sent to other users

Main Success Scenario:

Step 1. If the user is at his profile page, he requests to search for his friends

Step 2. The system displays a search form. The search form should be easy to understand.

Step 3. The user enter keywords

Step 4. The system displays a list of matching friends

Step 5. The user selects friends to be sent messages. The user should be able to select multiple friends efficiently.

Step 6. The user repeats step 3 to step 5 until he finds all needed friends

Step 7. The user requests to send a message to the friends

Step 8. The system delivers the message to the friends. The system should deliver the message immediately.

Extension:

In step 1, if the user is at another user's page, the use case continues at step 7.

3.1.5 Article writing

3.1.5.1 Use case: Write travel article

Description: a user writes a travel article

Pre-condition: The user has logged in

Post-condition: The new travel article is stored

Main Success Scenario:

Step 1. The user requests to create a travel article.

Step 2. System displays the article creation form to the user. The article creation form should be easy to use.

Step 3. The user enters the article content and submits it

Step 4. System validates the article content. The system should validate the article content in less than 1 second.

Step 5. If the article content is valid, the system stores the article into the database and displays confirmation message to the user; else the system displays the errors to the user and the user repeats step 3.

3.1.6 Review

3.1.6.1 Use case: Write review

Description: a user writes a review

Pre-condition: The user has logged in

Post-condition: The new review is stored

Main Success Scenario:

Step 1. The user selects to create a review.

Step 2. System prompts the user to select a review category.

Step 3. The user selects review category. The list of categories includes hotel, attraction and tour.

Step 4. System displays the suitable review creation form to the user. The review creation form should be familiar with typical Internet users.

Step 5. The user enters the review content and submits it

Step 6. The system validates the review. The system should validate the review in less than 1 second.

Step 7. If the review content is valid, the system stores the review into the database and displays confirmation message to the user; else the system displays the errors to the user and the user repeats step 5.

3.1.7 Trip planning

3.1.7.1 Use case: Create travel plan

Description: A user creates a travel plan

Pre-condition: The user has logged in

Post-condition: A travel plan is created

Main Success Scenario:

Step 1. The user requests to create a travel plan

Step 2. The system displays the travel plan creation form to the user

Step 3. The user enters the travel plan details and submits them

Step 4. System validates the travel plan

Step 5. If the travel plan is valid, the system stores it; else the system displays the errors to the user and the user repeats step 3.

3.1.7.2 Use case: Invite to trip

Description: A user invites his friends to a trip

Pre-condition: The user has logged in and the user has accessed the trip's page

Post-condition: An invitation is sent to the friends

Main Success Scenario:

Step 1. The user requests to search for his friends

Step 2. The system displays a search form. The search form should be easy to use.

Step 3. The user enters keywords

Step 4. The system displays a list of matching friends

Step 5. The user selects friends to be invited. The user should be able to select multiple friends efficiently,

Step 6. The user repeats step 3 to step 5 until he finds all needed friends

Step 7. The user requests to invite these friends to the trip

Step 8. The system delivers the invitation to the friends.

3.1.8 Travel expert

3.1.8.1 Use case: Register as travel expert

Description: A user registers as a travel expert

Pre-condition: The user has logged in

Post-condition: A travel expert request is sent to the administrator

Main Success Scenario:

Step 1. The user selects to send a travel expert request

Step 2. The system displays a travel expert request form to the user

Step 3. The user enters the request details into the form and submits it

Step 4. If the request's details are valid, the system stores it and displays a confirmation message to the user; else the system displays the errors to the user and the user repeats step 3

4 Other Non-functional Requirements

4.1 Performance Requirements

The system should be fast. It should respond quickly to users' requests. The system should also provide quick information validation. In addition, the system should minimize the delay when sending instant messages.

4.2 Safety and Security Requirements

The system should be secure. It should support user authentication. The privacy of users should also be protected. Users should be able to update their privacy settings easily. Moreover, to prevent virus attacks, the system should provide anti-virus feature.

4.3 Software Quality Attributes

The system should be easy to use. Specifically, users should not find it hard to create reviews, travel articles or travel plans. The management of profiles should be intuitive and familiar for users. Moreover, the system should be always available. Users shall be able to access the system via the Internet at all time.

Appendix A2 Artifact Specification Boilerplates

Notes:

- ( ) → optional
- […|…] → alternatives

Business goal boilerplates:

B1. <transitive action verb><object> ((for) <beneficiary>) ([with | to]) <reference> ([in | at | on] <location>) (for <purpose>)

B2. <transitive action verb><object> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) (for <purpose>)

B3. <intransitive action verb> ((for) <beneficiary>) ([with | to | of]) <reference> ([in | at | on] <location>) (for <purpose>)

B4. <intransitive action verb> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) (for <purpose>)

B5. <possessive verb><object> ([in | at | on] <location>) (for <purpose>)

B6. Be <quality> (([with | to]) <reference>) ([in | at | on] <location>) (for <purpose>)

B7. Be <quality> ((for) <beneficiary>) ([in | at | on] <location>) (for <purpose>)

Functional Feature Goal boilerplates:

FF1. (<agent> [shall (not)| shall (not) be able to]) <transitive action verb> <object> (for) <beneficiary> ([in | at | on] <location>) (from <source>) (to <destination>) (<functional manner>)

FF2. (<agent> [shall (not)| shall (not) be able to]) <transitive action verb> <object> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) (<functional manner>)

FF3. (<agent> [shall (not)| shall (not) be able to]) <transitive action verb> <object> ([with | to]) <reference> ([in | at | on] <location>) (<functional manner>) (for <purpose>)

FF4. (<agent> [shall (not)| shall (not) be able to]) <transitive action verb> <object> (with <company>) ([in | at | on] <location>) (<functional manner>) (for <purpose>)

FF5. (<agent> [shall (not)| shall (not) be able to]) <intransitive action verb> (for) <beneficiary> ([in | at | on] <location>) (from <source>) (to <destination>) (<functional manner>) (for <purpose>)

FF6. (<agent> [shall (not)| shall (not) be able to]) <intransitive action verb> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) (<functional manner>)

FF7. (<agent> [shall (not)| shall (not) be able to]) <intransitive action verb> ([with | to]) <reference> ([in | at | on] <location>) (<functional manner>) (for <purpose>)

FF8. (<agent> [shall (not)| shall (not) be able to]) <intransitive action verb> (with <company>) ([in | at | on] <location>) (<functional manner>) (for <purpose>)

Functional Service Goal boilerplates

FS1. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>) ([<event> | <condition>])

FS2. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>) ([<event> | <condition>])

FS3. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> ([with | to]) <reference> ([in | at | on] <location>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>) (for <purpose>) ([<event> | <condition>])

FS4. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> (with <company>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>) (for <purpose>) ([<event> | <condition>])

FS5. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>) (for <purpose>) ([<event> | <condition>])

FS6. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>) ([<event> | <condition>])

FS7. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> ([with | to]) <reference> ([in | at | on] <location>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>) (for <purpose>) ([<event> | <condition>])

FS8. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> with <company> ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>) (for <purpose>) ([<event> | <condition>])

Non-functional Product Goal boilerplates

NP1. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) (<manner>)

NP2. <positioner> shall (not) <possessive verb> <object> ([in | at | on] <location>) (for <purpose>)

NP3. <agent> shall (not) be <quality> ((for) <beneficiary>) ([in | at | on] <location>) (<manner>) (for <purpose>)

Non-functional Feature Goal boilerplates

NF1. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([for | in] <duration>) (<manner>) (for <purpose>)

NF2. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) ([for | in] <duration>) (<manner>) (for <purpose>)

NF3. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> ([with | to]) <reference>) ([in | at | on] <location>) ([for | in] <duration>) (<manner>) (for <purpose>)

NF4. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> with <company> ([in | at | on] <location>) ([for | in] <duration>) (<manner>) (for <purpose>)

NF5. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([for | in] <duration>) (<manner>) (for <purpose>)

NF6. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) ([for | in] <duration>) (<manner>) (for <purpose>)

NF7. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> ([with | to]) <reference>) ([in | at | on] <location>) ([for | in] <duration>) (<manner>) (for <purpose>)

NF8. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> with <company> ([in | at | on] <location>) ([for | in] <duration>) (<manner>) (for <purpose>)

NF9. <positioner> shall (not) <possessive verb> <object> ([in | at | on] <location>) ([for | in] <duration>) (<manner>) (for <purpose>)

NF10. <positioner> shall (not) be <quality> ((for) <beneficiary>) ([in | at | on] <location>) ([for | in] <duration>) (<manner>) (for <purpose>)

Non-functional Service Goal boilerplates

NS1. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (<manner>) (for <purpose>) ([<event> | <condition>])

NS2. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (<manner>) (for <purpose>) ([<event> | <condition>])

NS3. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> ([with | to]) <reference> ([in | at | on] <location>) ([by | with | via] <means>) ([for | in] <duration>) (<manner>) (for <purpose>) ([<event> | <condition>])

NS4. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> (with <company>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (<manner>) (for <purpose>) ([<event> | <condition>])

NS5. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (<manner>) (for <purpose>) ([<event> | <condition>])

NS6. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (<manner>) (for <purpose>) ([<event> | <condition>])

NS7. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> ([with | to]) <reference> ([in | at | on] <location>) ([by | with | via] <means>) ([for | in] <duration>) (<manner>) (for <purpose>) ([<event> | <condition>])

NS8. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> (with <company>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (<manner>) (for <purpose>) ([<event> | <condition>])

NS9. <positioner> shall (not) <possessive verb> <object> ([in | at | on] <location>) ([for | in] <duration>) (<manner>) (for <purpose>) ([<event> | <condition>])

NS10. <positioner> shall (not) be <quality> ([in | at | on] <location>) ([for | in] <duration>) (<manner>) (for <purpose>) ([<event> | <condition>])

Non-functional Use Case Constraint boilerplates

NU1. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<manner>) (for <purpose>) ([<event> | <condition>])

NU2. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<manner>) (for <purpose>) ([<event> | <condition>])

NU3. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> ([with | to]) <reference> ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<manner>) (for <purpose>) ([<event> | <condition>])

NU4. <agent> [shall (not)| shall (not) be able to] <transitive action verb> <object> (with <company>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<manner>) (for <purpose>) ([<event> | <condition>])

NU5. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<manner>) (for <purpose>) ([<event> | <condition>])

NU6. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<manner>) (for <purpose>) ([<event> | <condition>])

NU7. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> ([with | to]) <reference> ([in | at | on] <location>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<manner>) (for <purpose>) ([<event> | <condition>])

NU8. <agent> [shall (not)| shall (not) be able to] <intransitive action verb> (with <company>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<manner>) (for <purpose>) ([<event> | <condition>])

NU9. <positioner> shall (not) <possessive verb> <object> ([in | at | on] <location>) ([for | in] <duration>) (<manner>) (for <purpose>) ([<event> | <condition>])

NU10. <positioner> shall (not) be <quality> ([in | at | on] <location>) ([for | in] <duration>) (<manner>) (for <purpose>) ([<event> | <condition>])

Data Constraint boilerplates

D1. <positioner> shall <possessive verb> <object>

D2. <positioner> shall (not) be <quality>

Use Case Step boilerplates

US1. <agent> <transitive action verb> <object> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([for | in] <duration>) (for <purpose>) ([<event> | <condition>])

US2. <agent> <transitive action verb> <object> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) ([for | in] <duration>) ([<event> | <condition>])

US3. <agent> <transitive action verb> <object> ([with | to]) <reference> ([in | at | on] <location>) ([for | in] <duration>) (for <purpose>) ([<event> | <condition>])

US4. <agent> <intransitive action verb> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([for | in] <duration>) (for <purpose>) ([<event> | <condition>])

US5. <agent> <intransitive action verb> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) ([for | in] <duration>) ([<event> | <condition>])

US6. <agent> <intransitive action verb> ([with | to]) <reference> ([in | at | on] <location>) ([for | in] <duration>) (for <purpose>) ([<event> | <condition>])

Use Case Condition boilerplates

UC1. <agent> ([do not | does not | did not]) <transitive action verb> <object> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) (<functional manner>) (for <purpose>)

UC2. <agent> ([do not | does not | did not]) <transitive action verb> <object> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) (<functional manner>) (for <purpose>)

UC3. <agent> ([do not | does not | did not]) <transitive action verb> <object> ([with | to]) <reference> ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) (<functional manner>) (for <purpose>)

UC4. <agent> ([do not | does not | did not]) <transitive action verb> <object> (with <company>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) (<functional manner>) (for <purpose>)

UC5. <agent> ([do not | does not | did not]) <intransitive action verb> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) (<functional manner>) (for <purpose>)

UC6. <agent> ([do not | does not | did not]) <intransitive action verb> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) (<functional manner>) (for <purpose>)

UC7. <agent> ([do not | does not | did not]) <intransitive action verb> ([with | to]) <reference> ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) (<functional manner>) (for <purpose>)

UC8. <agent> ([do not | does not | did not]) <intransitive action verb> (with <company>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) (<functional manner>) (for <purpose>)

UC9. <positioner> ([do not | does not | did not]) <possessive verb> <object> ([in | at | on] <location>) (<manner>) (for <purpose>)

UC10. <positioner> ([is not | are not | was not | were not]) <quality> ([in | at | on] <location>) (<manner>) (for <purpose>)

Condition boilerplates

C1. <agent> [do not | does not | did not] <transitive action verb> <object> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>)

C2. <agent> [do not | does not | did not] <transitive action verb> <object> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>)

C3. <agent> [do not | does not | did not] <transitive action verb> <object> ([with | to]) <reference> ([in | at | on] <location>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>) (for <purpose>)

C4. <agent> [do not | does not | did not] <transitive action verb> <object> (with <company>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>) (for <purpose>)

C5. <agent> [do not | does not | did not] <intransitive action verb> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>) (for <purpose>)

C6. <agent> [do not | does not | did not] <intransitive action verb> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>)

C7. <agent> [do not | does not | did not] <intransitive action verb> ([with | to]) <reference> ([in | at | on] <location>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>) (for <purpose>)

C8. <agent> [do not | does not | did not] <intransitive action verb> with <company> ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) ([for | in] <duration>) (every <quantity> <time unit>) (<functional manner>) (for <purpose>)

C9. <positioner> ([do not | does not | did not]) <possessive verb> <object> ([in | at | on] <location>) ([for | in] <duration>) (<functional manner>) (for <purpose>)

C10. <positioner> ([is not | are not | was not | were not]) <quality> ([in | at | on] <location>) ([for | in] <duration>) (<functional manner>) (for <purpose>)

Event boilerplates

E1. <agent> <transitive action verb> <object> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) (<temporal manner>)

E2. <agent> [do not | does not | did not] <transitive action verb> <object> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) ([by | with | via] <means>) (<temporal manner>)

E3. <agent> [do not | does not | did not] <transitive action verb> <object> ([with | to]) <reference> ([in | at | on] <location>) ([by | with | via] <means>) (<temporal manner>) (for <purpose>)

E4. <agent> [do not | does not | did not] <transitive action verb> <object> (with <company>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) (<temporal manner>) (for <purpose>)

E5. <agent> [do not | does not | did not] <intransitive action verb> ((for) <beneficiary>) ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) (<temporal manner>) (for <purpose>)

E6. <agent> [do not | does not | did not] <intransitive action verb> ([with | to]) <reference> (for) <beneficiary> ([in | at | on] <location>) ([by | with | via] <means>) (<temporal manner>)

E7. <agent> [do not | does not | did not] <intransitive action verb> ([with | to]) <reference> ([in | at | on] <location>) ([by | with | via] <means>) (<temporal manner>) (for <purpose>)

E8. <agent> [do not | does not | did not] <intransitive action verb> with <company> ([in | at | on] <location>) (from <source>) (to <destination>) ([by | with | via] <means>) (<temporal manner>) (for <purpose>)

E9. <positioner> ([do not | does not | did not]) <possessive verb> <object> ([in | at | on] <location>)

E10. <positioner> ([is not | are not | was not | were not]) <quality> ([in | at | on] <location>) (for <purpose>)

Appendix A3 Correspondence between GUIMeta and Existing GUIM Approaches

The correspondence between GUIMeta's concepts and those in the existing Goal and Use Case Integrated Modeling approaches

Approach	Related Approach' concept	GUIMeta's Corresponding Concept(s)
Rolland et al. [133] & Kim et al. [76]	Business goal: ultimate purpose of the system	BG
	Design goal: possible manners of fulfilling a business goal	FFG
	Service goal: possible manners of providing services to fulfill design goals	FSG
Gorschek & Wohlin [55]	Product level requirement: product strategies	FFG
	Feature level requirement: high-level feature that the product supports	FFG
	Functional level requirement: action that users can perform	FSG
Lee et al. [91]	Rigid goal: goal that must be completely satisfied	FFG, FSG
	Soft goal: goal that can be partially satisfied	BG, NPG, NFG, NSG
	Actor-specific goal: actors' objectives with the system	FFG, NFG, FSG, NSG with agent is an actor
	System-specific goal: requirements on services that system provides	NPG, NFG, NSG with agent is a system or system component
	Functional goal	FFG, FSG
	Non-functional goal	NPG, NFG, NSG
	Original goal: intersection of rigid, actor-specific and function goal	FFG, FSG with agent is an actor
	Cooperative and conflict relationships between goals	Require/refine and exclude relationships respectively
	Satisfied/denied relationships	Satisfied/denied relationships can

	between an original goal and a use case	be represented in GUIMeta through operationalize, refine, require and exclude relationships. For instance, if G1 is operationalized by U1 and G1 refines G2, then G1 and G2 are both satisfied by U1. If G3 excludes G1, then G3 is denied by U1.
Cockburn [26]	Summary goal: system objectives	FFG
	User goal: user's task	FSG
	Sub-function goal: describe user activities	Use case step
Watahiki and Saeki [169]	Business goal, Feature goal and Service goal which are derived from Rolland et al. [133]	BG, FFG, and FSG respectively
	Constraint	NPG, NFG, NSG, NUCC
	Constrain link	Constraint relationship
van Lamsweerde & Willemet [164]	'Maintain' goal, 'Achieve' goal, 'Avoid' goal	These are not supported by GUIMeta, since we focus on abstraction-based goal classification
	AND, OR decomposition	AND, OR refinement relationship
Santander and Castro [141]	Strategic dependency modeling concepts: to model i.e., an actor is dependent on another actor with a certain goal	Not supported by GUIMeta. However, GUIMeta can be extended to support such modeling
	Goal: a condition or state of affairs in the world that the actor would like to achieve	BG, FFG
	Task: specify a particular way of doing something	FFG, FSG
	Soft goal: similar to goal, but the satisfaction is subject to interpretation	NPG, NFG, NSG
	Resource: an entity that is not considered problematic by the actor	Not supported. However, can be added to GUIMeta as an extension
	Goal-task link Soft goal-task link Soft goal-Soft goal link	Refinement relationships

	Task-Task link	
	Resource-task link	Not supported since resources are not supported. However, can be added to GUIMeta
Supakkul and Chung [154]	Non-functional requirement	NPG, NFG, NSG
	Actor association point	These are not directly supported by GUIMeta since the notion of actor is not used. They can be indirectly supported by declaring non-functional goals, non-functional categories, and constrain relationship.
	Use case association point	
	Actor-use case association	
	System boundary association point	
Kim et al. [77]	Structure view: it's about capturing business descriptions. A business description is the boundary of system and business, specified by activity, agent, event, object, candidate goal, and structured object	Not supported by GUIMeta since the business description components (except goals) are not included in GUIMeta. However, they can be added as extensions
	Abstraction view: capture refinement relationships of artifacts	Achieved by specifying goals and refinement relationship
	Function view: capture system-user interaction	Achieved by using use cases
	Quality view: goals should be connected to quality attributes they need to satisfy	Achieved by specifying non-functional goals (NPG, NFG, NSG) and constrain relationship

Appendix A4 Extraction Rule Creation and Quality Analysis

Extraction Rule Creation

In this appendix, we discuss the steps taken to write new extraction rules based on our defined rule categories and syntax. The following sentence (referred to as *original sentence*) is used to provide examples for illustrating the extraction rule creation techniques.

> *"The system will be designed to improve the quality of travel planning by facilitating the communication between travelers"*

Step 1: Identify ignorable parts or extractable artifact(s) and relationship(s) and their rationales. This step focuses on identifying the possible artifacts and relationships that can be extracted from a given sentence and/or which part of the sentence should be ignored. This also includes the analysis for the reasons behind such extraction and/or ignorance. Based on such analysis, the initial ideas about how necessary extraction rules should be written are generated.

From the original sentence, the following information is identified:

- Ignored part: *"The system will be designed."* This phrase should be ignored since in this context, it does not contribute to specify any business objectives, functionalities or constraints of a system. It is instead an introductory phrase for another phrase following it *("to improve the quality of travel planning"*). Note that in another context, it is not guaranteed that the same phrase should also be ignored. For instance, in a sentence such as *"The system will be designed in accordance to the HVCA standard,"* it is obviously an important part of a standard compliance goal. Therefore, what makes the phrase unimportant in this case is its use with a following verb phrase. In other words, it is concluded that if the phrase is used in the context of *"The system will be designed to do something,"* then it should be ignored. The sentence would then become: *"improve the quality of travel planning by facilitating the communication between travelers."*

- Extracted artifacts and relationships: there are two goals that can be extracted from this sentence, which are *"Improve the quality of travel planning"* (G1) and *"Facilitate the communication between travelers"* (G2). G2 refines G1, or G2 is a sub-goal of G1. The reason for this is that, G1 and G2 both describe important details about a system (a business objective a functionality respectively). In addition, the refinement relationship between these goals is recognized since the sentence has the structure of *"do something by doing something"* (*verb + object + by +verb_ing + object*).

This information suggests that two extraction rules would need to be generated to handle this single sentence. One helps ignore the unimportant phrase *"The system will be designed"* (called R1) and another helps reveal the artifacts G1 and G2 and the refinement relationships between them (called R2).

Step 2: Create rules based on the rule ideas. In the previous step, the initial ideas for rules are generated. In this step, the goal is to transform these ideas into actual rules classified and written in accordance with our rule categories and syntax. The following sub-steps are taken during this process:

det(system-2, The-1)	**nsubjpass(designed-5, system-2)**
aux(designed-5, will-3)	**auxpass(designed-5, be-4)**
root(ROOT-0, designed-5)	**xcomp(designed-5, improve-7)**
aux(improve-7, to-6)	dobj(improve-7, quality-9)
det(quality-9, the-8)	nn(planning-12, travel-11)
det(communication-16, the-15)	prep_of(quality-9, planning-12)
dobj(facilitating-14, communication-16)	prep_between(communication-16, travelers-18)
prepc_by(improve-7, facilitating-14)	

Figure A4-1. Dependency Parse Tree of the Original Sentence

- **Step 2.1: Obtain the dependency parse tree.** Extraction rules are specified based on the grammatical dependencies between words in a sentence. Therefore, obtaining dependency parse trees is the first important step to transform rule ideas into rule specifications. In addition, based on the obtained dependency parse tree, it is possible to locate the dependencies related to the phrases (or part

of sentence) involved in the rule ideas. The dependency tree is also an important input for choosing the categories of rules in the next step. Figure A4-1 shows the dependency parse tree of the original sentence. The bold dependencies are those relevant to rule R1 and the italic dependencies are related to rule R2.

- **Step 2.2: Determine the categories of the rule(s) being created.** It is critical to determine the types of the rules being created since different types of rules have different syntaxes and characteristics. For instance, the first rule in our example is for ignoring part of a sentence. Therefore, it could be either an ignorance rule or a navigation rule. According to Figure A4-1, `designed-5` is the root of the phrase to be ignored *("The system will be designed"*). However, it is also the root of the entire tree. Thus, it is unsuitable to specify it as an ignorance rule since an ignorance rule requires a primary action to have the root of the phrase to be ignored as its variable (`ignore(designed-5)`). However, this leads to the entire tree to be ignored since `designed-5` is also the root of the tree. Therefore, it should be specified as a navigation rule. In addition, the second rule is intended to extract both artifacts and relationships from a sentence. A relationship extraction rule would be suitable for this purpose.
- **Step 2.3. Write the rule(s).** Once the categories have been determined, the initial versions of rules can be generated. In this initial version, the matching dependencies identified in step 2.1 become the matching condition of the rules. Actions are added according to the requirements of the rule categories and what are expected to by done by the rules. If there is any word that appears in more than one dependency and action, a variable is created for it. For each word that appears only in one of the dependencies, it is specified inside the dependency with a pair of surrounding curly brackets ("{}"). Table A4-1 presents the initially generated rules. It can be seen that the matching condition in these rules corresponds to the highlighted dependencies in Figure A4-1. The words that appear multiple times in the matching conditions and actions are defined as variables (i.e., X, Y, Z in rule R1 and X, Y in rule R2).

Table A4-1. Initially Generated Extraction Rules

Rule R1	X={designed} Y/VB Z={system} root(X) det(Z, {the}) aux(X, {will}) nsubjpass(X, Z) auxpass(X, {be}) xcomp(X, Y) -> root(Y)
Rule R2	X/VB Y/VBG root(X) prepc_by(X, Y) -> sub_goal(goal(Y), goal(X))

Step 3: Generalize the rules. Extraction rules should be designed in a way that maximizes its applicability in as many as possible cases to support the model extraction process. In the previous step, initial versions of extraction rules are generated based on the specific context considered. In this step, the goal is to generalize such rules to extend its applicability in other similar contexts. This involves a deeper analysis on the original sentence and initial rules to verify whether those rules can be applied (or can be extended/modified to be applicable) in different variances of the original sentence. The variances of a sentence can be identified by replacing certain words in such a sentence with their alternatives (i.e., replacing *"the"* with *"a," "this," "that"*), or changing the structure of the sentence while keeping its meaning unchanged (i.e., rewrite a passive-voice sentence in active voice). Such an analysis enables the generalization of the rules to make them applicable to different variances of a sentence (i.e., no matter "the" or "this" or "that" is used, the rule is still applicable). In case it is identified that a single rule cannot handle all variances of a sentence, then a new rule (most of the time it is similar to the previously created rule, except some minor changes) needs to be created. This entire step can be viewed as a brainstorming process to identity as many as widely applicable rules based on a single original sentence.

Table A4-2. Generalized Extraction Rules

Rule R1	X={design|build|aim|intend|target} Y/VB root(X) nsubjpass(X, {system|project|software|application}) auxpass(X, {be}) aux(Y, {to}) xcomp(X, Y) -> root(Y)
Rule R1*	X={design|develop|build} Y/VB root(X) dobj(X, {system|application}) aux(Y, {to}) xcomp(X, Y) -> root(Y);
Rule R2	X/VB Y/VBG root(X) prepc_by(X, Y) -> sub_goal(goal(Y), goal(X))

Table A4-2 shows the generalization results of the rules created in the previous step. Regarding rule R1, it is identified that if the ignored phrase in the original sentence is rewritten as *"The system **is** designed to…"* (*"will be"* is replaced by *"is"*), or *"**This** system **shall** be intended to…"* (*"this"* replaces *"the," "shall"* replaces *"will," "intended"* replaces *"designed"*), its role is unchanged and should still be ignored. In addition, the situation is also not changed if *"system"* is updated to *"software,"* (*"The software is designed to…"*). Based on this observation, the initial R1 rule is modified to accommodate these variances. The dependency `det(z, {the})` is removed since regardless of what determiner is used, the role of the phrase is unchanged. Similarly, the dependency `aux(X, {will})` is removed to allow the rule to handle all variances of the to-be verb allowable in this phrase (i.e., *"is," "are," will be," "shall be"*).

When considering restructuring the entire original sentence in active voice, it is found that the generated rule R1 is no longer applicable since it has the matching condition `nsubjpass(X,{system|project|software})` and `auxpass(X, {be})` that can only be met by an passive voice sentence. In addition, while the phrase *"The system is*

intended to…" is commonly used, it is not usual to write *"Intend the system to…."* Therefore, a new (but similar) rule should be created to handle the case the original sentence is written in active voice. Such rule is showed as rule R1* in Table A4-2. Regarding rule R2, it is already quite general since its variables do not have values. Therefore, no variance can be generated.

In GUEST, the values of rule variables are recommended to be in the standard form of the words they refer to if possible. For instance, *"design"* is used instead of *"designs"* or *"designed,"* or *"application"* is used instead of *"applications."* This allows the rules to be general enough to handle different variances (i.e., cases in which the plural form is used is handled in the same way as when a singular form is used).

Extraction Rule Quality Analysis

Extraction rules play the key role in our approach to goal-use case model extraction. The accuracy of extraction result is directly affected by the quality of extraction rules. Therefore, it is critical to ensure the high quality of these rules. GUEST offers a number of features to verify the correctness, overlap and consistency of them.

Correctness

GUEST provides a syntax checker based on regular expression to guarantee that rules are specified in conformance to our defined syntax. For instance, there must be a matching condition and at least one action in a rule. In addition, a rule must conform to the requirements of the category it belongs to. For instance, a goal ignorance goal must have its primary action as an *"ignore"* action. The syntax checker is integrated with the rule editor to ensure rules to be specified correctly.

Redundancy

Since the number of extraction rules may evolve over times, or they may be developed by different users, managing the set of rules is challenging. It is thus inevitable that some rules may be redundant with some other. For instance, the rule R_1 has a set of dependencies D_1 as its matching condition and a set of actions A_1. Similarly, the rule R_2 has a set of dependencies D_2 and a set of actions A_2. Given D_1 is a subset of D_2 and A_1 is the same as A_2, then rule R_2 is redundant since if a sentence matches R_1, it definitely matches R_2. Redundancy does not affect the accuracy of the extraction. However, it

may create extra processing overhead since a sentence is found matching different (redundant) rules that then involves the execution of all these rules. GUEST is able to automatically identify redundancy within a given set of rules by comparing these rules' matching conditions.

Consistency

Consistency is another important property of extraction rules. In GUEST, we define inconsistency as the case in which there is a rule whose matching condition is equivalent to the matching condition of another rule while their sets of actions are not equal to each other. Inconsistency is a serious issue since the same sentence may match both rules with different characteristics and behaviors. This implies that at least one of these rules is not correctly specified. Similar to redundancy detection, GUEST automatically identifies rule inconsistencies by comparing rules by their matching conditions and actions.

Overlap

We define overlaps between extraction rules as the cases in which there is a rule whose matching condition is a subset of the matching condition of another rule, and these rules are not redundant or inconsistent. Overlap is often not a problem. However, overlap calculation it may be helpful as means to review or investigate the set of extraction rules. GUEST is able to calculate and display the degree of overlapping between each pair of rules based on the proportion of dependencies shared between them.

Appendix A5 Algorithms for Goal and Use Case Extraction

This appendix provides the high-level algorithms for the process of using extraction rules for identifying goals and use cases from text.

Goal Extraction

Figure A5-1 provides the algorithm for goal extraction.

```
Function extractGoals(sen)
   Input: sen → the sentence being considered for extraction
   Output: an extraction report that stores the association
   between a list of used extraction rules and the final
   extracted dependency collections, alternative extraction
   results, and any problems that occur during extraction.
   rootDepTree = getDependencyTree(sen)
   rootDC = getDependencyCollection(rootDepTree) //dependency
   collection (DC) is a data structure designed to wrap a
   dependency tree and record its relevant information, i.e.,
   its relationship with other dependency collections and the
   list of extraction rules used to derive it from the
   original sentence. Each rule execution outcome contains
   one or more dependency collections. Each non-empty DC
   corresponds to an extracted goal during the extraction.
   rootDC contains only dependencies obtained from sen.
   extractionReport = an empty extraction report
   extractGoals(rootDC, extractionReport) //search for
   matching rules, generate rule execution results and create
   reports for problems detected
   return extractionReport
End Function
```

```
Function ExtractGoals(rootDC, extractionReport)
   processedDCs = new Stack
   processedDCs→PUSH(rootDC)
   ExtractGoal(extractionReport, processedDCs)
END Function
```

```
Function ExtractGoals(extractionReport, processedDCs)
   WHILE !empty(processedDCs)
      currentDC = processedDCs→POP
      processDC(currentDC, extractionReport, processedDCs)
   END WHILE
END Function
```

```
Function processDC(currentDC, processedDCs, extractionReport)
   outputs = extractWithIgnoranceRules(currentDC) //list of
   extraction outputs, each for a matched rule. An output
   contains two dependency collections and their
   relationship. The dependency collections can be NULL
   depending on the matched rule.
   IF outputs != NULL //in case one or more matching
   ignorance rules are found and executed
      recordOutputs(outputs, processedDCs, extractionReport)
      //record the outputs into extraction report, update the
      processesdDCs stack with the new DCs from the outputs.
      In case multiple rules were matched, alternative
      extractions would be recorded.
   ELSE
      outputs = extractWithNavigationRules(currentDC)
      IF outputs = NULL
         outputs = extractWithRelationshipRules(currentDC)
         IF outputs = NULL
            outputs = extractWithSplittingRules(currentDC)
            IF outputs != NULL
               recordOutputs(outputs, processedDCs,
               extractionReport)
            END IF
         ELSE
            recordOutputs(outputs, processedDCs,
            extractionReport)
         END IF
      ELSE
         recordOutputs(outputs,
         processedDCs,extractionReport)
      END IF
   END IF
END Function
```

Figure A5-1. Algorithm for Goal Extraction

Use Case Extraction

Figure A5-2 provides the algorithms for use case extraction.

```
Function extractUseCase(text)

    Input: text → textual specification of the entire use
    case

    Output: an extraction report that stores a set of
    extracted dependency collections (DC). Each DC
    corresponds to an artifact. The report also captures the
    extracted relationships between these DCs, alternative
    extraction results and any problems that occur during the
    extraction. The extracted DCs and relationships will then
    be used to create a use case and its components, possibly
    use case goals and constraints and relationships between
    these artifacts.

    extractionReport = new empty extraction report

    usecase = new empty use case

    extractionReport.addUseCase(usecase)

    currentComponent = NULL //store the currently considered
    use case component

    lines = split(text) //split text by line breaks

    FOR EACH line: lines

       component = getComponent(line) //get component from
       the line

       IF component = NULL //in case the line is not a
       component indicator, then it is part of the
       specification of the currently considered component

          processComponent(currentComponent, line, usecase,
          extractionReport) //process the component
          specification. Depending on the component, suitable
          extraction rules are used. The algorithm for
          finding and executing matching rules is similar to
          that of goal extraction.

       ELSE

          currentComponent = component //update the currently
          considered component

       END IF

    END FOR

    return extractionReport

END Function
```

Figure A5-2. Algorithm for Use Case Extraction

Appendix A6 Modifying Rule Creation and Quality Analysis

In this appendix, we discuss the steps taken to write new modifying rules based on our defined rule syntax. The following specification (referred to as *original specification*) is used to provide examples for illustrating the modifying rule creation techniques.

"Users are capable of creating reviews"

Step 1: Identify how the specification should be properly written. In this step, the focus is on determining if the original specification is improperly written and if that is the case, how it should be specified. In this example, it is decided that the original should be instead written as *"Users create reviews."* We refer to it as a *modified specification* in our discussion. The desired specification should be *"Users shall be able to create reviews."* However, the modal verb phrase (*"shall be able to"*) addition should be handled in the next step of the polishing process rather than by modifying rules.

Step 2: Identify the necessary modifications to the dependency tree. Modifying rules are aimed to make modifications in specifications based on their dependency tree structures. Therefore, in this step, both the original specification and modified specification are first parsed to obtain their dependency trees. The dependency trees are then compared to identify the necessary modifications that need to be made. Table 6-10 presents these dependency trees. We refer to them as *original dependency tree* and *modified dependency tree* respectively. It can be seen that the two dependency trees share a number of important details. In both trees, `'users'` is the nominal subject (denoted by the dependency `nsubj(X, users)` in which `X` is the root of a tree) and `'reviews'` is the direct object of `'create.'` From this observation, it is clear that in order to obtain the modified dependency tree, the following modifications should be made to the original tree. Firstly, the dependencies that are related to the phrase *"are capable of"* should be removed (the ones in *italic* in Table A6-1). Secondly, the node `creating-5` in the original dependency tree should be changed to its standard form (i.e., `create`) and set as the root of the new tree. Thirdly, a new dependency that specifies the `nsubj` relationship between `create` and Users-1 should be created.

Table A6-1. Dependency Trees of the Original and Modified Specifications

Original Dependency Tree	Modified Dependency Tree
nsubj(capable-3, Users-1) *cop(capable-3, are-2)* *root(ROOT-0, capable-3)* *prepc_of(capable-3, creating-5)* dobj(creating-5, reviews-6)	nsubj(create-2, Users-1) root(ROOT-0, create-2) dobj(create-2, reviews-3)

Step 3: Determine the necessary actions to facilitate the modifications. Based on the necessary modifications identified in step 2, we determine the actions that can facilitate such modifications. These actions must be from the set of actions we have defined for modifying rules. In this example, we first look for an action that can be used to remove the dependencies corresponding to the phrase *"are capable of."* There are two types of actions that can be used for this purpose: *dependency removal action* and *dependency collection removal action*. Since the dependencies that we want to remove are linked directly to `capable-3`, a dependency collection removal action with the origin as `capable-3` can be used. In addition, because `dobj(creating-5, reviews-6)` is the only dependency that should be remained after this removal, we only need to add one terminal point, which is `creating-5`. Therefore, the action should be `remove(capable-3, creating-5)`. In addition, `creating-5` should be transformed to its standard form (i.e., `create-5`) and set as the root of the new dependency tree.

Based on our set of defined actions, a *verb adding action* can be applied in this case. As discussed earlier in this section, such an action has the syntax of `new_verb(name, {value}, POS_tag_ref, POS_tag_subject, node_before, node_after, is_negated, is_root)`. Since `creating-5` is an existing node, we do not need to fill in the `{value}`, `node_before`, and `node_after` parameters. Moreover, the new POS tag of this node should be the same as that of the node `are-2` in the original dependency tree. Therefore, the value of `POS_tag_ref` is `are-2`. Furthermore, the value of `is_negated` and `is_root` is `false` and `true` respectively since no negation should be added and `creating-5` is to be set as the root of the new dependency tree. To sum up the action should be `new_verb(<name of this node>, null, T, null, null, null, false, true)`. The name of this node will be determined in

the next step. Regarding the last needed modification about creating a new dependency `nsubj(create-5, Users-1)`, a *dependency adding action* can be applied.

Step 4: Write the rule. After all actions are determined, it is time to write the rule to facilitate the modifications in the original specification. An important task in this step is identifying the conditions of such a rule. In this example, it can be seen that if a specification is in the form of *"someone is/are capable of doing something,"* then it should be instead written in the form of *"someone do/does something."* From this analysis and the determined actions in step 3, the initial version of the modifying rule can be created. Figure A6-1 presents this rule.

```
X={capable}
Y/{NN|NNS|NNP|NNPS}
Z/VBG
T={be}
nsubj(X, Y)
cop(X, T)
root(X)
prepc_of(X, Z)
-> new_verb(Z, null, T, null, null, null, false, true),
remove_col(X, Z), nsubj(Z, Y);
```

Figure A6-1. The Initial Version of the Modifying Rule

Step 5: Generalize the rule. Modifying rules should be designed in a way that maximizes its applicability in as many as possible cases to support the specification polishing process. In the previous step, the initial version of the modifying rule is generated based on the considered specific context. In this step, the goal is to generalize such a rule to extend its applicability in other similar contexts. Consider the current version of the rule in our example. Although the conditions are sufficiently general, the actions are not adequate to deal with some variances this specification. For instance, in case the original specification is written as *"Registered users are capable of creating reviews"* (its dependency tree is shown in Figure A6-2), the dependency between `Registered-1` and `users-2` would be lost during the polishing process due to the action of `remove_col(X, Z)` (in which `X` refers to `capable-4` and `Z` refers to `creating-5`). To prevent this problem, an additional terminal point should be added

into this action to prevent the dependency tree rooted at `users-2` to be removed. The action thus should be `remove_col(X, Z, Y)` (`Y` refers to `users-2` in this rule).

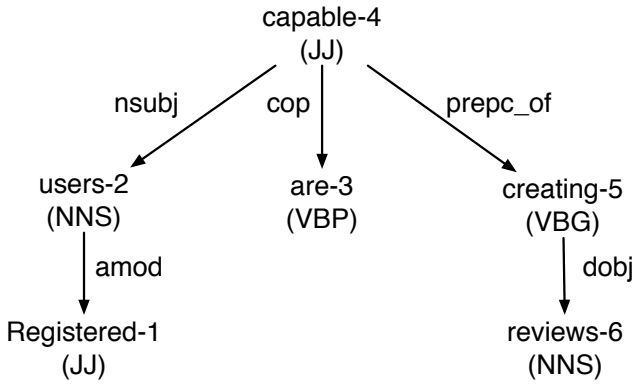


Figure A6-2. A Variance of the Original Specification

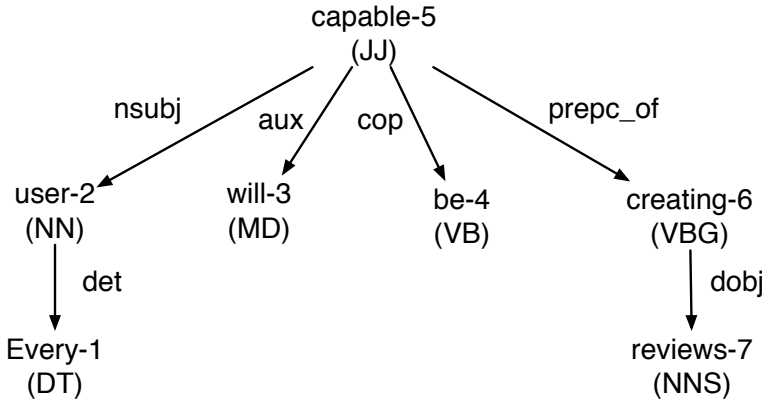


Figure A6-3. Another Variance of the Original Specification

Moreover, the dependency adding action `new_verb(Z, null, T, null, null, null, false, true)` is also not sufficient to handle some variances. Consider an example in which the original specification is *"Every user will be capable of creating reviews"* (its dependency tree is shown in Figure A6-3). Since the value of the `POS_tag_ref` parameter in this action is `T`, which refers to the node `be-4`, the POS tag of `creating_5` in the new dependency tree is the same as `be-4`, which is VB. This means the modified specification would become "Every user create reviews" which is grammatically incorrect (*"create"* should be instead *"creates"*). From this observation, it turns out that the POS tag of `creating_5` should be determined based on the subject of the entire specification instead of being referenced from `be-4`. Thus, the action should now be changed to `new_verb(Z, null, Y, null, null, null, false, true)`. The updated version of this modifying rule is presented in Figure A6-4.

```
X={capable}
Y/{NN|NNS|NNP|NNPS}
Z/VBG
T={be}
nsubj(X, Y)
cop(X, T)
root(X)
prepc_of(X, Z)
-> new_verb (Z, null, T, null, null, null, false, true),
remove_col(X, Z, Y), nsubj(Z, Y);
```

Figure A6-4. The Final Version of the Modifying Rule

Similar to the extraction rule quality analysis support discussed in Appendix A4, GUEST also provide users to analyze modifying rules for correctness, redundancy, consistency.

Appendix A7 Ontology Creation and Quality Analysis

Ontology Creation Process

In this section, we describe our steps to create ontologies. We adapted Noy et al. ontology building guidelines [113]. Protégé[16] is our choice for ontology development tool due to its comprehensive features. The examples used in this section come from our attempt of building ontology for the traveler social networking domain.

Step 1: Determine the domain and scope of the ontology. In this step, it is important to identify in which domain we are going to build an ontology. A domain often has a number of sub-domains. For instance, the sub-domains of the traveler social networking domain include: travel, social network (online community) and web application. In addition, the scope of the ontology needs to be determined. In the context of GUITAR, ontology is expected to contain the concepts and relationships of concepts in the relevant domains. Our ontology structure (cf. Figure 7.2 in Chapter 7) suggests what to look for to build ontology. For instance, they include:

- What the entities and properties used to describe the domain are (i.e., *traveler*, *booking*, *account*, *safe*, *expensive*)
- What activities to be carried out (i.e., *create review*, *book hotel*)
- What the characteristics of such activities. For instance, who usually book hotels? Who write reviews? Who is not allowed to write reviews? In what means a user can book hotels (i.e., *phone*, *internet*)?

Step 2: Consider reusing existing ontologies. It is always worth considering existing ontologies and checking the possibility of reusing them for our particular domain and tasks. In this step, the sub-domains identified in step 1 become extremely important. If no existing ontology were found for the main domain, looking into the sub-domains would be helpful in obtaining existing relevant ontologies. For instance, since no ontology was found for the traveler social networking domain, we then searched and obtained relevant ontologies in the domains of travel and online community. Some

[16] http://protege.stanford.edu/

ontology repositories such as Protégé Ontology Library[17], DAML Ontology Library[18] and Swoogle Semantic Web Search Engine[19] offer pre-built ontologies in different domains. Once relevant ontologies are obtained, the following steps are performed.

- **Step 2.1: Study the ontologies' contents and structures.** It is critical to gain understanding of what each ontology contains and how it is specified (i.e., its naming convention and class hierarchy). Such information is very important when merging these ontologies into one, especially if they are overlapped. Reading and studying the relevant domains' documentations are usually required to comprehend the contents of the ontologies.
- **Step 2.2: Merge the ontologies into a single ontology.** We use PROMPT [114], a plugin for Protégé, to merge multiple ontologies. PROMPT provides a semi-automated and interactive support to (1) identify identical or linguistically similar concepts in two ontologies, (2) suggest merging actions and (3) identify and resolve conflicts in the resulting ontology. PROMPT reportedly achieved over 90% precision and recall rates for its suggestion quality.
- **Step 2.3: Convert the resulting ontology to our format.** In GUITAR, we specify and use ontologies in Manchester OWL Syntax. Therefore, it is necessary to convert the resulting ontology into this format to be usable with our tool. This can be done automatically by Protégé. In addition, we need to make sure the ontology is specified according to our ontology meta-model. For instance, we use our notion of *activity* instead of object properties to describe the relationships between classes (the only allowable object properties are those defined under *relationship* category in our ontology meta-model). In fact, the relationship *"users create reviews"* is described with an activity *"create review"* and a *"hasSubject"* relationship between this activity and the class *"user."* Using activities enables the capturing of relationships between activities (i.e., two activities are identical, or one requires another), which is not possible by using object properties. In addition, it is important to make sure the names of concepts conform to our naming convention.

[17] http://protegewiki.stanford.edu/
[18] http://www.daml.org/ontologies/
[19] http://www.swoogle.umbc.edu

Step 3: Semi-automatically build an initial ontology. In case no existing ontology can be obtained, or if existing ontologies are incomplete (if we identify the need for additional details for the ontology) or unsuitable for our needs, we can start exploring ontology components from the domain descriptions or documents that we find. In this step, we use Text2Onto [24], an ontology learning tool, to build an initial ontology. Text2Onto provides a semi-automated support for extracting ontologies with concepts and relationships (i.e., sub-class, instances) from textual domain descriptions.

It thus helps us save time and effort in building an initial version of the ontology. Text2Onto is the best ontology learning tool currently available according to our investigation. It, however, has a number of limitations. First, it identifies all noun phrases in a sentence as concepts. This often creates false positives because not all noun phrases are valid domain concepts. Therefore, after the initial ontology is obtained, it is necessary to manually verify the validity of the identified concepts and relationships (i.e., remove unnecessary concepts). Second, it does not support the identification of quality properties (i.e., good, expensive), activities and a number of relationships (i.e., require, exclude, leadTo) that are required in our framework. Thus, additional steps need to be taken to further complete the ontology. In case an existing ontology is used, it would be merged into the initial ontology.

Step 4: Identify and classify concepts in the ontology. In this step, we focus on identifying additional concepts in the domain that have not been included in the initial ontology. We also classify all concepts (include the existing ones) according to our ontology structure. For example, some terms in the traveler social networking domain are *travel*, *traveler*, *hotel*, *'reserve room,' secure*. *Traveler* should then be classified as an *active entity*, *hotel* as an *inactive entity*, *'reserve room'* as an *activity* and *secure* as an *adjectival property*. For specific terms such as *English*, *French*, *white*, or *red*, we define them as instances of more general terms (i.e., *Language*, *Color*). We have developed our classification of around 300 commonly used verbs and their relationships (i.e., equivalent, disjoint)[20]. Therefore, in this step, verbs do not need to be identified and classified unless they do not exist in our collection.

[20] The verb collection can be found at http://goo.gl/gCUofM

Step 5: Identify relationships between concepts. In this step, we identify additional semantic relationships between the concepts. Examples of such relationships are equivalent, disjoint, sub-class, require, exclude, leadTo (between activities), hasAttribute (i.e., the entity *'service'* has attributes of *'good'* or *'poor'*). In textual domain descriptions, these relationships are usually recognized by a number of grammatical structures. For example, a require relationship between the activities *'register account'* and *'post comment'* can be identified from the sentence *"a user **must** register for an account **before** he can post comments in the system."* In this example, the structure 'must do something before doing something' implies a require relationship.

Ontology building is a time and effort-consuming task. It is necessarily an iterative process. For instance, in case step 5 is already completed, if new domain documentations are available, we would need to go back to step 3 to identify and classify additional terms. Additionally, if a new existing ontology were found, step 2 and later would be repeated. A sample ontology we created for the traveler social networking domain can be found at http://goo.gl/gCUofM.

While the creation of the domain ontology should be carried out at the beginning of the domain study phase, it can be extended at any time during the RE process when new knowledge about the domain is obtained. GUITAR provides an ontology editor to enable quick editing and on-the-fly update of ontologies (cf. section 7.4 in Chapter 7).

Ontology Quality Analysis

As the semantic analysis of requirements in our approach largely depends on the ontology's quality, it is critical to ensure the ontology is error-free before the analysis can be started. Table A7-1 presents a categorization of ontology anomalies in the context of our work. These categories were developed based on a collection of general ontology pitfalls identified and investigated by researchers and practitioners [125]. We identified the list of applicable anomalies and extended it with our identified possible ontology issues. The anomalies are classified into 3 categories: incorrectness, incompleteness and inconsistency. The complete description of these anomalies can be found in Table A7-2.

Table A7-1. Categories of Ontology Anomalies

Incorrectness	
A1: Creating unclassified classes A2: Merging different concepts into the same class A3: Missing annotations A4: Syntactically invalid relationships A5: Semantically invalid relationships A6: Specifying a hierarchy exceedingly	A7: Specifying the subject of an activity and object of a verb exceedingly A8: Wrapping intersection and union A9: Misusing ontology annotations A10: Using a miscellaneous class A11: Using different naming criteria in the ontology
Incompleteness	
A12: Missing classes	A13: Missing relationships
Inconsistency	
A14: Ontology is logically inconsistent	

In our work, we use OOPS! (**O**nt**O**logy **P**itfall **S**canner)[21], a web-based ontology validator, to identify these anomalies. OOPS! is able to automatically identify A2, A3, A8-11 and A14 (cf. Table A7-1). Our ontology editor can be used to automatically detect A1 and A4. A5, A6, A7, A12 and A13 are semantic problems. For instance, A5 is concerned with the semantic validity of a relationship between two classes (i.e., does activity A really *requires* activity B?). A12 and A13 are about the possibility that a class or a relationship in the domain of interest has not been identified by the ontology engineer. In our work, we partially support the identification of these problems. For instance, our ontology editor detects the cases when not all predefined classes (i.e., *active entity*, *inactive entity*, *adjectival property*) have a subclass. In addition, it identifies classes that are not connected to any pre-defined classes or does not have all of its predefined relevant relationships specified (i.e., an activity that does not have a *require* relationship with any other activity). Such detection helps ontology engineers enhance the ontology completeness.

[21] http://oops.linkeddata.es/

Table A7-2. Descriptions of Ontology Anomalies[22]

Incorrectness
A1. Creating unclassified classes*:** there exists a class that does not belong to any pre-defined class in the ontology meta-model. ***A2. Merging different concepts in the same class*:** a class is created whose identifier is referring to two or more different concepts. An example of this type of pitfall is to create the class *'StyleAndPeriod,'* or *'ProductOrService.'* ***A3. Missing annotations: ontology terms lack annotations properties. Annotation is critical to improve the ontology understanding ***A4. Syntactically invalid relationships:*** some relationships specifications are not (syntactically) correct according to the ontology meta-model. For instance, a *'leadTo'* relationship is specified between two verbs *'Book_v'* and *'Attend_v'* while they are only allowed between activities. **A5. Semantically invalid relationships:** some relationships are not semantically correct. For instance, two entities *'Library_e'* and *'Librarian_e'* have an *equivalent* relationship. **A6. Specializing a hierarchy exceedingly:** the hierarchy in the ontology is specialized in such a way that the final leaves cannot have instances, because they are actually instances and should have been created in this way instead of being created as classes. An example of this type of pitfall is to create the class *'RatingOfRestaurants'* and the classes *'1star_e,' '2stars_e,'* and so on, as subclasses instead of as instances. Another example is to create the classes *'Melbourne_e,' 'Sydney_e,' 'Canberra_e,'* and so on as subclasses of *'Place_e.'* This pitfall could be also named *'Individuals are not Classes.'* **A7. Specifying the subject of an activity or object of a verb exceedingly:** in some cases, the specification of an activity's subject (agent) or a verb's object is more restricted than expected. An example of this type of pitfall is to restrict the possible agent of the *'createReview_a'* activity to the class *'User_e,'* instead of allowing also the class *'Admin_e or User_e'* to create reviews. ***A8. Swapping intersection and union:*** in some cases, the use of intersection (of

[22] The descriptions of these common ontology anomalies are based on the Catalogue of common pitfalls http://oops.linkeddata.es/catalogue.jsp

classes) should instead be correctly specified with union. For instance, it would be incorrect if the agent of the *'createReview_a'* activity above is specified as *'Admin_e AND User_e'* since this equals to an empty class. ***A9. Misusing ontology annotations:*** The contents of some annotation properties are swapped or misused. An example of this type of pitfall is to include in the Label annotation of the class *'Crossroad_e'* with the following sentence *'the place of intersection of two or more roads'* and to include in the Comment annotation the word *'Crossroads.'* ***A10. Using a miscellaneous class*:** to create in a hierarchy a class that contains the instances that do not belong to the sibling classes instead of classifying such instances as instances of the class in the upper level of the hierarchy. An example of this type of pitfall is to create the class *'HydrographicalResource_e,'* and the subclasses *'Stream_e,' 'Waterfall_e,'* etc., and also the subclass *'OtherRiverElement_e.'* ***A11. Using different naming criteria in the ontology:*** Ontology elements are not named using the same convention within the whole ontology. It is considered a good practice that the rules and style of lexical encoding for naming the different ontology elements is homogeneous within the ontology.
Incompleteness
A12. Missing ontology classes and A13 Missing ontology relationships: ***One or more classes and relationships in the domain are missing from the ontology.***
Inconsistency
A14: Ontology is logically inconsistent: Inconsistency is identified in case there exist a class inferred to be equal to an empty class (owl:Nothing).
*******Anomalies in *italic* can be automatically identified by either OOPS! or GUITAR.

Appendix A8 Algorithms for Semantic 3Cs Problems Detection

This appendix presents some of the key algorithms for detecting semantic 3Cs problems in a goal-use case integrated model.

Table A8-1 and Table A8-2 present our algorithms for detecting missing relationships (P4 in Table 7-5) and missing use case exceptions or pre-conditions (P3.2.3 in Table 7-5) respectively. Algorithms for other problems solvable by this technique follow the same concepts.

Table A8-1. Algorithm for Detecting Missing Relationship

```
Function checkMissingRelationship(a1, a2)
   Input: a1 and a2 are artifact specifications
   Output: a problem report that contains the association between a
   missing relationship and the explanation for it. Such
   explanation is based on the explanation for the relationships
   between ontological concepts generated by Pellet. A problem
   report may contain more than one missing relationship
   IF  isAlternative(a1, a2) //if the artifacts are
       alternative of each other, then no need to identify
       relationship between them
       return NULL
   END IF
   a1_act = getActivity(a1) //extract activity of artifact a1
   a2_act = getActivity(a2) //extract activity of artifact a1
   a1_mos_act = getMOSDesc(a1_act) //get MOS representation
                                    of the activity a1_act
   a2_mos_act = getMOSDesc(a2_act)
   report = findPossibleRel(a1_mos_act, a2_mos_act)
   IF report != NULL
       return report
   END IF
   return NULL
END Function
```

Table A8-2. Algorithm for Detecting Missing Extension/Pre-condition

```
Function checkMissingExceptionOrPreCon(s)

   Input: s is a use case step specification

   Output: a problem report that contains the association between
   an identified exceptional case and the explanation for it. The
   explanation is based on the explanation of relationships between
   ontological concepts generated by Pellet. A problem report may
   contain more than one exceptional case

   s_act = getActivity(s) //get the activity of the use case
                            step

   s_mos_act = getMOSDesc(s_act)

   s_agent = getAgent(s)

   s_mos_agent = getMOSDesc(s_agent)

   s_obj = getObject(s)

   s_mos_obj = getMOSDesc(s_obj)

   exclActors = getExcludingActors(s_mos_act) //infer active
                 entities that has 'exclude' relationship
                 with the activity

   exclObjs = getExcludingObjects(s_mos_act) //infer inactive
                 entities that has 'exclude' relationship
                 with the activity

   //identify the actors and objects (in exclActors and
   exclObjs) that are sub-classes of s_mos_agent and s_mos_obj
   respectively

   relevantActorsObjs = getRelevantActorsObjs(actors, objs)

   IF relevantActorsObjs = NULL

       Return NULL

   END IF

   report = new MissingPreCon_ExtReport //create an empty
                                          report

   FOR EACH actor_obj in relevantActorsObjs

       //check if a pre-condition or an extension already
       exists to handle this actor or object.

       isActor = checkIsActor(actor_obj) //check if the class
       is actor (active entity) or object (inactive entity)

       uc = getUseCase(s)

       //check if a pre-condition or an extension exists to
       handle this entity. For instance, if an actor 'banned
       user' is known for not being allowed to do the action
       required by the use case or step (i.e. create review),
       then this function return true in case there is a pre-
       condition in the form of 'the user is not a banned
       user,' or 'the user is not banned'

       IF !checkPreconExists(actor_obj, isActor, uc) &

           !checkExtensionExists(actor_obj, isActor, uc)

           record(actor_obj, report) //this adds an
       association between actor_obj and the explanation
```

```
        why it is a missing exceptional case to handle
        END IF
    END FOR
END Function
```

Table A8-3 presents the algorithm for detecting inconsistency between two goals.

Table A8-3. Algorithm for Detecting Goal Pairwise Inconsistency

```
Function checkGoalPairwiseInconsistency(a1, a2)
    Input: a1 and a2 are two goal specifications
    Output: an explanation for the detected inconsistency. It is
    NULL if no inconsistency found
    sfs1 = getSemanticFunctions(a1) //get all semantic functions of
    a1. Verb and object are combine into an  activity function
    sfs2 = getSemanticFunctions(a2)
    IF sfs1 has at least an element which is not in sfs2 AND
    vice versa
        return NULL
    ENF IF
    //get common set of semantic functions of two a1 and a2
    common_sfs = getCommonSfs(sfs1, sfs2)
    conflicting_sfs = new List to store pairs of conflicting
    semantic functions
    overlapped_sfs = new List to store pairs of overlapped
    semantic functions. 2 semantic functions are overlapped if
    one's MOS representation is equivalent to or subclass of
    another's)
    FOR EACH sf in overlapped_sfs
        sf_val1 = getSemanticFunctionValue(a1, sf)
        sf_val2 = getSemanticFunctionValue(a2, sf)
        sf_mos_val1 = getMOSDesc(a1, sf)
        sf_mos_val2 = getMOSDesc(a2, sf)
        IF !relevant(sf_mos_val1, sf_mos_val2) //'relevant' means
            sf_mos_val1 and sf_mos_val2 are neither overlapped nor
            conflicting
            return NULL
        END IF
        IF isConflicting(sf_mos_val1, sf_mos_val2)
            conflicting_sfs.ADD(sf, sf_mos_val1, sf_mos_val2)
        ELSE IF isOverlapping(sf_mos_val1, sf_mos_val2)
            overlapped_sfs.ADD(sf, sf_mos_val1, sf_mos_val2)
        END IF
    END FOR
```

```
    IF count(conflicting_sfs) = 1
        expl = generateInconsistencyExpl(conflicting_sfs,
        overlapped_sfs)
        return expl
    END IF
    return NULL
END Function
```

Table A8-4 presents the algorithm to extract condition or event and main description of an artifact for the purpose of identifying cyclic inconsistent relationship.

Table A8-4. Algorithm for Extracting Conditions/Events

```
Function extractConditionEvent(goals, ontology)
  Input: goals is a list of goals in the model
  addedArtifacts = new empty map
  FOR EACH goal g in goals
      IF hasConditionOrEvent(g)
          g1 = getConditionOrEvent(g)
          g2 = getMainDesc(g) //main description is the rest
          of g, after removing its condition or event
          setRelationship(g1, g2, "require") //set the
          'require' relationship between these g1 and g2
          addedArtifacts.add(g1, g)
          addedArtifacts.add(g2, g)
      END IF
  END FOR
  FOR EACH artifact a in addedArtifacts
      duplicate = false
      FOR EACH goal g in goals
          IF isDuplicate(a, g) //check if they are
              semantically identical. Use Parameter
              matching technique (T4) to do this.
              duplicate = true
              mergeRelationships(a, g)
              BREAK
          END IF
      END FOR
      IF !duplicate
          goals.add(a)
      END IF
  END FOR
  //the updated version of goals from this algorithm is only used
  for cyclic relationship validation. It does not affect the
  original goal-use case model.
END Function
```

References

[1] J.-R. Abrial, and A. Hoare, "The B-book: assigning programs to meanings". Cambridge University Press. 2005.

[2] B.S. Ali and Z.M. Kasirun. "Developing tool for crosscutting concern identification using nlp". in Information Technology, 2008. ITSim 2008. International Symposium on. IEEE. 2008.

[3] D. Alrajeh, J. Kramer, A.v. Lamsweerde, A. Russo, and S. Uchitel. "Generating obstacle conditions for requirements completeness". in Proceedings of the 2012 International Conference on Software Engineering. IEEE Press. Zurich, Switzerland. 2012.

[4] V. Ambriola and V. Gervasi. "Processing natural language requirements". in Proceedings of the 12th International Conference on Automated Software Engineering. 1997.

[5] D. Amyot, X. He, Y. He, and D.Y. Cho. "Generating Scenarios from Use Case Map Specifications." in Proceedings of International Conference on Quality Software, 3: p. 108-115. 2003.

[6] A.I. Anton. "Goal-based requirements analysis". in Proceedings of the 2nd International Conference on Requirements Engineering. IEEE. 1996.

[7] A.I. Anton, R.A. Carter, A. Dagnino, J.H. Dempster, and D.F. Siege. "Deriving goals from a use-case based requirements specification." Requirements Engineering, 6(1): p. 63-73. 2001.

[8] A.I. Antón, W.M. McCracken, and C. Potts. "Goal decomposition and scenario analysis in business process reengineering". in Advanced Information Systems Engineering. Springer. 1994.

[9] J. Araùjo, E. Baniassad, P. Clements, A. Moreira, A. Rashid, and B. Tekinerdogan. "Early aspects: The current landscape." Technical Notes, CMU/SEI and Lancaster University. 2005.

[10] C. Arora, M. Sabetzadeh, L. Briand, F. Zimmer, and R. Gnaga. "Automatic checking of conformance to requirement boilerplates via text chunking: An industrial case study". in Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on. IEEE. 2013.

[11] E. Baniassad and S. Clarke. "Theme: An approach for aspect-oriented analysis and design". in Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society. 2004.

[12] B.W. Boehm, "Software engineering economics". Vol. 197. Prentice-hall Englewood Cliffs (NJ). 1981.

[13] B.W. Boehm. "Verifying and validating software requirements and design specifications." IEEE software, 1(1): p. 75. 1984.

[14] G. Boetticher, T. Menzies, and T. Ostrand, "The PROMISE Repository of Empirical Software Engineering Data, 2007."

[15] T.D. Breaux, A.I. Antón, and J. Doyle. "Semantic parameterization: A process for modeling domain descriptions." ACM Transactions on Software Engineering and Methodology (TOSEM), 18(2): p. 5. 2008.

[16] J. Bubenko, C. Rolland, P. Loucopoulos, and V. DeAntonellis. "Facilitating “fuzzy to formal” requirements modeling". in Requirements Engineering, 1994., Proceedings of the First International Conference on. IEEE. 1994.

[17] P. Buitelaar, D. Olejnik, and M. Sintek, "A protégé plug-in for ontology extraction from text based on linguistic analysis," in The Semantic Web: Research and Applications. 2004, Springer. p. 31-44.

[18] A. Casamayor, D. Godoy, and M. Campo. "Identification of non-functional requirements in textual specifications: A semi-supervised learning approach." Information and Software Technology, 52(4): p. 436-445. 2010.

[19] D.M. Cer, M.-C. De Marneffe, D. Jurafsky, and C.D. Manning. "Parsing to Stanford Dependencies: Trade-offs between Speed and Accuracy". in LREC. 2010.

[20] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese. "Model checking large software specifications." Software Engineering, IEEE Transactions on, 24(7): p. 498-520. 1998.

[21] E. Charniak. "Statistical techniques for natural language parsing." AI magazine, 18(4): p. 33. 1997.

[22] G.G. Chowdhury. "Natural language processing." Annual review of information science and technology, 37(1): p. 51-89. 2003.

[23] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. "Non-functional Requirements." Software Engineering. 2000.

[24] P. Cimiano and J. Völker, "Text2Onto," in Natural language processing and information systems. 2005, Springer. p. 227-238.

[25] E.M. Clarke, O. Grumberg, and D. Peled, "Model checking". MIT press. 1999.

[26] A. Cockburn. "Structuring Use Cases with Goals1." 1997.

[27] A. Cockburn. "Structuring Use Cases with Goals." 1997.

[28] A. Cockburn. "Basic use case template." Humans and Technology, Technical Report, 96. 1998.

[29] A. Cockburn, "Writing effective use cases, The crystal collection for software professionals." 2000, Addison-Wesley Professional Reading.

[30] I.C.S.S.E.S. Committee, I. Electronics Engineers, and I.-S.S. Board, "IEEE recommended practice for software requirements specifications: approved 25 June 1998". Vol. 830. IEEE. 1998.

[31] J. Cowie and W. Lehnert. "Information extraction." Communications of the ACM, 39(1): p. 80-91. 1996.

[32] B. Dano, H. Briand, and F. Barbier. "An approach based on the concept of use case to produce dynamic object-oriented specifications". in Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on. IEEE. 1997.

[33] B. Dano, H. Briand, and F. Barbier. "A use case driven requirements engineering process." Requirements Engineering, 2(2): p. 79-91. 1997.

[34] D. de Almeida Ferreira and A.R.d. Silva. "RSLingo: An information extraction approach toward formal requirements specifications". in Model-Driven Requirements Engineering Workshop (MoDRE), 2012 IEEE. IEEE. 2012.

[35] M.-C. De Marneffe and C.D. Manning. "The Stanford typed dependencies representation". in Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation. Association for Computational Linguistics. 2008.

[36] M.-C. De Marneffe and C.D. Manning. "Stanford typed dependencies manual." URL http://nlp. stanford. edu/software/dependencies manual. pdf. 2008.

[37] D.K. Deeptimahanti and R. Sanyal. "Semi-automatic generation of UML models from natural language requirements". in Proceedings of the 4th India Software Engineering Conference. ACM. 2011.

[38] T. DeMarco, "Structured analysis and system specification". Yourdon Press. 1979.

[39] D. Dermeval, J. Vilela, I.I. Bittencourt, J. Castro, S. Isotani, P. Brito, and A. Silva. "Applications of ontologies in requirements engineering: a systematic review of the literature." Requirements Engineering: p. 1-33. 2015.

[40] S.C. Dik and K. Hengeveld, "The theory of functional grammar: the structure of the clause". Walter de Gruyter. 1997.

[41] J. Drazan and V. Mencl, "Improved processing of textual use cases: Deriving behavior specifications," in SOFSEM 2007: Theory and Practice of Computer Science. 2007, Springer. p. 856-868.

[42] E. Dürr and J. van Katwijk. "VDM++, a formal specification language for object-oriented designs". in CompEuro'92.'Computer Systems and Software Engineering', Proceedings.: IEEE. 1992.

[43] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. "Patterns in property specifications for finite-state verification". in Software Engineering, 1999. Proceedings of the 1999 International Conference on. IEEE. 1999.

[44] D.V. Dzung and A. Ohnishi. "Improvement of quality of software requirements with requirements ontology". in Quality Software, 2009. QSIC'09. 9th International Conference on. IEEE. 2009.

[45] A. Fantechi and E. Spinicci. "A Content Analysis Technique for Inconsistency Detection in Software Requirements Documents". in WER. Citeseer. 2005.

[46] C. Fellbaum, "WordNet". Wiley Online Library. 1998.

[47] A. Ferrari, F. dell'Orletta, G.O. Spagnolo, and S. Gnesi, "Measuring and Improving the Completeness of Natural Language Requirements," in Requirements Engineering: Foundation for Software Quality. 2014, Springer. p. 23-38.

[48] N.E. Fuchs and R. Schwitter. "Attempto controlled english (ace)." arXiv preprint cmp-lg/9603003. 1996.

[49] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. "Specifying and analyzing early requirements in Tropos." Requirements Engineering, 9(2): p. 132-150. 2004.

[50] V. Gervasi and B. Nuseibeh. "Lightweight validation of natural language requirements." Software: Practice and Experience, 32(2): p. 113-133. 2002.

[51] V. Gervasi and D. Zowghi. "Reasoning about inconsistencies in natural language requirements." ACM Transactions on Software Engineering and Methodology (TOSEM), 14(3): p. 277-330. 2005.

[52] M. Glinz, "An integrated formal model of scenarios based on statecharts," in Software Engineering—ESEC'95. 1995, Springer. p. 254-271.

[53] M. Glinz. "Improving the quality of requirements with scenarios". in Proceedings of the Second World Congress on Software Quality. 2000.

[54] L. Goldin and D.M. Berry. "AbstFinder, a prototype natural language text abstraction finder for use in requirements elicitation." Automated Software Engineering, 4(4): p. 375-412. 1997.

[55] T. Gorschek and C. Wohlin. "Requirements abstraction model." Requirements Engineering, 11(1): p. 79-101. 2006.

[56] T. Gruber, "What is an Ontology." 1993.

[57] H.M. Harmain and R. Gaizauskas. "CM-Builder: an automated NL-based CASE tool". in Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on. IEEE. 2000.

[58] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. "Automated consistency checking of requirements specifications." ACM Transactions on Software Engineering and Methodology (TOSEM), 5(3): p. 231-261. 1996.

[59] J. Heumann. "Generating test cases from use cases." The rational edge, 6(01). 2001.

[60] J. Holtmann, J. Meyer, and M. Von Detten. "Automatic validation and correction of formalized, textual requirements". in Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on. IEEE. 2011.

[61] M. Horridge and S. Bechhofer. "The owl api: A java api for owl ontologies." Semantic Web, 2(1): p. 11-21. 2011.

[62] M. Horridge, N. Drummond, J. Goodwin, A.L. Rector, R. Stevens, and H. Wang. "The Manchester OWL Syntax". in OWLed. 2006.

[63] I. Horrocks, O. Kutz, and U. Sattler. "The Even More Irresistible SROIQ." KR, 6: p. 57-67. 2006.

[64] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. "SWRL: A semantic web rule language combining OWL and RuleML." W3C Member submission, 21: p. 79. 2004.

[65] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen. "Formal approach to scenario analysis." IEEE Software, (2): p. 33-41. 1994.

[66] H. Hu, L. Zhang, and C. Ye. "Semantic-based requirements analysis and verification". in Electronics and Information Engineering (ICEIE), 2010 International Conference On. IEEE. 2010.

[67] D. Jackson. "Alloy: a lightweight object modelling notation." ACM Transactions on Software Engineering and Methodology (TOSEM), 11(2): p. 256-290. 2002.

[68] I. Jacobson. "Object oriented software engineering: a use case driven approach." 1992.

[69] I. Jacobson. "The use-case construct in object-oriented software engineering." 1995.

[70] S. Jeffrey, J. Richards, F. Ciravegna, S. Waller, S. Chapman, and Z. Zhang. "The Archaeotools project: faceted classification and natural language processing in an archaeological context." Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, 367(1897): p. 2507-2519. 2009.

[71] H. Kaiya and M. Saeki. "Ontology based requirements analysis: lightweight semantic processing approach". in Quality Software, 2005.(QSIC 2005). Fifth International Conference on. IEEE. 2005.

[72] H. Kaiya and M. Saeki. "Using domain ontology as domain knowledge for requirements elicitation". in Requirements Engineering, 14th IEEE International Conference. IEEE. 2006.

[73] M. Kamalrudin, J. Grundy, and J. Hosking. "Tool support for essential use cases to better capture software requirements". in Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM. 2010.

[74] E. Kamsties and B. Paech. "Surfacing Ambiguity in Natural Language Requirement." 2000.

[75] E. Kamsties, D.M. Berry, and B. Paech. "Detecting ambiguities in requirements documents using inspections". in Workshop on Inspections in Software Engineering, Paris, France. Citeseer. 2001.

[76] J. Kim, S. Park, and V. Sugumaran. "Improving use case driven analysis using goal and scenario authoring: A linguistics-based approach." Data & Knowledge Engineering, 58(1): p. 21-46. 2006.

[77] J. Kim, J. Kim, S. Park, and V. Sugumaran. "A multi-view approach for requirements analysis using goal and scenario." Industrial Management & Data Systems, 104(9): p. 702-711. 2004.

[78] M. Kim, S. Park, V. Sugumaran, and H. Yang. "Managing requirements conflicts in software product lines: A goal and scenario based approach." Data & Knowledge Engineering, 61(3): p. 417-432. 2007.

[79] D. Klein and C.D. Manning. "Accurate unlexicalized parsing". in Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1. Association for Computational Linguistics. 2003.

[80] L. Kof, "Natural language processing for requirements engineering: Applicability to large requirements documents". na. 2004.

[81] L. Kof, "On the identification of goals in stakeholders' dialogs," in Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs. 2008, Springer. p. 161-181.

[82] L. Kof, "Requirements analysis: concept extraction and translation of textual specifications to executable models," in Natural Language Processing and Information Systems. 2010, Springer. p. 79-90.

[83] S.J. Körner and T. Brumm. "Natural language specification improvement with ontologies." International Journal of Semantic Computing, 3(04): p. 445-470. 2009.

[84] G. Kotonya and I. Sommerville, "Requirements engineering: processes and techniques". John Wiley & Sons, Inc. 1998.

[85] A. Lapouchnian. "Goal-oriented requirements engineering: An overview of the current research." University of Toronto: p. 32. 2005.

[86] K. Lauenroth and K. Pohl. "Dynamic consistency checking of domain requirements in product line engineering". in International Requirements Engineering, 2008. RE'08. 16th IEEE. IEEE. 2008.

[87] B.-S. Lee and B.R. Bryant. "Automated conversion from requirements documentation to an object-oriented formal specification language". in Proceedings of the 2002 ACM symposium on Applied computing. ACM. 2002.

[88] H. Lee, Y. Peirsman, A. Chang, N. Chambers, M. Surdeanu, and D. Jurafsky. "Stanford's multi-pass sieve coreference resolution system at the CoNLL-2011 shared task". in Proceedings of the Fifteenth Conference on Computational Natural Language Learning: Shared Task. Association for Computational Linguistics. 2011.

[89] H. Lee, A. Chang, Y. Peirsman, N. Chambers, M. Surdeanu, and D. Jurafsky. "Deterministic coreference resolution based on entity-centric, precision-ranked rules." Computational Linguistics, 39(4): p. 885-916. 2013.

[90] J. Lee and N.-L. Xue. "Analyzing user requirements by use cases: A goal-driven approach." IEEE software, 16(4): p. 92-101. 1999.

[91] J. Lee, N.-L. Xue, and J.-Y. Kuo. "Structuring requirement specifications with goals." Information and Software Technology, 43(2): p. 121-135. 2001.

[92] Y. Lee and W. Zhao. "An ontology-based approach for domain requirements elicitation and analysis". in Computer and Computational Sciences, 2006. IMSCCS'06. First International Multi-Symposiums on. IEEE. 2006.

[93] E. Letier and A. Van Lamsweerde. "Deriving operational software specifications from system goals." ACM SIGSOFT Software Engineering Notes, 27(6): p. 119-128. 2002.

[94] N. Leveson. "Completeness in formal specification language design for process-control systems". in Proceedings of the third workshop on Formal methods in software practice. ACM. 2000.

[95] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. "Requirements specification for process-control systems." Software Engineering, IEEE Transactions on, 20(9): p. 684-707. 1994.

[96] D. Liu, K. Subramaniam, A. Eberlein, and B.H. Far, "Natural language requirements analysis and class model generation using UCDA," in Innovations in Applied Artificial Intelligence. 2004, Springer. p. 295-304.

[97] L. Liu and E. Yu. "Designing information systems in social context: a goal and scenario modelling approach." Information systems, 29(2): p. 187-203. 2004.

[98] M. Luisa, F. Mariangela, and N.I. Pierluigi. "Market research for requirements analysis using linguistic tools." Requirements Engineering, 9(1): p. 40-56. 2004.

[99] S.G. MacDonell, K. Min, and A.M. Connor. "Autonomous requirements specification processing using natural language processing." arXiv preprint arXiv:1407.6099. 2014.

[100] G.A. Mala and G. Uma, "Automatic construction of object oriented design models [UML diagrams] from natural language requirements specification," in PRICAI 2006: Trends in Artificial Intelligence. 2006, Springer. p. 1155-1159.

[101] M.P. Marcus, M.A. Marcinkiewicz, and B. Santorini. "Building a large annotated corpus of English: The Penn Treebank." Computational linguistics, 19(2): p. 313-330. 1993.

[102] A.K. McCallum. "MALLET: A Machine Learning for Language Toolkit." 2002.

[103] R. McDonald, K. Lerman, and F. Pereira. "Multilingual dependency analysis with a two-stage discriminative parser". in Proceedings of the Tenth Conference on Computational Natural Language Learning. Association for Computational Linguistics. 2006.

[104] T. Morton, J. Kottmann, J. Baldridge, and G. Bierner, "Opennlp: A java-based nlp toolkit." 2005.

[105] T. Murata. "Petri nets: Properties, analysis and applications." Proceedings of the IEEE, 77(4): p. 541-580. 1989.

[106] T.H. Nguyen, J. Grundy, and M. Almorsy. "GUITAR: An ontology-based automated requirements analysis tool". in Requirements Engineering Conference (RE), 2014 IEEE 22nd International. IEEE. 2014.

[107] T.H. Nguyen, J. Grundy, and M. Almorsy. "Ontology-based automated support for goal–use case model analysis " Software Quality Journal, 23(3). 2015.

[108] T.H. Nguyen, J. Grundy, and M. Almorsy, "Rule-Based Extraction of Goal-Use Case Models from Text," in 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE/ESEC 2015). 2015, ACM: Bergamo, italy.

[109] T.H. Nguyen, J.C. Grundy, and M. Almorsy. "Integrating Goal-oriented and Use Case-based Requirements Engineering: The Missing Link". in ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems. Ottawa: ACM/IEEE. 2015.

[110] A.P. Nikora and G. Balcom. "Automated identification of LTL patterns in natural language requirements". in Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on. IEEE. 2009.

[111] N. Niu and S. Easterbrook. "Extracting and modeling product line functional requirements". in International Requirements Engineering, 2008. RE'08. 16th IEEE. IEEE. 2008.

[112] J. Nivre. "An efficient algorithm for projective dependency parsing". in Proceedings of the 8th International Workshop on Parsing Technologies (IWPT. Citeseer. 2003.

[113] N.F. Noy and D.L. McGuinness, "Ontology development 101: A guide to creating your first ontology." 2001, Stanford knowledge systems laboratory technical report KSL-01-05 and Stanford medical informatics technical report SMI-2001-0880.

[114] N.F. Noy and M.A. Musen. "The PROMPT suite: interactive tools for ontology merging and mapping." International Journal of Human-Computer Studies, 59(6): p. 983-1024. 2003.

[115] B. Nuseibeh. "To be and not to be: On managing inconsistency in software development". in Software Specification and Design, 1996., Proceedings of the 8th International Workshop on. IEEE. 1996.

[116] B. Nuseibeh and S. Easterbrook. "Requirements engineering: a roadmap". in Proceedings of the Conference on the Future of Software Engineering. ACM. 2000.

[117] N. Omar, J. Hanna, and P. McKevitt. "Heuristic-based entity-relationship modelling through natural language processing". in Artificial Intelligence and Cognitive Science Conference (AICS). Artificial Intelligence Association of Ireland (AIAI). 2004.

[118] J. Osis, "Model-Driven Domain Analysis and Software Development: Architectures and Functions: Architectures and Functions". IGI Global. 2010.

[119] S. Petrov, L. Barrett, R. Thibaux, and D. Klein. "Learning accurate, compact, and interpretable tree annotation". in Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics. Association for Computational Linguistics. 2006.

[120] K. Pohl, "Requirements engineering: fundamentals, principles, and techniques". Springer Publishing Company, Incorporated. 2010.

[121] K. Pohl, G. Böckle, and F.J. van der Linden, "Software product line engineering: foundations, principles and techniques". Springer Science & Business Media. 2005.

[122] B. Potter, D. Till, and J. Sinclair, "An introduction to formal specification and Z". Prentice Hall PTR. 1996.

[123] C. Potts. "Fitness for use: the system quality that matters most". in Proceedings of the Third International Workshop on Requirements Engineering: Foundations of Software Quality REFSQ. 1997.

[124] C. Potts, K. Takahashi, and A.I. Anton. "Inquiry-based requirements analysis." Software, IEEE, 11(2): p. 21-32. 1994.

[125] M. Poveda-Villalón, M.C. Suárez-Figueroa, and A. Gómez-Pérez, "Validating ontologies with oops!," in Knowledge Engineering and Knowledge Management. 2012, Springer. p. 267-281.

[126] N. Prat. "Goal formalization and classification for requirements engineering, fifteen years later". in Research Challenges in Information Science (RCIS), 2013 IEEE Seventh International Conference on. IEEE. 2013.

[127] A. Rago, C. Marcos, and J.A. Diaz-Pace. "Identifying duplicate functionality in textual use cases by aligning semantic actions." Software & Systems Modeling: p. 1-25. 2014.

[128] A. Rajan and T. Wahl, "CESAR: Cost-efficient Methods and Processes for Safety-relevant Embedded Systems". Springer. 2013.

[129] A. Rashid, P. Sawyer, A. Moreira, and J. Araújo. "Early aspects: A model for aspect-oriented requirements engineering". in Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on. IEEE. 2002.

[130] R. Rauf, M. Antkiewicz, and K. Czarnecki. "Logical structure extraction from software requirements documents". in Requirements Engineering Conference (RE), 2011 19th IEEE International. IEEE. 2011.

[131] B. Regnell, M. Andersson, and J. Bergstrand. "A hierarchical use case model with graphical representation". in Engineering of Computer-Based Systems, 1996. Proceedings., IEEE Symposium and Workshop on. IEEE. 1996.

[132] C. Rolland and C. Salinesi, "Supporting requirements elicitation through goal/scenario coupling," in Conceptual Modeling: Foundations and Applications. 2009, Springer. p. 398-416.

[133] C. Rolland, C. Souveyet, and C.B. Achour. "Guiding goal modeling using scenarios." Software Engineering, IEEE Transactions on, 24(12): p. 1055-1071. 1998.

[134] C. Rolland, G. Grosz, and R. Kla. "Experience with goal-scenario coupling in requirements engineering". in Requirements Engineering, 1999. Proceedings. IEEE International Symposium on. IEEE. 1999.

[135] L. Rosenhainer. "Identifying crosscutting concerns in requirements specifications". in Proceedings of OOPSLA Early Aspects. Citeseer. 2004.

[136] D.T. Ross. "Structured analysis (SA): A language for communicating ideas." Software Engineering, IEEE Transactions on, (1): p. 16-34. 1977.

[137] D.T. Ross and K.E. Schoman Jr. "Structured analysis for requirements definition." Software Engineering, IEEE Transactions on, (1): p. 6-15. 1977.

[138] K.S. Rubin and A. Goldberg. "Object behavior analysis." Communications of the ACM, 35(9): p. 48-62. 1992.

[139] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W.E. Lorensen, "Object-oriented modeling and design". Vol. 199. Prentice-hall Englewood Cliffs. 1991.

[140] K. Ryan. "The role of natural language in requirements engineering". in Requirements Engineering, 1993., Proceedings of IEEE International Symposium on. IEEE. 1993.

[141] V.F. Santander and J.F. Castro. "Deriving use cases from organizational modeling". in Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on. IEEE. 2002.

[142] J. Santos, A. Moreira, J. Araujo, V. Amaral, M. Alférez, and U. Kulesza. "Generating requirements analysis models from textual requirements". in Managing Requirements Knowledge, 2008. MARK'08. First International Workshop on. IEEE. 2008.

[143] K.K. Schuler. "VerbNet: A broad-coverage, comprehensive verb lexicon." 2005.

[144] V.S. Sharma, S. Sarkar, K. Verma, A. Panayappan, and A. Kass. "Extracting high-level functional design from software requirements". in Software Engineering Conference, 2009. APSEC'09. Asia-Pacific. IEEE. 2009.

[145] M. Shibaoka, H. Kaiya, and M. Saeki, "GOORE: Goal-oriented and ontology driven requirements elicitation method," in Advances in Conceptual Modeling–Foundations and Applications. 2007, Springer. p. 225-234.

[146] E. Sikora, M. Daun, and K. Pohl, "Supporting the consistent specification of scenarios across multiple abstraction levels," in Requirements Engineering: Foundation for Software Quality. 2010, Springer. p. 45-59.

[147] V. Simko, P. Hnetynka, T. Bures, and F. Plasil. "Formal verification of annotated use-cases." Charles University in Prague, Tech. Rep, 2. 2012.

[148] A. Sinha, S.M. Sutton, and A. Paradkar. "Text2Test: Automated inspection of natural language use cases". in Software Testing, Verification and Validation (ICST), 2010 Third International Conference on. IEEE. 2010.

[149] A. Sinha, A. Paradkar, P. Kumanan, and B. Boguraev. "A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases". in Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on. IEEE. 2009.

[150] E. Sirin, B. Parsia, B.C. Grau, A. Kalyanpur, and Y. Katz. "Pellet: A practical owl-dl reasoner." Web Semantics: science, services and agents on the World Wide Web, 5(2): p. 51-53. 2007.

[151] I. Sommerville, "Software engineering—9th ed. p. cm." 2011, McGraw-Hill Companies Inc., New York.

[152] I. Sommerville and P. Sawyer, "Requirements engineering: a good practice guide". John Wiley & Sons, Inc. 1997.

[153] J.M. Spivey, "Understanding Z: a specification language and its formal semantics". Cambridge University Press. 1988.

[154] S. Supakkul and L. Chung. "Integrating FRs and NFRs: A use case and goal driven approach." framework, 6: p. 7. 2005.

[155] A. Sutcliffe. "Scenario-based requirements engineering". in Requirements engineering conference, 2003. Proceedings. 11th IEEE international. IEEE. 2003.

[156] A.G. Sutcliffe, N.A. Maiden, S. Minocha, and D. Manuel. "Supporting scenario-based requirements engineering." Software Engineering, IEEE Transactions on, 24(12): p. 1072-1088. 1998.

[157] W. Taggart Jr and M.O. Tharp. "A survey of information requirements analysis techniques." ACM Computing Surveys (CSUR), 9(4): p. 273-290. 1977.

[158] C. Treude, M. Robillard, and B. Dagenais. "Extracting development tasks to navigate software documentation." 2015.

[159] A.C. Uselton and S.A. Smolka, "A compositional semantics for Statecharts using labeled transition systems," in CONCUR'94: Concurrency Theory. 1994, Springer. p. 2-17.

[160] A. Van Lamsweerde. "Requirements engineering in the year 00: A research perspective". in Proceedings of the 22nd international conference on Software engineering. ACM. 2000.

[161] A. Van Lamsweerde. "Goal-oriented requirements engineering: A guided tour". in Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on. IEEE. 2001.

[162] A. Van Lamsweerde. "Goal-oriented requirements enginering: a roundtrip from research to practice [enginering read engineering]". in Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International. IEEE. 2004.

[163] A. Van Lamsweerde. "Requirements engineering: from system goals to UML models to software specifications." 2009.

[164] A. Van Lamsweerde and L. Willemet. "Inferring declarative requirements specifications from operational scenarios." Software Engineering, IEEE Transactions on, 24(12): p. 1089-1114. 1998.

[165] A. Van Lamsweerde, R. Darimont, and E. Letier. "Managing conflicts in goal-driven requirements engineering." Software Engineering, IEEE Transactions on, 24(11): p. 908-926. 1998.

[166] M.Y. Vardi, "An automata-theoretic approach to linear temporal logic," in Logics for concurrency. 1996, Springer. p. 238-266.

[167] K. Verma and A. Kass, "Requirements analysis tool: A tool for automatically analyzing software requirements documents". Springer. 2008.

[168] J.B. Warmer and A.G. Kleppe. "The Object Constraint Language: Precise Modeling With Uml (Addison-Wesley Object Technology Series)." 1998.

[169] K. Watahiki and M. Saeki. "Combining goal-oriented analysis and use case analysis." IEICE TRANSACTIONS on Information and Systems, 87(4): p. 822-830. 2004.

[170] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer. "Scenario usage in system development: a report on current practice". in Requirements Engineering, 1998. Proceedings. 1998 Third International Conference on. IEEE. 1998.

[171] N. Weston, R. Chitchyan, and A. Rashid. "A framework for constructing semantically composable feature models from natural language requirements". in Proceedings of the 13th International Software Product Line Conference. Carnegie Mellon University. 2009.

[172] K. Wiegers and J. Beatty, "Software requirements". Pearson Education. 2013.

[173] J. Woodcock, P.G. Larsen, J. Bicarregui, and J. Fitzgerald. "Formal methods: Practice and experience." ACM Computing Surveys (CSUR), 41(4): p. 19. 2009.

[174] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie. "Automated extraction of security policies from natural-language software documents". in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM. 2012.

[175] E. Yu. "Modelling strategic relationships for process reengineering." Social Modeling for Requirements Engineering, 11: p. 2011. 2011.

[176] E. Yu and J. Mylopoulos. "Why goal-oriented requirements engineering". in Proceedings of the 4th International Workshop on Requirements Engineering: Foundations of Software Quality. 1998.

[177] E.S. Yu. "Modeling organizations for information systems requirements engineering". in Requirements Engineering, 1993., Proceedings of IEEE International Symposium on. IEEE. 1993.

[178] E.S. Yu. "Towards modelling and reasoning support for early-phase requirements engineering". in Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on. IEEE. 1997.

[179] K. Yue. "What does it mean to say that a specification is complete?". in Proc. IWSSD-4, Fourth International Workshop on Software Specification and Design. 1987.

[180] P. Zave. "Classification of research efforts in requirements engineering." ACM Computing Surveys (CSUR), 29(4): p. 315-321. 1997.

[181] D. Zowghi and V. Gervasi. "On the interplay between consistency, completeness, and correctness in requirements evolution." Information and Software Technology, 45(14): p. 993-1009. 2003.

[182] D. Zowghi and C. Coulin, "Requirements elicitation: A survey of techniques, approaches, and tools," in Engineering and managing software requirements. Springer. p. 19-46. 2005