2011

An Interactive Approach of Ontology-based Requirement Elicitation for Software Customization

Xieshen Zhang
University of Windsor

An Interactive Approach of Ontology-based Requirement Elicitation for Software Customization

by

Xieshen Zhang

A Thesis
Submitted to the Faculty of Graduate Studies
through Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the

University of Windsor

Windsor, Ontario, Canada

2011

An Interactive Approach of Ontology-based Requirement Elicitation

for Software Customization

by

Xieshen Zhang

APPROVED BY:

_____

Dr. G. Bhandari
Odette School of Business

_____

Dr. A. Mukhopadhyay
School of Computer Science

_____

Dr. X. Yuan, Advisor
School of Computer Science

_____

Dr. D. Wu, Chair of Defense

School of Computer Science

May 04, 2011

DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

Software product lines allow reusing a collection of related software engineering assets to develop custom-made high quality software with reduced time and cost. In this thesis, an interactive approach of requirement elicitation for software customization is presented. It first adopts an ontology-based requirement model to represent the commonalities and variabilities among a group of related requirement artefacts. The development of a dialogue system further enables users to interactively customize software products by means of text-based dialogue. While the ontology model directs the dialogue system to perform requirement elicitation, its instantiation is accomplished with the support of decomposition-based requirement refinement in Service-Oriented Architecture. Besides the design details, a case study is presented to demonstrate how customizing an online book shopping system could be achieved with interactive requirement elicitation. Finally, the reliability and efficiency of the proposed method are evaluated with experimental comparison.

DEDICATION

This thesis is dedicated to my dear parents and brother for their endless love and support.

ACKNOWLEDGEMENTS

First, I would like to express my deep and sincere gratitude to my supervisor Dr. Xiaobu Yuan for the continuous support to my study and research, for his patience, motivation, and wide knowledge. I could not have imagined accomplishing the thesis without his guidance.

I am also heartily thankful to Dr. Asish Mukhopadhyay and Dr. Gokul Bhandari, whose insightful comments and encouragement inspired me throughout the work.

Finally, I would like to thank my friends Xiewei and Yifeng. Their long-term emotional support gave me the courage to carry on.

TABLE OF CONTENTS

DECLARATION OF ORIGINALITY iii
ABSTRACT iv
DEDICATION v
ACKNOWLEDGEMENTS vi
INTRODUCTION 1
1.1 Introduction 1
1.2 Problem Statement 1
1.3 Contribution 2
1.4 Structure of the Thesis 3
SOFTWARE PRODUCT LINE AND SERVICE-ORIENTED ARCHITECTURE 4
2.1 Software Product Line 4
2.2 Service-Oriented Architecture 5
INTERACTIVE REQUIREMENT ENGINEERING 8
3.1 Interactive Requirement Engineering 8
3.2 Dialogue System 9
3.3 Knowledge-based Requirement Engineering 13
3.4 Ontology 14
3.4.1 Overview 14
3.4.2 Ontology for Requirement Engineering 17
3.4.3 Ontology for SOA 18
PROPOSED ONTOLOGY-BASED REQUIREMENT MODEL 21
4.1 Introduction 21
4.2 Concepts 21
4.3 Relationships 23
4.4 Rules 28
4.5 Ontology Instantiation 29
PROPOSED REQUIREMENT ELICITATION METHOD 32
5.1 Introduction 32
5.2 Structure of the Proposed Dialogue System 32
5.3 Process of Requirement Elicitation 33
5.3.1 Machine-directed Requirement Elicitation 34
5.3.2 User Requirement Customization 37
5.4 Output of Requirement Elicitation 37
5.4.1 Output Overview 37
5.4.2 Output Generation 39
5.5 Considerations for System Implementation 43
IMPLEMENTATION AND CASE STUDY 45
5.6 Implementation 45

5.7 Case Study 49
5.7.1 Case Overview 49
5.7.2 Book Locating Service 50
COMPARISON ANALYSIS 58
7.1 Introduction 58
7.2 Problem Instance Generation 58
7.3 Experiment with the Proposed Method 60
7.4 Experiment with the Undirected Method 61
7.5 Results and Analysis 62
CONCLUSION AND FUTURE WORK 66
8.1 Conclusion 66
8.2 Future Work 66
REFERENCES 68
APPENDICES 73
The Complete Requirement Model for the Case Study 73
VITA AUCTORIS 78

CHAPTER I

INTRODUCTION

1.1 Introduction

Software product line (SPL) engineering is a paradigm to develop software applications with reusable software assets, which are tailored to individual customers' needs [1]. By reusing software engineering artefacts (e.g. software components) rather than developing them from scratch, software systems are expected to be customized, while the costs can be effectively cut down. Meanwhile, with the same primary goal of software reuse, Service-Oriented Architecture (SOA) separates system functionalities into loosely coupled and reusable services that communicate with each other via autonomous messages [2]. Although SPL and SOA differ, as different software engineering paradigms, in many respects, they actually complement each other [3]. By reusing services, and adopting SOA-based methods in SPL engineering, especially the Semantic Web Service techniques (e.g. automatic service discovery and composition) [4], the goal of automating software development could be achieved. Furthermore, the main focus of SPL engineering will then shift from repetitive system design and implementation to functionalities (i.e. services) customization.

On the other hand, in order to actualize completely automated SPL engineering, an approach is required for guiding human-machine interaction in software products customization. However, managing the complexity and variability of product features inherent in software product lines is very challenging [5]. In addition, a supporting tool for directing the automatic and interactive product customization is still lacking [6].

1.2 Problem Statement

A solution to automatic requirement elicitation is critical for the realization of automated SPL. While an increasing number of publications in SOA have addressed the

problem of automatic system implementation, few studies investigate the automation of requirement engineering.

To automate the requirement engineering process, first of all, a supporting tool for human-machine interaction is required, which is used to conduct the communications during requirement elicitation. Meanwhile, it must be capable of managing the knowledge related to SPL requirement engineering, thus the knowledge could be naturally presented to users. Furthermore, the tool should be able to generate service-oriented outputs for the automation of system implementation.

On the other hand, knowledge for automatic requirement elicitation is supposed to be presented in formats understandable to machines. In other words, a semantic way to represent the knowledge is required. Moreover, as a knowledge engineering solution to SPL engineering, it must be suitable to describe the common and variable features of requirement engineering artefacts of software systems that are given. Since the major challenge rooted in requirement engineering lies in maintaining the completeness and consistency of requirement products, it is necessary to tackle them properly. Last but not the least, an approach to express the knowledge about human-machine interaction should be investigated.

1.3 Contribution

To facilitate the realization of automated SOA-based SPL, this thesis presents a dialogue-based interactive approach for guiding software product customization. An abstract ontology-based requirement model, which represents the knowledge of the product features as well as their business logic, is developed. Besides, a frame-based dialogue system [7] is designed based on the knowledge model. It helps elicit users' requirements and then outputs service-oriented system description for the implementation of the candidate applications.

Though not mentioned in the thesis title, the proposed approach is designed

specifically for customizing SOA applications. In other words, it first guides users to order the services they need, and then generate corresponding service descriptions for automatic service discovery and composition.

1.4 Structure of the Thesis

The rest of the thesis is structured as follows. Chapter II presents the introduction to SPL and SOA, while Chapter III outlines the work related to interactive requirement engineering. The ontology model is proposed and explained in Chapter IV. Chapter V reports the proposed dialogue system and the interactive requirement elicitation method, followed by Chapter VI within which the implementation of the dialogue system and a case study is presented. Chapter VII demonstrates a group of experimental comparisons between the proposed machine-directed interactive requirement elicitation method and an undirected method, and the results are also analyzed in this chapter. Finally, conclusion and future work are discussed in Chapter VIII.

CHAPTER II

SOFTWARE PRODUCT LINE AND SERVICE-ORIENTED ARCHITECTURE

2.1 Software Product Line

The key idea of SPL engineering is gathering, analyzing and reusing the software engineering artefacts of closely related software systems. These reusable artefacts provide development options in each software engineering stage. Consequently, the software development activities will mainly focus on system customization rather than creation.

To develop an application with SPL framework, there are two processes: domain engineering and application engineering [1]. During domain engineering process, the commonalities and variabilities of the reusable artefacts are defined. Vertical tractability links are established between artefacts of different software engineering phases. In the application engineering stage, applications are developed. The variabilities that bind to the candidate application are identified. Then based on these common and variable artefacts, the development of the application is carried out.

In SPL, the products of the domain engineering process are supposed to be reused. So the price of developing a new application is mainly charged at the application engineering stage. By customizing rather than creating, the application engineering is cost-effective compared to traditional software engineering approaches. Therefore, as long as the domain engineering process is controllable, SPL can effectively reduce cost, time and human effort in software engineering.

Nonetheless, *Rabiser et~al.* [6] point out that, compared to the effort spending on developing and modeling the software product lines, little support is available for enhancing their utilization in practice,. Without effective approaches to utilize the product lines, particularly the automated approaches, SPL could not be widely accepted in industry. In other words, they will be of more academic value than practical value. In [6], *Rabiser et~al.* further define 6 requirements for facilitating the application of product

lines in practice:

- Automated and interactive variability resolution
- Adaptability and extensibility
- Application requirements management support
- Flexible and user-specific visualizations of variability
- End-user guidance
- Project management support

The idea of interactive requirement elicitation is inspired by these suggestions. With an automated and interactive solution to requirement elicitation, the variability of application requirements can be automatically managed and interactively elicited. The by-product of a requirement knowledge base further enables applying the product lines adaptively and extensively. Consequently, an approach to improve the practical value of SPL is suggested.

2.2 Service-Oriented Architecture

Since SOA captures many best practices from previous software engineering experiences, and makes business systems more flexible and reusable, it has gained an increasing popularity in industry as well as academic communities in the past decade. Technically speaking, SOA represents a model in which the software systems are decomposed into loosely coupled units of functionalities (i.e. services), while each of these units must be autonomous, reusable, discoverable, and is able to communicate with other units via autonomous messages [2]. Thus the units could be distributed, and collaborate through message exchange.

Typically, there are three roles involved in SOA engineering: provider, broker and requestor (Fig. 2.1). Service provider develops services and publishes the services by registering the service descriptions as well as corresponding access information in service broker's depository. Service requestor then tries to find the services by consulting the

service broker. Service broker matches requestor's demands with the services registered in the depository, and return the appropriate service access information to the requestor. Later on, requestor visits and retrieves the provider's services according to the access information. Therefore, to access certain services, the requestor must first discover the services from broker's registry, and then bind to the provider so as to invoke and compose, if necessary, the services.

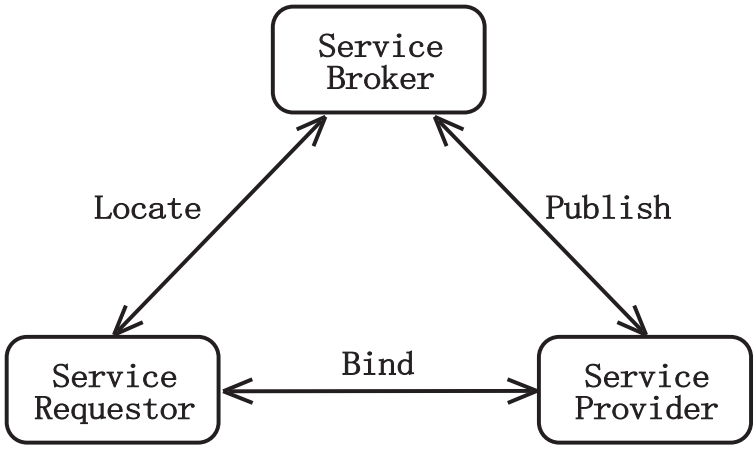


Fig. 2.1: Model of SOA

Service discovery is the process of finding appropriate services from brokers' registry. Traditionally, UDDI mechanism is applied in publishing, matching and discovering services. However, it only defines a set of syntactic search criteria. Matching on semantic level is not supported, which results in unwanted feedbacks. Inspired by the development of Semantic Web, the idea of Semantic Web Service comes out. Semantic Web Service approaches offer semantics to web services. Consequently, they are self-describable and machine-processable, and the discovery of these services is more promising. [8] presents an Semantic Web Service solution as well as a list of its applications in addressing the problem of matchmaking-based automatic service discovery.

Moreover, in this thesis, a method which is the inversion of service composition is adopted for the domain engineering of SPL. In other words, the functionalities of SOA

systems are analyzed by decomposition. Then results of the decomposition are further used to instantiate the proposed ontology-based requirement model.

Service composition is very significant in SOA engineering. The implementation of a complex web service often involves the invocation of other services. However, services are distributed. Their collaboration relies on exchanging autonomous messages. Therefore, the syntax as well as semantics of the messages, particularly the order of their exchanges, should be defined explicitly [9]. Message exchange in SOA is called service composition. Service composition represents the process of combining certain services' functionalities to implement a composite service's functionalities [10]. It can be performed by composing either primitive or composite services [10]. In this thesis, composite services are decomposed into less complex services, in order to obtain the knowledge related to systems' functionalities and business logic.

In addition, nowadays, people are also working on automating the service composition process. Semantic Web Service plus AI planning methodologies suggests approaches to solve this problem. In spite of lacking a comprehensive solution, successful improvement has been achieved [8].

Automatic service discovery and composition are critical to the realization of automated SPL. While the interactive requirement elicitation method proposed in this thesis is expected to automate the requirement engineering process, automated SOA methods are the best solutions to the automation of system implementation so far.

CHAPTER III

INTERACTIVE REQUIREMENT ENGINEERING

3.1 Interactive Requirement Engineering

In conventional software engineering, computers are treated simply as impersonal machines providing functions, objects or models, while their personality and characterization are neglected. In [11], *Knaus* states: "In the eyes of the software developer, computer behaves more like a human with extreme skills and obedience. " He further asserts that an interactive software engineering paradigm, which redistributes computers' responsibilities, can overcome the long-term software development and maintenance issues rooted in conventional programming paradigms. In addition, by defining a metaphor for the computer, building a concept model as a programming paradigm and designing an appropriate user interface, it is possible to find such an interactive paradigm.

Though *Knaus* promises a bright future, little progress has been made. The task of software engineering is very complicated. It is very challenging to redefine computers' responsibilities. Machines cannot deal with the complexity of a specific software component. Meanwhile, to build a concept model requires much effort from both software engineering and human-computer interaction. Thus how to build the interactive software engineering paradigm is still a question. Fortunately, some inspiring ideas came out in recent studies. SOA encapsulates software functionalities into loosely coupled services, which helps the machine software engineers get rid of the lower-level complexity and simplify their jobs. On the other hand, with SPL paradigm, their responsibilities are further specified as managing the variable software engineering artefacts. Therefore, in interactive software engineering, machines can play the role in directing users to select the reusable software assets and implementing the candidate application by composing the ordered services.

Since a relatively concrete specification to the machines' responsibilities in interactive software engineering is now available, this thesis further proposes a requirement elicitation approach for SOA-based SPL engineering as a programming model for realizing the interactive requirement engineering. A frame-based dialogue system is applied as the interaction interface. Work related to dialogue systems will be presented in the next section.

3.2 Dialogue System

Dialogue systems are a kind of computer systems designed to communicate with human beings, extracting and analyzing information from their dialogue-based expressions, so as to accomplish certain tasks (e.g. exchanging information and providing services) in relatively natural manners. Language is the most efficient way for human beings to exchange information between each other. Most human communications in history are based on dialogues. Thus dialogue system provides a more natural, comfortable and convenient way for human-machine interaction.

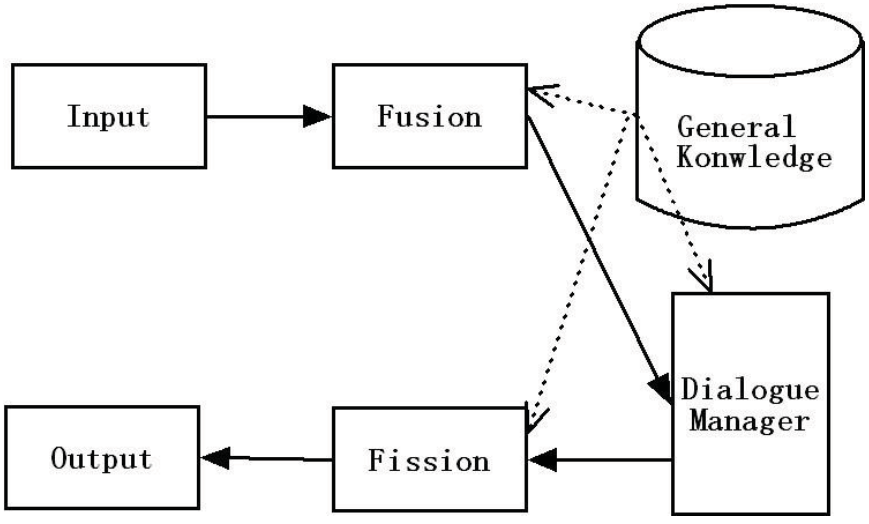


Fig. 3.1: Structure of dialogue system

Typically, a dialogue system consists of six components (Fig. 3.1): Input, Fusion, Dialogue Manager, Knowledge Base, Fission and Output [7].

- Input:

  Handle different modes of inputs.

- Fusion:

  Extract, recognize, interpret, and fuse information from different modes of inputs.

- Dialogue Manager:

  Control the flow of the dialogue by deciding how the system should respond to the inputs [12].

- Knowledge Base:

  Manage information like dialogue history, task knowledge, general dialogue knowledge, domain specific knowledge and user information.

- Fission:

  Transform the responses to different modes of outputs.

- Output:

  Handle the outputs.

The core components of a dialogue system are dialogue manager and knowledge base. Dialogue managers can be classified into four categories [7]:

- Finite-state and frame-based:

  Finite state-based dialogue managers are the simplest dialogue managers. The dialogue structure is represented in the form of state transition network, and the dialogue control is system-driven and all the system's utterances are predetermined [7]. As an extension of finite state-based dialogue managers, frame-based models simulate the approach of form filling, which allows some degree of flexibility. In this thesis, a frame-based dialogue system is developed for conducting requirement elicitation interactively.

- Information state and the probabilistic based:

  For the information state-based approaches, a group of states are predefined, and the state of the dialogue will be changed dynamically according to certain interaction strategies. Some probabilistic techniques (e.g. partially observable Markov decision process) are applied to manage the transition strategies.

- Plan based:

  Plan-based dialogue managers are based on the plan-based theories of communicative action and dialogue [2]. They are more complex than the previous dialogue managers.

- Collaborative agent based:

  Collaborative agent-based approaches try to capture the motivation behind a dialogue and the mechanisms of dialogue itself. As a result, managers based on these approaches contribute to the most complicated dialogue systems, which allow very high level of flexibility.

For the knowledge source in dialogue system, typically there are five different models of knowledge [13]:

- Dialogue Model:

  Dialogue Model holds the general information about the construction of a dialogue, which is used to control the dialogue. Grammar-based modeling and Plan-based modeling are two main approaches to model the knowledge for Dialogue Model.

- Dialogue History:

  Dialogue History records the history of the dialogue. It is used for dialogue control, disambiguation of context dependent utterances, and context sensitive interpretation [13].

- Domain Model:

  Domain Model holds the domain knowledge that will be referred to in the conversations. Knowledge in Domain Model is mainly used to guide the semantic interpretation of user's utterances, find the relevant items and relations that are under discussing, supply default responses and son on [13]. Domain Model usually contains the structure of the domain and comprises a subset of the general world knowledge. Its simplification is Conceptual Model, which represents the conceptual relationships between the objects in the domain [13]. Often, Conceptual Model alone is enough for the domain knowledge of the dialogue system.

- Task Model:

  Task Model, which often consists of a hierarchical representation structure, describes how the system's communicative and other tasks should be carried out [13].

- User Model:

  User Model represents the user's goal and plans, capabilities, attitudes, knowledge and belief [13].

In this thesis, the knowledge base of the proposed dialogue system contains domain knowledge of requirement elicitation and the task knowledge for guiding users to customize a specific type of software applications.

In addition to the structure, by considering the source of information which determines the interaction, tasks of dialogue systems can be classified into four categories [14]:

- Slot-filling task:

  The user has his goal and has the information about accomplishing the task.

- Database search task:

  The user has his goal but needs to retrieve information for completing the task.

- Explanation task:

  The user doesn't have or has little knowledge about the task.

- More complex tasks:

  The task is a combination of the other three tasks.

In the proposed method, a slot-based dialogue system is adopted, and requirement elicitation is modeled as a group of slot-filling tasks. These tasks will be performed according to the knowledge related to requirement engineering, which is built in knowledge base of the dialogue system. Work related to knowledge-based requirement engineering will be introduced in the next section.

3.3 Knowledge-based Requirement Engineering

Requirement engineering is recognized as the most critical stage of the entire software development process [15]. Typically, over 40% of errors in a software project are from requirement engineering, while they need 10 more times of costs to repair than other errors [16].

Conventional process-based or scenario-based requirement engineering methods predefine a group of processes and their corresponding guidelines. Accordingly, the requirement engineering activities and deliverables are carried out following the guidelines [17]. However, it is very often that when the processes are ongoing, some important information is not yet available. So, engineers have to repeat the processes, which results in project delay and additional cost [18].

Unlike traditional process-driven requirement engineering, knowledge-driven requirement engineering, as a novel requirement engineering paradigm, is conducted under the guidance of domain knowledge. Hence, information hidden in the domain can

be retrieved without much help from domain experts. The information is further used to guide the traditional requirement engineering process. As a result, the validity, completeness as well as consistency of requirement engineering product are maintained. Moreover, changes to the software development project will be detected and predicted in an early stage, and fewer waste efforts will be made. Finally, the outcome of the project is expected to be more mature and complete, while rework can be dramatically reduced [18].

Furthermore, among the group of knowledge-driven requirement engineering methods, ontology-based requirement engineering is very popular. It [19]:

- Provides formal representation for both requirement documents and knowledge.
- Describes the problem domain with varying degrees of formalization and expressiveness.
- Is well suited as an evolutionary approach.
- Is used to support requirements management and improve the traceability of requirement artefacts.

Thus it outperforms other traditional knowledge-based approaches [19]. By now, a number of ontology-based requirement engineering approaches have been proposed. Detailed introduction will be presented in the next section.

3.4 Ontology

3.4.1 Overview

In theory, an ontology is a formal, explicit specification of a shared conceptualization [20]. In other words, ontology is used to represent the common knowledge within a domain.

The reasons to develop an ontology can be roughly classified into five categories [21]:

- To share common understanding of the structure of information among people or software agents
- To enable reuse of domain knowledge
- To make domain assumptions explicit
- To separate domain knowledge from the operational knowledge
- To analyze domain knowledge

They are all closely related to domain knowledge representation. Generally, an ontology provides a shared vocabulary, which can be used to model a domain or a task. Here，modeling means constructing the concepts, objects as well as their properties and relations that exist in the domain or in the solution to the task [22].

Conventionally, knowledge engineering methods, like propositional logic, predicate logic and other rule-based methods, mainly investigate topics like logic, knowledge representation, search, and so on [23]. They focus on how to solve the problem rather than the knowledge itself. So the resulting knowledge is often implicit and difficult to be maintained, shared or reused. On the contrary, the main concern of ontology is the contents of knowledge and approaches to accumulate it. It builds the foundation for common knowledge.

Moreover, roughly speaking, ontology consists of task ontology, which characterizes the computational architecture of a knowledge-based system for certain tasks, and domain ontology, which characterizes the knowledge of a specific task domain [23].

To develop an ontology, typically, includes the following steps [21]:

1. Define classes (concepts in the domain) in the ontology
2. Arrange the classes in a taxonomic (subclass-superclass) hierarchy

3. Define slots (properties of classes and instances) and describing allowed values for these slots
4. Fill in instances
5. Fill in the values for slots of the instances

If all the classes, slots, instances and their relationships are properly defined, the ontology for knowledge of a task or a domain is created.

There are many important benefits in applying ontology. First of all, the knowledge is formal, explicit and shared, which means the knowledge is accessible to everyone. With ontology, the common standards of a domain can be established by the experts. People with different background will have opportunities to acquire the knowledge without much professional training. Meanwhile, the taxonomy-based representation is very concise and straight-forward, which decreases ambiguities and errors. Finally, ontologies are machine-oriented. Some of the ontology languages are XML-based, which can be easily shared among different machines. So, currently, ontology is one of the most popular and powerful knowledge engineering methods widely applied in different applications.

In this thesis, Web Ontology Language (OWL) [24], one of the most successful ontology languages recommended by W3C, is adopted. OWL uses XML syntax and is partially mapped to Description Logic, which is a subset of Predicate Logic. Thus OWL provides users with various inference capabilities. Actually, the realization of some OWL reasoners is based on tableau algorithms, which is an algorithm for Description Logic reasoning. OWL consists of three sublanguages: OWL Lite, OWL DL and OWL Full. OWL Full is the most expressive among the three. But there is not any reasoners supporting its inference. In contrary, while promising the decidability, the expressiveness of OWL DL and OWL Lite is sacrificed [25]. Thus Semantic Web Rule Language (SWRL) [26], which supplements OWL DL and OWL Lite with Horn-like rules, was proposed. The DL-safe version of SWRL is also decidable [25].

Moreover, as explained above, ontology has mechanism to describe implicit knowledge. In fact, methods for retrieving the implicit knowledge are based on ontology reasoning. Some of these approaches are derived from Description Logic reasoning. For example, OWL DL is based on *SHIQ* Description Logic. Thus algorithms for Description Logic reasoning, such as tableau algorithms, can be used to infer with OWL DL ontology. Furthermore, many stable reasoners are available for OWL DL reasoning. For example, Protégé, an OWL ontology development platform, provides interfaces for plugging in reasoners like Pellet, FaCT++, Jena and RACER. In this thesis, Pellet is used for ontology reasoning. It supports reasoning with both OWL DL and the DL-safe version of SWRL.

3.4.2 Ontology for Requirement Engineering

As discussed in section 3.3, ontology-base requirement elicitation is a popular topic nowadays. However, there is a long history of applying ontology for requirement engineering. The very first research effort dedicated to utilizing ontologies in the requirement engineering can be dated back to the early 1980s [25]. Since then, a number of ontology-based requirement engineering approaches have been studied, developed and proposed. Among the most notable publications, [27] introduces an ontology-based requirement model that facilitates detecting incompleteness and inconsistency of requirement artefacts, measuring the quality of requirement engineering, and predicting potential changes in later software engineering phases. A very complete group of requirement engineering related ontological relationships is defined in the model. In [28], a minimum model for describing requirement knowledge is presented. Goal, quality constraint and softgoal are proposed as the fundamental ontology concepts in requirement engineering. In addition, a framework for ontology-based requirements elicitation is introduced in [29]. Types of functional requirements as well as their relationships which facilitate requirement elicitation are outlined in the ontology model.

Meanwhile, [15] presents a well-structured ontology-based requirement model called *SoftWiki*, which is capable of capturing and managing the requirement engineering artefacts for all stages of system development.

Although all the approaches introduced above make great contribution to ontology-based requirement engineering, they are not suitable for representing requirement knowledge for automated SPL. [27] places its emphasis on artefacts verification, while the model proposed in [28] is more theoretical than practical. Besides, the objective of the method from [29] is to ease the communication between requirement engineers and clients in requirement elicitation. Similarly, *SoftWiki* [15] is developed for supporting the collaboration of all stakeholders in all software engineering stages.

Actually, contributions from most ontology-based requirement engineering studies fall into the following three categories:

- Improving the quality of the requirement engineering artefacts (e.g. [27], [29], [30]).
- Defining a shared understanding among engineers and clients (e.g. [31], [32], [33]).
- Developing new knowledge-based requirement engineering methods (e.g. [34], [35], [36]).

Issues critical to the realization of SOA-based automated SPL, like providing automatic guidance for product customization and generating service-oriented system specification, are not well covered by these approaches.

3.4.3 Ontology for SOA

As mentioned in Chapter II, the idea of Semantic Web Service is proposed for automating SOA system implementation activities. Different from Feature Driven Development [37] and Model-Driven Achitecture [38], where system functionalities are mapped to system features and platform-independent models, SOA encapsulates

application functionalities into loosely coupled services. Thus, instead of designing and realizing the features or models, software applications can be implemented by discovering, composing and invoking the services in SOA. Moreover, Semantic Web Service methods further specify the web service descriptions on the semantic level, thus suggest solutions for automatic service discovery and composition [4].

Semantic Web Service approaches are also based on ontology. Currently, there are mainly three ontologies developed for Semantic Web Service: Web Service Modeling Ontology (WSMO) [39], Semantic Markup for Web Services (OWL-S) [40] and SOA Ontology [20].

- WSMO is a conceptual model related to Semantic Web Service. It supports the Semantic Web Service deployment and interoperation.
- OWL-S is also an ontology for describing Semantic Web Service. It enables automatically discover, invoke, compose, and monitor web services under specified constraints.
- "SOA Ontology defines the concepts, terminology and semantics of SOA in both business and technical terms" [41]. It creates a foundation for facilitate SOA understanding, SOA related communication, and SOA system modeling. Meanwhile, it potentially, contributes to model-driven SOA implementation [41].

The first two ontologies are relatively low-level. They are techniques for describing concrete Semantic Web Services.

In this thesis, OWL-S is applied. OWL-S is based on the ontology language OWL. It is an ontology of services that makes automatic service discovery and composition possible [40]. The instances of its class *ServiceProfile* describe the characteristics of the services which are used to match clients' requests, while information for service composition is contained in instances of the class *ServiceModel*. When discovering the services, the requestors' *ServiceProfile*s will be matched automatically with service

providers' *ServiceProfile*s through semantic capability matching [42]. If the matching succeeds, the desired services are found. Then the *ServiceModel*s, associated with the discovered services, will carry information about the process of composing and invoking the services. So by reasoning the knowledge contained in *ServiceModel*s, automatic service composition will be performed and the desired functionalities can be obtained [8]. On the other hand, although OWL-S provides descriptions for web service functionalities, it has few mechanisms for non-functional service description [43]. Fortunately, [43] proposes a quality extension for OWL-S to offset this drawback. It inherits from the class *ServiceParameter* defined in OWL-S, which is designed for extending OWL-S with more specific service descriptions.

CHAPTER IV

PROPOSED ONTOLOGY-BASED REQUIREMENT MODEL

4.1 Introduction

This chapter reports the ontology-based requirement model. Different from the ontology-based requirement engineering methods discussed in Chapter III, the model developed in this research integrates the requirement engineering knowledge with service-oriented knowledge. While the key concepts and relationships proposed in [27] and [28] are kept for maintaining the completeness and consistency of the product requirements, a service-oriented decomposition approach is applied for instantiating the modeling, as well as organizing the commonalities and variabilities in SPL. Furthermore, information for directing requirement elicitation, such as ranks of the requirements, is also expressed in the ontology model. Therefore, knowledge contained in the model is expected to guide the automatic product customization and facilitate generating service-oriented system specification for system implementation.

In this chapter, construction of the ontology-based requirement model (Fig. 4.1) is presented according to the ontology engineering steps proposed in [21]. First, concepts of the model are defined. Second, relationships describing the taxonomic hierarchy of the ontology are outlined. Third, as supplement to OWL ontology, this chapter proposes a group of SWRL rules for the model. Finally, instantiation of the ontology is discussed.

4.2 Concepts

The concepts of the model are illustrated in Fig. 4.1 with class diagram notations. In the domain of requirement elicitation, according to their different roles, Requirements can mainly be classified into three categories: *Function*, *Quality* and *Softgoal*. Meanwhile, *Rank* is used to represent the importance of the requirements with respect to the entire product software. Besides, the proposed ontology model also contains concepts to

describe detailed information about a requirement, which offers helps to clients' evaluation.

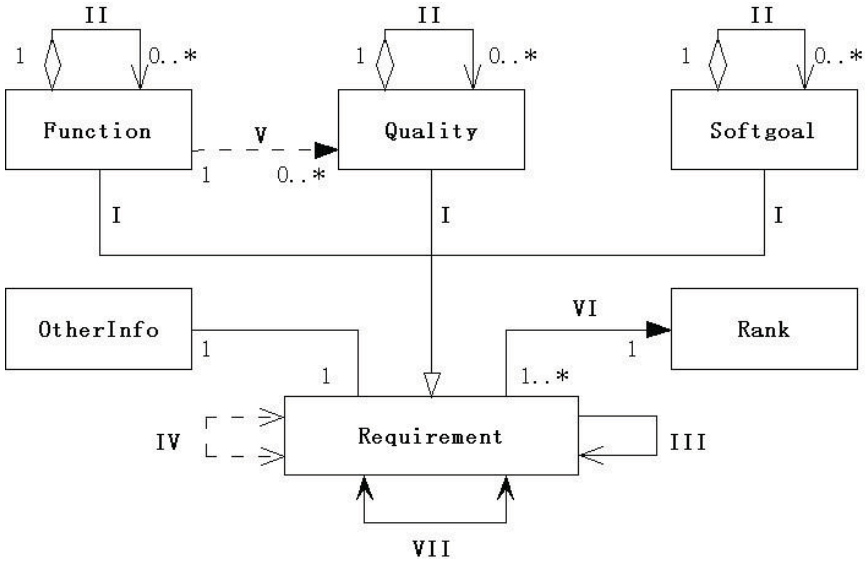


Fig. 4.1: The proposed ontology-based requirement model

- *Requirement*: An instance of *Requirement* is a system feature can be ordered by users. There are three types of *Requirement*: *Function*, *Quality* and *Softgoal*.
- *Function*: An instance of *Function* represents a functionality that users can order. It may be a primitive function offered by the product software or a composition of several primitive functions. From the service-oriented point of view, a function is actually a service. In this research, functions are organized in forest-like structure, where composite functions decompose into less complex composite functions or primitive functions.
- *Quality*: An instance of *Quality* is a non-functional constraint imposed on a function. Mainly, they are used to further specify a functionality. Therefore, a quality instance is always related to a particular function. A quality cannot be chosen if the corresponding function is dropped. Besides, a quality can also be further specified with sub-qualities in a decomposition tree.

- *Softgoal*: Instances of *Softgoal* are also non-functional constraints. However, instead of imposing on a particular function, they describe the global environment within which the product software system works. They are often very abstract, and may be related to a subset of the candidate functions and qualities, but all selected requirements must submit to them. Similar to instances of *Quality*, softgoals can also decompose into sub-softgoals.
- *Rank*: Instances of *Rank* represent the importance of the requirements with respect to the potential system. On the other hand, they also specify the evaluation order of the requirements in the requirement elicitation process.
- *OtherInfo*: Concepts generalized from *OtherInfo* may be general requirement engineering related concepts (e.g. stakeholders) or specific concepts within a domain. Instances of them are used to describe detailed information about the requirements. Users can request such relevant information when evaluating the requirements.

4.3 Relationships

$REQ = FUN \cup QUA \cup SOF$

I. $generalize(REQ, x),\ x \in \{FUN, QUA, SOF\}$

II. $decompose(x, y),\ x \in R,\ y \in R,\ R \in \{FUN, QUA, SOF\}$

III. $rely(x, y),\ x \in REQ,\ y \in REQ$

IV. $contradict(x, y),\ x \in REQ,\ y \in REQ$

V. $associate(x, y),\ x \in FUN,\ y \in QUA$

VI. $hasRank(x, r),\ x \in REQ,\ y \in RANK$

VII. $invalid(x, y),\ x \in REQ,\ y \in REQ$

Fig. 4.2: Notation for the relationships

The relationships between the concepts are depicted in Fig. 4.1, and their notations are outlined in Fig. 4.2. In the proposed method, these relationships not only enable checking the consistency and completeness of the customized requirements, but

also facilitate machines to direct the requirement elicitation.

I. *Generalize*:

An instance of *Function*, *Quality* or *Softgoal* is also an instance of *Requirement*. *Generalize* represents the *IS-A* relationship.

II. *Decompose*:

Requirement *x* decomposes into less complex requirement *y* of the same type. *y* describes part of *x*'s characteristics or is a more detailed alternate to *x*. *x* is called the parent, while *y* is called the child of *x*.

A requirement can decompose into zero children; otherwise it must decompose into at least two children. A child has at most one parent. Logically, a requirement cannot decompose into more complex requirements (e.g. its parent or the parent of its parent). Thus the decompose relationship forms decomposition trees. In practice, it is possible that a requirement participates in the decompositions of several more complex requirements. However, if a requirement is allowed to have two parents, when it is picked up during requirement elicitation, the composition that the requirement is supposed to join in the product software will be unknown. Hence, in this case, two copies of the requirement are required for participating in the two decompositions.

When functions decompose into sub-functions, the parent functions represent functionalities that are the results of their children's composition (i.e. service composition). In other words, a parent implies a composition strategy rather than any concrete functionalities. Only the leaves in a function decomposition tree are primitive functionalities. Besides, it is not necessary to select a composition strategy if one only needs some primitive functionalities.

In a quality or softgoal decomposition tree, the children denote

refinements to the parent. So logically, it actually doesn't make any difference to have a parent quality directly replaced by its children in an instantiated ontology model. The decomposition relationship only eases the requirement elicitation interaction or the ontology instantiation work.

III. *Rely*:

The realization of requirement *x* relies on the implementation of requirement *y*. If *x* is ordered by the clients, *y* must also be selected; otherwise the resulting system will not function properly.

So *Rely* describes the completeness of the requirement elicitation product. In addition, when a requirement relies on two other requirements, this implies it needs them both. In practice, it is possible that the requirement only requires one of them. In this case, two copies of the same requirement are created and a *Contradict* relationship is established between them; then each of the two copies relies on one of the two required requirements.

A parent function relying on its children or children of its children implies the composition strategy requires the involvement of the corresponding children. If a child function relies on its parent or parent of its parent, this indicates the child function is designed deliberately for the composition. Normally, a function relying on another function means the input of one function is the output of the other function.

When a parent quality or softgoal relies on its children, it means the children are essential to the parent constraint. In this research, children qualities and softgoals are supposed to rely on their parents. This promises that during the requirement elicitation process, children qualities and softgoals will not be explored if their parents are abandoned.

Qualities and softgoals may rely on functions. This suggests realization of

the constraints requires the implementation of some functions. Functions may also need some quality or softgoal constraints to function properly. Moreover, if a function relies on a quality, it also relies on the function associated with this quality constraint.

IV. *Contradict*:

Requirement *x* contradicts requirement *y*. Requirement *x* and requirement *y* are not supposed to simultaneously realized in the product software.

Contradict describes the consistency of the elicited requirements. This relationship is symmetric and non-reflective. A requirement cannot contradict its children, parent or the requirements it relies on. A function should not contradict the quality constraints associated with it.

Normally, if two requirements play the same role in the candidate application, which means they represent the same functionality or constraint, there is a *Contradict* relationship between them. In addition, if two requirements cannot be met simultaneously in the product software, they contradict each other.

V. *Associate*:

Function *x* is associated with quality constraint *y*. *y* is a quality constraint that can be imposed on function *x*.

As a quality cannot be realized on the customized software if its associated function is not implemented. *Associate* relationship also implies the quality constraint relies on its corresponding function. Moreover, with the same problem and solution as *Decompose* relationship, two functions are not supposed to be associated with the same quality.

If a composite function is associated with a quality, this suggests the quality constraint is imposed on the composition rather than any primitive functionalities. Constraints for primitive functionalities should be directly

related to the concrete functions. Moreover, as children qualities are refinements to their parents, if a function is associated with a parent quality, it is also associated with the corresponding children qualities. Therefore, if an *Associate* relationship is explicitly defined between a function and a quality, the function is associated with the entire quality decomposition sub-tree which is rooted on the quality. Meanwhile, the parent of the root quality, if there is one, is not supposed to have *Associate* relationship with any functions.

VI. *hasRank*:

Requirement *x* has a rank of *r*. A requirement can have exactly one rank.

During the requirement elicitation process, requirements with higher ranks will be offered to users for evaluation before those with lower ranks. Hence, if a requirement has strong influence on the candidate application or other requirements, it should be assigned with a high rank. Besides, the parent requirements should always have higher ranks than their children. If several requirements are closely related and supposed to be evaluated one after another, they should be of the same rank.

Furthermore, requirements of the highest rank are treated as essential requirements. They represent the common features of the SPL artefacts. As a result, they will be picked mandatorily before the evaluation of any other non-essential requirements. In addition, no requirement should contradict essential requirements.

VII. *Invalid*:

There is an invalid relationship between requirement *x* and requirement y.

Invalid relationships are used to denote the invalidity in the instantiated ontology model. It is applied with rules, and can be generalized into types of more specified Invalid relationships. Types of invalidity will be

presented in the next section.

4.4 Rules

i. $contradict(x,x) \Rightarrow invalid(x,x)$
ii. $decompose(x,x) \Rightarrow invalid(x,x)$
iii. $rely(x,y) \wedge rely(y,z) \Rightarrow rely(x,z)$
iv. $contradict(x,y) \Rightarrow contradict(y,x)$
v. $contradict(x,y) \wedge rely(x,y) \Rightarrow invalid(x,y)$
vi. $contradict(x,y) \wedge decompose(x,y) \Rightarrow invalid(x,y)$
vii. $contradict(x,y) \wedge associate(x,y) \Rightarrow invalid(x,y)$
viii. $decompose(x,y) \wedge decompose(y,x) \Rightarrow invalid(x,y)$
ix. $decompose(x,y) \wedge decompose(y,z) \Rightarrow decompose(x,z)$
x. $decompose(x,y) \wedge decompose(z,y) \Rightarrow invalid(x,y) \wedge invalid(z,y)$
xi. $associate(x,y) \wedge associate(z,y) \Rightarrow invalid(x,y) \wedge invalid(z,y)$
xii. $decompose(x,y) \wedge x \in R \wedge y \in R \wedge R \in \{QUA, SOF\} \Rightarrow rely(y,x)$
xiii. $associate(x,y) \Rightarrow rely(y,x)$
xiv. $rely(x,y) \wedge contradict(y,z) \Rightarrow contradict(x,z)$
xv. $decompose(x,y) \wedge associate(z,x) \Rightarrow associate(z,y)$
xvi. $contradict(x,y) \wedge hasRank(x,r) \wedge r = TopRank \Rightarrow invalid(x,y)$

Fig. 4.3: Rules for the proposed ontology model

Fig. 4.3 illustrates the group of SWRL rules applied in the research. They adopt horn-like presentation. By reasoning with these rules, implicit knowledge for requirement elicitation and ontology instantiation can be retrieved. Followings are the explanations to the rules.

i. *Contradict* relationship is non-reflective.

ii. *Decompose* relationship is non-reflective.

iii. *Rely* relationship is transitive.

iv. *Contradict* relationship is symmetric.

v. *Contradict* relationship and *Rely* relationship are disjointed.

vi. *Contradict* relationship and *Decompose* relationship are disjointed.

vii. *Contradict* relationship and *Associate* relationship are disjointed.

viii. *Decompose* relationship is asymmetric.

ix. *Decompose* relationship is transitive.

x. *Decompose* relationship is inverse-functional.

xi. *Associate* relationship is inverse-functional.

xii. Children qualities and softgoals rely on their parents

xiii. A quality relies on its corresponding function.

xiv. If requirement *x* relies on requirement *y*, *x* contradicts the requirements that *y* contradicts.

xv. If function *z* is associated with quality *x*, *z* is associated with *x*'s children.

xvi. Requirements cannot contradict top rank requirements.

In fact, rules i-xi can be expressed with OWL elements. However, some of them cannot be reasoned with available reasonors. Even if reasoners can deal with them, the invalid relationships will not be explicitly pointed out by the reasoners. Thus rules are applied here.

Rules iii, iv, xiii and xiv reflect the nature of the relevant relationships, while rules xii is used to facilitate the requirement elicitation process.

Besides, rules i, ii, v, vi, vii, viii, ix, x, xi, xv and xvi are used to verify the validity of an instantiated requirement model. During requirement elicitation, the explicitly defined *Decompose* and *Associate* relationships determine the order of requirement evaluation. Therefore, rule xv is not activated in requirement elicitation process. Moreover, rule ix contradicts rule x, and it also violates the decomposition tree structure. But rule ix facilitates discovering the invalidity that parents cannot contradict the children of their children. Hence, it is applied but not activated together with rule x or in requirement elicitation process.

4.5 Ontology Instantiation

Instantiating the ontology model is actually the domain engineering process in

SPL engineering.

Before instantiating the ontology model for a specific type of software systems, the requirement engineers must analyze this type of service-oriented applications. Decompose these systems into primitive services and find out the commonalities and variabilities. Then they can instantiate the model according to the following procedure presented in Fig. 4.4.

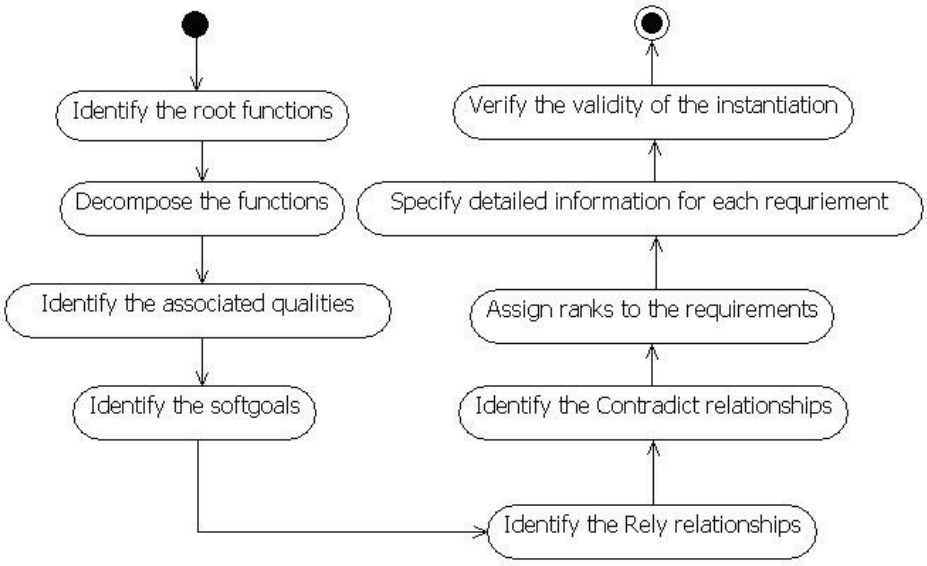


Fig. 4.4: Procedure of instantiating the ontology model

1. Identify the main functions which are roots of the decomposition trees.
2. For each of the roots, if it represents certain composition strategies, identify the children that contribute to the composition. Then establish the *Decompose* relationship between the parent and children. If the children are also decomposable, repeat this decomposition process, until all primitive functions are discovered.
3. Find the corresponding quality constraints that can be imposed on the functions. Organize the qualities with decomposition trees. As all the qualities constraints contained in a decomposition sub-tree are related with the same function, establish an *Associate* relationship between the corresponding

function and the root of the sub-tree. *Associate* relationships between children and the corresponding function are not supposed to be explicated defined. In requirement elicitation process, when a function is picked by users, its associated qualities will be pre-evaluated immediately. However, the children qualities are expected to be explored in a later stage.

4. Identify the softgoals, and decompose them if necessary.
5. Establish the *Rely* relationship for the requirements.
6. Establish the *Contradict* relationships.
7. Identify the essential requirements, and assign ranks to the requirements with respect to their importance and expected positions in the elicitation process.
8. Specify the detailed descriptions for each requirement. Bind each requirement with the corresponding service description which will be used in generating the service-oriented output. Details will be discussed in Chapter V.
9. Verify the validity of the instantiated ontology model. Make modifications if necessary. Moreover, the generally acknowledged facts, like requirement which could not decompose into exactly one child, should also be checked manually.

Then a valid instance of the ontology model is built. Typically, for a type of medium-sized software systems like online book shopping service, there will be dozens of requirements and more than a hundred relationships created in the instantiated ontology model.

CHAPTER V

PROPOSED REQUIREMENT ELICITATION METHOD

5.1 Introduction

A frame-based dialogue system is developed in this thesis, which takes the instantiated ontology model as knowledge base. It is applied to elicit users' demands through human-machine interaction. Though to maintain the completeness and consistency of the customized requirements is very complicated and requires ontology reasoning, interactions for requirement elicitation are actually a group of slot-filling tasks. Questions such as whether users need a specific requirement will be proposed by the machine, and users will respond with their decisions on the very requirement. Therefore, users know what they are going to do and how it is going to be done, which means the requirement elicitation process can be modeled as a set of slot-filling subtasks, while the utterances, slots as well as value options for each slot will be retrieved from the knowledge base, hence a framed-based dialogue system is capable of handling the interactions for requirement elicitation, in spite of its limited communication ability.

In this chapter, the structure of the dialogue system, the requirement elicitation process and the output of the elicitation will be discussed.

5.2 Structure of the Proposed Dialogue System

The frame-based dialogue system designed in this research consists of four components: interface, I/O controller, dialogue manager and knowledge base (Fig. 5.1).

The dialogue interface is text-based. It displays machine generated utterances and provides one slot for users to fill in. Typically, the utterances will be questions like "Would you like to select the requirement …?". Users are expected to answer "Yes" or "No". Then the users' response will be passed onto the I/O controller. It will try to match the input with a set of predetermined information. If the matching fails, an utterance that

asks users to correct their response will be generated by the I/O controller, sent to the interface and get displayed as the output of current interaction. Otherwise, the input will be converted into format processable to machines and passed onto the dialogue manager. The dialogue manager then knows users' decision on the requirement currently being evaluated. It will consult the ontology knowledge base with the decision, and customize the requirements based on the related requirement knowledge as well as the input. After that, an output will be generated by the dialogue manager according to the result of the customization and sent to the I/O controller. The I/O controller will convert the output into natural language and have it displayed by the interface, which will initiate the next round of interaction.

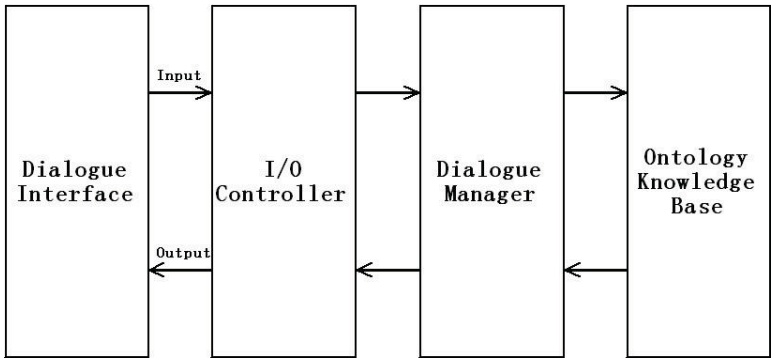


Fig. 5.1: Structure of the proposed dialogue system

5.3 Process of Requirement Elicitation

Before the commencement of requirement elicitation, the implicit knowledge (e.g. indirect relationships) contained in the instantiated ontology model will be extracted by reasoning.

The requirement elicitation process (Fig. 5.2) is divided into two stages. First, requirement elicitation will be conducted under the guidance of machine. Then users will have chances to change their decisions made in the first stage and further customize the product software.

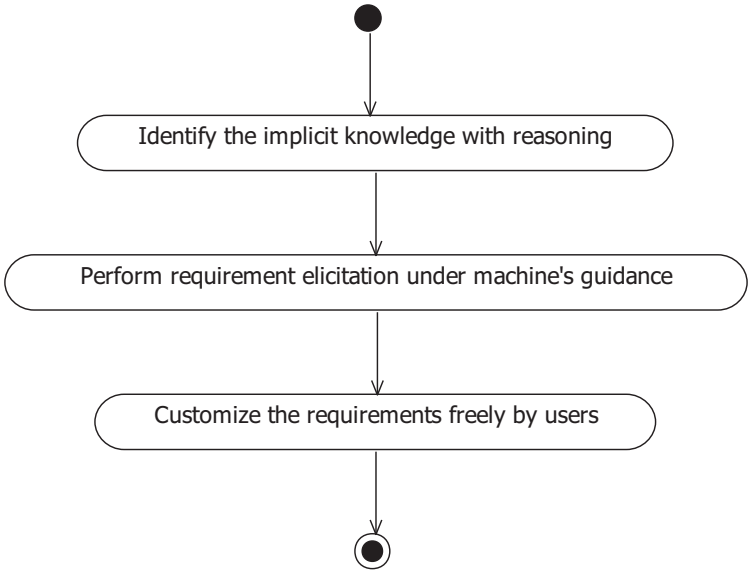


Fig. 5.2: Process of interactive requirement elicitation

5.3.1 Machine-directed Requirement Elicitation

During the first stage, each requirement will be offered to users for evaluation in turns. At the beginning, all essential requirements will be picked automatically without being evaluated. Then the functions will be evaluated, and the evaluation of qualities will follow. Finally, the softgoals will be customized. Among the requirements of the same category, one with higher rank will be evaluated before those with lower ranks.

Fig. 5.3 presents the pseudo code for evaluating the requirements. When evaluating a requirement, there could be four cases.

1. If a requirement *R* is essential to the system, actions for selecting the requirement will be performed.
2. If the requirement R is non-essential and pre-selected, actions for selecting a requirement will be performed. These actions include call *selectRequirement* to have *R* selected; call *preSelectRequirement* to have the requirements that *R* relies on pre-selected; call *preDropRequirement* to have the requirements that *R* contradicts pre-dropped; call *preEvaluateRequirement* to have the

requirements that *R* decomposes into pre-evaluated; and if *R* is a function, call *preEvaluateRequirement* to have the qualities that *R* is associated with pre-evaluated.

```
1.   FOR each requirement R to be evaluated
2.       IF R is essential to the system THEN
3.           CALL PerformRequirementSelecting with R
4.       ELSE IF R is pre-selected THEN
5.           CALL PerformRequirementSelecting with R
6.       ELSE IF R is pre-dropped THEN
7.           CALL PerformRequirementDropping with R
8.       ELSE
9.           CALL evaluateRequirement with R
10.          IF R is to be selected THEN
11.              CALL PerformRequirementSelecting with R
12.          ELSE
13.              CALL PerformRequirementDropping with R
14.          END IF
15.      END IF
16.  END FOR

17.  PerformRequirementSelection with R
18.          CALL selectRequirement with R
19.          CALL preSelectRequirement with the requirements R relies on
20.          CALL preDropRequirement with the requirements R contradicts
21.          CALL preEvaluateRequirement with the requirements R decomposes into
22.          IF R is a function THEN
23.              CALL preEvaluateRequirement with the qualities R is associated with
24.          END IF

25.  PerformRequirementDropping with R
26.          CALL dropRequirement with R
27.          CALL preDropRequirement with the requirements that relies on R
```

Fig. 5.3: Pseudo code for requirement evaluation process

3. If the requirement *R* is non-essential and pre-dropped, action for dropping a requirement will be performed. These actions include call *dropRequirement* to have *R* dropped and call *preDropRequirement* to have the requirements that rely on *R* pre-dropped.
4. If the requirement *R* is non-essential and has not been pre-selected or pre-dropped, *evaluateRequirement* will be called to have *R* evaluated by users. Then if users choose to select *R*, actions for selecting a requirement will be

performed. Otherwise, actions for dropping a requirement will be performed.

Followings are the explanations to the subroutines used in the pseudo code.

- *selectRequirement* will put the requirement to a set to have it labelled as "selected" if it is unlabelled.
- *droptRequirement* will put the requirement to a set to have it labelled as "dropped" if it is unlabelled.
- *evaluationRequirement* will present the requirement to users through dialogue interface. Users can choose to select or drop the requirement, or request detailed description to the requirement before making the decision.
- *preSelectRequirement* will put the requirement to a set to have it labelled as "pre-selected" if it is unlabelled.
- *preDropRequirement* will put the requirement to a set to have it labelled as "pre-dropped" if it is unlabelled.
- *preEvaluateRequirement* will first call *evaluateRequirement* to have the requirement evaluated if it haven't been evaluated yet. Then based on users' choice, *preSelectRequirement* or *preDropRequirement* will be called. Moreover, if the requirement is to be pre-selected, pre-select the requirements it relies on, and pre-drop the requirements contradicting it. Otherwise, pre-drop the requirements relying on it.

During the requirement elicitation process, all requirements will be expanded at most once (in pre-evaluation or in formal evaluation). Here, expanding a requirement means retrieving the detailed information of the requirement. Besides, each *Decompose* and *Associate* relationship will be visited at most once by the parents. Each *Rely* and *Contradict* relationship will be visited at most twice by the two involved requirements. Therefore, let V be the number of requirement instances in the ontology and E be the number of the four relationships. Then the complexity of the algorithm is $O(V+E)$.

5.3.2 User Requirement Customization

During the second stage, users can order the machine to select or drop an arbitrary requirement. In other words, they can change their decisions made in the first stage. If a selected requirement is to be dropped, the selected requirements that rely on it will also be dropped. If a dropped requirement is to be selected, the selected requirements that contradict it will be dropped and the dropped requirements that it relies on will be selected. Therefore, the completeness and consistency of the customization are maintained all over the two stages.

5.4 Output of Requirement Elicitation

5.4.1 Output Overview

To build software with SOA methods, the services must first be discovered. Thus the output of the requirement elicitation process is a set of service descriptions which can be used to discover the services satisfying the selected requirements.

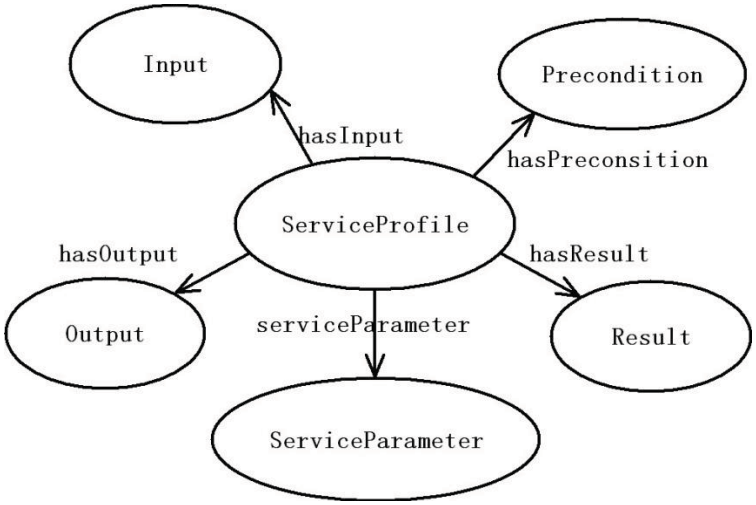


Fig. 5.4: Selected classes and properties in OWL-S functionality description

In this research, OWL-S is used to describe the services. OWL-S makes use of an instance of *ServiceProfile* to represent the information needed to discover a service. *ServiceProfile* has four functionality related properties: *hasInput*, *hasOutput*,

hasPrecondition and *hasResult*. They associate an instance of *ServiceProfile* with respective instances of *Input*, *Output*, *Precondition* and *Result*. And an instance of *Input*, *Output*, *Precondition* and *Result* would respectively represent: the information the service requires to work, the message the service returns, the condition within which the service executes properly and the effects as well as outputs of the service execution.

Quality constraints to services are not explicitly defined in OWL-S. The extension proposed in [43] is used to describe the qualities and softgoals. In this extension, *Quality_Property* which is generalized from OWL-S class *ServiceParameter* is used to represent a constraint. For those measurable qualities, instances of *Attribute*, inherited from *Quality_Property*, can be used to express them as well as their metrics. For those abstract constraints (e.g. softgoals), *Quality_Model*, which connects *Quality_Property* via property *defined_by*, can be used to specify their standards. Instances of *ServiceProfile* are associated with instances of *Quality_Property* through property *serviceParamter*.

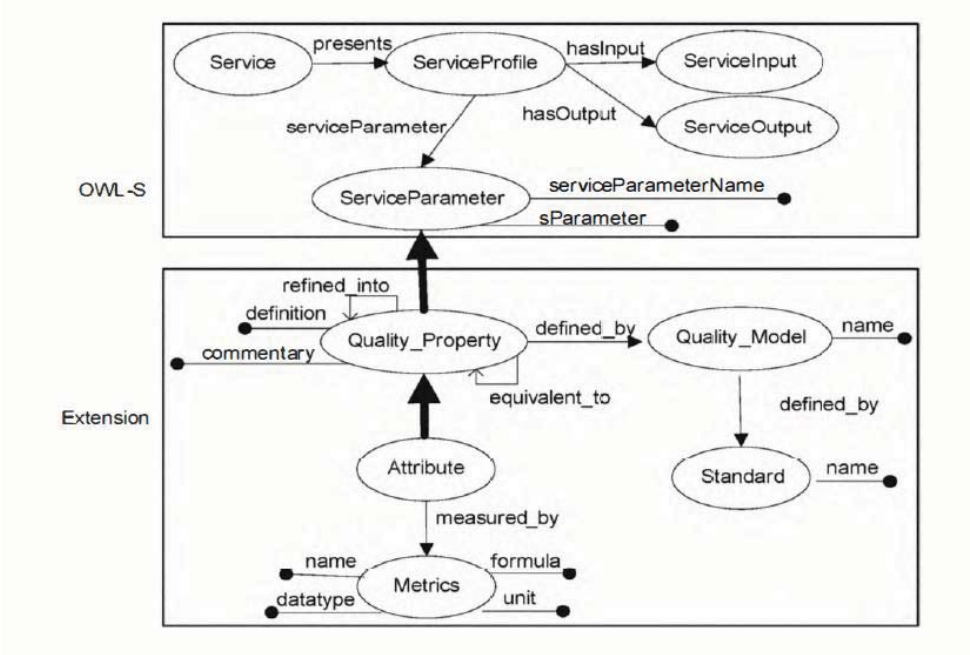


Fig. 5.5: OWL-S quality extension proposed in [43]

So for each requirement, it is related with a piece of service discovery

information: either functionalities represented with *Input*, *Output*, *Precondition* and *Result*, or quality constraints represented by *Quality_Property*. Converting requirements into service descriptions is actually to combine the information that belongs to the selected requirements. As requirements are organized in decomposition trees, the selected requirements also form a group of selected sub-trees. The integration process can be carried out in a way of merging nodes in the selected sub-trees.

5.4.2 Output Generation

The output generation process (Fig. 5.6) is divided into four phases.

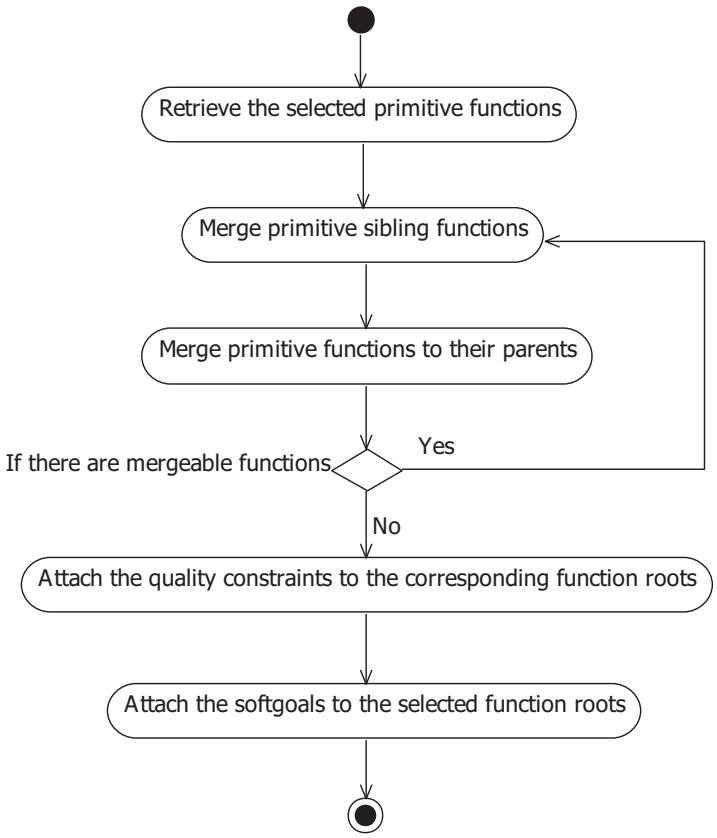


Fig. 5.6: Procedure of generating service description

1. Retrieve the selected primitive functions. Normally, primitive functions have information about input and output of the service. If there is a *Rely*

relationship between two selected sibling primitive functions, it implies that some inputs of one primitive function are from the outputs of the other function. Then those relevant inputs and outputs are not necessarily to be expressed in the service description. So merge the two siblings with algorithm outlined in Fig. 5.7.

```
1.  //Let f1 and f2 be the selected siblings
2.  //Let f1 rely on f2
3.  //Let f3 be the result of the merge
4.  CREATE function f3
5.  FOR each f2's input, precondition or result
6.      IF f3 doesn't have it THEN
7.          Attach it to f3's inputs, preconditions or results
8.      END IF
9.  END FOR
10. FOR each f1' output, precondition or result
11.     IF f3 doesn't have it THEN
12.         Attach it to f3's outputs, preconditions or results
13.     END IF
14. END FOR
15. FOR each f2's output
16.     IF the output is not an input of f1
17.         AND f3 doesn't have it
18.     THEN
19.         Attach it to f3's outputs
20.     END IF
21. END FOR
22. FOR each f1's input
23.     IF the input is not an output of f2
24.         AND f3 doesn't have it
25.     THEN
26.         Attach it to f3's inputs
27.     END IF
28. END FOR
29. Establish Decompose relationship between f1, f2's parent and f3
30. FOR each Rely relationship f1 and f2 participate in
31.     IF the relationship is not between f1 and f2
32.       AND it involves another requirement r4
33.     THEN
34.         Establish the same relationship between f3 and r4
35.     END IF
36. END FOR
37. Remove f1 and f2 from the selected requirement set
38. Put f3 to the selected requirement set
```

Fig. 5.7: Pseudo code for merging siblings

The algorithm first creates a new function. Then attach inputs, outputs, preconditions and results of the two siblings to the new function, and removing the unnecessary inputs and outputs. Establish the same *Decompose* and *Rely* relationships that the two siblings participate in for the new function, while the relationships between the two siblings should be eliminated. Then replace the two siblings with the new function in the corresponding selected sub-tree.

Repeat the sibling merging process until no more primitive siblings can be merged any more.

2. Merge the primitive children functions into their parents in the selected function sub-trees. Composite functions usually don't contain information about input or output. Since they represent composition strategies, they are associated with preconditions and results. The children should be merged to their parents so as to make descriptions for the compositions complete. The algorithm for merging children into parent is presented in Fig. 5.8.

   The algorithm first attaches the child's inputs, outputs, preconditions and results to the parent. Then establish the same *Rely* relationships that the child participates in for the parent, and remove the child from selected requirement set.

   It is possible that a parent function is selected but none of its primitive children is selected. If this happens and no other requirements rely on the parent function, it is supposed that users don't really need this function. Hence, it will be removed as if it has never been selected. If some selected requirements rely on it, it cannot be removed directly. In this case, there will be some default primitive functional descriptions predefined for the composite function. With these descriptions, the composite function can be treated as a primitive function.

After primitive children are merged into their parents, the merged parent forms new "primitive" functions. It is also possible that the new primitive functions rely on some of their siblings. In this case, run the merging sibling algorithm for them. Repeat merging the new primitive children into parents until only the roots of the selected sub-trees are left.

```
1.  //Let p1 be the selected parent
2.  //Let c2 be the selected child
3.  FOR each c2's input, output, precondition or result
4.      IF p1 doesn't have it THEN
5.          Attach it to p1's inputs, outputs, preconditions or results
6.      END IF
7.  END FOR
8.  FOR each Rely relationship c2 participates
9.      IF the relationship is not between p1 and c2
10.        AND it involves another requirement r3
11.     THEN
12.         Establish the same relationship between p1 and r3
13.     END IF
14. END FOR
15. Remove c2 from the selected requirement set
```

Fig. 5.8: Pseudo code for merging child into parent

3. Attach the selected leaf qualities to the corresponding function roots. Parent qualities don't represent any concrete constraints, so they don't carry any service related information. For each leaf quality, first find the selected function that is associated with it. Then further trace the root of selected sub-tree which contains the function. Attach the service information carried by the leaf quality to the root's service description. If a parent quality is selected but none of its children is selected, the same solution for function will be applied to handle it.
4. Attach the selected leaf softgoals to the function roots. Parent softgoals also don't carry service related information. Attach the leaf softgoals' information to all the selected function roots. If a parent softgoal is selected but none of its children is selected, the same solution for function and quality will be applied.

Finally, the integrated service descriptions (i.e. the OWL-S files), carried by the roots of the selected function sub-trees, form the output of the requirement elicitation. When the service for a root function is discovered, descriptions to the primitive services, on which the root is built, are also carried by this service's specification. Therefore, when generating the output, service information for the primitive functions is merged into the root functions' descriptions, and there is no need to describe the primitive functions separately. On the other hand, merging service descriptions into root functions is optional. Users may choose to have the quality and softgoal constraints attached on the primitive functions directly, and then discover and evaluate the selected primitive functions separately. As a result, instead of applying the service compositions offered by the service providers, they can define the service composition strategies on their own.

5.5 Considerations for System Implementation

Implementation of service-oriented systems involves two stages: service discovery and service composition. Solutions to automatic service discovery and composition with OWL-S are discussed in [8]. A brief overview is presented in this section.

Different from UDDI, which supports keyword based service discovery, OWL-S can describe the semantics of the services. With semantic capability matching, limitations of syntactic service matching can be overcome, and service discovery will be more intelligent. Basically, there are two types of service capability presentation: one is to use an extensive class hierarchy to specify the detailed functionalities; the other is to define the state transformation resulted from the execution of the service. OWL-S supports both presentations [42], which means capability matchmaking with OWL-S is more promising. Besides, plenty of matching as well as other service discovery algorithms have been proposed in literature for automating OWL-S based service discovery (refer to [8] for the list of algorithms). Therefore, with the OWL-S based requirement elicitation outputs,

appropriate services can be found automatically.

When OWL-S based services are discovered, their compositions should be performed in order to provide the requested functionalities. In OWL-S, the composition is described with instances of *ServiceModel* which will be retrieved when the services are discovered. Each *ServiceModel* actually represents process models of composing and executing the corresponding service. The process models can be used to construct generic procedures and plans for implementing the functionalities. Several AI planning techniques have been proposed for automatic OWL-S based service composition [8]. Although most of the approaches are not mature enough and further work for realizing automatic OWL-S service composition is still needed, automated composition for OWL-S based information-gathering services can already be performed [42].

By now, the complete process from requirement elicitation to system realization has been presented. With an SPL approach, the service-oriented software is ordered and expected to be implemented automatically. Furthermore, by directly interacting with the text-based dialogue system developed in this thesis, users can properly customize the expected software product without producing any errors.

CHAPTER VI

IMPLEMENTATION AND CASE STUDY

5.6 Implementation

The dialogue system was developed in Java 6.0 on x86 platform with Windows operating system. Eclipse IDE (3.6) was used to facilitate coding and debugging.

Fig. 6.1 depicts the interface of the dialogue system. It is divided into three parts. The utterances generated by the dialogue manager are displayed in the upper left textbox. Users can type their response in the lower left textbox. Meanwhile, the three lists on the right side contain the selected, dropped and to-be-evaluated requirements respectively.

The ontology is built with Protégé ontology editor (4.2), and Pellet (2.2.2) is applied as the ontology reasoner. Pellet supports reasoning with both OWL DL and DL-safe SWRL. The version of OWL language adopted in the thesis is 1.1.

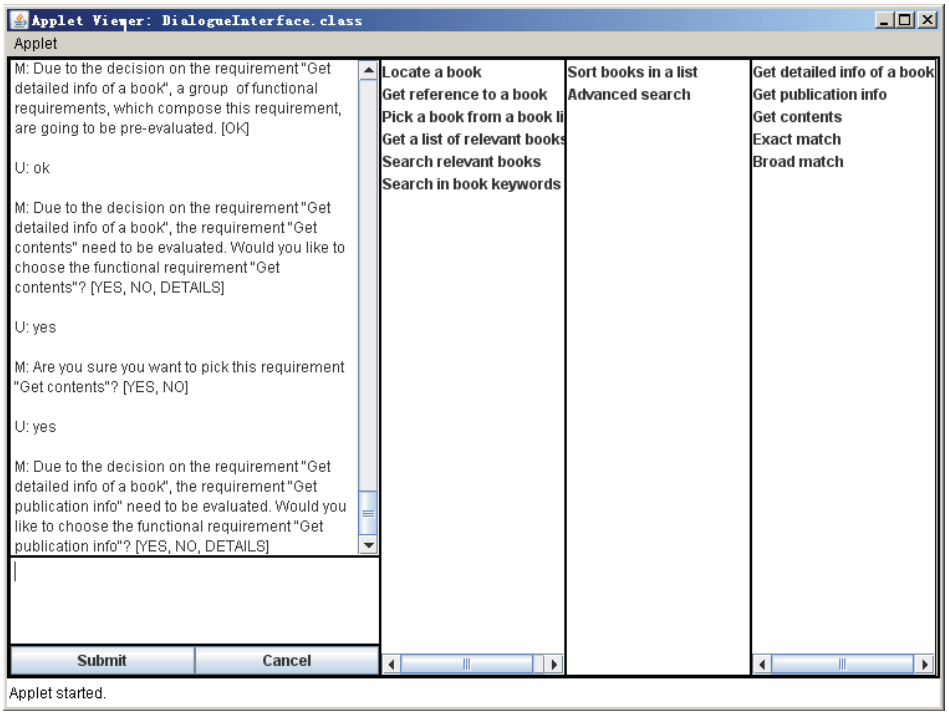


Fig. 6.1: Interface of the proposed dialogue system

The concepts, object relationships, data relationships, rules and instances of the

requirement model created with Protégé editor are shown below. Fig. 6.2 illustrates the concepts for the ontology-based ontology model, while the relationships between these concepts are presented in Fig. 6.3. In Fig. 6.4, the group of data relationships describing the concepts with primitive data types is created. Meanwhile, the SWRL rules designed for the requirement model are given in Fig. 6.5, followed by Fig. 6.6, where instances for the case study, which will be discussed in the next section, are constructed with the ontology model.

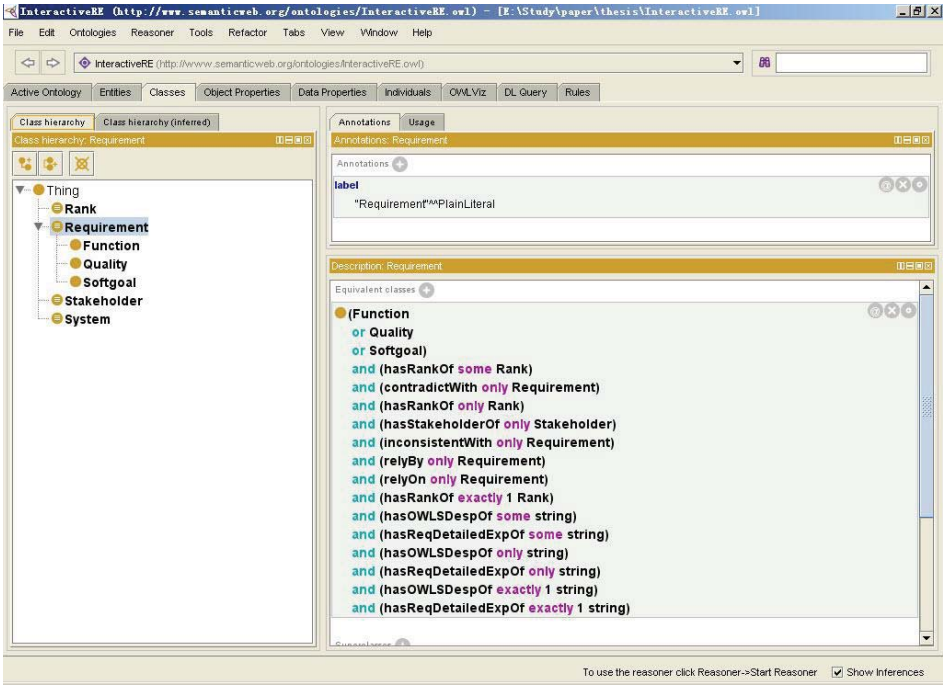


Fig. 6.2: Classes created for the proposed ontology model

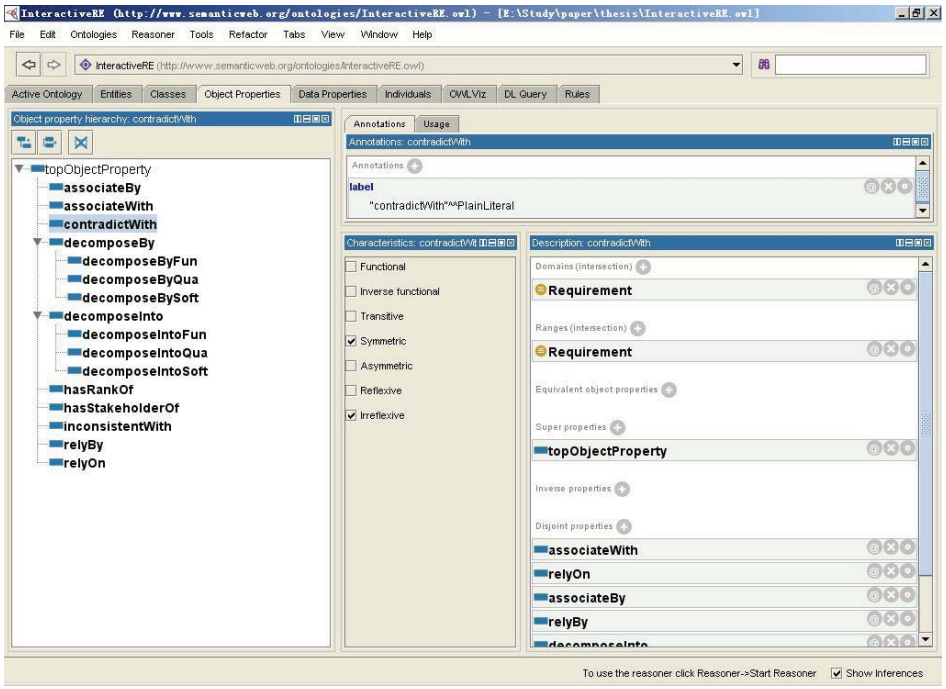


Fig. 6.3: Object properties created for the proposed ontology model

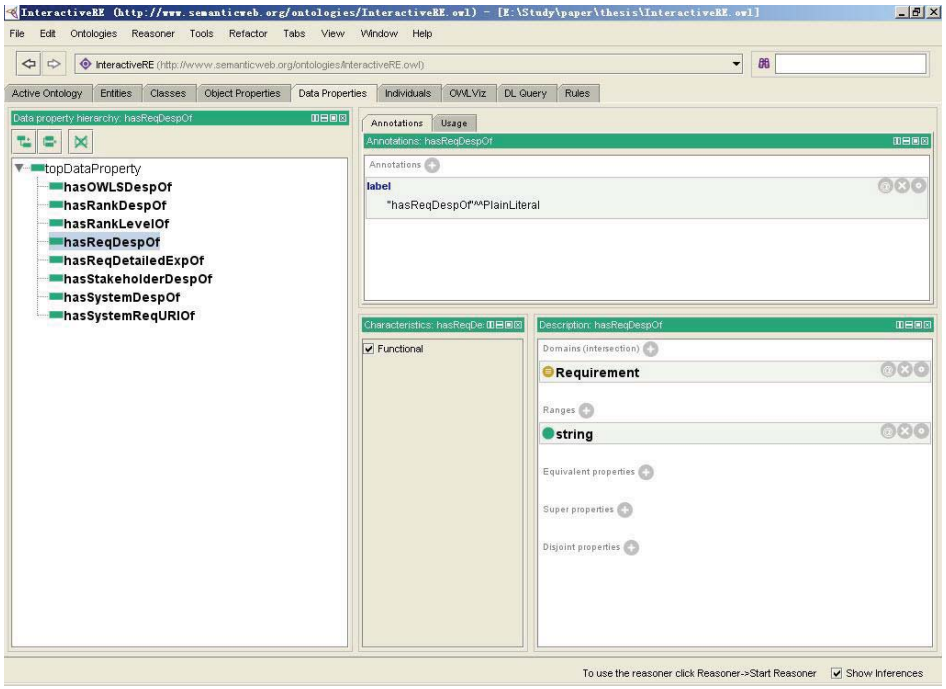


Fig. 6.4: Data properties created for the proposed ontology model

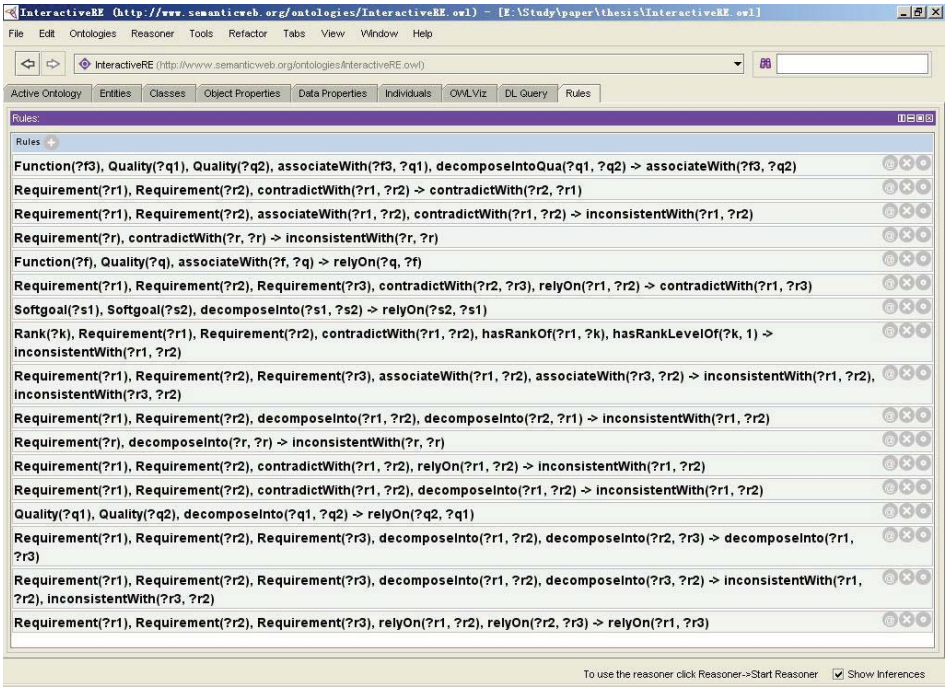


Fig. 6.5: Rules created for the proposed ontology model

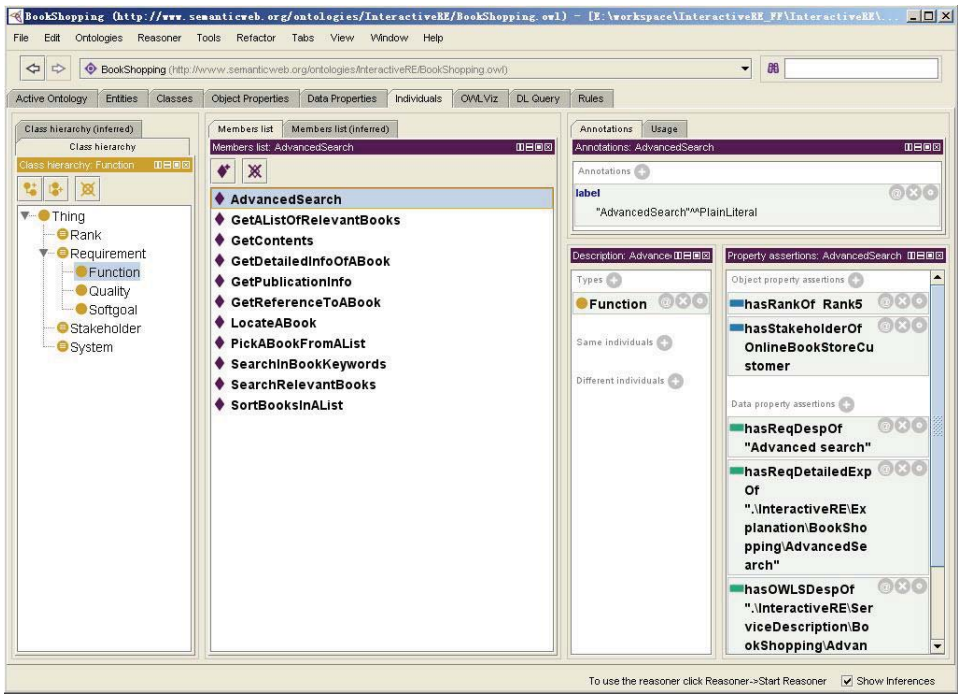


Fig. 6.6: Instances created with the proposed ontology model

5.7 Case Study

5.7.1 Case Overview

An online book shopping system is used as a case study in this research. The structure of a typical but simplified online book shopping system is illustrated in Fig. 6.7. There are basically four modules: book locating, cart management, account management and order placing. Book locating module is responsible for book searching and retrieving book information; cart management module provides a list where users can save the references of the books they want to buy; account management module manages users' personal, delivery and payment information; order placing module gathers information such as shopping list, payment, delivery, and total price, and helps users to place the order. It is assumed that account management is not necessary for an online book shopping system. Users can specify necessary information for each purchase without having it saved in the online bookstore.

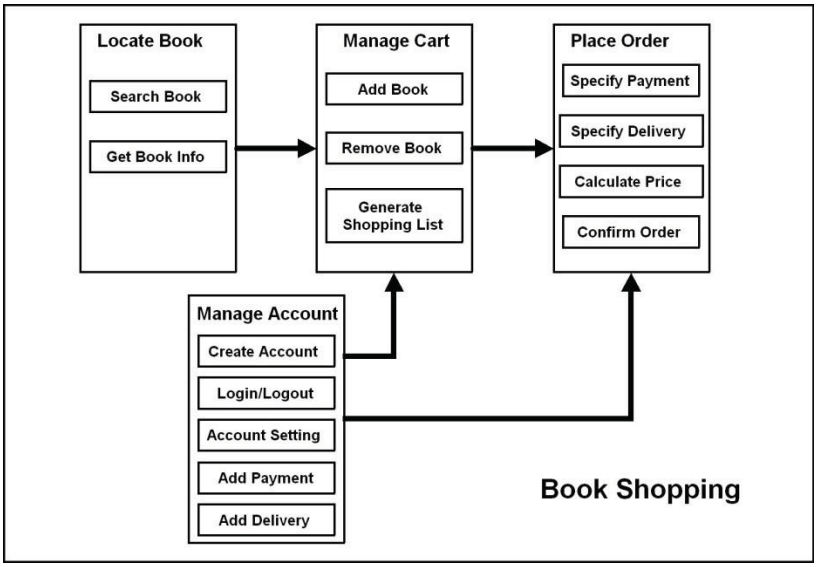


Fig. 6.7: Functionalities of an online book shopping system

The requirement model was instantiated for the entire book shopping system (refer to Appendix A). There are totally 52 functions, 6 qualities and 2 softgoals. For the relationships, there are 48 *Decompose*s, 102 *Rely*s, 6 *Contradict*s and 6 *Associate*s. It is

too complicated to explain all the details. In this thesis, process of customizing the book locating module with the proposed approach is presented in details as the case study.

5.7.2 Book Locating Service

1. Function Decomposition

First, the ontology model is instantiated with requirements of the book locating module. Basically, book locating module provides two functionalities: get reference to a book and get detailed information about a book. Details of a book may include the publication information, the contents, sample chapters and so on. Here publication information and contents are used as examples. In addition, in order to get the reference of a book, the most common method is search. Search will return a list of relevant books, so users need to point out the very book from the list. In addition, to facilitate users in finding the book among a list of books, sorting could be applied. Thus getting a list of books may contain two sub-processes: search a book and sort the search results. Furthermore, there are two ways of book searching. One is to match user inputs with the predefined keywords of the books. The other is advanced composite search. Users may provide detailed information such as authors and publication to narrow down the search domain. There are two levels of keywords matching: broad match and exact match. Exact match tries to return the results that are most relevant to the inputs, while broad match allows returning something appearing similar to the inputs but not exactly related to the inputs. On the other hand, broad match may return something unexpected but interesting. Thus they are two different levels of search quality constraints, and mutually exclusive.

Now, book locating is fully decomposed into primitive functions and quality constraints. Followings are descriptions to each of the functions.

- *Search in book keywords*: input – phrases (from users); output – a list of relevant books and a list of references to the books

- *Advanced search*: input – the fields to be matched and phrases for each field (from users); output – a list of relevant books and a list of references to the books
- *Search relevant books*: input – (from sub-functions); output – (from sub-functions)
- *Sort books in a list*: input – the sorting order (from users) and a list of book references (from *Search relevant books*); output – a list of relevant books and a list of references to the books
- *Get a list of relevant books*: input – (from sub-functions); output – (from sub-functions)
- *Pick a book from a list*: input – book index in the list (from users) and a list of book references (from *Get list of relevant books*); output – a reference to a book
- *Get reference to a book*: input – (from sub-functions); output – (from sub-functions)
- *Get publication info*: input – a reference to a book (from *Get reference to a book*); output – publication information of a book
- *Get contents*: input – a reference to a book (from *Get reference to a book*); output – contents of a book
- *Get detailed info of a book*: input – (from sub-functions); output – (from sub-functions)
- *Locate a book*: input – (from sub-functions); output – (from sub-functions)

2. Ontology Instantiation

Fig. 6.8 presents the instantiated ontology model.

Broad match and *Exact match* are mutually exclusive, so there is a *Contradict* relationship between them. They are quality constraints for *Search in book keywords*. As

a result, they are related with *Search in book keywords* via *Associate* relationship. Meanwhile, *Search in book keywords* is essential to *Search relevant books*; *Search relevant books* is essential to *Get a list of relevant books*; *Get a list of relevant books* and *Pick a book from a list* are essential to *Get reference to a book*; *Get reference to a book* is essential to *Locate a book*. Thus there are *Rely* relationships pointing from the parents to the children. In addition, *Sort books in a list* relies on the output of *Search relevant books*; *Pick a book from a list* relies on the output of *Get a list of relevant books*; *Get detailed info of a book* relies on the output of *Get reference to a book*. So there are *Rely* relationships between each pair. Meanwhile, *Pick a book from a list* can only contribute to *Get reference to a book*; *Get reference to a book* can only contribute to *Locate a book*. As a result, there are *Rely* relationships from the two children to the two parents.

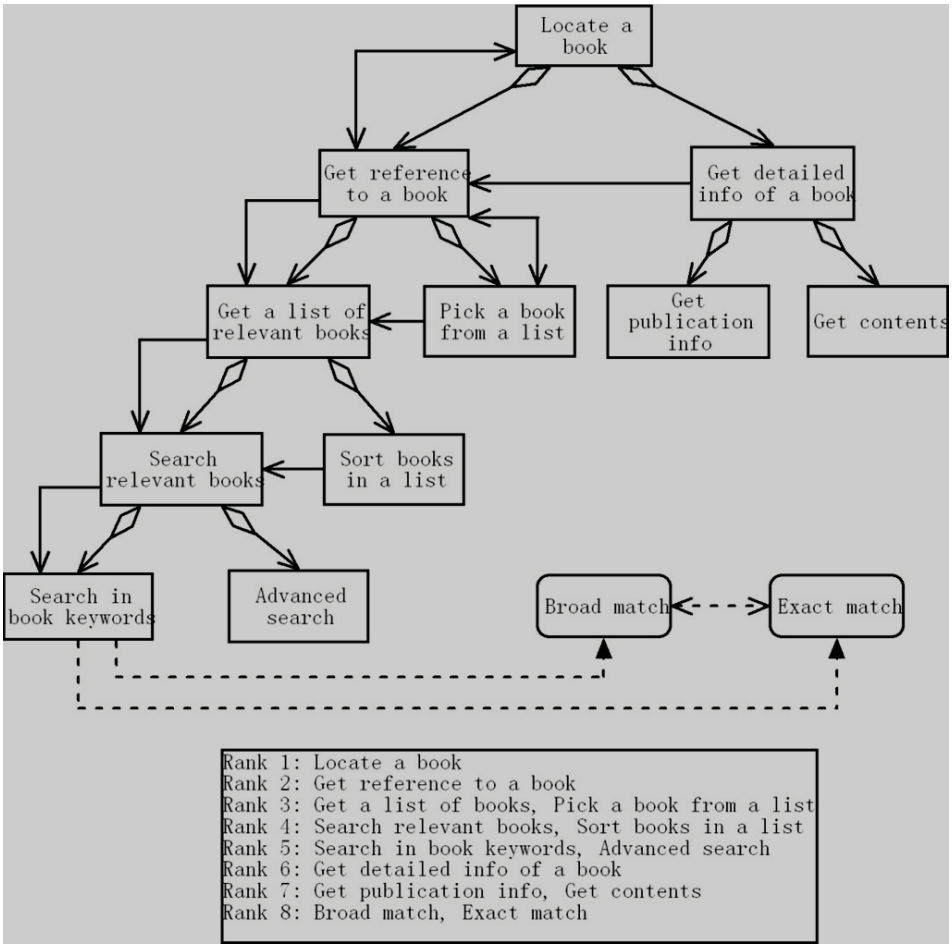


Fig. 6.8: Requirement model instantiated with book locating service

Ranks of the requirements are also shown in Fig. 6.8. As the core of *Locate a book* is to get the reference of the book, requirements describing *Get reference to a book* are expected to be evaluated first. Thus *Get detailed info of a book* as well as its children has a relatively low rank.

3. Requirement Elicitation

When the ontology model is ready, requirement elicitation can be performed. The sample utterances between users and machine are presented in Fig. 6.9 - 6.15. In this case, users need the software system to retrieve detailed information about a book. So they only want to search a book with exact matching and retrieve all the available information about the book.

In Fig. 6.9, users are first welcomed and informed that the essential requirement *Locate a book* is selected mandatorily. Then the two sub-requirements *Get reference to a book* and *Get detailed info of a book* should be pre-evaluated. But *Get reference to a book* is relied by *Locate a book*. Thus it has already been pre-selected, and its pre-evaluation is skipped. *Get detailed info of a book* is pre-evaluated, but users need more explanation to it. After the users know the role of *Get detailed info of a book* in the system to be built, they select it.

Fig.6.10 shows the evaluation of *Get reference to a book* as well as its two children. As *Rely* relationship is transitive, they are all relied by *Locate a book*, and have been pre-selected. In Fig. 6.11, *Search relevant books* and *Sort books in a list* are evaluated. Then it's the turn to evaluate requirement *Search in book keywords* and to pre-evaluate its two associated quality constraints. During this process, the users input some information that the machine cannot handle (Fig. 6.12). Then the machine notifies the users to correct their response. With the interaction presented in Fig. 6.13 – Fig. 6.15, users pick the functions and quality constraints they need. After that they are not willing to change their decisions, and the requirement elicitation process is finished.

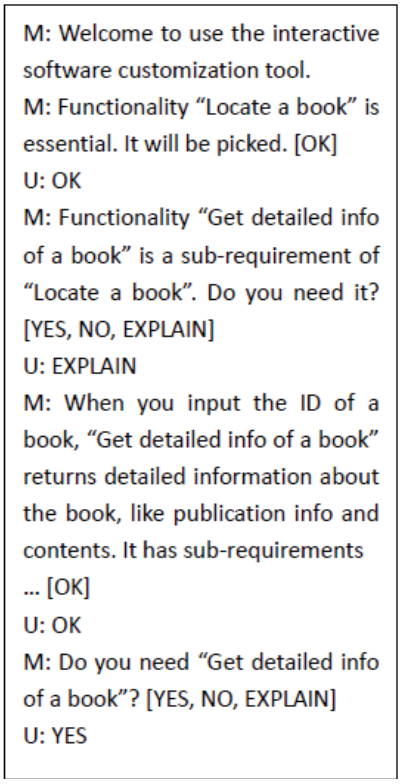


Fig. 6.9: Dialogue utterances for customizing book locating service (part 1)

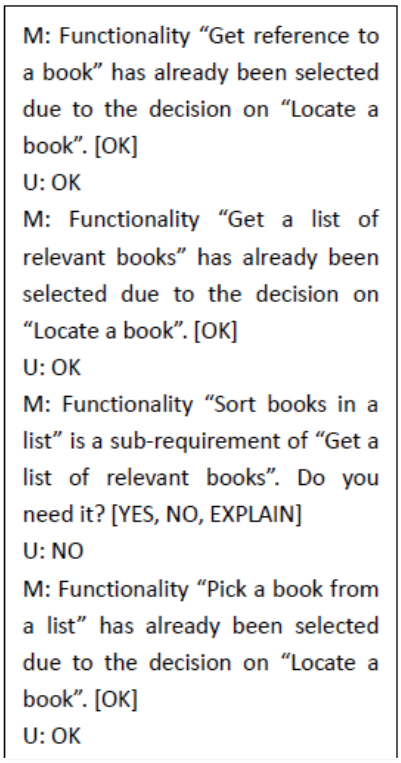


Fig. 6.10: Dialogue utterances for customizing book locating service (part 2)

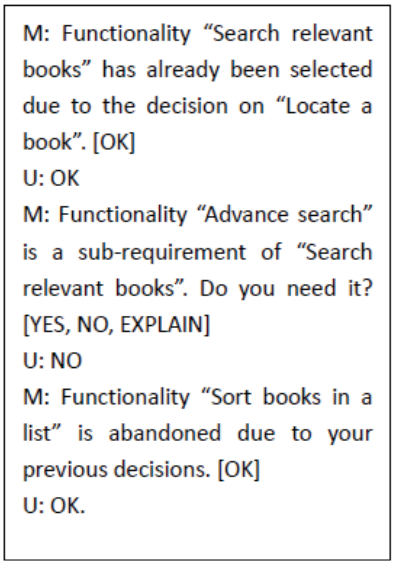


Fig. 6.11: Dialogue utterances for customizing book locating service (part 3)

M: Functionality "Search in book keywords" has already been selected due to the decision on "Locate a book". [OK]
U: OK
M: Quality constraint "Broad match" is related to "Search in book keywords". Do you need it? [YES, NO, EXPLAIN]
U: What
M: Your response cannot be recognized. Please answer with the suggested responses. [OK]
U: OK
M: Quality constraint "Broad match" is related to "Search in book keywords". Do you need it? [YES, NO, EXPLAIN]
U: EXPLAIN
M: When you search a book, not only books with keywords exactly like your inputs will be returned, but books with keywords similar to your inputs will also be returned. If you choose constraint "Broad match", quality constraint "Exact match" cannot be selected. ... [OK]
U: OK

Fig. 6.12: Dialogue utterances for customizing book locating service (part 4)

M: Do you need "Broad match"? [YES, NO, EXPLAIN]
U: NO
M: Quality constraint "Exact match" is related to "Search in book keywords". Do you need it? [YES, NO, EXPLAIN]
U: YES
M: Functionality "Advanced search" is abandoned due to your previous decisions. [OK]
U: OK

Fig. 6.13: Dialogue utterances for customizing book locating service (part 5)

M: Functionality "Get detailed info of a book" is selected due to your previous decisions. [OK]
U: OK
M: Functionality "Get publication info" is a sub-requirement of "Get detailed info of a book". Do you need it? [YES, NO, EXPLAIN]
U: YES
M: Functionality "Get contents" is a sub-requirement of "Get detailed info of a book". Do you need it? [YES, NO, EXPLAIN]
U: YES
M: Functionality "Get publication info" is selected due to your previous decisions. [OK]
U: OK
M: Functionality "Get contents" is selected due to your previous decisions. [OK]
U: OK

Fig. 6.14: Dialogue utterances for customizing book locating service (part 6)

M: Quality constraint "Broad match" is abandoned due to your previous decisions. [OK]
U: OK
M: Quality constraint "Exact match" is selected due to your previous decisions. [OK]
U: OK
M: All the requirements have been evaluated. Would you like to make any changes? [ADD, REMOVE, NO]
U: NO
M: The customization process is complete. Thank you for your cooperation.

Fig. 6.15: Dialogue utterances for customizing book locating service (part 7)

4. Output Generation

Finally, requirements *Search in book keywords*, *Search relevant books*, *Get a list of relevant books*, *Pick a book from a list*, *Get reference to a book*, *Get publication info*, *Get contents*, *Get detailed info of a book*, *Locate a book* and *Exact match* are selected.

Then the output will be generated in the way of merging the sub-tree rooted on *Get reference to a book*, which includes the sibling merging between *Get a list of relevant books* and *Pick a book from a list*, and merging *Get publication info*, *Get contents* into *Get detailed info of a book*. As a result, there will be three functions left: *Get reference to a book*, *Get detailed info of a book* and *Locate a book*. Because *Get detailed info of a book* relies on the output of *Get reference to a book*, they will be

merged to form a new function, which will immediately merged into *Locate a book*. Finally, quality *Exact match* will be attached directly to *Locate a book*. The output BookShoppingProfile.owl looks like the OWL-S document presented in Fig. 6.16. The instances of inputs, outputs and qualities are defined in documents BookShoppingProcess.owl and BookShoppingQuality.owl. They are imported by the profile document.

Now, with this OWL-S description, services are expected to be discovered by semantic capability matching. Then the composition and execution of the services will be performed based on the corresponding service composition information offered by the service providers.

```
<?xml version="1.0" encoding="UTF-8"?>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:profile="http://www.daml.org/services/owl-s/1.2/Profile.owl#"
         xmlns:owl="http://www.w3.org/2002/07/owl#">

<owl:Ontology rdf:about="">
         <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.2/Profile.owl"/>
         <owl:imports rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns"/>
         <owl:imports rdf:resource="ttp://www.w3.org/2002/07/owl"/>
         <owl:imports rdf:resource="http://www.semanticweb.org/ontologies/
                InteractiveRE/BookShoppingProcess.owl"/>
         <owl:imports rdf:resource="http://www.semanticweb.org/ontologies/
                InteractiveRE/BookShoppingQuality.owl"/>
</owl:Ontology>

<profile:Profile rdf:ID="Locate_a_book">
         <profile:textDescription>Locate a book</profile:textDescription>

         <profile:has_process ref:resource="http://www.semanticweb.org/ontologies/
                InteractiveRE/BookShoppingProcess.owl#LocateABookProcess"/>

         <profile:hasInput rdf:resource="http://www.semanticweb.org/ontologies/
                InteractiveRE/BookShoppingProcess.owl#PhrasesFromUserInput"/>
         <profile:hasInput rdf:resource="http://www.semanticweb.org/ontologies/
                InteractiveRE/BookShoppingProcess.owl#BookIndexInTheList"/>

         <profile:hasOutput rdf:resource="http://www.semanticweb.org/ontologies/
                InteractiveRE/BookShoppingProcess.owl#ListOfRelevantBooks"/>
         <profile:hasOutput rdf:resource="http://www.semanticweb.org/ontologies/
                InteractiveRE/BookShoppingProcess.owl#BookContents"/>
         <profile:hasOutput rdf:resource="http://www.semanticweb.org/ontologies/
                InteractiveRE/BookShoppingProcess.owl#BookPublicationInfo"/>

         <profile:serviceParameter rdf:resource="http://www.semanticweb.org/ontologies/
                InteractiveRE/BookShoppingQuality.owl#ExactMatch"/>

</profile:Profile>
</rdf:RDF>
```

Fig. 6.16: The output service description (BookShoppingProfile.owl)

CHAPTER VII

COMPARISON ANALYSIS

7.1 Introduction

In the proposed approach, implicit knowledge hidden in the ontology model is retrieved to direct the requirement elicitation process. Meanwhile, the approach adopts a strategy that always maintains the completeness and consistency of the elicitation results. When a requirement is selected, all other requirements on which it relies will be selected, and those that contradict it will be removed. When a requirement is dropped, the requirements that rely on it will also be dropped. On the other hand, without the implicit knowledge and guidance, people normally have to resolve the incompleteness and inconsistency step by step. In this thesis, simulation experiments were conducted to compare the performance of the proposed method with the undirected method.

7.2 Problem Instance Generation

To generate a problem instance is to construct a requirement decomposition forest. A subset of the requirements contained in the trees will be chosen as the expected requirements.

In this research, problem instances are generated randomly. Functions, qualities and softgoals are not differentiated. They are all treated simply as requirements. When constructing a problem instance, the number of requirements n and the number of *Rely*, *Contradict* relationships m to be generated are two input parameters for the problem instance.

The algorithm for generating a decomposition forest with n nodes is presented in Fig. 7.1. First, when creating the ith node, a random integer j ranging from [0, i-1] is generated. j decides the position of the forest to add the ith node. If it is zero, create the ith node as a root. Otherwise, the ith node is attached as a child of the previous generated

*j*th nodes. If the *j*th node is a leaf, when the *i*th node is attached to it, the (*i*+1)th node is also attached to it. This is because a requirement is not supposed to decompose into only one child. If the *j*th node is not a leaf, only the *i*th node is attached to it. If it is the turn to create the *n*th node, as there are no more nodes available, it cannot be attached to a leaf. It can be created as a root or attached to a non-leaf node as a child. Then the decomposition forest is generated, and it is assumed that the rank to the *i*th node is *i*. Thus parent always has a higher rank than its children.

```
1.  //Let n be the number of requirements to be generated
2.  i ← 1
3.  WHILE i <= n
4.      j ← a random number from [0, i)
5.      IF j == 0 THEN
6.          Create the ith requirement as a root
7.          i ← i+1
8.      ELSE IF the jth requirement is not a leaf THEN
9.          Create the ith requirement as a child of the jth requirement
10.         i ← i+1
11.     ELSE IF i != n
12.         Create the ith requirement as a child of the jth requirement
13.         Create the (i+1)th requirement also as a child of the jth requirement
14.         i ← i+2
15.     END IF
16. END WHILE
```

Fig. 7.1: Pseudo code for constructing decomposition forest

After that, the *m Rely* and *Contradict* relationships should be generated. Each time, two nodes are randomly picked from the forest, and *Rely* or *Contradict* is randomly picked as the candidate relationship between them. The validity of the forest will be checked with ontology reasoning after the candidate relationship is attached to the forest. If the candidate relationship is valid and is not a duplicated relationship, generate the next relationship. Otherwise, remove the candidate relationship from the forest, and generate a new one.

When *m* valid and non-duplicated relationships are generated, the problem instance is generated. A subset of the *n* requirements will be randomly chosen as the

requirements that users expect to select. This subset of requirements will be shuffled in a list. The requirement with a lower index in the list is assumed to have a higher expected priority, which means it is more demanded. It is often true that this subset of requirements are not consistent or complete with each other. Users are supposed to give preference to requirements with higher expected priorities, which means they will give up an expected requirement if it contradicts requirements with higher expected priorities. With this guideline, users will try to take fewer rounds of interaction to accomplish the tasks and select as many of the expected requirements as possible, while keeping the selected requirements complete and consistent with each other.

7.3 Experiment with the Proposed Method

The proposed approach is designed to evaluate the requirements one by one, from the highest rank to the lowest rank. In the experiments, users that are simulated were to evaluate the requirements. When evaluating a requirement, if it is not expected, users will go through the expected requirements from the highest expected priority to the lowest. If the requirement being evaluated is relied by an expected requirement while it doesn't contradict expected requirements with higher expected priority, it will be selected. Otherwise, it will be dropped. If the requirement is expected, go through the expected requirements from the highest expected priority to itself. If it doesn't contradict the requirements with higher expected priority, select it, otherwise drop it. As is explained in Chapter IV, selecting or dropping a requirement with the proposed approach will cause all the relevant requirements to be handled accordingly.

After all the requirements have been selected or dropped, the simulated users will double check their decisions. Every selected unexpected requirement will be checked if they are relied by any selected expected requirements. It is possible that the expected requirement which relies on an unexpected requirement will be dropped after selection of the corresponding unexpected requirement. If no more selected expected requirement

relies on an unexpected requirement, the unexpected requirement will be removed. When all unnecessary unexpected requirements have been removed, the unselected expected requirements will be checked from the highest expected priority to the lowest expected priority. If an expected requirement doesn't contradict any selected requirements, it will be selected. After all the expected requirements have been rechecked, the users finish the requirement elicitation.

To evaluate the performance of the proposed approach, one round of interaction will be charged for each requirement evaluation, and each time users change their mind on a requirement, it will take one round of interaction to have it handled.

7.4 Experiment with the Undirected Method

For the undirected approach, the simulated users don't take care of the completeness or consistency issues initially. Instead, they will first tell the machine the requirements they want. Suppose there are k expected requirements. Then it will take k rounds of interaction to have the k requirements ordered.

After that, users will try to maintain the completeness and consistency. However, they only know the explicitly defined relationships and don't have the implicit knowledge. They will go through the selected requirements from the highest expected priority to the lowest. If an expected requirement relies on some other requirements, users will order machine to select these unselected required requirements, and these requirements will be treated as of the same expected priority as the corresponding expected requirement. If an expected requirement contradicts some selected requirements with lower expected priority, these requirements will be dropped. Whenever any changes are made, users will recheck all the currently selected requirements from the highest expected priority to the lowest expected priority. Every time users want to make some changes for a requirement, one round of interaction will be charged. When no more incompleteness or inconsistency exists among the selected requirements, the elicitation

process is complete. Otherwise, if the amount of interaction excess 10 times of the total number of requirements in the problem instance, users will give up. As a result, the iteration of experiment is unsuccessful.

7.5 Results and Analysis

Experiments were programmed in Java 6.0 with Eclipse 3.6 and performed on x86, Windows platform.

Experiments were separated into three groups. The first group fixes the number of relationships m to be 20, and varies the number of requirements n from 10 to 100, with an increment of 10. The second group fixes n to be 50, while increases m from 10 to 100, and the interval of each increase is 10. The third group changes n together with m, from (10, 10) to (100, 100). Each time, both of the two inputs were raised by 10.

Besides, for each input pair, 25 iterations of experiments were performed. In each iteration, a new problem instance was generated, and both methods were applied to solve it. As the proposed method promises the completeness and consistency, the number of iterations in which the undirected approach can successfully produce complete and consistent requirements was recorded. Those iterations of experiments are called successful iterations. Among the successful iterations, the numbers of interactions charged with both methods are compared. Moreover, the numbers of expected requirement selected by both methods are compared. Among those iterations where equal amount of expected requirements were selected, the amount of unexpected requirements selected by both approaches were compared.

Table 7.1, Table 7.2 and Table 7.3 respectively present results of the three groups of experiments.

A	B	C	D	E	F	G
10 / 20	0	-	-	-	-	-
20 / 20	11	27.3%	5	0	0	0
30 / 20	8	28.8%	2	0	1	0
40 / 20	15	32.9%	5	0	1	0
50 / 20	16	51.1%	4	1	1	0
60 / 20	21	58.9%	3	0	0	0
70 / 20	18	73.5%	3	0	0	0
80 / 20	18	66.4%	1	0	0	0
90 / 20	22	77.9%	6	0	0	0
100 / 20	19	73.5%	1	0	0	0

Table 7.1: Results for the first group of experiments (*n*:*m*=10:20-100:20)

A	B	C	D	E	F	G
50 / 10	22	80.9%	2	0	0	0
50 / 20	16	50.5%	3	1	2	0
50 / 30	5	30.2%	3	0	0	0
50 / 40	6	6.8%	2	3	0	0
50 / 50	2	6.4%	2	0	0	0
50 / 60	2	0.0%	1	1	0	0
50 / 70	0	-	-	-	-	-
50 / 80	0	-	-	-	-	-
50 / 90	0	-	-	-	-	-
50 / 100	0	-	-	-	-	-

Table 7.2: Results for the second group of experiments (*n*:*m*=50:10-50:100)

A	B	C	D	E	F	G
10 / 10	12	58.3%	5	1	0	1
20 / 20	8	0.0%	4	1	1	0
30 / 30	8	5.9%	2	1	0	1
40 / 40	5	11.5%	1	2	1	0
50 / 50	0	-	-	-	-	-
60 / 60	3	6.8%	0	1	2	0
70 / 70	0	-	-	-	-	-
80 / 80	0	-	-	-	-	-
90 / 90	0	-	-	-	-	-
100 / 100	0	-	-	-	-	-

Table 7.3: Results for the third group of experiments (*n*:*m*=10:10-100:100)

Followings are explanations to the columns.

- Column A: number of requirements / number of relationships
- Column B: number of successful iterations performed by the undirected method
- Column C: average percentage of more interactions the proposed method was charges compared to the undirected method (the percentage is calculated in this way: （*number of interactions the proposed method was charged－number of interactions the undirected method was charged*）／*number of interactions the undirected method was charged*×100%)
- Column D: number of successful iterations in which the proposed method selected more expected requirements
- Column E: number of successful iterations in which the undirected method selected more expected requirements
- Column F: number of successful iterations in which the proposed method selected equal number of expected requirements as the undirected method and fewer unexpected requirements
- Column G: number of successful iterations in which the undirected method selected equal number of expected requirements as the proposed method and fewer unexpected requirements

From all the results, it can be observed that the undirected approach could not promise to produce complete and consistent results, while the proposed approach is designed to overcome this problem. In the best cases, within 22 out of 25 iterations, valid results could be generated by the undirected method. Moreover, in a considerable amount of successful iterations, the proposed method had more expected requirements selected. On the other hand, most of the requirements were evaluated at least once by the proposed approach, while the undirected approach only concerns the expected requirements. Thus the proposed approach often requires more rounds of interaction.

In the first group of requirements, when *n* increased and *m* was kept unchanged,

the undirected approach performed better and better. It took fewer and fewer rounds of interactions to complete the tasks while its success rate grew very fast. Meanwhile, the proposed method was never defeated. There were always some successful iterations within which the proposed method obtained more expected requirements.

For the second group of requirements, when *m* grew and *n* was fixed, performance of the undirected method decreased dramatically. When there were more than 40 relationships, it could hardly generate any valid result. What is worth mentioning is that, while the proposed method could easily defeat the undirected method, in the rare cases of successful iterations, undirected method found more expected requirements. Mainly, this result was due to the simulation strategy rather than the proposed elicitation method. The simulated user always tries to obtain the expected requirement with the highest expected priority. Sometimes, the most demanded requirement is obtained but other expected requirements are neglected.

In the third group, even if *n* and *m* were raised with the same pace, the performance of the undirected approach decreased very fast as the complexity of the problem increased. Meanwhile, the distance between the amount of interactions the proposed method requires and the amount of interactions the undirected approach needs was shortened when the problem was complicated.

In conclusion, these experiments show that requirement elicitation is not an easy task. Without implicit knowledge and proper guidance, it is almost impossible to get a valid and expected result. However, the method proposed in this thesis can successfully help to solve this problem. Though, it always needs certain amount of interactions, this is the price necessary for accomplishing the task. Besides, when the problem gets tougher, the price is not as remarkable as before.

CHAPTER VIII

CONCLUSION AND FUTURE WORK

8.1 Conclusion

Aiming at realizing automated SPL with service-oriented methods, an approach to interactive requirement elicitation is proposed in this thesis. It adopts ontology to represent the requirement engineering related knowledge, which directs a slot-filling dialogue system to communicate with clients. With this method, users are capable to customize the application requirements that satisfy their demands by interacting with machines, while the completeness and consistency of the customization is ensured. The ordered requirements will further be converted into OWL-S based service descriptions for system implementation. A case study is presented in this thesis to prove the feasibility of the proposed method, while simulation experiments were conducted to verify its efficiency and reliability.

On the other hand, though this thesis made an effort to achieve automated requirement elicitation, the proposed requirement model is still preliminary and light-weighted. Since the model is static, it cannot be applied in dynamic environment. Moreover, in order to avoid additional complexity, not all requirement engineering related ontological relationships are directly described. Finally, the reported requirement elicitation approach only supports customizing requirements based on the knowledge that machine owns. Users cannot order anything unknown to the machine, which is not always the case in practice.

8.2 Future Work

For the future works, first, in order to implement automated SPL, approaches related to automatic application implementation, such as automatic service discovery, composition and delivery, will be further explored. Meanwhile, it is necessary to have the

ontology model optimized (e.g. improve its expressiveness, and extend it with domain properties). In addition, analysis about methods other than OWL-S for utilizing the requirement elicitation results and describing abstract information (e.g. softgoals) is also worth performing. Last but not the least, topics about enriching the experience of human-computer interaction in requirement engineering are very interesting. Related studies (e.g. visualize the interactive requirement elicitation) will be conducted in future.

REFERENCES

[1] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*. Berlin: Springer, 2005.

[2] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ: Prentice Hall PRT, 2005.

[3] A. Helferich, et al., "Software Product Lines, Service-Oriented Architecture and Frameworks: Worlds Apart or Ideal Partners?" in *Proceedings of the 2nd International Conference on Trends in Enterprise Application Architecture*, 2006, pp. 187-201.

[4] S. A. McIlraith, T. C. Son, and H. Zeng, "Semantic Web Services," *IEEE Intelligent Systems*, 16(2), 46-53, 2001.

[5] H. Gomaa and M. E. Shin, "Automated Software Product Line Engineering and Product Derivation," in *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, 2007, pp. 285.

[6] R. Rabiser, P. Grünbacher, and D. Dhungana, "Requirements for Product Derivation Support: Results from a Systematic Literature Review and an Expert Survey," *Information and Software Technology*, 52(3), 324-346, 2010.

[7] T. H. Bui, "Multimodal Dialogue Management: State of the Art," Centre for Telematics and Information Technology, University of Twente, Enschede, Netherlands, Tech. Rep. TR-CTIT-06-01, 2006.

[8] D. Martin, et al., "Bringing Semantics to Web Services with OWL-S," *World Wide Web*, 10(3), 243-277, 2007.

[9] B. Srivastava and J. Köhler, "Web Service Composition: Current Solutions and Open Problems," in *Proceedings of the 13th International Conference on Automated Planning and Scheduling*, 2003. Available: CiteSeer, http://citeseerx.ist.psu.edu.

[10] S. Dustdar and W. Schreiner, "A Survey on Web Services Composition," *International Journal of Web and Grid Services*, 1(1), 1–30, 2005.

[11] C. Y. Knaus, "Feature - Interaction design for software engineering: Boost into programming future," *Interactions*, 15(4), 71-74, 2008.

[12] A. Flycht-Eriksson and A. Jönsson, "Dialogue and Domain Knowledge Management in Dialogue Systems," in *Proceedings of the 1st SIGdial Workshop on Discourse and Dialogue*, 2000, pp. 121-130.

[13] A. Flycht-Eriksson, "A Survey of Knowledge Sources in Dialogue Systems," *Electronic Transactions on Artificial Intelligence*, 3(D), 5-32, 1999.

[14] M. Araki, et al., "A Dialogue Library for Task-Oriented Spoken Dialogue Systems," in *Proceedings of the IJCAI Workshop on Knowledge and Reasoning in Practical Dialogue Systems*, 1999, pp. 1-7.

[15] T. Riechert, et al., "Towards Semantic based Requirements Engineering," in *Proceedings of the 7th International Conference on Knowledge Management*, 2007, pp. 144-151.

[16] C. I. Lin and C. Ho, "A Generic Ontology-Based Approach for Requirement Analysis and its Application in Network Management Software," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 13(1), 37-61, 1999.

[17] M. Kossmann, "Ontology-Driven Requirements Engineering with Reference to the Aerospace Industry," in *Proceedings of the 2nd International Conference on the Applications of Digital Information and Web Technologies*, 2009, pp. 95-103.

[18] M. Kossmann, "Ontology-Driven Requirements Engineering: Building the OntoREM Meta Model," in *Proceedings of the 3rd International Conference on Information and Communication Technologies: From Theory to Applications*, 2008, pp. 1-6.

[19] H. J. Happel and S. Seedorf, "Applications of Ontologies in Software Engineering," in *Proceedings of the 2nd Workshop on Semantic Web Enabled Software Engineering*, 2006. Available: CiteSeer, http://citeseerx.ist.psu.edu.

[20] T. R. Gruber, "A Translation Approach to Portable Ontology Specifications," *Knowledge Acquisition*, 5(2), 199-220, 1993.

[21] N. F. Noy and D. L. McGuinness, "Ontology Development 101: A Guide to Creating Your First Ontology," Stanford Knowledge Systems Laboratory, Tech. Rep. KSL-01-05, 2001.

[22] F. Arvidsson and A. Flycht-Eriksson, "Ontologies I," 2008. [Online]. Available: http://www.ida.liu.se/~janma/SemWeb/Slides/ontologies1.pdf.

[23] R. Mizoguchi, "Tutorial on Ontological Engineering," 2004. [Online]. Available: http://www.ei.sanken.osaka-u.ac.jp/japanese/tutorial-j.html.

[24] D. L. McGuinness and F. van Harmelen, "OWL Web Ontology Language Overview," 2004. [Online]. Available: http://www.w3.org/TR/owl-features.

[25] G. Dobson and P. Sawyer, "Revisiting Ontology-Based Requirements Engineering in the Age of the Semantic Web," in *Proceedings of the International Seminar on Dependable Requirements Engineering of Computerised Systems at NPPs*, 2006. Available: CiteSeer, http://citeseerx.ist.psu.edu.

[26] I. Horrocks, et al., "SWRL: A Semantic Web Rule Language Combining OWL and RuleML," 2004. [Online]. Available: http://www.w3.org/Submission/SWRL.

[27] H. Kaiya and M. Saeki, "Ontology Based Requirements Analysis: Lightweight Semantic Processing Approach," in *Proceedings of the 5th International Conference on Quality Software*, 2005, pp. 223-230.

[28] I. Jureta, J. Mylopoulos, and S. Faulkner, "Revisiting the Core Ontology and Problem in Requirements Engineering," in *Proceedings of the 16th IEEE International Requirements Engineering Conference*, 2008, pp. 71-80.

[29] D. V. Dzung and A. Ohnishi, "Ontology-Based Reasoning in Requirements Elicitation," in *Proceedings of the 7th IEEE International Conference on Software Engineering and Formal Methods*, 2009, pp. 263-272.

[30] P. Kroha, R. Janetzko, and J. E. Labra, "Ontologies in Checking for Inconsistency of Requirements Specification," in *Proceedings of the 3rd International Conference on Advances in Semantic Processing*, 2009, pp. 32-37.

[31] I. Omoronyia, et al., "A Domain Ontology Building Process for Guiding Requirements Elicitation," in *Proceedings of the 16th International Working Conference on Requirements Engineering: Foundation for Software Quality*, 2010, pp. 188-202.

[32] T. H. Al Balushi, et al., "ElicitO: A Quality Ontology-Guided NFR Elicitation Tool," in *Proceedings of the 13th International Working Conference on Requirements Engineering: Foundation for Software Quality*, 2007, pp. 262-276.

[33] R. Roy, et al., "Design Requirements Management Using an Ontological Framework," *CIRP Annals: Manufacturing Technology*, 54(1), 109-112, 2005.

[34] J. Lin, M. S. Fox and T. Bilgic, "A Requirement Ontology for Engineering Design," *Concurrent Engineering: Research and Applications*, 4(3), 279-291, 1996.

[35] M. Shibaoka, H. Kaiya and M. Saeki, "GOORE: Goal-Oriented and Ontology Driven Requirements Elicitation Method," in *Proceedings of the 2007 Conference on Advances in Conceptual Modeling: Foundations and Applications*, 2007, pp. 225-234.

[36] K. Jarosla, "Passing from Requirements Specification to Class Model Using Application Domain Ontology," in *Proceedings of the 2nd International Conference on Information Technology*, 2010, pp. 129-132.

[37] S. R. Palmer and J. M. Felsing, A Practical Guide to Feature-Driven Development. Upper Saddle River, NJ: Prentice Hall, 2002.

[38] A. Brown, "An introduction to Model Driven Architecture, " 2004. [Online]. Available: http://www.ibm.com.

[39] D. Roman, et al., "Web Service Modeling Ontology," *Applied Ontology*, 1(1), 77-106, 2005.

[40] D. Martin, et al., "OWL-S: Semantic Markup for Web Services," 2004. [Online]. Available: http://www.w3.org/Submission/OWL-S.

[41] C. Harding, "SOA Ontology Draft 2.0," 2008. [Online]. Available: http://www.opengroup.org/projects/soa-ontology/doc.tpl?gdid=16940.

[42] D. Martin, et al., "Bringing Semantics to Web Services: The OWL-S Approach," in *Proceedings of the 1st International Workshop on Semantic Web Services and Web Process Composition*, 2004, pp. 26-42.

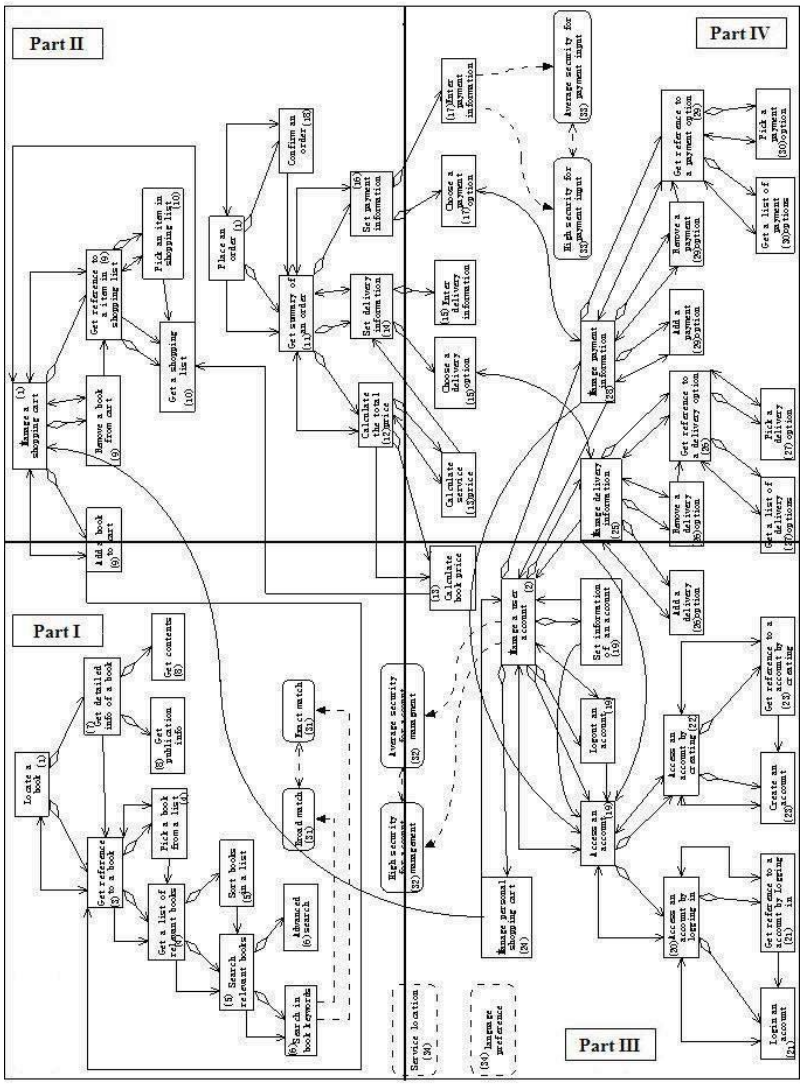
[43] S. Jean, et al., "An Extension of OWL-S with Quality Standards," in *Proceedings of the 4th IEEE International Conference on Research Challenges in Information Science*, 2010, pp. 483-494.

APPENDICES

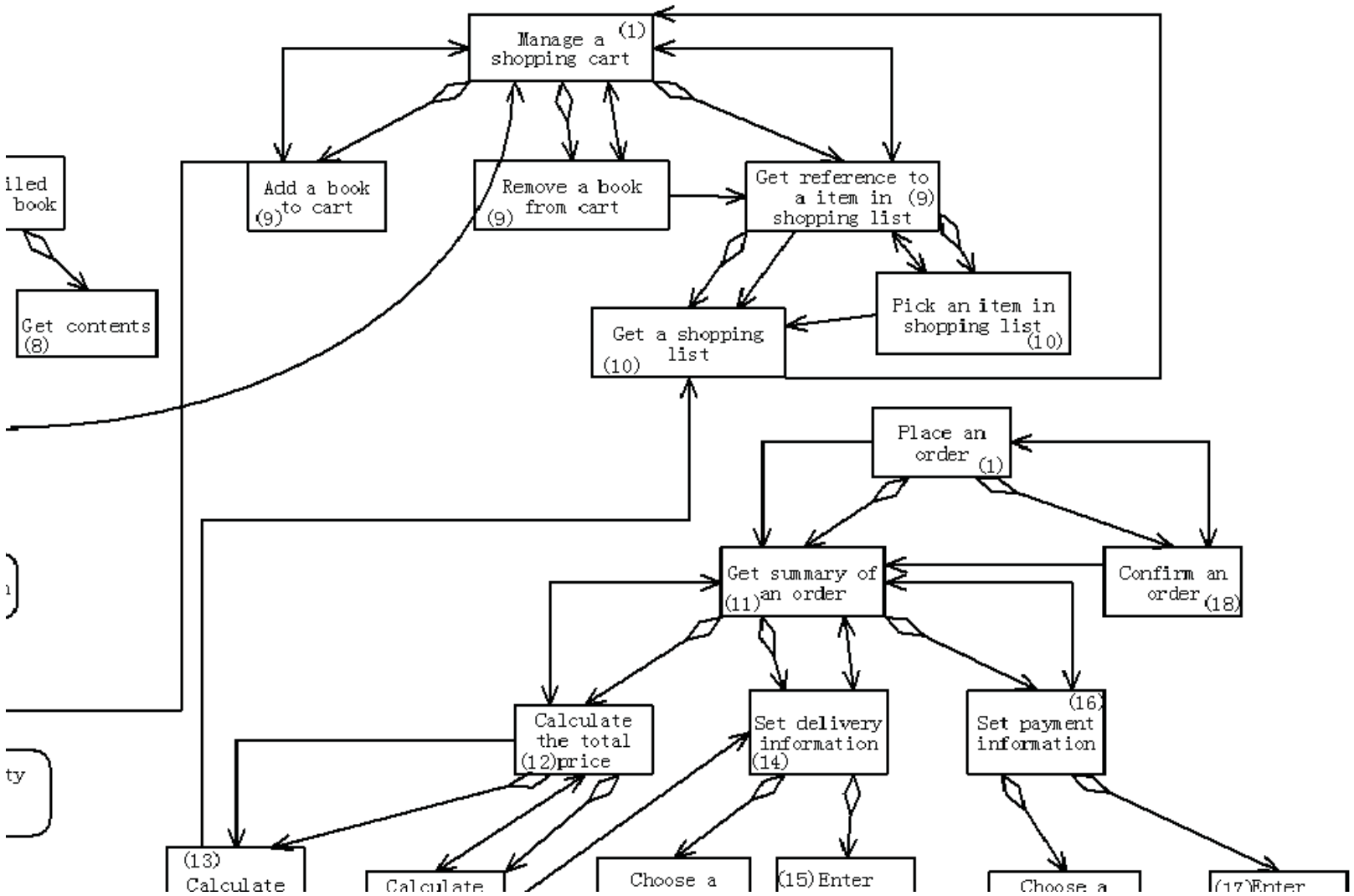
APPENDIX A

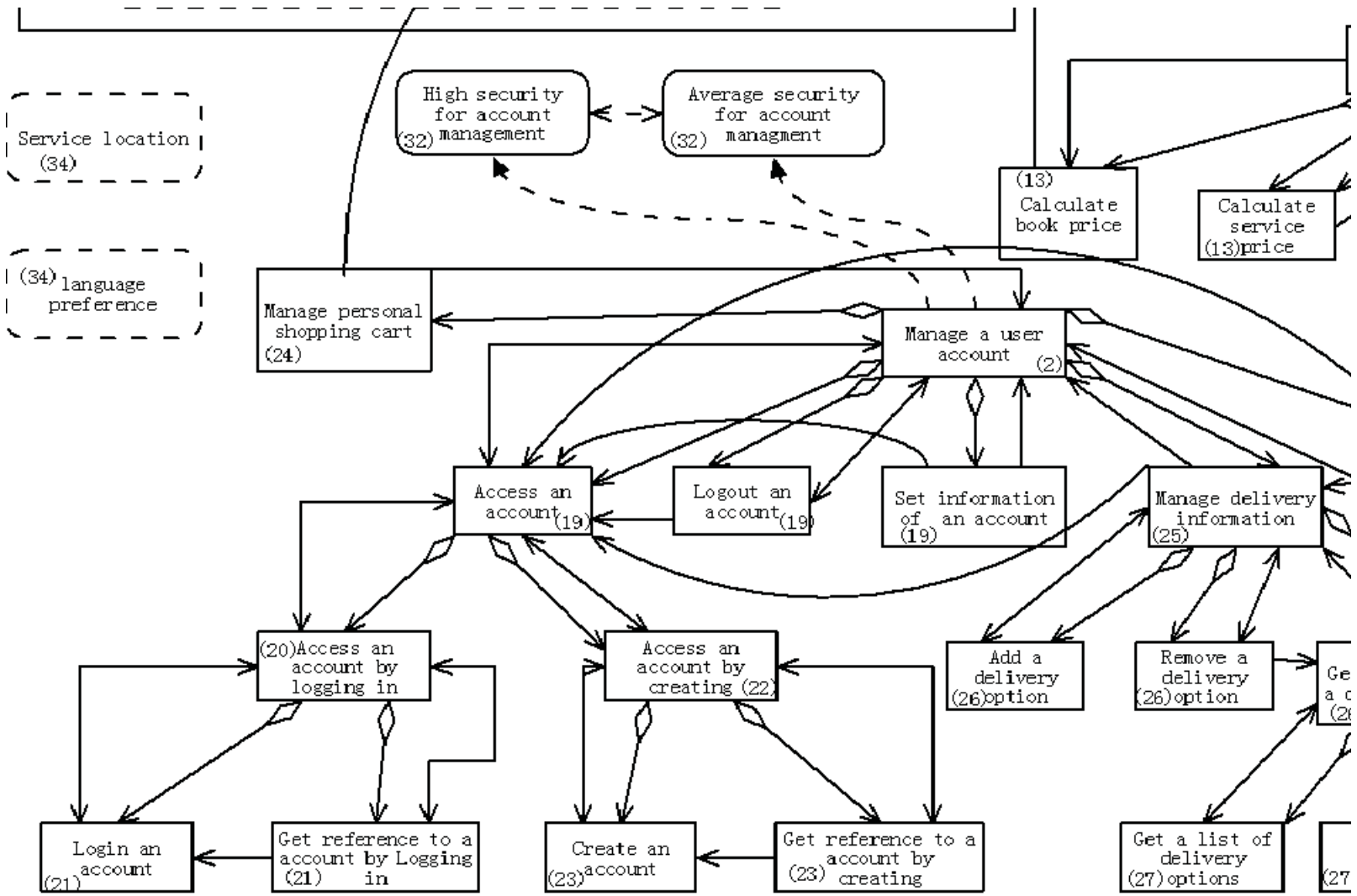
The Complete Requirement Model for the Case Study

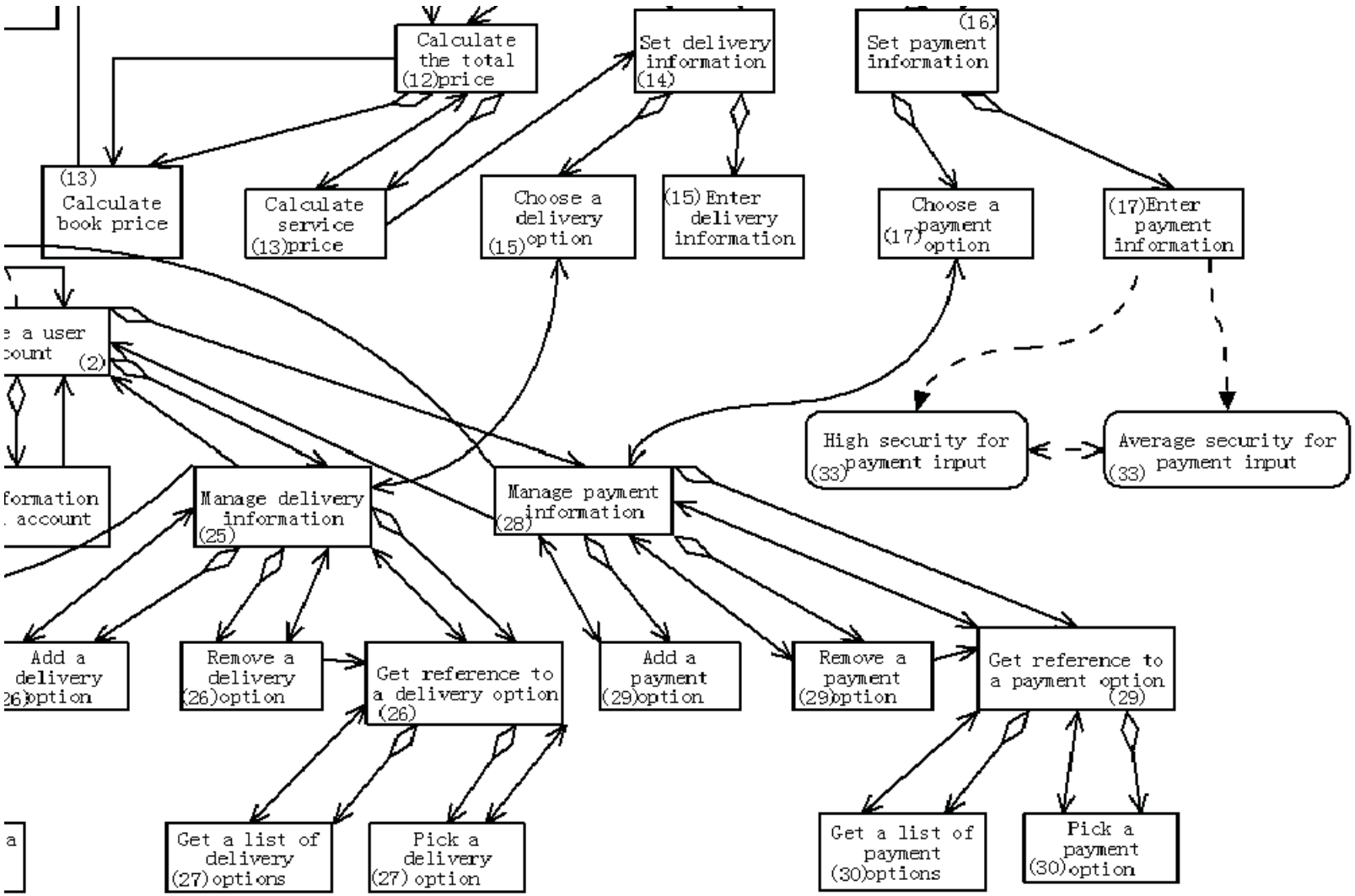
The following figure illustrates the complete ontology-based requirement model instantiated with the case study of online book shopping service. The figure is divided into four parts. The magnified figures for Part I, Part II, Part III and Part IV are presented on page 74, 75, 76 and 77 respectively.



Locate a book (1)
Get reference to a book (3)
Get detailed info of a book (7)
Get contents (8)
Get publication info (8)
Pick a book from a list (4)
Get a list of relevant books (4)
Sort books in a list (5)
Search relevant books (5)
Advanced search (6)
Search in book keywords (6)
Add a book to cart (9)
Exact match (31)
Broad match (31)
Average security for account management (32)
High security for account management (32)
Service location (34)
(13) Calculate


Manage a shopping cart (1)
Add a book to cart (9)
Remove a book from cart (9)
Get reference to a item in shopping list (9)
Get contents (8)
Get a shopping list (10)
Pick an item in shopping list (10)
Place an order (1)
Get summary of an order (11)
Confirm an order (18)
Calculate the total price (12)
Set delivery information (14)
Set payment information (16)
(13) Calculate
Choose a
(15) Enter
Choose a
(17) Enter


Service location (34)
(34) language preference
High security for account management (32)
Average security for account managment (32)
(13) Calculate book price
Calculate service (13)price
Manage personal shopping cart (24)
Manage a user account (2)
Access an account (19)
Logout an account (19)
Set information of an account (19)
Manage delivery information (25)
(20)Access an account by logging in
Access an account by creating (22)
Add a delivery (26)option
Remove a delivery (26)option
Login an account (21)
Get reference to a account by Logging in (21)
Create an account (23)
Get reference to a account by creating (23)
Get a list of delivery (27)options


Calculate the total (12)price
Set delivery information (14)
(16) Set payment information
(13) Calculate book price
Calculate service (13)price
Choose a delivery (15)option
(15) Enter delivery information
Choose a payment (17)option
(17)Enter payment information
e a user count (2)
formation account
Manage delivery information (25)
Manage payment information (28)
High security for (33)payment input
Average security for (33) payment input
Add a delivery (26)option
Remove a delivery (26)option
Get reference to a delivery option (26)
Add a payment (29)option
Remove a payment (29)option
Get reference to a payment option (29)
a
Get a list of delivery (27)options
Pick a delivery (27) option
Get a list of payment (30)options
Pick a payment (30)option

VITA AUCTORIS

NAME	Xieshen Zhang
PLACE OF BIRTH	Wujiang, Jiangsu, China
YEAR OF BIRTH	1984
EDUCATION	Shanghai Jiao Tong University
	2003-2007 B.Eng.