Doctoral Thesis

Studies on Ontology-based Verification of Software Requirements

September, 2014

Doctoral Program in Integrated Science and Engineering

Graduate School of Science and Engineering

Ritsumeikan University

Dang Viet Dung

Doctoral Thesis reviewed
by Ritsumeikan University

Studies on Ontology-based Verification of Software Requirements
(オントロジーを用いたソフトウェア要求の検証手法に関する研究)

September, 2014
2014 年 9 月

Doctoral Program in Integrated Science and Engineering
Graduate School of Science and Engineering
Ritsumeikan University
立命館大学大学院理工学研究科
総合理工学専攻博士課程後期課程

Dang Viet Dung
ダン ヴィエット ズン

Supervisor: Professor Atsushi Ohnishi
指導教員: 大西 淳教授

Abstract

Insufficient or incorrect software requirements will cause big loss in later stages of software development. To minimize loss, requirements quality assurance aims to gain requirements document of high quality. However, it is difficult to attain that purpose without knowledge of application domain. Current techniques of requirements quality assurance do not utilize domain knowledge for improving quality of requirements. To overcome this drawback, we propose an ontology-based verification method of software requirements. In the method, requirements ontology is a knowledge structure that contains functional requirements, their attributes and relations among them. To find errors in requirements document, software requirements are compared with information in the requirements ontology, then rules are used for reasoning of indispensable or inaccurate requirements. On the basis of the results, analysts can ask questions to customers and correctly and efficiently revise requirements. To support this method, an ontology-based verification tool of software requirements has been developed. Through experiments, we can show the effectiveness of requirements ontology in verification of software requirements.

On the other hand, the effect of ontology-based requirements verification depends on quality of requirements ontology. With the increasing size of ontology, it is difficult to verify the correctness of information stored in ontology. We then propose another method of using rules for verification of the correctness of requirements ontology. We provide a rule description language to specify properties that requirements ontology should satisfy. Then, by checking whether requirements ontology is consistent with specified rules, we verify the correctness of the ontology. We have developed a verification tool to support the method and evaluated the tool through experiments.

This page intentionally left blank.

Acknowledgments

I would like to express my deepest thank to my research supervisor, Prof. Atsushi Ohnishi. He gave me valuable advices throughout the three years of doctoral study. From his guidance, I gradually learnt how to do research, how to write conference papers and journal papers. He encouraged me and always helped me when I had problems. Without his strictness on my research and paper writing, my study could not be completed. Thank you my professor! To my vise supervisors, Prof. Katsuhisa Maruyama and Prof. Hiromitsu Shimakawa, I am thankful for their time and effort reviewing my research and helpful comments which contributed to the success of this thesis. To Assoc. Prof. Hiroya Itoga, I would like to thank a lot for his numerous suggestions during my seminars. My thesis would not have been finished without the help of participants to do experiments. They were very enthusiastic and suggested me to establish the experiments more objectively. Thanks to all participants of the experiments: Eiji Shiota, Kennya Kojima, Kazuki Kakimoto, Takao Umekage, Takashi Iseki, Shinta Takeda, and many other students in Software Engineering laboratory 2013. I also would like to appreciate members of Software Engineering laboratory. They gave me encouragement and friendship. I owe people of ontology research group, especially Ryuichi Sakamoto and Bui Quang Huy who have developed ontology builder and ontology verifier, respectively, so that I can reuse their works. Finally, I would like to acknowledge financial support for my study from Monbukagakusho scholarship via recommendation by Ritsumeikan University.

This page intentionally left blank.

Contents

This page intentionally left blank.

List of Figures

List of Tables

Chapter 1

Introduction

The topic of this thesis is in software engineering field and focuses on improvement of quality of a software artifact: software requirements specification. Specifically the thesis presents an usage of requirements ontology to help enhancement of quality of software requirements specification. There are two studies: the first study is how to use requirements ontology to help that purpose, the second study is how to guarantee the quality of requirements ontology because that quality affects the performance of the first study. To provide an overview of the research topic, this chapter will explain problems to be addressed in the thesis and briefly touch upon solutions to them. The organization of the thesis will be also provided.

1.1 Problems Statement

There are two major reasons that make development of software of high quality difficult. The first reason is the nature of software: software is often single built product instead of mass production as in other industries [38]. Therefore, we often say software development, but not software production. The second reason is the complexity of nowadays software systems. For instance, operating systems such as Windows or Linux (being a software themselves) might consist of tens of thousands of components. Some components can be reused, but a lot of them are newly developed or customized. The difficulties in guarantee quality of all components and cooperating them smoothly cause troublesome for development of high quality software.

Since software is often built as a single instance, each time a new software is about to be developed, descriptions of the software should be collected and specified. These descriptions are called software requirements, which are statements of what the software should

do. The activities to gather and compose software requirements are named requirements engineering. The final output of requirements engineering is a requirements document which includes a list of detailly written software requirements.

Software requirements are important because if requirements were wrong, the software would be useless. Wrong requirements or lack of requirements will cause a lot of revisions or changes in latter phases of software development. Changes in requirements will take as a lot of efforts and budgets as late they are, so an important aim in requirements definition phase is getting as sufficient and correct requirements as possible. Studies have shown the cost to fix a requirements error could increase 100 times if fixing it in later phases of software development [3,11,25]. Another study found that it took average 30 minutes to fix a requirements error, but it cost five to 17 hours to correct a defect identified during system testing [22].

Because software requirements are important, it needs to guarantee quality of requirements document before moving on later phases of software development. That activity can be called requirements quality assurance. Main techniques for requirements quality assurance are: inspections and reviews, queries on a requirements database, animation-based validation, and formal verification [53]. Inspections and reviews are widely used, in which individual inspectors find errors in requirements document and then they meet to discuss. Queries technique is suitable for requirements in diagrammatic notations. Animation-based technique simulates working of software to find defects. The simulation-based validation transforms requirements to an executable format, so part of the requirements needs to be specified formally (formally means in a machine processable format). Formal methods are effective in verification of requirements document since they can be automated by tools, but these techniques demand that requirements must be specified formally.

However, there are still problems to the existing techniques for requirements quality assurance. Though inspections and reviews are common, if inspectors do not have domain knowledge, they might not detect errors in requirements documents. Lack of domain knowledge will lessen quality of requirements document and indirectly weaken quality of software product. With the complexity, diversity, and uncertainty (evolve very fast) of modern information systems, it is impossible for requirements engineers to have knowledge of every application domain. The usage of formal verification is still limited in certain systems which need high reliability [14, 55]. It is difficult for stakeholders to learn formal languages in order to understand specification documents (Stakeholders are people that will be affected by a system: customers, engineers, project managers, etc.). The translation from natural language to formal language required a lot of work-load and hardly implemented automatically [30]. Although there are many ways to write software requirements,

using natural languages is still the best for stakeholders. Even diagrammatic presentations (used in queries technique) still require stakeholders to learn some notations, and not all customers of software systems are willing to learn them.

Fig. 1.1 summarises the above discussions. There exists the need of new requirements quality assurance technique which uses domain knowledge and should support checking of natural languages requirements.

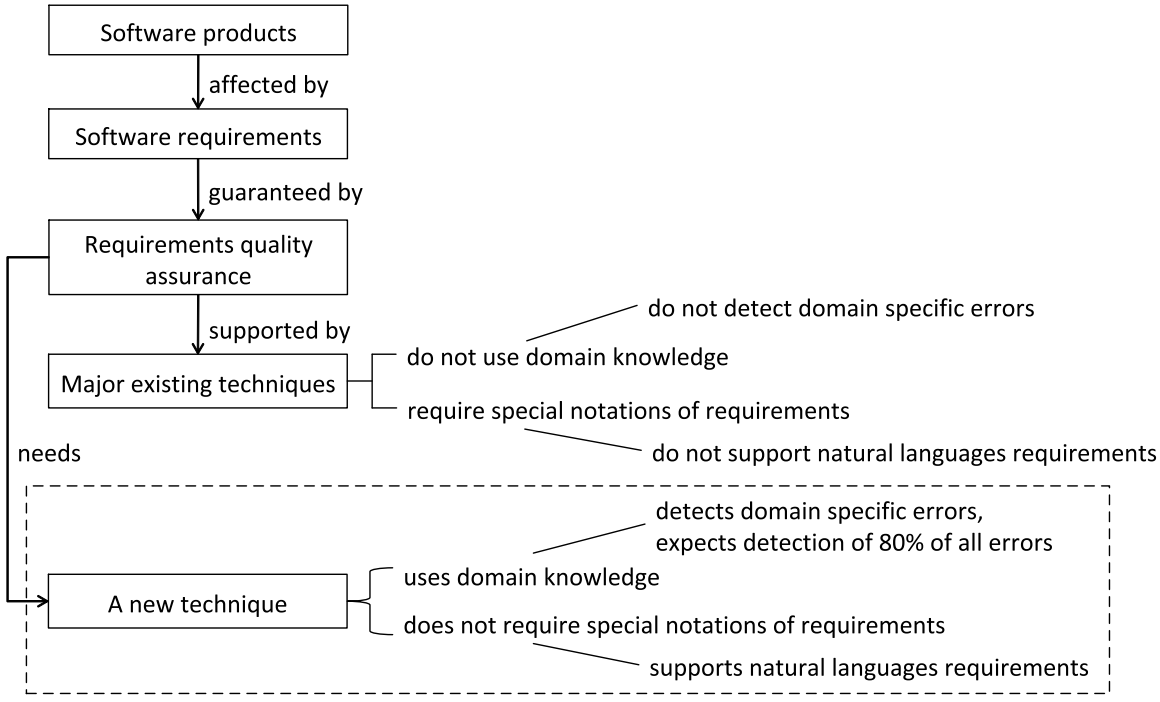


Figure 1.1 Summary of problems statement.

Therefore, a first question to be solved in our research is: "Is it possible to support requirements quality assurance with a technique that utilizes domain knowledge and does not require writing software requirements in a special notation other than natural languages?" In addition, if using that technique, the effectiveness of the method will depend on the correctness of the domain knowledge. Hence, a second research question is: "How to guarantee the correctness of the domain knowledge using in the answer of the first research question?" We expect that by using domain knowledge, we could detect more than 80% of errors in software requirements (a quantitative target).

1.2 Solution Approach

To answer the first research question above, we proposed an ontology-based verification method of software requirements. In the method, domain knowledge about existing software requirements is collected and stored in a repository named requirements ontology. Then requirements ontology is used for verification of new software requirements. To answer the second research question, we guarantee the correctness of requirements ontology with rules which specify accurate characteristics of an ontology. Fig. 1.2 introduces these studies in this thesis.

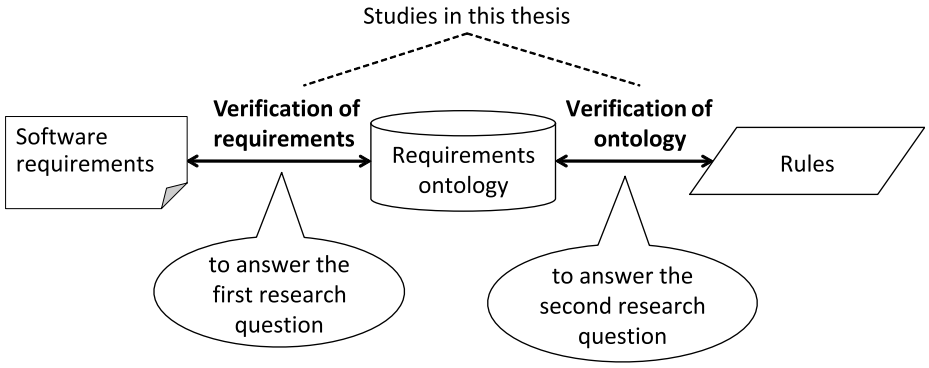


Figure 1.2 Topics to be studied in this thesis.

In the first study, verification of requirements will detect errors and improve quality of software requirements. Similarly, in the second study, verification of ontology will detect errors and improve quality of requirements ontology. The two studies are not separate but co-operative with each other in a common aim of improvement of quality of software requirements specification. In ontology-based requirements engineering, improving quality of ontology indirectly helps improving quality of software requirements. Fig. 1.3 presents an overview of the two studies: verification of requirements and verification of ontology.

Requirements ontology model has been studied at Software Engineering lab at Ritsumeikan University [35, 43, 51, 68]. Requirements ontology contains a hierarchy of functional requirements in a problem domain. That hierarchy can be retrieved from existing software documents in the same domain or from knowledge of domain experts. To check software requirements which are written in natural languages, we compare requirements sentences with functional nodes in requirements ontology to find errors. Fig 1.4 illustrates these ideas. This approach meets the first research question in the above section which requires: usage of domain knowledge to check software requirements, and usage of natural languages to specify software requirements.

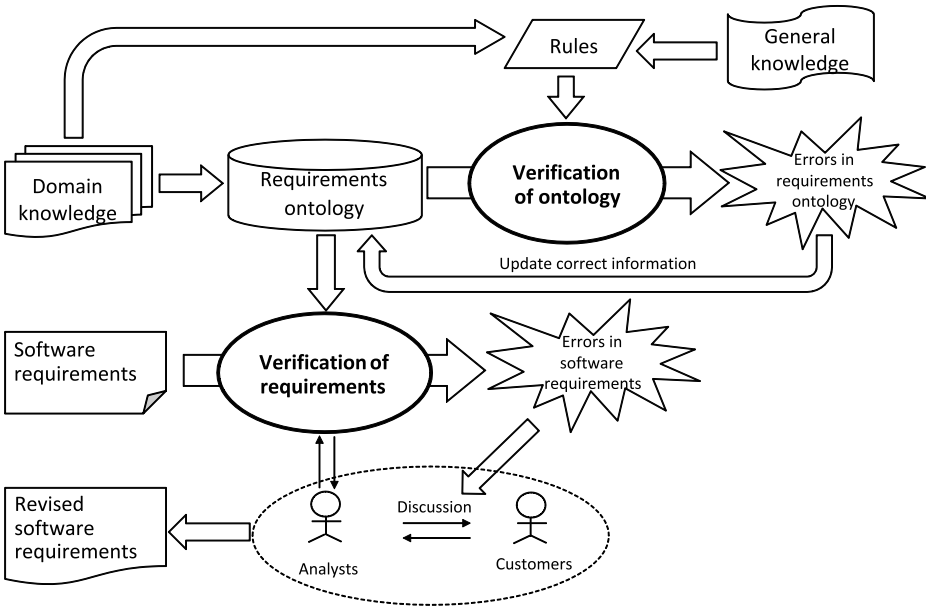


Figure 1.3 An overview of verification of software requirements and verification of requirements ontology.

To provide a solution to the second research question, we provide a rule-based verification method of the correctness of requirements ontology. We separate rule definition and rule interpretation, so new verification rules can be defined specifically for new requirements ontology. Fig. 1.5 presents this approach as a solution to the second research question.

Several issues to be studied in the research are:

1. How to use requirements ontology to detect errors in software requirements?
2. How to specify rules and how to use rules to detect errors in requirements ontology?
3. Which tools should be developed to support the methods?
4. How to evaluate the efficiency of the methods?

For issue 1, our ontology-based verification method works as follows. Firstly, each requirement sentence is mapped to a node in requirements ontology. Secondly, by using reasoning rules and knowledge stored in requirements ontology, errors in software requirements can be detected. Thirdly, errors are used to revise and improve requirements specification.

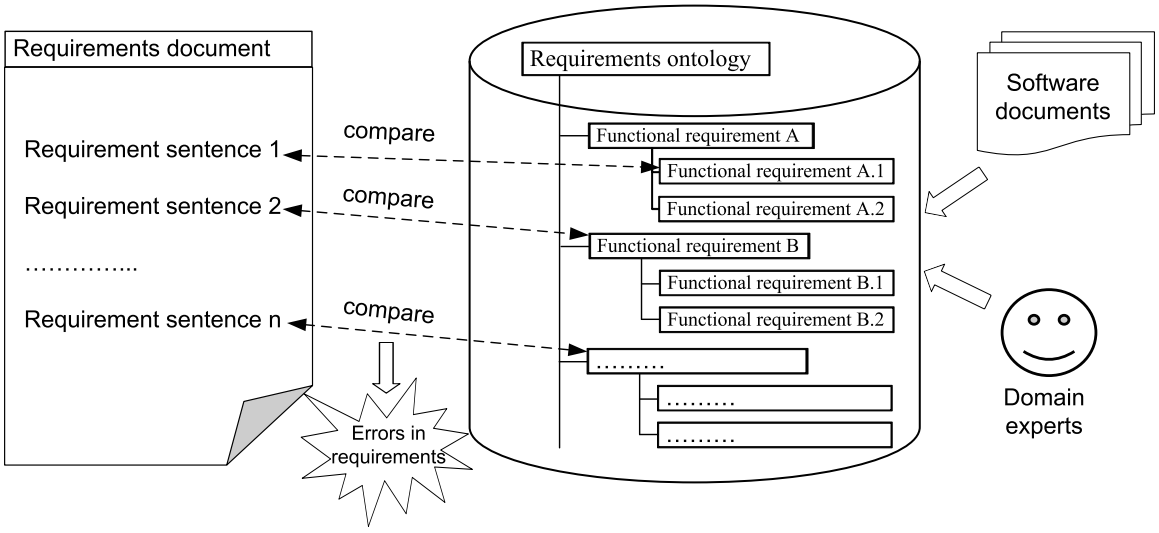


Figure 1.4 An approach to solve the first research question.

For issue 2, to detect errors in order to improve quality of requirements ontology, we specify verification rules from domain knowledge and general knowledge. Each rule specifies a characteristic that requirements ontology should satisfies. Rules are described according to rule templates, so we can use a rule interpreter to interpret rules when verifying requirements ontology against the rules. This solution to issue 2 is presented in Fig. 1.5.

For issue 3, we have developed two verification tools. The first verification tool which bases on Prolog has been developed to support the verification method of software requirements using requirements ontology. The second verification tool which includes a rule interpreter (as explained in above paragraph) is to support the verification method of requirements ontology using rules.

For issue 4, we evaluated the efficiency of the methods by conducting comparative experiments, in which one group of subjects used our methods and the other group did not. We expected that by using the methods, the number of errors detected would be more, and the quality of requirements specification and requirements ontology would be improved.

1.3 Outline of Thesis

The thesis is organized as follows. The next chapter discusses background of the research in requirements engineering and ontology, reviews related works and compares with our works. Then Chapter 3 introduces requirements ontology model and construction of requirements ontology. Chapter 4 proposes ontology-based verification method of software

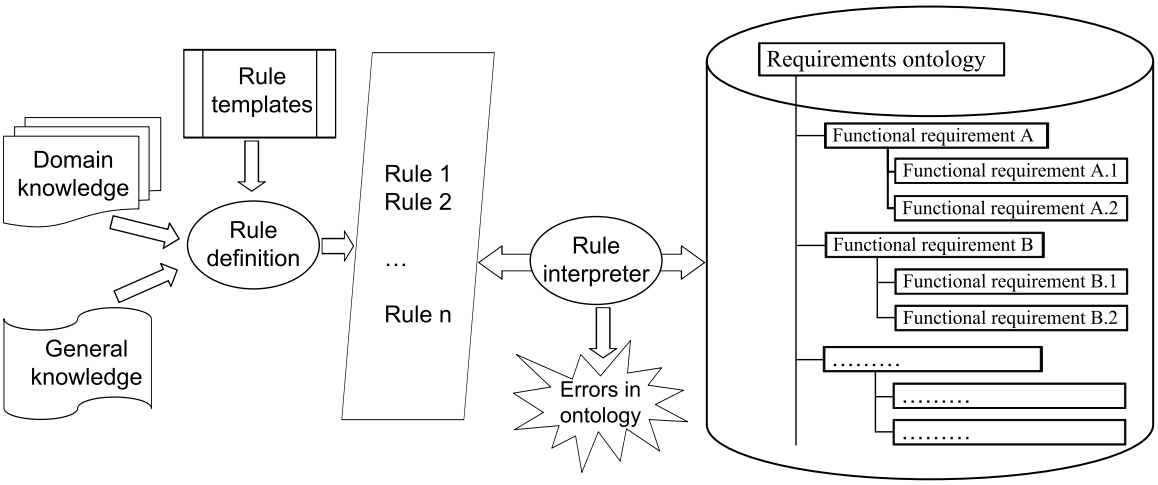


Figure 1.5 An approach to solve the second research question.

requirements. To evaluate the method in Chapter 4, Chapter 5 presents several comparative experiments. In order to guarantee quality of requirements ontology, Chapter 6 provides a verification method of the correctness of requirements ontology. Chapter 7 will evaluate the method in Chapter 6 through experiments. Finally, Chapter 8 summarises research results and discusses future works.

This page intentionally left blank.

Chapter 2

Background and Related Works

In this chapter, we briefly explain the research area of this thesis, and propose using of several definitions of terminologies in this thesis. We then discuss recent related works and compare with our research.

2.1 Background

2.1.1 Software engineering

The original definition of software engineering was by Fritz Bauer in the first NATO conference of software engineering, 1968 [34]:

> "Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."

The definition emphasizes the applying of engineering principles to software development. The fact that many large software projects were over budget or behind schedule is named "software crisis." To face with such crisis, engineering disciplines is applied to yield cost-effective software with good quality.

Another definition of software engineering by Sommerville [45]:

> "Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stage of system specification through to maintaining the system after it has gone into use."

Sommerville's definition also stresses on "engineering discipline" and covers "all aspects of software production." It means that we should consider not only software process

but also tools and methods to support software process; not only product of software process (the runnable code) but also other artifacts of software process such as documents, management, etc.

Although applying software engineering techniques, many software projects still have problems such as late delivery or do not meet users' needs. The reasons are not capability of software engineers or software processes which are applied, but problems with requirements of software systems [28]. If problems exist in software requirements, bigger issues will raise in latter phases of software development or in software product.

2.1.2 Software requirements

Software requirements are statements of characteristics that the software should have. Several examples of software requirements are as follows.

1. We want to build a students portal.
2. Customers can login our online web store using their usernames and passwords.
3. Students are forbidden to update his/her name, birthday, and student ID.
4. Our website should be implemented using PHP and MySQL.

Requirements can be general statement (as in example 1), functional description (as in example 2), constraint (as in example 3), or specific implementation (as in example 4). Wiegers discussed that a requirement statement might be classified into several types: business requirement, user requirement, system requirement, functional requirement, technical requirement, and constraint [63]. It depends on type of stakeholders who describe requirements.

Standing from point of view of software users, software requirements are descriptions of what the software should do. From side of software engineers, software requirements might include information of how to implement the system. Some authors claim that requirements should contain only what information, instead of how information. However, software requirements might contain a mixture of problem information, expected behavior of software, and environmental constraint which includes characteristic of running platform or deployment organization. If we strictly eliminate all how information from software requirements, subsequent design solution might conflict with constraints in requirements or regulation of the organization.

In this thesis, we use the definition of software requirements as 'what should be implemented', by Kotonya and Sommerville [27]. They consider a requirement as any statement of system service or constraint.

2.1.3 Requirements engineering

Requirements engineering is a set of techniques to discover, document, and maintain requirements of a system. The term "engineering" implies that systematic and repeatable techniques should be applied to attain requirements of high quality [46]. Though the techniques can be applied in development of software, hardware, or any type of systems, but most works in requirements engineering focused in "software-intensive system" [59]. Regardless of nature of a system whether it is hardware or software or other types, it can be considered as a machine. Therefore, requirements engineering techniques are not much different for software and non-software systems.

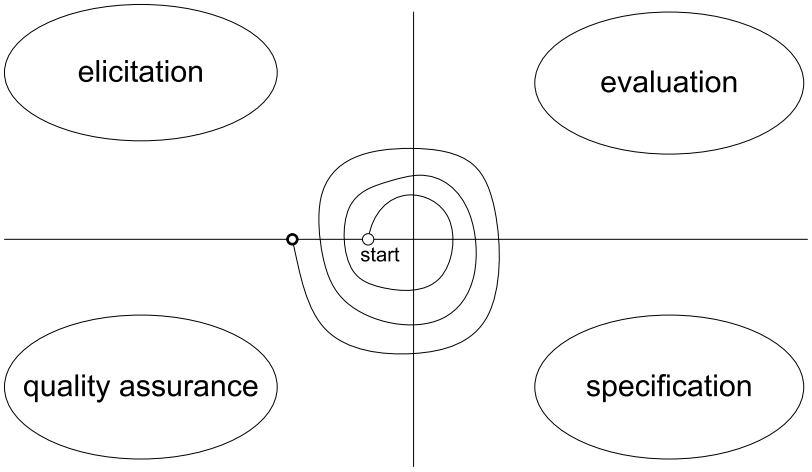


Figure 2.1 A spiral requirements engineering process [58].

Typical requirements engineering activities include: elicitation, evaluation, specification, and quality assurance. Requirements elicitation concerns with discovery of candidate requirements of the system-to-be (the system which is being developed). Requirements evaluation focuses on analysis and negotiation of elicited requirements. Requirements specification consists of organizing and writing agreed requirements in a requirements document. Requirements quality assurance consolidates requirements document with detection of errors in requirements document and fixing them. These activities have not to be linear, but can be spiral, as shown in Fig. 2.1. In the figure, requirements engineering activ-

ities can be iterated several rounds until final requirements reach certain quality standards of organizations or projects.

In this thesis, we focus on requirements quality assurance, but our study also support other activities such as requirements elicitation or requirements evaluation.

2.1.4 Requirements document

Requirements document is the output of requirements engineering process. It is an official statement of requirements of the system-to-be. Stakeholders agree on requirements document, and software developers base on requirements document to proceed to further steps in software development. Depending on organization, requirements document can be called "software requirements specification" (SRS), "functional specification", "requirements definition" [26]. In this thesis, we abbreviate requirements document as SRS.

There is no global standard for writing SRS. Structure and style of SRS depend on organizations. IEEE std. 830-1998 is a common reference document when writing SRS [17]. An example of a recommended format of SRS is in Fig. 2.2.

Our research aims to improve quality of SRS, thus indirectly help improving quality of the software-to-be (the software which is being developed). The structure of SRS, as in Fig. 2.2, can be reviewed by human quickly. Instead of the structure, we focus on semantic of software requirements statements. In this thesis, we consider SRS as a list of software requirements. Each requirement is simply a statement (a sentence) of the software-to-be .

2.1.5 Requirements quality assurance

Writing SRS is not enough. We need to make sure that SRS is correct and good enough to serve as a basis for design, construction, and testing [62]. Lamsweerde summarised requirements quality assurance as follows [56] (italic texts are our inserted words).

> "Requirements quality assurance consists of detecting defects in the requirements document (*SRS*), reporting them, analysing their cause and undertaking appropriate actions to fix them... The main target of this process is the completeness, consistency, adequacy, unambiguity, measurability and comprehensibility of *SRS* items. The later such defects are found with respect to these target qualities, the more costly their repair is."

As described in Sect. 1.1, there are four major techniques in requirements quality assurance: inspections and reviews, queries on a requirements database, animation-based

1. Introduction
    1.1. Purpose
    1.2. Scope
    1.3. Definition, acronyms, and abbreviations
    1.4. References
    1.5. Overview
2. Overall description
    2.1. Product perspective
    2.2. Product functions
    2.3. User characteristics
    2.4. Constraints
    2.5. Assumptions and dependencies
3. Specific requirements
    3.1. External interface requirements
    3.2. Functional requirements
        3.2.1. User class 1
            3.2.1.1. Functional requirement 1.1
                .
                .
            3.2.1.n. Functional requirement 1.n
                .
                .
        3.2.2. User class m
            3.2.2.1. Functional requirement m.1
                .
                .
            3.2.2.m. Functional requirement m.n
    3.3. Performance requirements
    3.4. Design constraints
    3.5. Software system attributes
    3.6. Other requirements
4. Appendices
5. Index

Figure 2.2 A template of SRS (organized by user class) in IEEE std. 930-1998 [17].

validation, and formal verification [53]. However, there are still problems with existing techniques. Inspections and reviews are widely used, but it does not utilize domain knowledge. It is difficult for analysts to review SRS without domain knowledge of the software-to-be, but it is not always possible to find analysts in a software project who have such

domain knowledge. The other three methods: queries, animation validation, and formal verification require the whole or a part of SRS written in special notations (diagram, formal specification). Instead of special notations, software stakeholders prefer using natural languages in SRS [30].

How to improve quality of SRS which is written in natural languages? The format of SRS and the conformance to standard should be check manually by requirements analysts, but how to check the semantics of requirements specification, especially the detail description of functionalities of the software-to-be (section 3 in Fig. 2.2). The semantic checking of functionalities demands domain understanding from analysts. In our research, we study how to support requirements quality assurance with domain knowledge. We also focus on requirements which are stated in natural languages.

There are two terminologies related to requirements quality assurance: validation and verification (e.g., animation validation, formal verification). Boehm has expressed the difference between them [5]:

- "Validation: Are we building the right product?"
- "Verification: Are we building the product right?"

According to Boehm's descriptions, in software development, software validation ensures that software product meets user's needs, while software verification ensures that software product has been built in conformance with requirements and design specifications.

In scope of requirements quality assurance, Lamsweerde discussed that requirements "should be validated with stakeholders in order to pinpoint inadequacies with respect to actual needs"; requirements "should also be verified against each other in order to find inconsistencies and omissions" [57]. Requirements validation means comparing SRS with customer needs, which demands the involvement of customers. Requirements verification means checking SRS against some standards. Between the two, verification is focused in this thesis.

The verb "verify" originates from two Latin words "*vērus*" and "$facio$" meaning "true" and "make," respectively [64]. It means "to confirm something as true." In essence, "to verify" implies "to prove formally" (formally: in a machine processable form [54]). In this sense, "verification of software requirements" means proving formally that software requirements satisfy some quality standards.

We use the term "checking" to refer to general activities of confirmation of quality of software requirements. "To check" means "to see if something is correct or accurate". Checking is a process of validation, verification, or confirmation.

In this thesis, we will cover verification of SRS both automatically and manually. In our research, verification of SRS are mainly methods for detection of errors in a list of software requirements. If no error exists, the requirements list satisfies quality standards.

Software requirements should be verified as soon as possible, but how early can verification of software requirements start? Wiegers stated that SRS should be inspected when requirements are just ten percent complete [61]. It means that we can start quality assurance activities even in requirements elicitation. The assumption that SRS includes a list of software requirements, besides advantage of independence of structure of SRS, has another strength that we can also apply our verification method to a list of elicited requirements or a list of initial requirements in early stages of requirements engineering.

2.1.6 Ontology

Ontology, or the nature of being, originally refers to the question "what kinds of things exist" [44]. The topic has been studied by philosophers from ancient time. When Aristole studied about "essence of thing", he established a system of categories to classify things in the world [8]. Through a numerous other researches, gradually ontology represents abstract form of organizations and systems in universe. To philosophers, ontology is a classification of categories [13].

To engineers, ontology has to be in a machine processable language [50]. In information community, a recent definition of ontology by Gruber is gained some agreement [12]:

> "An ontology is a formal, explicit specification of a shared conceptualization for a domain of interest."

The above short definition expresses three informations. Firstly, an ontology is specified formally, meaning in a machine processable format. Secondly, an ontology contains shared conceptualization which means agreement on concepts. Thirdly, an ontology is for a domain of interest.

In practice, general ontology model includes classes (concepts), attributes, and relations. Fig. 2.3 displays a representation of the model.

However, in some implementations, an attribute can be also considered as a class, and there exists a relation of hasAttribute between a parent class and an attribute class. For example, Pizza ontology is a popular example of ontology[1]. A part of class diagram for Pizza ontology is displayed in Fig. 2.4. The Pizza ontology includes common classes

[1] http://www.co-ode.org/ontologies/pizza/

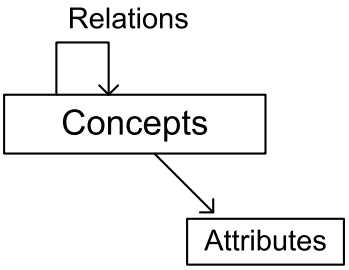


Figure 2.3 A model of general ontology.

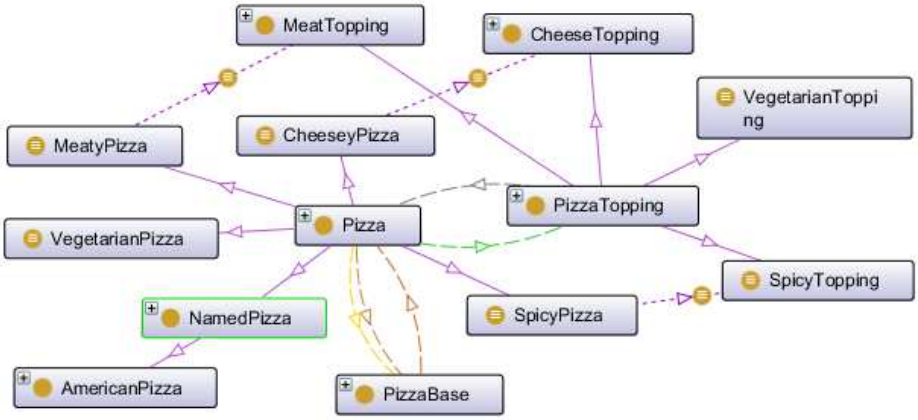


Figure 2.4 A part of class diagram of Pizza ontology.

(drawn by OntoGraf plugin for Protege [41])

related to Pizza cookie such as PizzaBase, PizzaTopping, SpicyPizza, VegetarianPizza, and so on. The class Pizza has relation of hasTopping with the class PizzaTopping and relation of hasBase with the class PizzaBase.

What ontology is used for? Clearly we see that ontology is reuse-oriented. Ontology is used as a dictionary which includes vocabularies in a domain and specification of meanings of the vocabularies. Ontology can be used as a reference book when you work in a domain but lack knowledge of the domain.

In our research, we focus on using ontology to support requirements engineering. We will propose a type of ontology to support requirements elicitation or verification (which will be explained in Chapter 3).

As a conclusion of this section, we summarise a list of short definitions of several terms in Table 2.1. Though there are many other ways of definitions, in this thesis we make agreement to use these meanings of terms.

Table 2.1 A list of short definitions of several terms in this thesis.

Term	Meaning
a requirement	any statement of system service or constraint
SRS	a list of software requirements
to check	to see if something is correct
verification	check something against standards
ontology	classification of categories

2.2 Related Works

There are several related works in requirements engineering using ontology. Some works focuses on early stage—the construction of ontology [2, 6, 69], and some others supports requirements elicitation [9, 19, 23, 24, 32, 67, 72], and some works relate to ontology-based detection of errors in software requirements [29, 70, 71].

2.2.1 Researches support construction of ontology

Some works focused on the construction of ontology. Breitman and Leite used language extended lexicon (LEL) to represent terms and phrases in application language, and then proposed framework to construct ontology [6]. Similarly, Zhang, Mei, and Zhao provided feature diagram to represent domain knowledge [69]; Bao et al. proposed maintenance framework for domain ontology with focus on formal representation of process changes [2].

The above three researches focused on construction of ontology, but did not focus on using ontology to detect errors in software requirements like our research.

2.2.2 Researches support requirements elicitation

A number of researchers have explored the usage of ontology to support requirements elicitation, notably Kaiya, Saeki, Kitamura and colleagues proposed ontology-based requirements elicitation method [19, 23]. Their method proposed general rules for requirements elicitation, but did not provide user definition rules for detection of errors of requirements belonging to specific domains, as described in Section 4.5. Kluge et al. described business requirements and software characteristics in terms of ontology, somewhat similar to our representation of functional requirements in ontology [24]. Their method helps busi-

ness people to compare and match their functional requirements with functions of available commercial software products, and choose a suitable one. Zong-yong and colleagues divided ontology into multiple stages: domain ontology, task ontology, and application ontology [72]. Domain users participate in the requirements elicitation by fill in the questionnaires directed by ontology, but the method does not describe how to reason new requirements. Dobson, Hall, and Kotonya proposed non-functional requirements ontology to be used to discover non-functional requirements [9]. Our ontology model also contains non-functional requirements and also supports elicitation of those requirements. Xiang et al. divided initial requirements to a list of goal; each goal is narrowed down to a list of sub-goals [67]. Using relation among goals, they can refine initial requirements. Similarly, Liu and colleagues used ontology model consisted of actor, goal, task to do reasoning [32], but their approach is more formal than our research.

The above researches focused on elicitation using ontology, but their methods did not support detection of 4W1H errors, redundant errors, and off-topic errors in SRS; especially their methods did not allow user definition rules like our method (The terms: 4W1H, redundant, off-topic errors, and user definition rules will be explained in Chapter 4).

2.2.3 Researches support detection of errors in software requirements

Some literatures related to detection of errors in requirements. Zong-yong and colleagues represented requirements model as UML diagrams and checked the model using domain ontology and rules [71]. Zhu and Jin proposed method to detect inconsistency in requirements using state transition diagram [70]. Kroha and colleagues transform static parts of UML diagrams to ontology and find inconsistency [29].

The above three researches focused on checking diagrammatic elements in requirements specification, but they did not focus on improvement of quality of requirements which are stated in natural languages.

Chapter 3

Requirements Ontology

This chapter will introduce requirements ontology model which derives from previous research [35, 43, 51, 68], then it discusses how to construct requirements ontology with illustration by examples and a supporting tool.

3.1 Requirements Ontology Model

At the initial phase of software development, customers usually think about desired system in terms of functionality and usability: "Which functions does the system provide? What are steps for using it?" Their views are from outside the system; structures of the system might be invisible to users, but interfaces and functionalities are more perceptible to them. For example, customers may state their needs like: "We want to manage insurance records of our customers." However, customers might not understand their own requirements very well, and software engineers might misinterpret what customers have said [39]. In that case, knowledge in certain problem domain is very useful since it helps analysts to elicit correct and sufficient requirements. For example, analysts of insurance system know that the function "manage insurance records" implies four functions: "add records", "retrieve records", "edit records", and "delete records." This knowledge of functions can be stored in advance, in which each function can have descriptions, and scenarios of using it. Functions might have relations, which are helpful in requirements elicitation, for example: function "edit records" requires function "save records"; function "delete records" may require function "undelete records." Relations among functions will help analysts and stakeholders of software systems elicit requirements methodologically and do not miss functions that are needed.

To store requirements knowledge of software functions in a domain, we propose re-

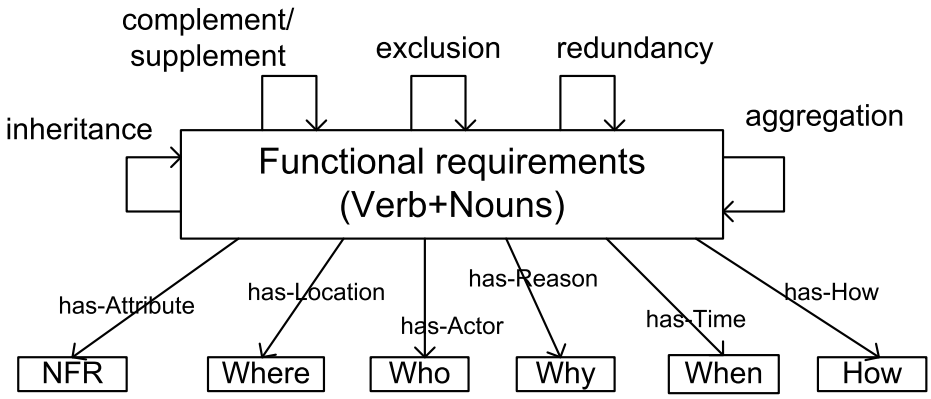


Figure 3.1 Meta-model of requirements ontology.

quirements ontology model. Requirements ontology is a type of domain ontology (domain ontology is an ontology of a certain domain knowledge), but it is specialized for supporting requirements engineering. It consists of functions structure and attributes of functions of software systems. Our requirements ontology meta-model is illustrated in Fig. 3.1. As discussing in the above paragraph, from users' point of view, functional requirements of a system-to-be often refer to actions and objects, as verbs and nouns. Therefore, each functional requirement that includes one verb and several nouns becomes a node in requirements ontology, and relations including inheritance and aggregation can represent a functional structure of systems of a certain domain. We provide other four relations: complement, supplement, inconsistency, and redundancy which are explained below.

- Complement: If functional requirements A and B have a relation of complement, when A is deleted, B also has to be deleted. In addition, when A is added, it is necessary to add B. The opposite also holds. Using complementary relation, we can detect lack of mandatory functional requirements.

- Supplement: If functional requirement A supplements functional requirement B, when B is deleted, A also has to be deleted. In addition, B can appear independently in requirements specification, while A cannot appear independently. Using supplementary relation, we can detect optional functional requirements.

- Exclusion: When functional requirements A and B are exclusive, both A and B should not exist in a requirements document. In other words, if A exists in an SRS, then B should not exist in the requirements document and vice versa. Using this relation, we can detect inconsistent functional requirements.

- Redundancy: When functional requirements A and B are redundant, both A and B can exist in an SRS, but modification of only A may lead to be inconsistent between modified A and B. So, we can detect potential inconsistency with this relation between two functional requirements and modification of just one of them.

Functional requirements can include other information such as agent of function (who), location of function (where), time to do function (when), reason of function (why), and method to do function (how), and non-functional requirements (NFR). The five factors (who, where, when, why, how) are called 4W1H attributes of functional requirements. The 4W1H and NFR attributes are illustrated in bottom of Fig. 3.1.

Fig. 3.2 shows an example of a part of requirements ontology of university portal site. The requirements ontology contains a hierarchy of functions of university portal site; some of the functions are: "search courses, register courses, cancel courses" and so on. The function "register courses" has sub-functions such as: "register main courses", and "register subsidiary courses". The two functions "register courses" and "cancel courses" have a relation of complement which means that they should both exist in a requirements specification. If a software allows students to register courses, it should allow students to cancel courses. However, the software might have the function "print registration results" since it is supplementary to the function "register courses". Figure 3.2 also shows examples of 4W1H attributes (who, when, where, why, how) and NFR for the function "register courses", e.g., "portal" is value of the attribute "where" of the function "register courses".

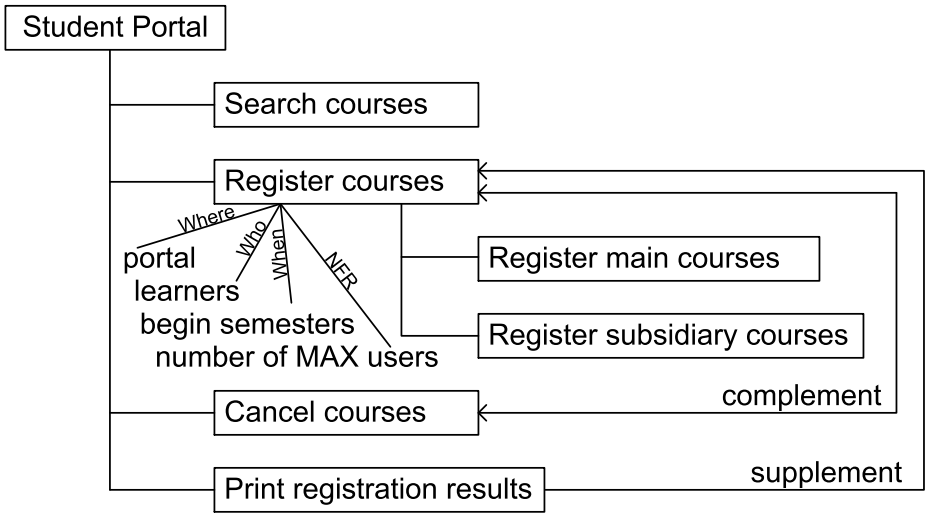


Figure 3.2 A part of the requirements ontology of university portal site

3.2 Construction of Requirements Ontology

Requirements ontology, as in Fig. 3.2, can be built by some ways. One way is to analyze manual documents of existing software systems in the application domain. Another way is to construct a hierarchy of functions and attributes of functions directly by interviewing with domain experts.

User-guide documents usually include list of functions of software systems and guideline about how to use the functions, so we can extract from these documents list of functions and usage context of functions: agent, method, time, location, reason. These informations can be used to build requirements ontology. The extraction of these informations from manual documents has been studied at Software Engineering lab, Ritsumeikan University [51, 68]. There also has another study of how to use these informations to construct requirements ontology [20].

However, the detail of development of requirements ontology is out of scope of this thesis. We suppose that requirements ontology already exists, and use it for verification of quality of requirements specification.

3.3 An Editor Tool of Requirements Ontology

To support construction and management of requirements ontology, an ontology editor tool has been developed. This tool was written with Java, and was two person-months product; the number of source code lines was 7,576. The original of the tool was developed for ontology in Japanese by Sakamoto at Software Engineering lab, Ritsumeikan University [43]. Based on Japanese version, we also developed an English version of the ontology editor tool. Fig. 3.3 shows a screenshot of the ontology editor which allows creating and managing requirements ontology. In the figure, the visual view of requirements ontology is on the left, and the attributes of functions and relations among functions are on the right.

In the tool, requirements ontology is described with OWL [65]. A part of requirements ontology of university portal site in OWL format is shown in the left part of Fig. 3.4. That part of OWL file contains specification for the function “register courses”. The OWL specification in the figure describes that: the functional requirement “register courses” is a subfunction of “student portal”; “register courses” has when attribute as “begin semesters”; “register courses” has a complementary relation with the function “cancel courses”.

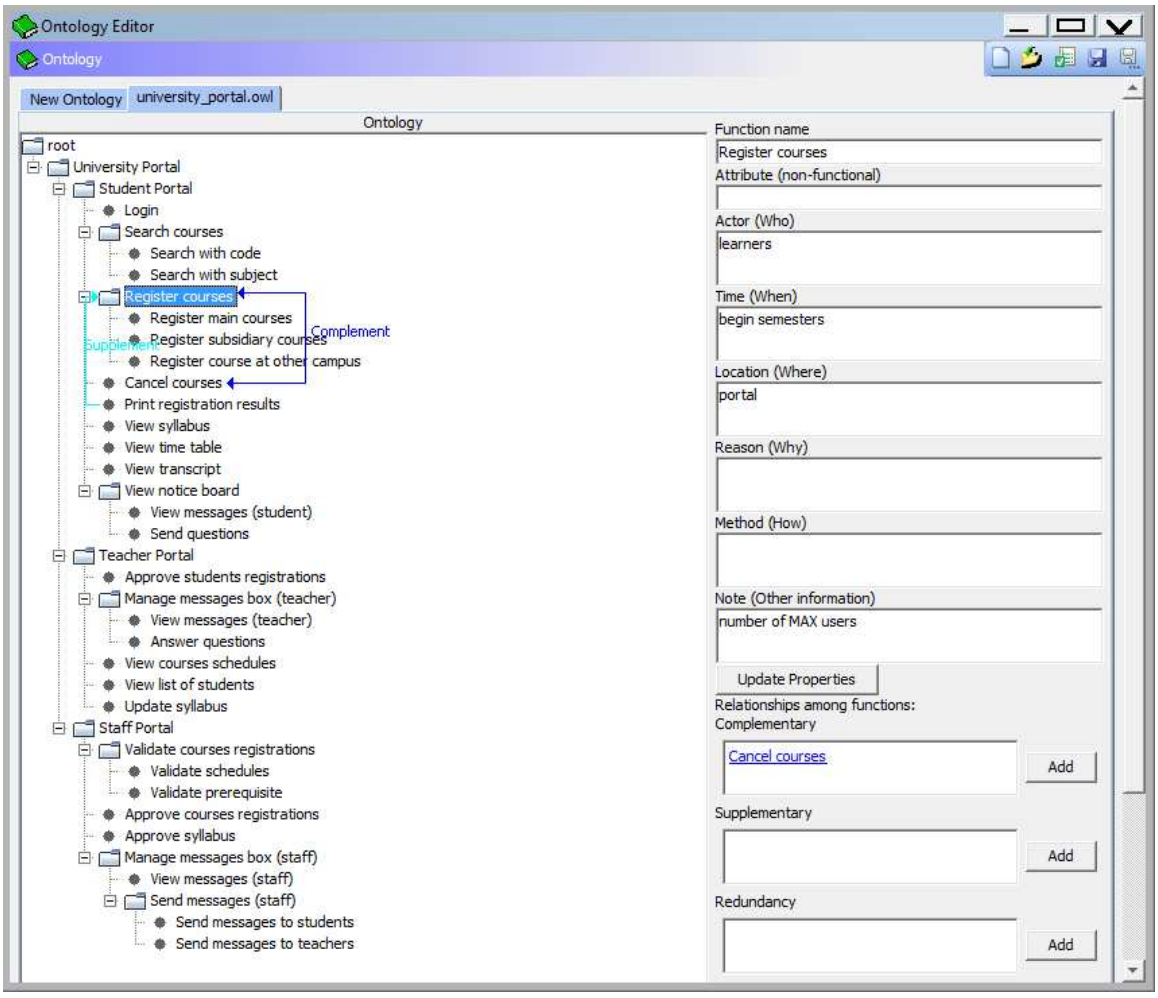
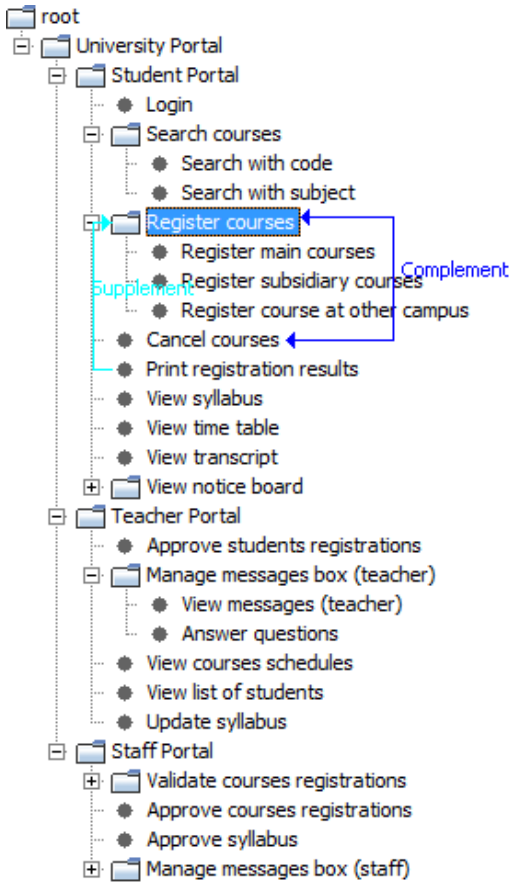


Figure 3.3 Screenshot of requirements ontology editor.

```
<?xml version="1.0" encoding="UTF-8" ?>
<rdf:RDF  xmlns:base="selab.is.ritsumei.jp"
   xmlns:owl="http://www.w3.org/2002/07/owl#">

<owl:Class rdf:ID="Register courses">
   <rdfs:subClassOf>
    <owl:Class rdf:ID="Student Portal"/>
   </rdfs:subClassOf>
   <rdfs:subClassOf>
    <owl:Restriction>
     <owl:onProperty>
      <owl:ObjectProperty rdf:ID="hasWhen"/>
     </owl:onProperty>
     <owl:allValuesFrom>
      <owl:Class rdf:ID="begin semesters"/>
     </owl:allValuesFrom>
    </owl:Restriction>
   </rdfs:subClassOf>
   <rdfs:subClassOf>
    <owl:Restriction>
     <owl:onProperty>
    <owl:ObjectProperty rdf:ID="hasComplementary"/>
     </owl:onProperty>
     <owl:allValuesFrom>
      <owl:Class rdf:ID="Cancel courses"/>
     </owl:allValuesFrom>
    </owl:Restriction>
   </rdfs:subClassOf>
  </owl:Class>
```

(a) Ontology in OWL format



(b) Hierarchy of functions in ontology

Figure 3.4 An example of OWL format for storing of requirements ontology.

Chapter 4

Ontology-based Verification Method of Software Requirements

In our ontology-based verification method, software requirements will be compared with functional nodes in the requirements ontology to find errors. This chapter will explain the techniques of how to compare requirements with ontology and how to detect errors in requirements. Firstly, we discuss type of errors in software requirements that we can detect with requirements ontology. Secondly, we propose a framework of ontology-based verification of software requirements. Thirdly, we illustrate a simple example of finding lack of requirements using requirements ontology. Fourthly, we explain a guideline as a simple approach of verification method which can be applied manually. Fifthly, we elaborate verification method with reasoning rules. Finally, we present a supporting tool for the method which has been developed.

4.1 Types of Errors in Software Requirements

What errors can appear in software requirements? What are characteristics of a good requirements specification? Boehm has stated that major quality attributes of software specification are: "complete, consistent, feasible, testable" [4]. In the IEEE std. 830-1998 "Recommended Practice for Software Requirements Specifications," it is recommended that a good SRS should be: "correct, unambiguous, complete, consistent, ranked for importance and/or stability, verifiable, modifiable, traceable" [17]. Pressman wrote in his popular "Software Engineering" book that problems such as "contradictions, ambiguities, vagueness, incompleteness, and mixed levels of abstraction" can exist in requirements specification [40]. In this thesis, we focus on some types of common errors in SRS that can be

detected using requirements ontology. They are: incorrectness, incompleteness, inconsistency, ambiguity, redundancy.

- Incorrectness: A SRS is correct if all requirements states about what the software shall meet [17]. A requirement might be any description of the software being developed. So if a requirement is not a description of the software, the requirement might be incorrect. However, it is claimed that "there is no tool or procedure that ensures correctness" [17]. Requirements validation only raises the possibility that the specification will meet the real need of customers [47]. For example, a statement that an online store will serve an unlimited number of customers concurrently is considered incorrect since it is unrealistic.

- Incompleteness: A SRS is incomplete if it lacks requirements that are needed. For example, a SRS of an online store software which does not contain description of a function which allows customers to buy products is considered incomplete. Incompleteness is one of the most common problems in requirements specification [40].

- Inconsistency: In a SRS, two requirements are inconsistent if they conflict with each other [48]. For example, one requirement states that warning message should be in red color while another requirement says that the message should be yellow [31].

- Ambiguity: A requirement is ambiguous if it can be understood in several ways [40]. For example, a statement in SRS of library software: "Almost everyone can borrow things from our library." In that statement, who is "almost everyone": user, student, or people in the town; what are "things" in the library: book, magazine, or video tape? That statement is considered ambiguous.

- Redundancy: Repeated requirements or requirements which express the same meaning in SRS are redundant. However, elimination of redundant items from SRS is not always strictly forced. The reason is that sometimes if a requirement is stated only once, and many other places refer to the requirement, the documents becomes difficult to read. In that case, it is recommended to repeat a short version of the requirement and put a reference in parenthesis [31].

4.2 A Verification Framework

We suppose that requirements ontology is already built for an application domain, then we use the requirements ontology to detect above types of errors in SRS. In our method,

SRS are described as a list of sentences. Each sentence is mapped to a node in requirements ontology. After mapping, we use relations and information in ontology for reasoning to find lacking functions or incorrect descriptions of functions. Based on detected errors, we can revise and improve quality of software requirements. Fig. 4.1 presents an ontology-based verification framework of software requirements.

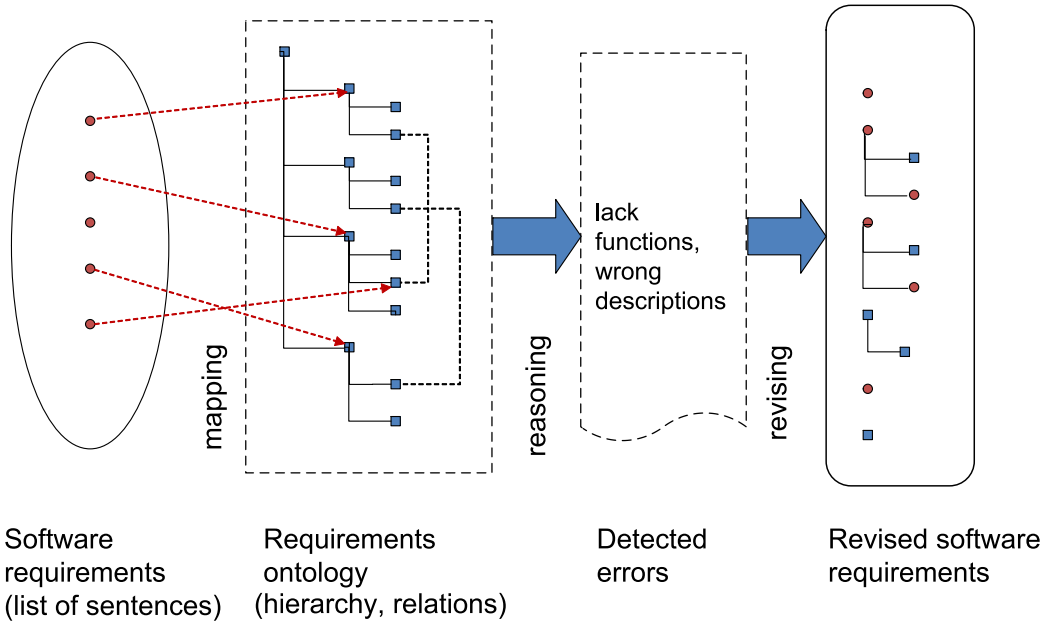


Figure 4.1 An ontology-based verification framework of software requirements.

In this thesis, we focus on clarification of the two activities: mapping requirements to ontology and reasoning for detection of errors in software requirements. The third activity—revising requirements based on errors—is similar to other existing requirements quality assurance techniques (e.g., inspections and reviews).

4.3 A Preliminary Example

We will show a simple example to explain the above verification framework. Firstly, we map requirements sentences to requirements ontology. For examples, a requirement "Student can register for courses online" is mapped to node "Register courses" in ontology model of university portal site, as illustrated in Fig. 4.2.

After mapping, we use relations in ontology for reasoning errors in requirements list. In Fig. 4.2, using relation of complement, the function "Cancel course" should be added to requirements list. Similarly, through relation of supplement, the function "Print registration

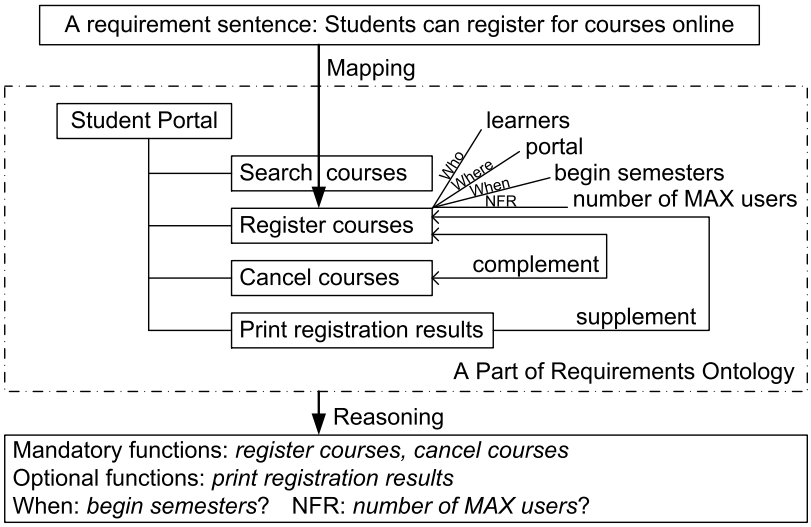


Figure 4.2 An example of verification of requirements using ontology.

results" is optionally added. For optional functions, requirements team can discuss with software stakeholders whether adding them to requirements specification. After doing so, the requirements list will be more complete. In addition, by comparing 4W1H information of requirements sentences with 4W1H attributes of function nodes in requirements ontology, lack or wrong descriptions of 4W1H in requirements can be detected. For example, after mapping a requirement sentence as in Fig. 4.2, we compare "students" with "learners", "online" with "portal", which are attributes "who" and "where" of the function "Register courses", respectively. If these phrases are not equivalent, the requirements should be revised. Moreover, the ontology node "Register courses" has NFR attribute about the maximum number of concurrent users, so the requirements specification of students portal should refer to this issue.

In this way, we can detect lack of indispensable requirements and improve the completeness and correctness of software requirements specifications.

4.4 A Guideline for Verification of Software Requirements using Requirements Ontology

We provide a guideline for ontology-based verification of software requirements. The guideline follows verification framework in Sect. 4.2. In the guideline, firstly, we map software requirements to nodes in requirements ontology. Secondly, we can detect several

type of errors in software requirements as following strategies.

- Incorrectness: We can try to identify potentially incorrect requirements of a software-to-be such as requirements which might be out of scope of the domain. A requirement which does not map to any node in requirements ontology might be out of scope. We find such not-mapped requirements and evaluate if they are out of scope (off-topic) of the software problem.

- Incompleteness: We can detect lack of functional requirements in SRS using the relations among functions in ontology (Fig. 3.2). For example, in Fig. 4.2, we detect the lack of function "Cancel courses" through the relation of complement between two functions "Register courses" and "Cancel courses".

- Inconsistency: To find inconsistent requirements, we search for two requirements mapping to a same node in requirements ontology but having contradict meaning, or two requirements mapping to two functions and having a relation of exclusion.

- Ambiguity: We can find potentially ambiguous requirements by searching for requirements which map to several separate nodes in requirements ontology. Such requirements might be ambiguous since they might express several different functions. We then evaluate if these requirements are really ambiguous.

- Redundancy: We detect redundancy requirements by finding two requirements mapping to a same node in requirements ontology and evaluate if they express the same meaning. If so, the two requirements are redundant. We also search for two requirements which map to two nodes in requirements ontology which have a relation of redundancy.

- 4W1H errors: We compare 4W1H information in requirements statements with 4W1H attributes in requirements ontology to find lack or wrong description of these attributes. We can classify 4W1H errors into the above types of errors (incorrectness, incompleteness, inconsistency), but in order to see the effect of 4W1H attributes in errors detection, we keep this type of errors separately.

Fig. 4.3 summarizes method for finding the above potential errors in SRS using requirements ontology. However, the detected problems are just possibility of errors and should be checked again by human.

Fig. 4.4 show the guideline which includes seven steps. In step (1), each requirement in SRS is mapped to functions in requirements ontology. In step (2), 4W1H information

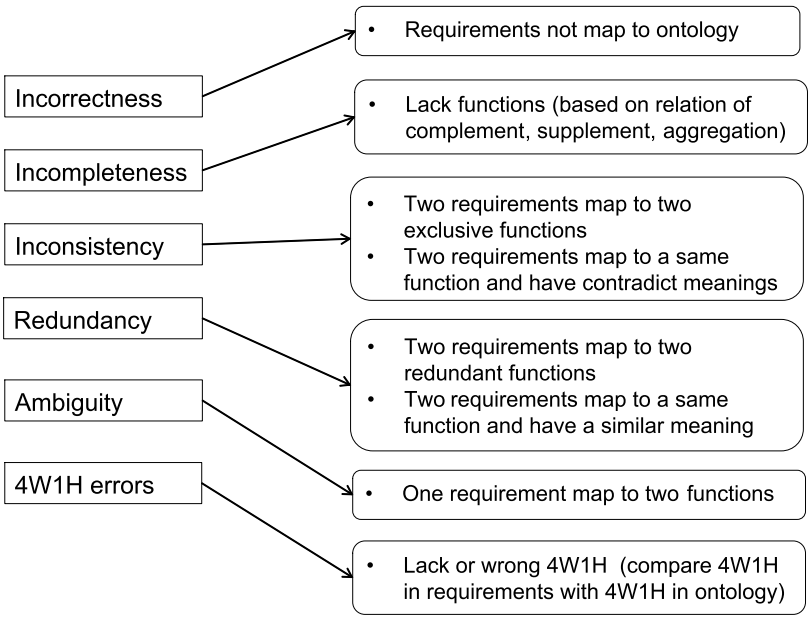


Figure 4.3 Method for detection several types of errors in SRS using requirements ontology.

in requirements are compared with 4W1H attributes in ontology to find errors of wrong or lacking specification of 4W1H. If a function in ontology specifies 4W1H attributes, the mapping requirement should also specify a similar 4W1H information. In step (3), incomplete errors are detected by using relations of complement, supplement, inheritance, and aggregation in ontology. After that, in step (4), two requirements mapping to a same function, or two requirements mapping to two functions having a redundant relation, are detected and evaluated if they are redundant. Step (5) is for finding inconsistent errors: two requirements mapping to two exclusive functions in ontology might be inconsistent; two requirements mapping to a same function but having contradict information might also be inconsistent. In step (6), ambiguous errors are detected by finding requirements mapping to two separate functions and evaluating if they have multiple meanings. In step (7), requirements which do not map to any function in ontology will be evaluated if they are out of scope of application problem (off-topic errors).

Guideline for verification of requirements using ontology: Use following checking table and seven steps under it.

Steps	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Requirements	Mapping to functions	4W1H errors	Lacking potential functions	Redundant with	Contradict with	Ambiguous	Off-topic
R1							
R2							

(1) Map each requirement to functions in ontology

(2) 4W1H errors: Compare 4W1H in requirements and 4W1H in ontology and find errors:

  (a) If ontology describes 4W1H, requirements should also describe about 4W1H, otherwise they might lack 4W1H information.

  (b) If both ontology and requirements have 4W1H information, they should be similar, otherwise they might be wrong 4W1H.

(3) Incomplete errors: Find lacking functions of potential function:

  (a) If a parent function in ontology has been in SRS, its sub-functions might be also in SRS.

  (b) If a function in ontology has been in SRS, other functions having relations of complement with it should be in SRS, and functions having relations of supplement or inheritance with it might also be in SRS.

  (c) Use transitive relations, e.g., if function A is in SRS, and A complements B, and B supplements C, then both B and C should be in SRS.

(4) Redundant errors: Find two requirements mapping to a same function, or two requirements mapping to two redundant functions, then evaluate if they are the same meaning (redundant).

(5) Inconsistent errors: Find two requirements mapping to a same function, or two requirements mapping to two exclusive functions, then evaluate if they have inconsistent meanings.

(6) Ambiguous errors: Find requirements mapping to two functions, then evaluate if they are ambiguous.

(7) Off-topic errors: Find requirements not mapping to any function, then evaluate if they are out of scope of SRS.

Figure 4.4 A guideline for verification of software requirements using requirements ontology.

The above detections of potential errors use some assumptions: a requirement which does not map to any node in ontology might be off-topic; two requirements mapping to a same node in ontology might be redundant or inconsistent depending on whether they express similar or contradict meanings; a requirement mapping to two different nodes in ontology might have two meanings and might be ambiguous. The method detects potential problems in SRS that might be errors; human will check those potential problems finally whether they are errors.

For example, Table 4.2 shows verification steps of the requirement statement "Students can register for courses online." The requirement is mapped to function "Register courses" in step 1. In step 2, we compare 4W1H information of the requirement with 4W1H attributes of the function "Register courses", and we detect the lack of when and NFR attributes. In step 3, through relations of complement and supplement, we find the possibility of lack of functions: "Cancel courses, Print registration results".

4.4.1 A case study

To illustrate the application of above guideline, we apply the method for verification of an SRS of online web store. An example of SRS for online web store software is listed as follows.

R1. Customers can purchase commodities through our online store.
R2. Customers can make payment on delivery of commodities.
R3. We will be informed when the store is nearly empty.
R4. Customer can select commodity to buy from our online catalogues.
R5. We require customers to pay in advance.
R6. Customers can upload images of commodities they like.
R7. We want to send emails of new commodities to customers.
R8. Customers can cancel buying products if they want.
R9. We want to find potential customers and introduce them our store.
R10. We want to sum up the number of products left in store.

Table 4.1 shows an example of a part of requirements ontology for an online store. This ontology represents a case where an online store has several functions such as "F.1 Place order, F.2 Process order, F.3 Manage inventory, F.4 Manage customers relation" and so on. The function "F.1 Place order" has sub-functions: "F.1.1 Search commodity, F.1.2 Select commodity, F.1.3 Remove commodity, F.1.4 Change quantity of commodity..." The function "F.1.2 Select commodity" and the function "F.1.3 Remove commodity" have relationship of complement: if an online store allows customers to select products to add to

order, it should allow customers to remove products from order. However, the software might have the function "F.1.4 Change quantity of commodity" since it supplements to the function "F.1.2 Select commodity." Table 4.1 also shows 4W1H attributes (who, when, where, why, how) for some functions, e.g., "customer" is value of who attribute of the function "F.1 Place order."

Table 4.1 A part of requirements ontology of online store.

Function	Relation	4W1H
F.1 Place order	c F.2	Who: customer
F.1.1 Search commodity	s F.1.2	
F.1.2 Select commodity	c F.1.3	
F.1.3 Remove commodity	c F.1.2	When: order is not finished
F.1.4 Change quantity of commodity	s F.1.2	
F.1.5 View order		
F.1.6 Input shipping address		
F.1.7 Pay order	c F.2.1	How: bank transfer, credit card, payment on delivery
F.1.8 Cancel order	c F.2.3	When: order is not yet shipped
F.2 Process order	c F.1	Who: staff
F.2.1 Process payment	c F.2.2	
F.2.2 Process shipment	c F.2.1	
F.2.3 Process cancellation	c F.1.8	
F.3 Manage inventory	s F.1, F.2	
F.3.1 Update category	c F.3.2	Who: staff
F.3.2 Update commodity	c F.3.1	Who: staff
F.3.3 Statistic commodity		
F.4 Manage customers relation		
F.4.1 Register customer account	s F.4.2	Who: customer
F.4.2 Send email to customer		Who: staff

Abbreviation of relations: c - complement, s - supplement

Table 4.3 shows examples of verification of the SRS of online store using the guideline in Fig. 4.4. In step (1), above requirements (R1–R10) are mapped to ontology as shown in the column "Mapping to functions" of Table 4.3. For example, requirement R1 "Customers can purchase commodities through our online store." is mapped to function "F.1.2 Select commodity" in ontology; requirement R2 "Customers can make payment on delivery of commodities." is mapped to function "F.1.7 Pay order," and so on.

Table 4.2 Examples of verification of a requirement of student portal site.

Steps	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Requirements	Mapping to functions	4W1H errors	Lacking potential functions	Redundant with	Contradict with	Ambiguous	Off-topic
Students can register for course online	Register courses	Lack When, Lack NFR	Cancel courses, Print registration results				

Table 4.3 Examples of verification of SRS of online store using requirements ontology.

Steps	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Requirements	Mapping to functions	4W1H errors	Lacking potential functions	Redundant with	Contradict with	Ambiguous	Off-topic
R1	F.1.2		F.1.1, F.1.3, F.1.4	R4			
R2	F.1.7		F.2.1		R5		
R3	F.3.3			R10			
R4	F.1.2		F.1.1, F.1.3, F.1.4	R1			
R5	F.1.7	lack How	F.2.1, F.2.2		R2		
R6	F.3.2	wrong Who	F.3.1				
R7	F.4.2		F.4.1				
R8	F.1.3, F.1.8	lack When	F2.3, F.1.2			R8	
R9	–						R9
R10	F.3.3			R3			

In step (2), 4W1H information in requirements are compared with 4W1H attributes in ontology to detect the lacking or wrong description. For example, requirement R5 "We require customers to pay in advance." is mapped to the function "F.1.7 Pay order"; the function F.1.7 in ontology specifies how attribute, but requirement R5 does not specify how to pay, so requirement R5 might lack description of how attribute. Requirement R6 "Customers can upload images of commodities they like." is mapped to the function "F.3.2 Update commodity" in ontology, and the requirement R6 specifies the agent of the action as "customers", but the function F.3.2 in ontology specifies who attribute as "staff", so requirement R6 might have wrong description of who attribute. The results of 4W1H errors are listed in column (2) of Table 4.3.

In step (3), incomplete errors are detected by reasoning with relations among functions in ontology. For instance, requirement R1 is mapped to function F.1.2, but in requirements ontology of online store in Table 4.1, function F.1.2 has complement relation with function F.1.3, so function F.1.3 should be in SRS (mandatory function). In addition, function F.1.1 and function F.1.4 supplement the function F.1.2, so the two functions F.1.1 and F.1.4 might be added to SRS (optional functions). The reasoning method for finding incomplete errors is similar to other requirements.

In step (4), redundant errors are detected by finding two or more requirements mapping to a same function and having the same meaning. Both requirements R1 and R4 are mapped to the function F.1.2, and they express similar meanings, so they might be redundant. Requirements R3 and R10 also might be redundant since they are both mapped to the same function F.3.3. The method detects potential problems that might be errors, and human will check these problems finally.

In step (5), inconsistent errors are detected by finding two requirements mapping to a same function but having contradict meanings. Requirement R2 "Customers can make payment on delivery of commodities" and R5 "We require customers to pay in advance" are both mapped to the function "F.1.7 Pay order", but R2 and R5 have contradict meanings, so they are inconsistent.

In step (6), ambiguous errors are detected by finding requirements mapping to two separate functions. Requirement R8 "Customers can cancel buying products if they want." is mapped to two functions: "F.1.3 Remove commodity" and "F.1.8. Cancel order", so requirement R8 might be ambiguous.

In step (7), off-topic errors are detected by finding requirements not mapping to any node in ontology. Requirement R9 "We want to find potential customers and introduce them our store." is not mapped to any function in ontology, so R9 might be out of scope of online store software.

To provide a more visual illustration of the above detection steps, Fig 4.5 shows the mapping and errors detection for some requirements in the SRS of online store. For example, in the figure, requirements R1 and R4 map to a same function "Select commodity", and they have similar meanings, so R1 and R4 are redundant. Requirements R2 and R5 map to a same node "Pay order", but they have contradict meanings, so R2 and R5 are inconsistent. Requirement R8 maps to two function "Remove commodity" and "Cancel order", so it might be ambiguous. Requirement R9 does not map to any node in requirements ontology, so R9 might be out of scope of the software-to-be (off-topic).

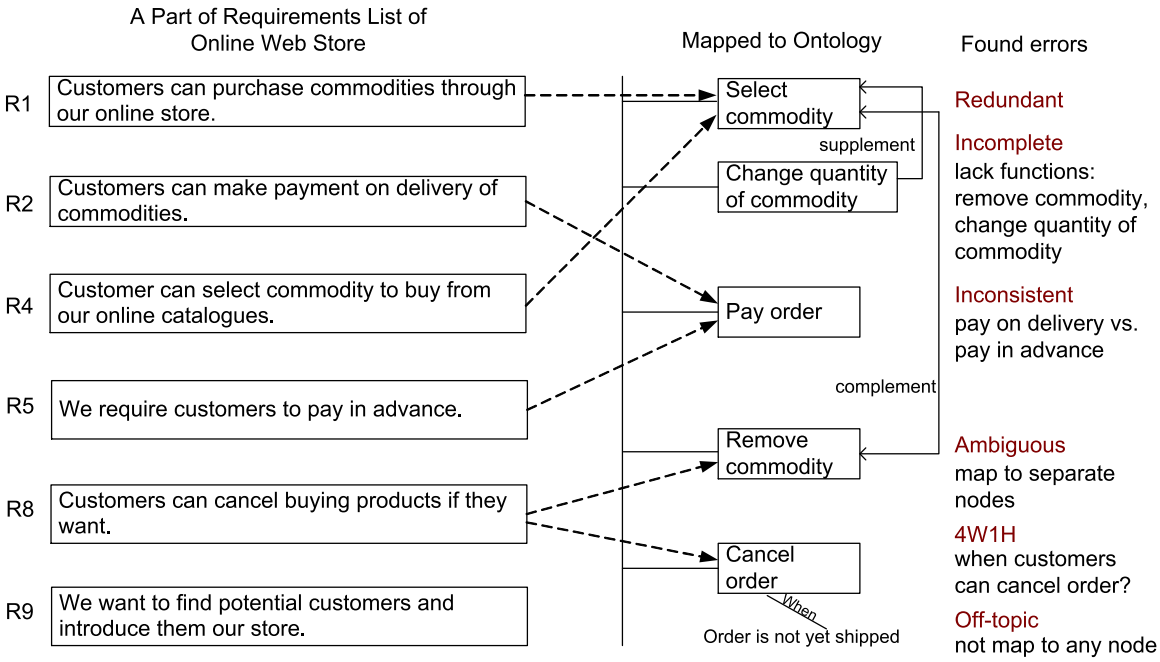


Figure 4.5 Illustration of detection of errors in some requirements of online store.

Based on the checking sheet as in Table 4.3, the checking results of SRS of online store are summarised and listed in Fig. 4.6. In total, there are three 4W1H errors, nine incomplete errors, two redundant errors, one inconsistent error, one ambiguous error, and one off-topic error.

This sub-section has described a guideline for detection of errors in requirements specification. However, with a large ontology and requirements specification, the verification process will require a lot of efforts. Therefore, the verification process should be supported by a reasoning mechanism such as using rules which can be automated by tool. The next section will discuss about reasoning of potential errors in software requirements using rules.

I. 4W1H errors (lacking or wrong description of functional requirements):
   - R5: (lack How) Which methods of payment are provided? Credit card, bank transfer?
   - R6: (wrong Who) Staff, instead of customer, should upload image of products.
   - R8: (lack When) Customer may cancel order only if the order has not yet shipped.

II. Incomplete errors (lacking functions):
   - F.1.1. Search commodity
   - F.1.3. Remove commodity
   - F.1.4. Change quantity of commodity
   - F.2.1. Process payment
   - F.2.2. Process shipment
   - F.2.3. Process cancellation
   - F.3.1. Update category
   - F.3.2. Update commodity
   - F.4.1. Register customer account

III. Redundant errors (duplicate requirements):
   - R1 and R4 seem the same.
   - R3 and R10 refer to the same function of statistic commodity

IV. Inconsistent errors (contradict requirements):
   - R5 contradicts with R2: pay in advance and pay on delivery.

V. Ambiguous errors (multiple-meaning requirements):
   - R8 seems to express two functions: remove commodity from order and cancel order

VI. Off-topic errors (out of scope requirements):
   - R9 seems not a function of online store system.

Figure 4.6 Checking results of SRS of online store using requirements ontology.

4.5 Detail Verification Method of Software Requirements using Requirements Ontology

In general, software requirements are compared with functional nodes in the requirements ontology, then rules are used to find errors in requirements. On the basis of the results, requirements team can ask questions to customers in order to revise requirements.

In detail, each requirement sentence is parsed to get a list of verb and nouns, then the list is compared and mapped to a node in requirements ontology with the help of a thesaurus. For examples, a requirement “Student can register for courses online” is parsed to get a list of verb and noun as {V: register, N: courses}, and then mapped to node “register courses”

in ontology model, as illustrated in Fig. 4.7.

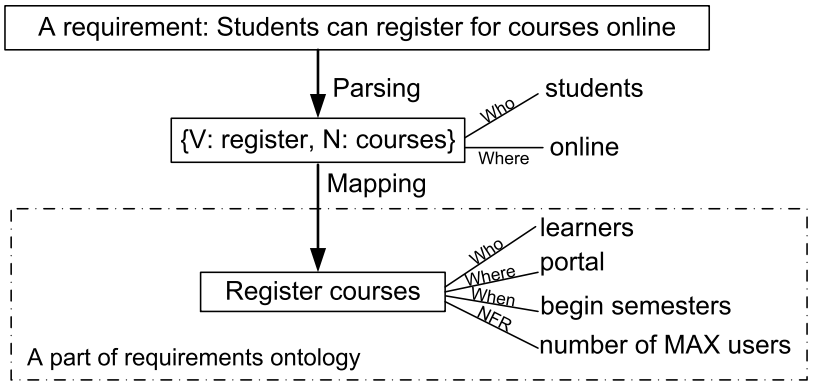


Figure 4.7 An example of parsing requirements and mapping to ontology.

After mapping, we use rules to find errors in SRS (rules will be described later). We summarize the detail method of ontology-based verification of software requirements as four main stages as follows:

1. Parse software requirements;
2. Map requirements to nodes in ontology;
3. Detect errors in requirements using ontology and rules;
4. Interpret results, suggest revising requirements.

Rules are necessary for verification of requirements specification because new rules of a specific system can be used to verify SRS of the system more precisely. We separately provide rules interpreter and rules, because it becomes easy to add new rules. Before describing rules for checking requirements in the following section, we will define some terms related to rules as follows.

- Users: are users of software systems.
- Users of the verification method: are people who want to verify requirements specification documents; they can be requirements elicitors, analysts, managers, or stakeholders. We refer to users of the method briefly as analysts.
- Domain independent rules: are applied to requirements belonging to any domains. Domain independent rules are provided in advance.
- User definition rules: are specified by analysts. User definition rules include domain specific rules and system specific rules.
- Domain specific rules: are used to verify requirements belonging to each specific

domain.

- System specific rules: are used to verify specific requirements documents of a specific system.

4.5.1 Rules definition

Figure 4.8 introduces some notations to describe rules, e.g., the fact that a function X is in the requirements specification is denoted by inSRS(X).

MO(R $\mapsto$ X): requirements R is mapped to ontology node X.
inSRS(X): function X is already in requirements specification.
shouldbeInSRS(X): function X is mandatory.
maybeInSRS(X): function X is optional.
maybeIncorrect(R), maybeAmbiguous(R): requirements R may be incorrect, or ambiguous, respectively.
maybeInconsistent(R, S): requirements R and S may be inconsistent.
maybeRedundant(R, S): either requirements R or S may be redundant.
relation(X, Y): there is a relation between X and Y.
hasWho(X,WX): function X has who attribute as WX.
hasWhere(), hasWhen(), hasWhy(), hasHow(), hasNFR(): similar to hasWho().
shouldSimilar(WR,WX): the two terms WR and WX should be similar.
lackWho(R): requirements R is lacking information about the agents.
lackWhen(), lackWhere(), lackWhy(), lackHow(): similar to lackWho().

Figure 4.8 Predicates used in rules.

Rules describe conditions for whether adding, deleting, or revising a function in requirements list. It depends on the type of relations that a node in ontology has with other nodes which have already been in requirements specification. From relations and functional requirements already in specification, rules will help to find incomplete, inconsistent, or redundant requirements.

Figure 4.9 lists 27 domain independent rules for checking requirements. Rules (1), (5), (6), and (9) are for reasoning about mandatory requirements, while rules (7), (8), and (10) are for finding optional requirements. Mandatory functions should be in SRS, while optional functions are not compulsory. For example, in the requirements of students portal above, the two functions “register courses” and “cancel courses” are mandatory functions,

(1) MO(R ↦ X) → inSRS (X)
(2) complement(X,Y) → complement(Y,X)
(3) exclusion(X,Y) → exclusion(Y,X)
(4) redundancy(X,Y) → redundancy(Y,X)
(5) complement(X,Y) ∧ inSRS(X) → shouldbeInSRS(Y)
(6) supplement(X,Y) ∧ inSRS(X) → shouldbeInSRS(Y)
(7) supplement(X,Y) ∧ inSRS(Y) → maybeInSRS(X)
(8) aggregation(X,Y) ∧ inSRS(Y) → maybeInSRS(X)
(9) inheritance(X,Y) ∧ inSRS(X) → shouldbeInSRS(Y)
(10) inheritance(X,Y) ∧ inSRS(Y) → maybeInSRS(X)
(11) exclusion(X,Y) ∧ MO(R ↦ X) ∧ MO(S ↦ Y) → maybeInconsistent(R,S)
(12) redundancy(X,Y) ∧ MO(R ↦ X) ∧ MO(S ↦ Y) → maybeRedundant(R,S)
(13) ¬MO(R ↦ X) → maybeIncorrect(R)
(14) MO(R ↦ X) ∧ MO(R ↦ Y) → maybeAmbiguous(R)
(15) MO(R ↦ X) ∧ MO(S ↦ X) → maybeRedundant(R,S)
(16) MO(R ↦ X) ∧ hasWho(R,WR) ∧ hasWho(X,WX) → shouldSimilar(WR,WX)
(17) MO(R ↦ X) ∧ ¬hasWho(R,WR) ∧ hasWho(X,WX) → lackWho(R)
(18) MO(R ↦ X) ∧ hasWhen(R,WR) ∧ hasWhen(X,WX) → shouldSimilar(WR,WX)
(19) MO(R ↦ X) ∧ ¬hasWhen(R,WR) ∧ hasWhen(X,WX) → lackWhen(R)
(20) MO(R ↦ X) ∧ hasWhere(R,WR) ∧ hasWhere(X,WX) → shouldSimilar(WR,WX)
(21) MO(R ↦ X) ∧ ¬hasWhere(R,WR) ∧ hasWhere(X,WX) → lackWhere(R)
(22) MO(R ↦ X) ∧ hasWhy(R,WR) ∧ hasWhy(X,WX) → shouldSimilar(WR,WX)
(23) MO(R ↦ X) ∧ ¬hasWhy(R,WR) ∧ hasWhy(X,WX) → lackWhy(R)
(24) MO(R ↦ X) ∧ hasHow(R,HR) ∧ hasHow(X,HX) → shouldSimilar(HR,HX)
(25) MO(R ↦ X) ∧ ¬hasHow(R,HR) ∧ hasHow(X,HX) → lackHow(R)
(26) MO(R ↦ X) ∧ hasNFR(R,NR) ∧ hasNFR(X,NX) → shouldSimilar(NR,NX)
(27) MO(R ↦ X) ∧ ¬hasNFR(R,NR) ∧ hasNFR(X,NX) → lackNFR(R)

Figure 4.9 Rules for verifying requirements.

but the function "print registration results" is an optional function. Mandatory requirements and optional requirements will contribute to the completeness of SRS. Rule (11) supports reasoning about inconsistent requirements; and rule (12) is for finding redundant requirements. The elimination of inconsistent requirements and redundant requirements also improves quality of SRS.

If a requirement is not mapped to any node in requirements ontology, it might be out of scope and might be incorrect, as specified in rule (13). Since each function is represented by

one node in requirements ontology, one requirements sentence which maps to two separate nodes in ontology might have multiple meaning and might be ambiguous, which is defined in rules (14). For example, a requirements sentence "Students can reserve library items through portal" is mapped to two ontology nodes: "reserve book" and "reserve video tape", so the requirements sentence is ambiguous. Two requirements that map to a same node in ontology might leads to redundancy, as specified in rule (15). The above matters are machine detectable issues, and they might be errors; human will check finally whether they are really errors.

Fig. 4.10 shows some reasoning examples to illustrate several of the above rules. For complement rule (5), if "register courses" is in SRS then "cancel courses" should be in SRS. For supplement rule (6), if "print registration results" is in SRS then "register courses" should be in SRS. For aggregation rule (8), if "manage commodity" is in SRS then its sub-functions ("add commodity, delete commodity, edit commodity") may be in SRS. For inheritance rule (9), if "save HTML file" is in SRS then its predecessor ("save document") should also be in SRS. For exclusion rule (11), users should choose either "use web interface" or "use desktop interface" but not both. For redundancy rule (12), "add commodity" and "insert product" should not both exist in SRS because they are redundant.

Rules from (16) to (27) in Fig. 4.9 are for reasoning of errors in 4W1H attributes. Rule (16) states that if a requirement R is mapped to an ontology node X, the agents (who attribute) of R and X should be similar. Otherwise it is considered inconsistent and needed to be revised. Rules (17) specifies conditions when a requirement is lacking descriptions of the agents, but the mapped ontology node has specified them. Other rules (18–27) for reasoning of errors in when, where, why, how, and NFR attributes are similar to rules (16) and (17).

Using this reasoning mechanism, mandatory requirements, optional requirements, inconsistent requirements, redundant requirements, and ambiguous requirements can be detected. From these errors, analysts can recommend whether to add, remove, or revise some requirements. Then the requirements team will make a revised version of requirements specification. Through verification, new requirements can be found and can be added to SRS. Analysts can start again another round of verification of the revised SRS, until no more errors are found. This refinement makes the requirements list satisfies the quality attributes such as completeness, consistency, unambiguous, and non-redundancy.

There are errors that cannot be detected by domain independent rules. These errors are violation of specific domain properties or system constraints. For example, in checking SRS of an education management system (EMS), analysts want to assure that students cannot access scores. Access includes several activities such as retrieve, change, and delete.

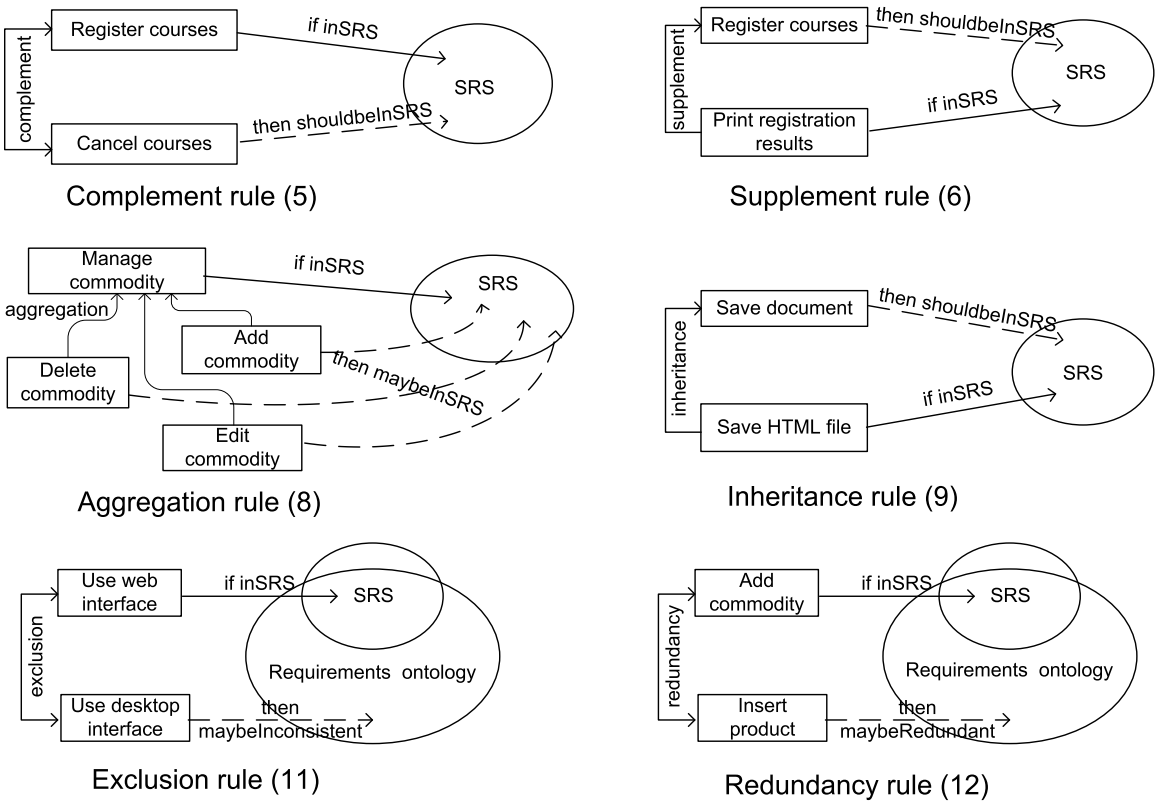


Figure 4.10 Examples of reasoning using several rules in Fig. 4.9.

This constraint applies to many applications in education domain and relates to a group of functions, and it is not general enough to set as domain independent rules.

To allows analysts to specify user definition rules, a grammar is presented in Fig. 4.11. This grammar is in BNF form and uses pre-defined predicates in Fig. 4.8; it allows to define two common types of business rules: integrity rule and derivation rule [33, 42]; it also allows analysts to specify new predicates. For example, analysts can specify a domain rule which states that only administrators can update users of a software, and update implies add, edit, or delete:

(DR1) Vupdate ::= 'add' | 'edit' | 'delete'
(Vupdate 'users').who SHOULD BE 'administrator'

A rule that a student should not update his/her name, his/her student ID, his/her birthday, and his/her scores can be written such as:

(DR2) Nreserved ::= 'ID' | 'name' | 'birthday' | 'scores'
(Vupdate Nreserved).who SHOULD NOT BE 'student'

<DS-RULE >::= <INTEGRITY-RULE>|<DERIVATION-RULE>
<INTEGRITY-RULE>::= <V N>.<attribute> [SHOULD|SHOULD NOT] BE <constraint-value>
<DERIVATION-RULE>::= IF {<predicate>} THEN <predicate >
<predicate>::= <functor>({<term>})
<term>::= <V>|<N>|<V N>|<V N>.<attribute>|<variable>
<functor>::= <pre-defined-functor>|<user-defined-funtor>
<pre-defined-functor>::= (list of predicates in Fig. 4.8)
<V>::= {list of verbs in problem domains} |<variable>
<N>::= {list of nouns in problem domains}|<variable>
<attribute>::= who|when|where|why|how
<variable>::= X|Y|WX|WY

Figure 4.11 A grammar for extension of rules.

The above rule describes that: reserved nouns include ID, name, birthday, and scores; the agent of the action of updating of reserved nouns should not be student.

Another rule that if there is a function of insertion of item, and there is not a function of deletion of item, it should have a function to hide or disable the item:

(DR3) Vins ::= 'insert' | 'add' | 'create'
Vdel ::= 'delete' | 'drop' | 'remove'
Vdis ::= 'hide' | 'conceal' | 'disable'
IF inSRS(Vins X) AND NOT inSRS(Vdel X) THEN shouldInSRS(Vdis X)

We will show examples to illustrate the usage of user definition rules. Suppose that an university want to develop a students portal site as a new module of the existing EMS. Initial requirements are described as follows.

1) We want to create a new module as students portal.
2) On the portal, teachers can create accounts for new students.
3) Students can update their accounts with: name, password, and address.

Using rules (DR1), (DR2), and (DR3) above, analysts find that the requirement 2) is inconsistent with rule (DR1), since rule (DR1) states that only administrators can create new accounts. They can also use rule (DR2) to detect a problem with the requirement 3), because rule (DR2) specifies that students cannot change their names themselves. In addition, by applying rule (DR3) to the requirement 2), since the software has the function "create accounts," but it does not have the function "delete accounts" analysts will find the

need of the function "disable accounts." Totally, using above three user definition rules, analysts can find three errors which cannot be detected by domain independent rules.

User definition rules are useful when requirements analysts want to verify certain properties of domains or systems. Normally, an analyst who verifies a SRS should not participate in preparing requirements documents, and (s)he wants to assure that the SRS satisfy domain properties, so (s)he can describe domain properties such as user definition rules for verification of SRS. In case an analyst is a domain expert, (s)he still may need supportive rules to verify a large requirements specification.

4.5.2 Case studies

This section will illustrate the capability of the verification method by applying it with requirements specification of real software projects. We will examine the SRS and use requirements ontology of the corresponding domains and reasoning rules to detect potential errors. The below subsections focus on two SRS: (1) student registration module and (2) online shopping mall. We will also present examples of user definition rules in the second case study.

Case study 1

This case study is about verification of requirements of student registration module at a university [1]. A part of SRS for the module is as follows.

R1. Student can be register on the system and fill in all detail and forward to choose project/ supervisor.
R2. Student can change detail if information is incorrect such as telephone number.
R3. Student can change his login password at any time for security reason.
R4. Student can request his password if he/she forgotten the password.

The student registration module belongs to the category of user registration software. A part of requirements ontology for user registration system is as follows.

F.1. Register account [supplement: Log in system]
F.2. Log in system [complement: Log out system]
F.3. Log out system
F.4. Change password (When: first time log in/ periodic/ any time)
F.5. Reset password (How: sending new password to registered email address)
F.6. Update personal information

[1] URL http://briggs.myweb.port.ac.uk/jimapp/SUMS/Ivan/SRS-v1.2.doc

In the above representation of ontology, descriptions which are inside parentheses() are 4W1H attributes of functions, and descriptions which are inside square brackets[] are relationships among functions.

We will use the above ontology to verify the SRS of student registration module. The parsing of requirements, mapping to ontology, reasoning rules, and errors found are listed below.

R1. Parsing: (V: register, N: student, Who: student), (V: fill, N: detail, Who: student), (V: forward, N: project/supervisor, Who: student)
Mapping: F.1. Register account, F.6. Update personal information
Reasoning rules: (14), (6), (5)
Potential errors: Ambiguous, Lack functions (F.2. Log in system, F.3. Log out system)

R2. Parsing: (V: change, N: detail, Who: student, When: information is incorrect)
Mapping: F.6. Update personal information
Reasoning rules: (15)
Potential errors: Redundant (with requirement R1)

R3. Parsing: (V: change, N: password, Who: student, When: any time)
Mapping: F.4. Change password
Reasoning rules: (18)
Potential errors: Lack 4W1H (When: first time log in/ periodic)

R4. Parsing: (V: forget, N: password, Who: student, When: forgotten password)
Mapping: F.5. Reset password
Reasoning rules: (25)
Potential errors: Lack 4W1H (How)

Requirement *R1* is parsed and mapped to functions *"F.1. Register account"* and *"F.6. Update personal information"* in ontology. Using rule (14), since *R1* maps to two different functions, it might be ambiguous. Using supplement rule (6) and complement rule (5), because the function *"Register account"* is in SRS, we detect that the functions *"Log in system"* and *"Log out system"* should also be in SRS.

Requirement *R2* maps to the function *"F.6. Update personal information"* in ontology. Because requirement *R1* also maps to this function, so it might be a redundant error, due to rule (15). But after checking again by analysts, requirements *R1* and *R4* are not redundant. The verification method suggests potential errors, and analysts will check these potential errors finally.

Requirement *R3* maps to the function *"F.4. Change password"*. The requirement *R1* specifies when attribute such as that student can change password any time. Instead, the

function *"F.4. Change password"* in ontology describe when attribute such as that users should change password at the first time log in, change periodically, or change any time. Therefore, using rule (18), the requirement *R3* might lack a part of description of when attribute about *"first time log in/ periodic"*

Requirement *R4* maps to the function *"F.4. Reset password"*. That function in ontology has how attribute, but the requirement *R4* does not describe about the how attribute. Hence, using rule (25), the requirement *R4* might lack description about the method to reset password.

In this case study, we illustrated the errors detection with each requirement individually for easy of understanding, but actually our method and the requirements verification tool which has been developed (will be described in Sect. 4.6) verifies the requirements set as a whole and return results as a set of all detected errors.

Case study 2

This case study examines requirements of an online shopping mall (OSM) web application [2] to detect potential errors. A part of SRS for OSM application is as follows.

R1. (User: Administrator) Secure registration and profile management facilities for Customers.
R2. (User: Customer) Browsing through the e-Mall to see the items that are there in each category of products like Apparel, Kitchen accessories, Bath accessories, Food items etc.
R3. (User: Customer) Adequate searching mechanisms for easy and quick access to particular products and services.
R4. (User: Administrator) Creating a Shopping cart so that customers can shop 'n' no. of items and checkout finally with the entire shopping carts.
R5. (User: Employee) Regular updates to registered customers of the OSM about new arrivals.
R6. (User: Employee) Uploading 'Most Purchased' Items in each category of products in the Shop like Apparel, Kitchen accessories, Bath accessories, Food items etc.

The OSM application also belongs to the category of online store software. We has described a part of requirements ontology of online store in Table 4.1, but for easy of reading, we repeat a part of requirements ontology of online store as follows.

F.1. Place order (Who: customer)
  F.1.1. Search product (How: keyword, name, price)
  F.1.2. Select product [complement: Remove product]
  F.1.3. Change quantity [supplement: Select product]
  F.1.4. Remove product

[2] URL https://code.google.com/p/osmlite/

F.1.5. Pay order (How: bank transfer, credit card, payment on delivery) [complement: Cancel order]
F.1.6. Enter shipping address
F.1.7. Cancel order (When: before shipment)
F.2. Process order placement(Who: employee)
F.2.1. Process order payment
F.2.2. Process order shipment
F.2.3. Process order cancellation
F.3. Manage inventory
F.3.1. Update category (Who: employee)
F.3.2. Update product (Who: employee)
F.4. Manage customer relation
F.4.1. Register user (Who: customer)
F.4.2. Update profile (Who: customer) [supplement: Register user]
F.4.3. Dispatch information (Who: employee, How: send email)

We will use the above ontology to verify the SRS for online shopping mall. For example, the requirement *R1* is parsed and mapped to functions *"F.4.1. Register user"* and *"F.4.2. Update profile"* in ontology. Using rule (16), the requirement *R1* specifies that the agent of the action *"register customer"* is *"administrator"*, but the ontology specifies that the who attribute of the function *"F.4.1. Register user"* is *"customer"*, so it might be an error of 4W1H. The requirement *R1* also maps to two nodes in ontology, so it might include multiple requirements (One requirements sentence should be unity and include only one requirement [17]). The fact that a requirement maps to multiple functions nodes in ontology, especially when those functions do not have relationships, might lead to ambiguity in the requirement, according to rule (14). The parsing of requirements, mapping to ontology, reasoning rules, and errors found are listed below.

R1. Parsing: (V: register, N: customer, Who: administrator), (V: manage, N: profile, Who: administrator)
Mapping: F.4.1. Register user, F.4.2. Update profile
Reasoning rules: (16), (14)
Potential errors: Wrong 4W1H (Who), Ambiguous

R2. Parsing: (V: browse, N: item, N: product, Who: customer)
Mapping: F.1.2. Select product
Reasoning rules: (5), (6)
Potential errors: Lack functions (F.1.3. Change quantity, F.1.4. Remove product)

R3. Parsing: (V: search, N: product, Who: customer)
Mapping: F.1.1. Search product
Reasoning rules: (25)
Potential errors: Lack 4W1H (How)

R4. Parsing: (V: create, N: shopping cart, Who: administrator), (V: check out, N: shopping cart, Who: administrator)
Mapping: F.1. Place order, F.1.5. Pay order
Reasoning rules: (16), (25), (14), (5)
Potential errors: Wrong 4W1H (Who), Lack 4W1H (How), Ambiguous, Lack functions (F.1.6. Cancel order)

R5. Parsing: (V: update, N: customers, N: OSM, N: arrivals, Who: employee)
Mapping: F.4.2. Update profile, F.4.3. Dispatch information
Reasoning rules: (14), (25)
Potential errors: Ambiguous, Lack 4W1H (How)

R6. Parsing: (V: upload, N: 'most purchased' item, Who: employee)
Mapping: -
Reasoning rules: (13)
Potential errors: Off-topic

We will explain the errors detection with other requirements. Requirement *R2* maps to the function *"F.1.2. Select product"*, but that function has complement and supplement relationships with two functions *"F.1.4. Remove product"* and *"F.1.3. Change quantity"*, respectively. Therefore, the SRS should specify that two functions, according to rules (5) and (6).

Requirement *R3* maps to the function *"F.1.1. Search product"*, and that function in ontology specifies How attribute, but the requirement *R3* does not specify how information, so it suggests the lacking of how description in requirement *R3*, due to rule (25).

Requirement *R4* maps to the functions *"F.1. Place order"* and *"F.1.5. Pay order"*. Reasoning using rule (16), the function *"F.1. Place order"* in ontology has who attribute as "customer", but the requirement *R4* describes the agent of the action as "administrator", so it is a 4W1H error in requirement *R4*. In addition, the function *"F.1.5. Pay order"* in ontology has How attribute, but the requirement *R4* does not specify how attribute; therefore, *R4* lacks description of how, according to rule (25). In ontology, the function *"F.1.5. Pay order"* has relationship of complement with the function *"F.1.7. Cancel order"*, so another error is the lacking of function cancelling order in SRS, due to rule (5).

Requirement *R5* maps to the functions *"F.4.2. Update profile"* and *"F.4.3. Dispatch information"*, but the two mapped functions do not have relationships, so the requirement *R5* seems to be ambiguous, due to rule (14). Using rule (25), The function *"F.4.3. Dispatch information"* has attributes of who and how, but the requirement *R5* describe about only about who; so the requirement *R5* lacks how attribute.

Requirement *R6* does not map to any function in ontology, so it seem to be an out of scope requirement (off-topic error), according to rule (13).

We will show some examples of applying domain rules to check requirements. A domain rule which states that customer cannot make changes to products in shopping mall can be specified as follows.

Vchange ::= 'add' | 'update' | 'delete'
(Vchange 'product').Who MUST not 'customer'

If we use the above rule to verify the SRS of online shopping mall, the result will be that the SRS does not violate that rule because there is no requirements about customers changing products in shopping mall.

Another rule which states that if there is function for customer to access something in store, then should have function for employee to update that thing.

Vaccess ::= 'search' | 'select' | 'view'
IF inSRS(Vaccess <N>) and (Vaccess <N>).Who = 'customer' THEN shouldbeInSRS(Vchange <N>) and (Vchange <N>).Who = 'employee'

Using the above rule to verify the SRS of online shopping mall, because there are functions for customer to *"search product"* and *"select product"* in SRS, it needs a function for employee to add or update product. Therefore, the SRS lacks the function *"update product"*.

If customers make any change to order, employee should process that change. We can write a rule such as: if there is a function for customers to make change to orders, then it should also have a corresponding function for employee to process that change in customers' orders.

Vor ::= 'place' | 'pay' | 'cancel'
PPor ::= 'placement' | 'payment' | 'cancellation'
IF inSRS(Vor 'order") and (Vor 'order').Who = 'customer' THEN shouldbeInSRS('process order' PPor) and ('process order' PPor).Who = 'employee'

Using the above rule, we detect the lack of three functions: *"process order placement"*, *"process order payment"*, and *"process order cancellation"*. The reason is that the three

functions: *"place order", "pay order", "cancel order"* (inferred by requirement *R4*) already should be in SRS.

This section have described reasoning of errors in software requirements using rules and ontology. However, it is difficult to do reasoning manually with a large ontology and requirements specification. Therefore, a tool is needed to support this method. The next section will introduce an ontology-based verification tool that has been developed.

4.6 Development of an Ontology-based Verification Tool of Software Requirements

We have developed an ontology-based verification tool of software requirements. The tool was written with Java language, using OpenNLP library [36] and SWI-Prolog library [52]. It was a six person-months product; the total lines of codes was 27,851.

The ontology-based verification tool takes a requirements list, ontology and rules as inputs. It then applies reasoning to find errors in requirements and display results to analysts, and they use the results to construct suggestions to revise requirements. Based on these suggestions, analysts will work with customers to revise the requirements list. The tool supports four main activities in ontology-based verification method: (1) Parse requirements specification, (2) Map requirements to nodes on ontology, (3) Detect errors in requirements using ontology and rules, and (4) Interpret results. We will describe these functions in detail in what follows.

4.6.1 Parse requirements

Software requirements are often stated in natural languages, so we need to extract verbs and nouns from requirements statements to construct functional concepts in our format. We use OpenNLP (Open source Natural Language Processing) library [36] to parse requirements specification. The requirements should be stated using plain language, so it is easier to parse and process.

For example, a requirement sentence "We will post notices for students on portal." is parsed and extracted a list of verb and noun as: {V:post, N:notice}, as illustrated in Fig. 4.12. Firstly we use OpenNLP library to parse the sentence to get a sentence structure. Secondly, we execute regular expressions on that sentence structure to obtain verb and noun.

Unlike verbs and nouns, it is more difficult to extract 4W1H from requirements sen-

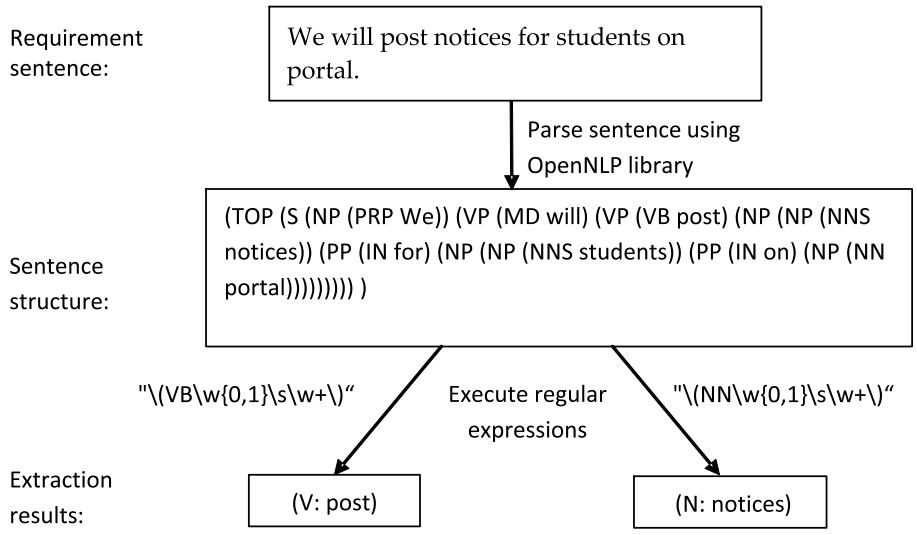


Figure 4.12 An example of parsing a requirement.

tences. With OpenNLP library, we can parse each requirement sentence to a sentence structure (as in Fig. 4.12) or a parsing tree (as in Fig. 4.13) and extract subject, main verb, nouns, and preposition phrases of the main verb. The subject and preposition phrases would become 4W1H attributes of requirements, but how to classify them into five types of 4W1H attributes: who, when, where, why, how? We can classify preposition phrases by types of words in 4W1H phrases: agent or subject (who), location (where), time (when), verb phrase (why or how). For example, in Fig. 4.13, we extract a preposition phrase (with PP tag in the parsing tree): “on portal”. Since the preposition “on” in this case aims to refer to a place, we can classify this 4W1H phrase as a where attribute. However, classification of preposition phrases is a difficult problem [10]. For instance, “on” might also refer to time such as: “on Monday, on Christmas.” We do not focus on natural language processing, so in cases the tool cannot classify 4W1H automatically, we let analysts to select the types of 4W1H manually.

The parsing results in terms of lists of verb and nouns are then used to map requirements to nodes in ontology. The mapping method will be introduced in following sub-section.

4.6.2 Map requirements to nodes in ontology

Fig. 4.14 illustrates the mapping of requirements to ontology. The left side depicts the requirements for building a portal for students to register courses; the right side is a part of requirements ontology of university portal site. Each requirement sentence on the left

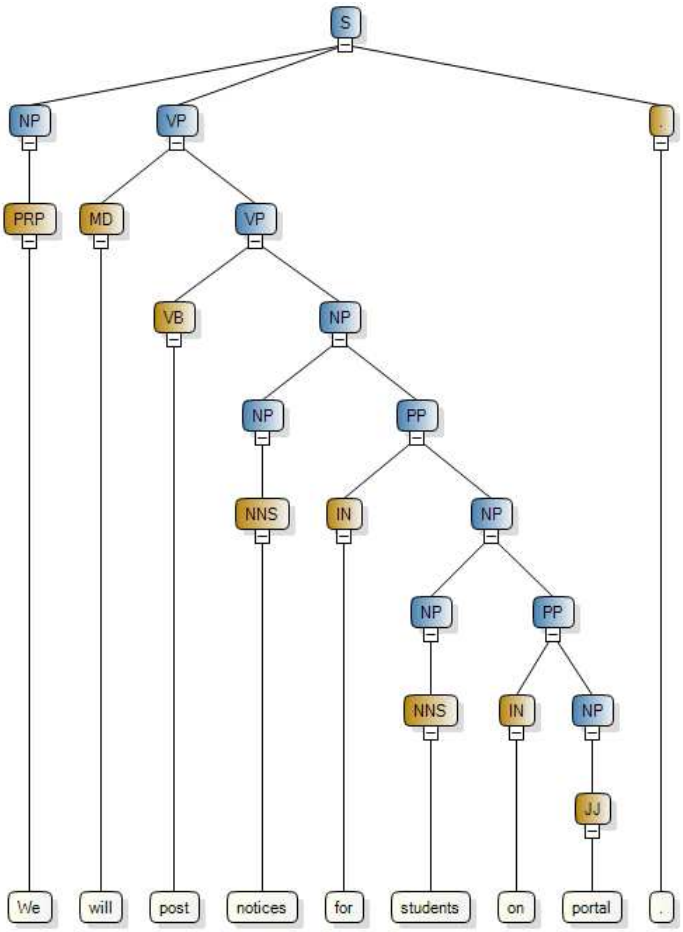


Figure 4.13 An example of a parsing tree.

(drawn by OpenNLP Parser Demo [49])

is parsed and then mapped to functional nodes in requirements ontology on the right. For example, the requirement number three: "We will post notices for students on portal" is parsed to a list of verb and nouns as: {V:post, N:notices}, and is then mapped to a node "Send messages" in requirements ontology of university portal site, as in Fig. 4.14.

Although requirements can be mapped to nodes in ontology simply by string comparison, but to enable more efficient comparison, we used thesaurus to support translation. Because a word may have different synonyms in different domains, so besides using common thesaurus, we added the capability of building domain thesaurus which has a similar structure to requirements ontology model. Each term in thesaurus includes: concept, part of speech, definition, relationships of synonym to other terms, and similarity values between terms, as illustrated in Fig. 4.15. The figure displays a screenshot of the thesaurus editor

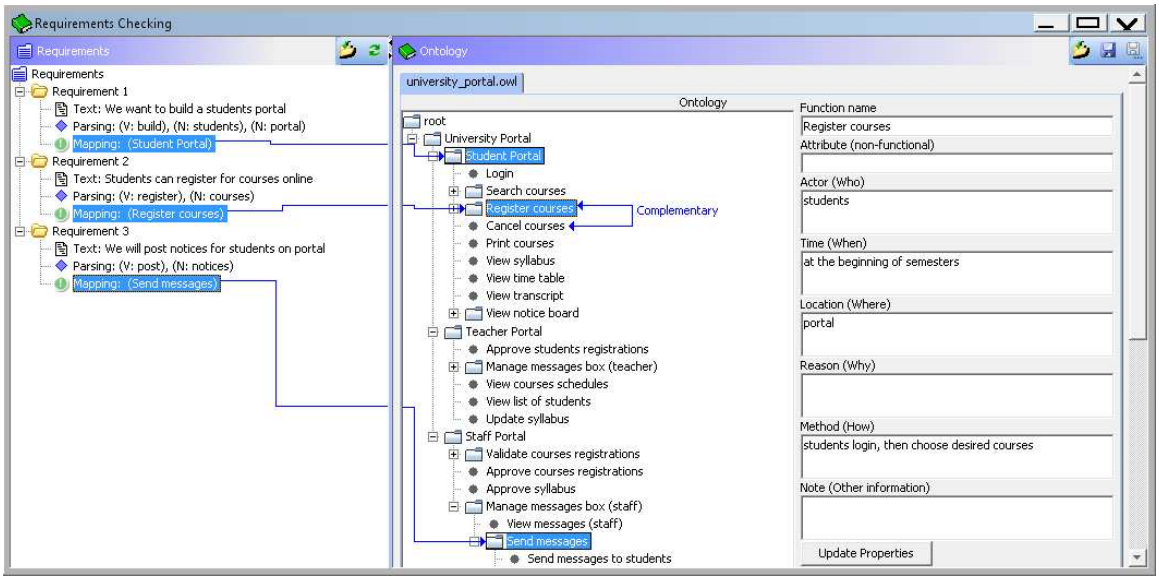


Figure 4.14 Screenshot of mapping software requirements to requirements ontology.

for the web portal domain. The web portal thesaurus includes common terms and relations among them in web portal domain. For example, the word "message" is synonymous with the word "notice", and the word "post" is synonymous with the word "send". The relation of synonym will help the comparison of words in requirements statements with words in ontology. In case of too many matches, analysts can choose which requirements map to which node in ontology (Analysts can drag requirements sentences and drop to ontology nodes).

We used a function to calculate the similarity level between words in requirements with words in ontology as follows.

- Given two lists or words: $S(s_1, s_2, , s_n), T(t_1, t_2, , t_m)$
- The similarity between them is computed as:
  $similar(S,T) = \prod_{i:1-n}(max_{j:1-m}(similar\_value(s_i, t_j))$

A requirement R is mapped to node N if $similar(R,N) > mapping\_threshold$, otherwise it is not mapped. For example, we compare the list R = {post, notice} and the list N = {send,

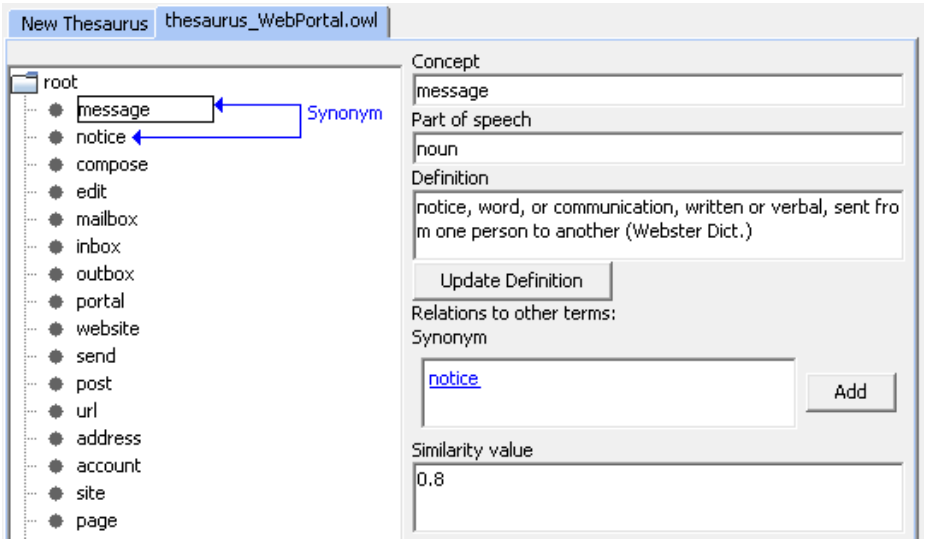


Figure 4.15 Thesaurus used for mapping.

message} as follows.

$$similar\_value(post, send) = 0.7 \qquad \text{(in thesaurus)}$$

$$similar\_value(notice, message) = 0.8 \quad \text{(in thesaurus)}$$

$$similar(post\ notice, send\ message) = 0.7 * 0.8 = 0.56$$

Suppose that $mapping\_threshold$ is set at 0.5, we have the result that $similar(R, N)$ is satisfied for mapping R to N, so the list {V:post, N:notices} is mapped to ontology node "send messages", as in Fig. 4.14.

4.6.3 Detect errors in requirements using ontology and rules

After mapping requirements to nodes in ontology, we use relations and rules to detect errors in the set of mapped nodes. Our requirements ontology is stored in OWL files (Web Ontology Language), but the reasoning power of OWL language is limited [15], so we use Prolog—a general purpose logic languages—for verifying requirements. Rules (1–27) in Sub-sect. 4.5.1 are transformed to Prolog rules, as in Fig. 4.16. We also transform requirements ontology from OWL files to terms and relations in Prolog, e.g., we use correlative keywords between OWL and Prolog as in Table 4.4. We execute following examples of commands to detect errors in SRS:

"shouldbeInSRS(X)."

"maybeInSRS(X)."

Table 4.4 Corresponding keywords in OWL and Prolog.

OWL	Prolog
<owl:Class rdf:ID="X"> <owl:ObjectProperty rdf:ID= "hasComplementary"/> <owl:Class rdf:ID="Y"/>	complement(X, Y).
<owl:Class rdf:ID="X"> <owl:ObjectProperty rdf:ID= "hasSupplementary"/> <owl:Class rdf:ID="Y"/>	supplement(X, Y).
<owl:Class rdf:ID="X"> <rdfs:subClassOf> <owl:Class rdf:ID="Y"'/> </rdfs:subClassOf> <owl:Class rdf:ID="Y"/>	aggregation(X, Y).

"maybeInconsistent(X,Y)."

The above Prolog commands will reason using rules in Fig. 4.16 to find inaccurate or indispensable requirements in SRS. The command *"shouldbeInSRS(X)."* will return the results for mandatory requirements, while the command *"maybeInSRS(X)."* returns the results for optional requirements, and the command *"maybeInconsistent(X,Y)."* returns the results for potentially inconsistent requirements. In order to detect possibly incorrect requirements, we search for requirements that are not mapped to nodes in ontology by using the command *"maybeIncorrect(X,Y)."* in rule (13).

To allow analysts define new rules, a rule definition tool is developed, as shown in Fig. 4.17. The definition of rule has two steps: define it using rule grammar, transform it to Prolog to run in checking environment. In the first step, rules can be described gradually by the rule grammar: each non-terminal symbol can be replaced by a selection from a list of expressions, according to the production rules defined in rule grammar. For example, in Fig. 4.17, in the rule which is defining, the symbol <term> is going to be replaced by the symbol <V-N>. Users definition rules will then be transformed from rule grammar to Prolog by a macro in order to execute in Prolog environment.

To illustrate examples of verification results by the tool, Fig. 4.18 shows results of errors detection in the requirements list of student portal site (from Fig. 4.14). There exists several detected issues in the SRS: optional functions, lack of 4W1H information, possibly wrong 4W1H information. By fixing these problems, we can improve quality of SRS.

```
(1)  inSRS(X) :– mapping(R,X).
(2)  complement(Y,X) :– complement(X,Y).
(3)  exclusion(Y,X) :– exclusion(X,Y).
(4)  redundancy(Y,X) :– redundancy(X,Y).
(5)  shouldbeInSRS(Y) :– complement(X,Y), inSRS(X), not(inSRS(Y)).
(6)  shouldbeInSRS(Y) :– supplement(X,Y), inSRS(X), not(inSRS(Y)).
(7)  maybeInSRS(X) :– supplement(X,Y), inSRS(Y), not(inSRS(X)).
(8)  maybeInSRS(X) :– aggregation(X,Y), inSRS(Y), not(inSRS(X)).
(9)  shouldbeInSRS(Y) :– inheritance(X,Y), inSRS(X), not(inSRS(Y)).
(10) maybeInSRS(X) :– inheritance(X,Y), inSRS(Y), not(inSRS(X)).
(11) maybeInconsistent(R,S) :– exclusion(X,Y), mapping(R,X), mapping(S,Y).
(12) maybeRedundant(R,S) :– redundancy(X,Y), mapping(R,X), mapping(S,Y).
(13) maybeIncorrect(R) :– not(mapping(R,X)).
(14) maybeAmbiguous(R) :– mapping(R,X), mapping(R,Y).
(15) maybeRedundant(R,S) :– mapping(R,X), mapping(S,X).
(16) shouldSimilar(WR,WX) :– mapping(R,X), hasWho(R,WR), hasWho(X,WX).
(17) lackWho(R) :– mapping(R,X), not(hasWho(R,WR)), hasWho(X,WX).
(18) shouldSimilar(WR,WX) :– mapping(R,X), hasWhen(R,WR), hasWhen(X,WX).
(19) lackWhen(R) :– mapping(R,X), not(hasWhen(R,WR)), hasWhen(X,WX).
(20) shouldSimilar(WR,WX) :– mapping(R,X), hasWhere(R,WR), hasWhere(X,WX).
(21) lackWhere(R) :– mapping(R,X), not(hasWhere(R,WR)), hasWhere(X,WX).
(22) shouldSimilar(WR,WX) :– mapping(R,X), hasWhy(R,WR), hasWhy(X,WX).
(23) lackWhy(R) :– mapping(R,X), not(hasWhy(R,WR)), hasWhy(X,WX).
(24) shouldSimilar(WR,WX) :– mapping(R,X), hasHow(R,HR), hasHow(X,HX).
(25) lackHow(R) :– mapping(R,X), not(hasHow(R,HR)), hasHow(X,HX).
(26) shouldSimilar(NR,NX) :– mapping(R,X), hasNFR(R,NR), hasNFR(X,NX).
(27) lackNFR(R) :– mapping(R,X), not(hasNFR(R,NR)), hasNFR(X,NX).
```

Figure 4.16 Reasoning rules in Prolog.

4.6.4 Interpret results, suggest revision of requirements.

From verification results, requirements analysts propose mandatory requirements and optional requirements to customers. Analysts also pose to them questions concerning contradict requirements and redundant requirements. The verification tool can generate questions automatically from verification results. For example, if the Prolog command *"shouldbeInSRS(X)."* gives result of mandatory requirements as "cancel courses", then the tool generate questions such as: "Do you agree to add the function 'cancel courses' to requirements list?" Similarly, if the Prolog command *"redundant(X,Y)."* gives the result as a list of two similar functions such as: "view message detail" and "show message", then the tool generates

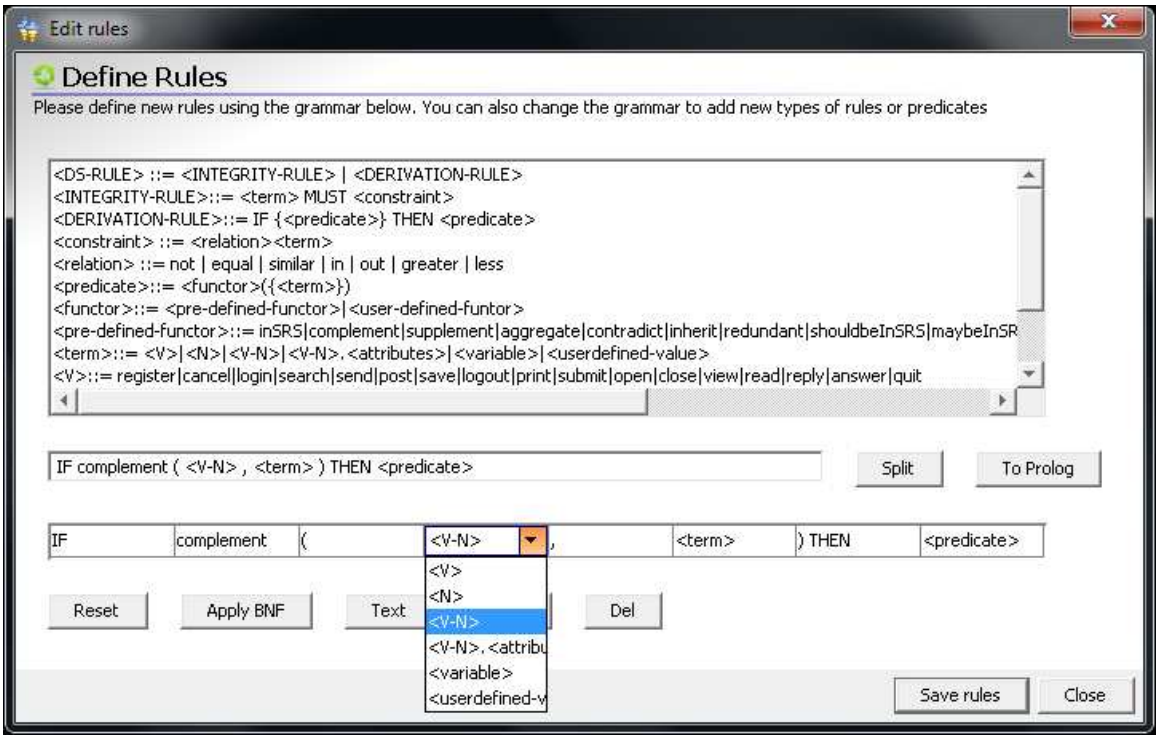


Figure 4.17 Screenshot of rules definition tool.

questions as follows: "The two functions 'view message detail' and 'show message' are redundant. Which one do you want to remove from requirements?" To generate questions for 4W1H attributes, the tool use some templates, e.g., "Please describe steps to perform the function... When to do the function..." Fig. 4.19 shows a list of questions which are generated from results in Fig. 4.18. There are questions about lack of 4W1H information, questions asking for insertion of new functions to SRS, and questions requesting revision of 4W1H information. Requirements engineers can use these questions to discuss with customers how to revise the requirements. After receiving customers' answers, requirements team will revise requirements list, and analysts can proceed verification again, until they find no errors in requirements specification.

4.7 Chapter Summary

This chapter have described ontology-based verification method of software requirements in a simple approach using a guideline and a detail approach using reasoning rules. The difference between using guideline and using reasoning rules is that the former can be

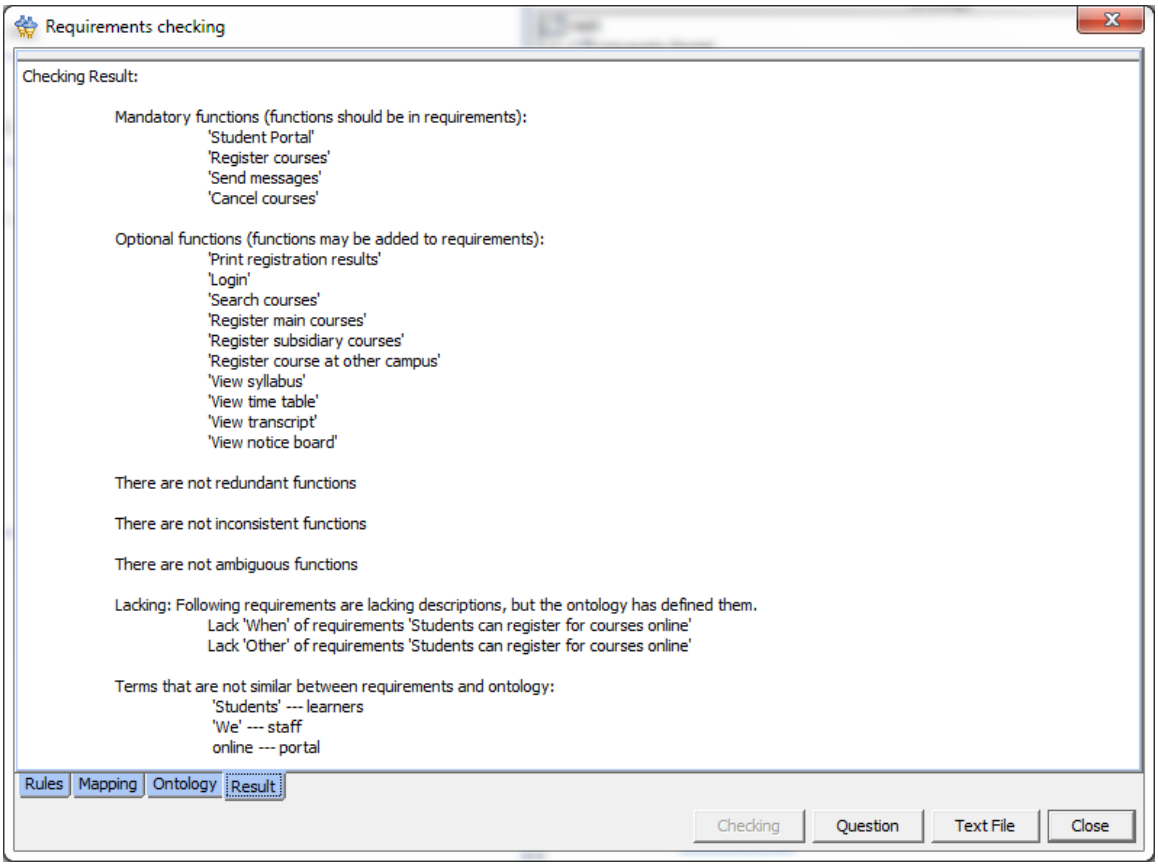


Figure 4.18 Screenshot of showing results of errors detection in software requirements using requirements ontology.

conducted manually while the latter should be used with a supporting tool. The next chapter will evaluate our proposed method through experiments.

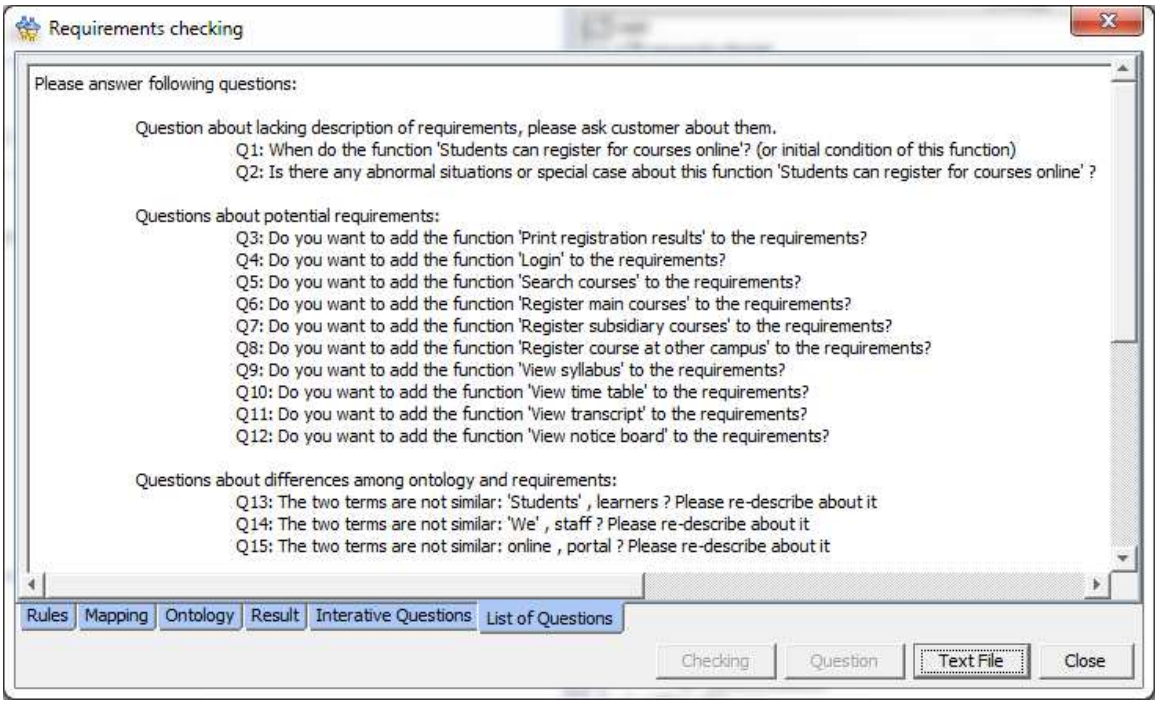


Figure 4.19 Screenshot of generating questions from detected errors.

This page intentionally left blank.

Chapter 5

Evaluation of Ontology-based Verification Method of Software Requirements

This chapter will present several comparative experiments to evaluate the effect of ontology-based method in verification of software requirements. In the first experiment, two groups of subjects checked a same SRS, but one group used ontology-based method and the other did not. In other two experiments, two subjects exchanged working method to obtain a more objective result. Finally, the last two experiments in this chapter had a larger number of participants (15) for purpose of confidence in statistics of results. We also see whether the quantitative target (at the end of Sect. 1.1) has been achieved.

5.1 Experiment 1

In Experiment 1, we assigned two groups of subjects working on a same list of requirements, but one group using our method and the other not. We compare the results of two groups in the number of errors they found, time they used, and the number of functional requirements in the revised list of requirements.

5.1.1 Overview of the experiment

There were four students who acted as requirements analysts. They were provided with initial requirements of a queue management system (QMS) for a city hall in Fig. 5.1. Among the four subjects, two subjects worked freely and two subjects were provided with

ontology and verification tool. Two groups were asked to check the requirements list, find errors and lacking functions, and suggest revision of requirements. Another graduate student who had experience in development of QMS systems took the roles as customer, answering questions and discussing with analysts to revise requirements. The customer was asked to give the same answers to the same questions from both groups of analysts.

1. Citizens get tickets at the reception desk.
2. Citizens wait until their turn.
3. Staffs call next citizen.
4. The speaker informs the citizen's turn.
5. LED displays show the citizen's turn.
6. Citizen goes to the calling counter.

Figure 5.1 Initial requirements of QMS system.

It took us eight hours to build a QMS requirements ontology from existing user-guide documents of similar QMS systems. The QMS requirements ontology included 127 nodes, 54 relations among nodes, and about 190 information of 4W1H. It was provided to the group using the method.

The discussion methods were meeting and email. The groups conducted totally 8 meetings, each lasted roughly 1.5 hour, and sent totally 48 emails of discussion. Subjects S1 and S2 did not use ontology and verification tool, whereas subjects S3 and S4 did use them. To illustrate our verification method, we will explain the initial steps of requirements verification by subject S3 in the next sub-section.

5.1.2 Example of verification of requirements by a subject with method

Subject S3, after receiving the initial requirements in Fig. 5.1, used our tool to parse them into lists of verb and nouns, then he mapped these lists to nodes in QMS ontology. The parsing and mapping results are displayed in Fig. 5.2. After mapping requirements to ontology, subject S3 proceeded checking using our tool. The checking engine found that the node "call next customer" has relation of complement to the node "call any customer" and the node "recall customer", so it suggested to add the two functions to the requirements list. Similarly, the node "display recent tickets" was complement to the node "display calling ticket", so it was also recommended to be added to requirements. In addition, the checking engine detected that the initial requirements number two and six are not mapped

to ontology, so they were not functions of QMS system and recommended to be removed from requirements list.

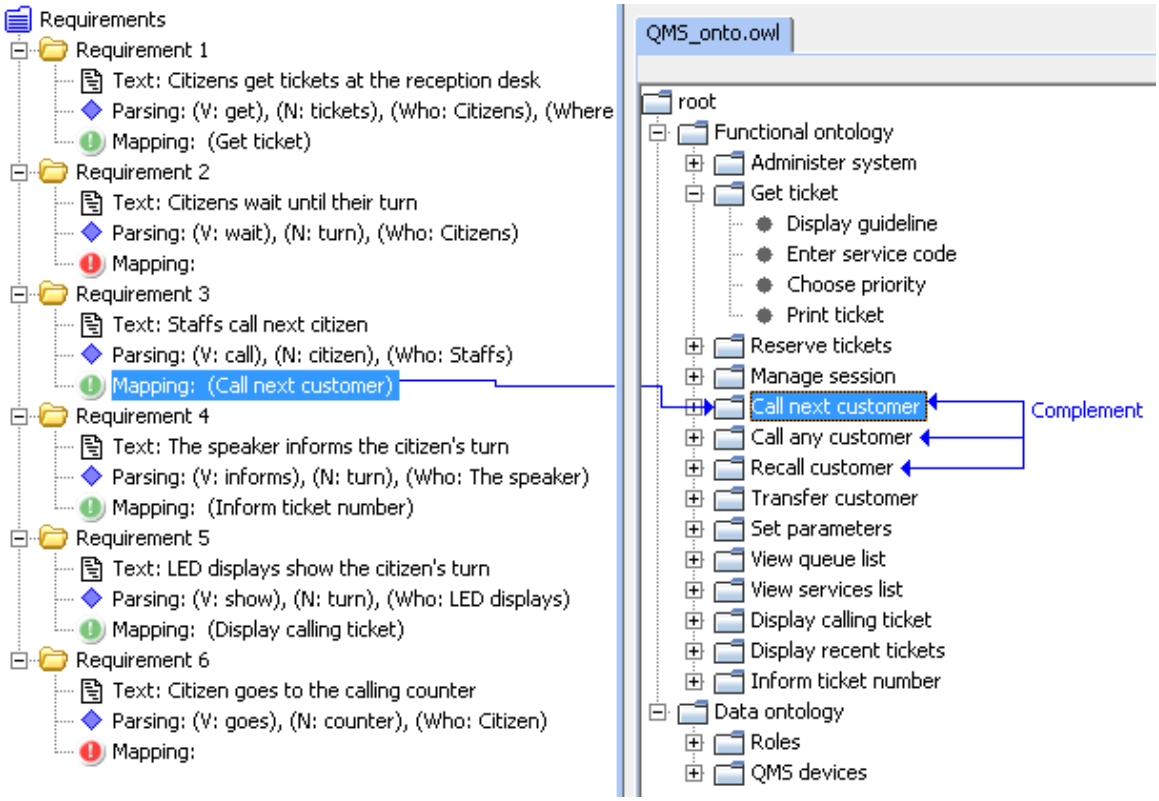


Figure 5.2 Mapping initial requirements to QMS ontology.

After verification in the first round as above, subject S3 found eight requirements, as shown in Fig. 5.3. After revising requirements, he continued with a second round of verification. Finally, after four rounds he could find total 47 functional requirements. The final requirements list is in Appendix A. Following sections will compares used time and results by two groups of subjects and discusses reasons why they had that results, and evaluates the effect of using the verification method in improving quality or requirements specification.

5.1.3 Time used to check requirements

As described in Section 4.5.1, analysts can check SRS in several rounds: after revising SRS in each round, analysts can check the revised SRS again in another round until they do not find errors in SRS anymore. In the experiment, subject S1 conducted requirements

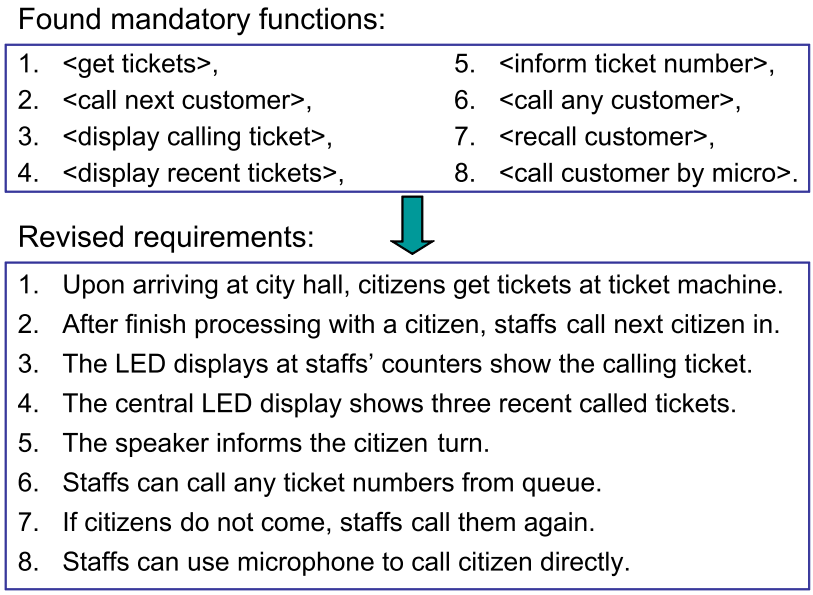


Figure 5.3 A part of requirements list revised by subject S3.

checking in one round, and then he stopped, while subject S2 proceeded checking to the second round. Instead, subjects S3 and S4, after each round, found out more requirements and continued process of requirements checking until the fourth round (The number of requirements that each subject found in each round are shown in Sub-sect. 5.1.6).

We summarize time that each subject used in each round in Table 5.1. In the table, time is separated into two categories: time for checking requirements; time for discussion with customer plus time for revising requirements. For example, in the first round, subject S1 used three hours for checking and four hours for discussion and revision, while subject S3 used only five minutes for checking and three hours for discussion and revision.

Table 5.1 Time used: checking time | discussion and revision time.

Group	Subject	1st round	2nd round	3rd round	4th round	Total
without method	S1	3h | 4h				3h | 4h
	S2	3h | 5h	1h | 2h			4h | 7h
with method	S3	5m | 3h	8m | 5h	10m | 4h	10m | 3h	33m | 15h
	S4	5m | 1h	5m | 1h	5m | 2h	5m | 2h	20m | 6h

m - minutes | h - hour(s)

The group with method used a very short time for checking SRS (5~10 minutes) in

each round because they were supported by verification tool. The tool imported SRS file, automatically parsed requirements sentences, automatically mapped requirements to ontology, then checked SRS by reasoning engine. Instead, the group without method did not have the support of tool and did not have reference to requirements ontology, so they used much longer time (one to several hours) for checking SRS in each round. It suggests that the verification method and tool can help shortening time of checking SRS.

In the group with method, subject S4 used only one or two hours for discussion and revision in each round because he used the support of tool for these activities, but subject S3 did not. Subject S4 used verification tool to generate questions, asked customers directly, then entered answers to the verification tool. The tool then generated a revised requirements list and continued checking. Unlike subject S4, subject S3 discussed with customer via email. Subject S3 did not use the tool to generate lists of revised requirements automatically, but he composed them by himself. That were reasons why subject S3 used more time for discussion and revision than subject S4 did in each round.

In addition, because of the above reasons, subject S4 used only five minutes for checking requirements in each round but subject S3 used more minutes for checking in the second, third, and fourth rounds. After each round, subject S3 imported a new SRS file and parsed it by tool. It took him some minutes to adjust parsing and mapping results. Instead, subject S4 did not do these steps in the second, third, and fourth rounds.

5.1.4 Errors found by two groups of subjects

Table 5.2 lists the number of errors that each subject detected and average time used on each error (including checking time, discussion time, and revision time). In total, the subjects with method detected many more errors than the subjects without method detected. That was the reason totally the subjects with method spent a lot of time for discussion and revision (Table 5.1). However, in average time used on each error, subjects without method spent more time than the subjects with method did. It suggests that our method could shorten not only checking time, but also discussion and revision time for each error in SRS.

Table 5.3 classifies types of errors found by two groups of subjects as: incomplete requirements, inconsistent requirements, redundant requirements, ambiguous requirements, off-topic (incorrect) requirements, and 4W1H errors (lacking 4W1H or wrong description of 4W1H). The group with method outperformed the group without method in number of incomplete requirements. The group without method did not detect errors such as incorrectness, ambiguity, redundancy, but the group with method did.

Table 5.2 The number of errors found by each subject and average time used on each error.

Group	Subject	no. errors found	average time on each error
without method	S1	20	21 minutes
	S2	36	18.3 minutes
with method	S3	128	7.3 minutes
	S4	114	3.3 minutes

Table 5.3 Classification of errors found by two groups.

	without method		with method	
	S1	S2	S3	S4
Incomplete	13	11	63	67
Inconsistent	0	0	0	0
Redundant	0	0	2	0
Ambiguous	0	0	0	1
Off-topic	0	0	2	2
4W1H errors	7	25	61	44

We use two metrics to compare the performance of errors detection:

- Recall: the number of correctly detected errors / the number of errors.
- Precision: the number of correctly detected errors / the number of detected errors.

Table 5.4 summarizes the recall and precision metrics of errors found by two groups. The group with method had higher recall metrics but had lower precision metrics (It is said that when recall tends to increase, precision would be decreased and vice versa [1]). Since recall metrics of results with ontology-based verification method were higher, it suggests that the method helps increasing the number of correctly detected errors in software requirements.

Table 5.4 Recall and precision metrics of errors found in Experiment 1.

Group	Subject	Recall			Precision		
		raw data	%	average	raw data	%	average
without method	S1	18 / 112	16.0%	22.2%	18 / 20	90.0%	89.4%
	S2	32 / 112	28.5%		32 / 36	88.8%	
with method	S3	86 / 112	76.7%	71.8%	86 / 128	67.1%	66.4%
	S4	75 / 112	66.9%		75 / 114	65.7%	

Table 5.5 The number of questions.

	without method		with method	
	S1	S2	S3	S4
Proposals to add requirements	12	11	63	67
Proposals to remove requirements	0	0	4	2
4W1H questions	14	25	61	45

5.1.5 Questions for revising requirements by two groups

Based on errors found in requirements list, subjects suggested revisions of requirements. Subjects S1 and S2 worked freely and prepared questions by themselves, while subjects S3 and S4 used verification tool to generate questions. Table 5.5 lists number of questions that the customer received from two groups.

There were three types of questions received from subjects: proposals to add functional requirements, proposals to remove functional requirements, and questions about descriptions of functional requirements such as 4W1H information. We see that subjects in the group with method generated much more proposal questions to add functional requirements than the other two subjects did. In particular, the group without method did not ask any question to remove functional requirements. This was because they did not find errors such as incorrectness, inconsistency or redundancy in the requirements list. For example, the initial requirement number 2: “Citizens wait until their turn,” and number 6: “Citizen goes to the calling counter” are not functions of QMS system and can be eliminated from requirements list.

Figure 5.4 lists some questions from the group with method, and answers from the

Q5: Do you want to add the function 'Reserve ticket' to the requirements?
A5: Yes, citizens can reserve ticket in advance.
Q6: Do you want to add the function 'Display guideline' to the requirements?
A6: Yes. We want to display a guideline on how citizens can get tickets.
Q16: When does staff perform the function 'Call any citizen'?
A16: Staffs want to be able to call any citizens from queue. For example, citizens who have priority, or citizens who have been skipped by the system due to their being absent at a previous call.
Q17: Please describe the steps to perform the function 'Call any citizen'?
A17: Staff views the list of tickets in queue. Then he/she selects a ticket to call in.
Q25: The type of tickets printer depends on number of customers per day. Normally, how many citizens come to your office per day?
A25: From hundreds to a thousand.

Figure 5.4 A part of questions from group with method and answers.

customer. Questions five and six are proposals to add requirements; questions 16, 17, and 25 are descriptions of requirements. The verification tool found that the functions "Reserve ticket" and "Display guideline" are optional requirements, so it generated questions five and six. It also detected that there were no information about when or how to do the function "Call any citizen" in requirements list, so it generated questions 16 and 17. Question 25 was more difficult to generate: the tool detected the lacking of non-FR about maximum number of citizens related to the function "Get tickets", so it generated questions in a simple format, then analysts revised the questions to ask the customer.

5.1.6 The number of functional requirements found by two groups of subjects

Table 5.6 summarizes the number of functional requirements that each subject found after each round. We see that totally the group with method established many more requirements than the group without method. The reason is that subjects in group with method clarified each function into many sub-functions down to a detailed level, whereas the other students did not. For example, subjects in group without method only described general function about "view report", while the other two subjects divided it to several sub-functions such as: "report on number of customers, report on transaction time." These sub-functions were modeled in QMS ontology, so the group using method could reason out them.

Table 5.6 The number of functional requirements found by two groups of subjects.

	Student	1st round	2nd round	3rd round	4th round
without	S1	17			
method	S2	10	12		
with	S3	8	19	30	47
method	S4	9	19	34	44

Figure 5.5 shows the intersection of functional requirements lists by two groups. That figure only represents some functional requirements shortly in terms of verb and nouns (The complete list of functional requirements are in Appendix A). Six functional requirements that found by all subjects (shown in the left text box of Fig. 5.5) are common functions of QMS system. Therefore, every subject could detect them; even subjects S1 and S2 did not use ontology and verification tool. The 30 functional requirements that the group with method found, but the group without method did not find—in the right text box of Fig. 5.5—are mainly administrative functions and specific knowledge of QMS system. The checking by subjects with method was directed by ontology and tool, so they could infer these functions.

There were nine functional requirements that the group without method found, but the group with method did not. Of the nine functional requirements, three were incorrect and needed to be eliminated from specification. The other six functional were ambiguous, and needed to be revised. The reason that the group with method did not find these functional requirements is that they were not included in the QMS requirements ontology, so the subjects did not reason out them. It suggests inserting these functional requirements to requirements ontology for later use.

5.1.7 Discussion

Subjects S3 and S4 found more errors in requirements list and produced more questions to add or remove functions. Subjects S1 and S2 did not detect errors such as incorrectness, redundancy, but did students S3 and S4. The group using ontology and verification tool found double number of requirements to the group working freely. These results suggests that using our verification method can improve the quality of requirements specification.

Comparing results of students inside each group: the group with method did not have much difference in results between subjects S3 and S4, but the group working freely had

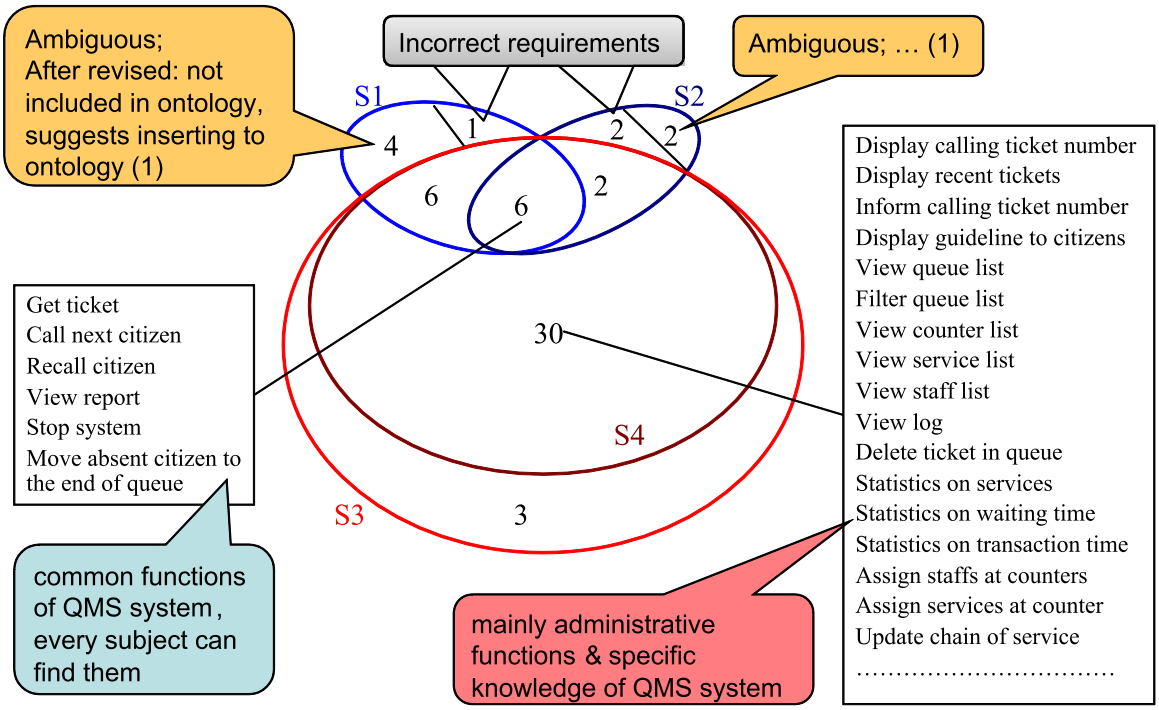


Figure 5.5 Intersection of functional requirements lists by two groups of subjects.

a larger difference in results between subjects S1 and S2. The group using verification method was directed by ontology and tool, so they arrived at similar results. It suggests that adding people to the group with method will be not efficiently, except assignment of each analyst working on a different part of requirements specification.

Though both subjects S3 and S4 used tool and ontology, but subject S3 found more three incomplete requirements than subject S4 did. From checking results, subject S3 revised the questions before asking the customer, but subject S4 kept original questions generated by tool and sent to the customer. Here we do not conclude that revising questions before sending to customers or using original questions generated by tool is better, but we expect to elaborate this issue in future work.

The verification tool supports automatically parsing requirements, mapping to ontology, checking, and generating questions for revising requirements. It suggests that the verification tool which provides semi-automatic verification has potential to improve the checking speed. However, two groups used similar total of time—including time for checking requirements, revising requirements, and discussion with customer. In order to shorten total of time used by the group with method, one way is to reduce time for revising requirements. This issue needs to be evaluated in future research.

5.2 Two Experiments with Exchange of Working Method

The effect of ontology-based requirements verification method might depend on skill and domain knowledge of subjects. To evaluate the method more objectively, we conducted two other experiments: Experiment 2 and Experiment 3. In each experiment, two subjects checked a same SRS, but one subject used requirements ontology and verification tool, and the other subject worked freely. The working methods of two subjects interchanged in the two experiments: the subject who used ontology-based method in Experiment 2 worked freely in Experiment 3, and vice versa. Two subjects in the experiments were software engineers; both had five years' experience in software development.

5.2.1 Experiment 2

In this experiment, two subjects were asked to check a SRS of 40 requirements of an accounting software for a high school. To make the SRS for the experiment, firstly we start with an existing SRS, and then we revise and intentionally put some errors on the SRS. The first subject worked freely and the second subject was provided with ontology and tool. We built requirements ontology of accounting software from manuals of available commercial products, and provided the ontology to the second subject. When receiving checking results from two subjects, a requirements specialist reviewed their results to examine whether each reported problem is adequate to be considered an error. We also use two metrics to compare the results: recall and precision.

Table 5.7 Number of errors found by two subjects in Experiment 2.

	Recall		Precision	
	w/o. method	w. method	w/o. method	w. method
Incomplete	2 / 6	4 / 6	2 / 4	4 / 4
Inconsistent	0 / 1	1 / 1	0 / 0	1 / 1
Redundant	3 / 7	6 / 7	3 / 3	6 / 6
Ambiguous	2 / 7	5 / 7	2 / 2	5 / 5
Off-topic	4 / 7	6 / 7	4 / 6	6 / 9
4W1H errors	1 / 6	5 / 6	1 / 1	5 / 5

Table 5.7 summarizes the checking results from two subjects. Each error type is listed in the table with the metrics of recall and precision. We see that all of the recall metrics of

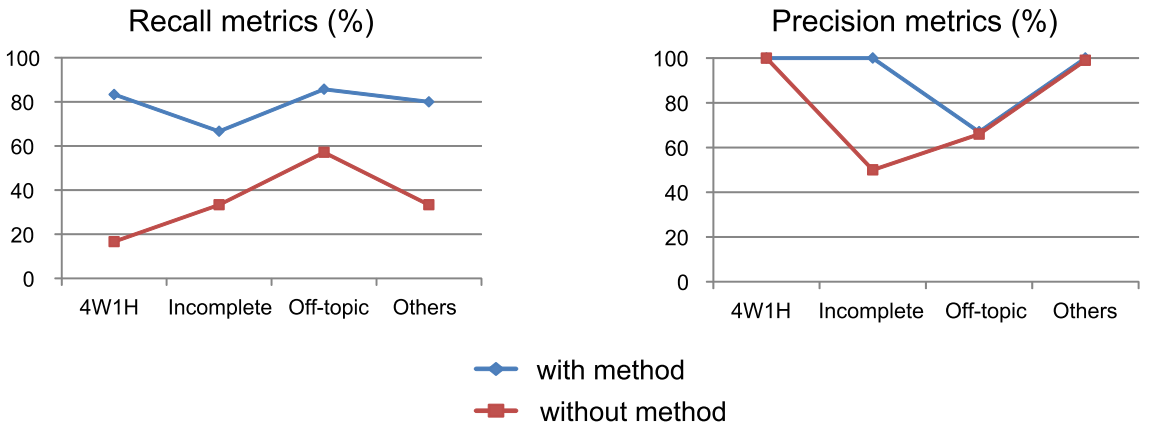


Figure 5.6 Recall and precision metrics of results in Experiment 2.

Table 5.8 Averages of recall and precision metrics in Experiment 2 (higher is better).

	with method	without method
Recall	78.9%	23.3%
Precision	91.7%	78.7%

the results from the subject using checking method are higher than those from the subject not using the method. However, in the precision metrics, results from the subject with method are only higher than results from the subject without method in the incomplete errors. The precision metrics of off-topic errors in the results from the subject with method is only 6/9. This is because the checking method use an assumption about off-topic errors: a requirement which does not map to any function in requirements ontology might be out of scope of the problem domain. The verification tool only detects potential problems which might be errors, and requirements verifiers have to determine if those problems are really errors.

Fig. 5.6 displays the distribution of recall and precision metrics in scale of 100% and Table 5.8 shows the averages of these metrics. From the graph and the table, we see that the recall and precision of results by the subject with method are better than these metrics of results by the subject without method.

5.2.2 Experiment 3

In Experiment 3, two subjects checked a SRS of 33 requirements of a hotel management system. As saying above, the working methods of two subjects interchanged in this experiment: the first subject used ontology and verification tool while the second subject did not. Fig. 5.7 illustrates some examples of requirements of hotel management system and potential errors detected by ontology-based verification method. For example, in the figure, requirement R1 maps to functional node "Manage VIP customers", but the node specifies when attribute, so R1 should describe about when customers become VIP. Requirements R2 and R3 map to a same function "Reserve room", and they express similar meanings, so R1 and R4 are redundant. Requirement R4 does not map to any node in requirements ontology, so R4 might be not a requirement of hotel management system (off-topic error). Requirement R5 maps to two separate nodes, so R5 might be ambiguous.

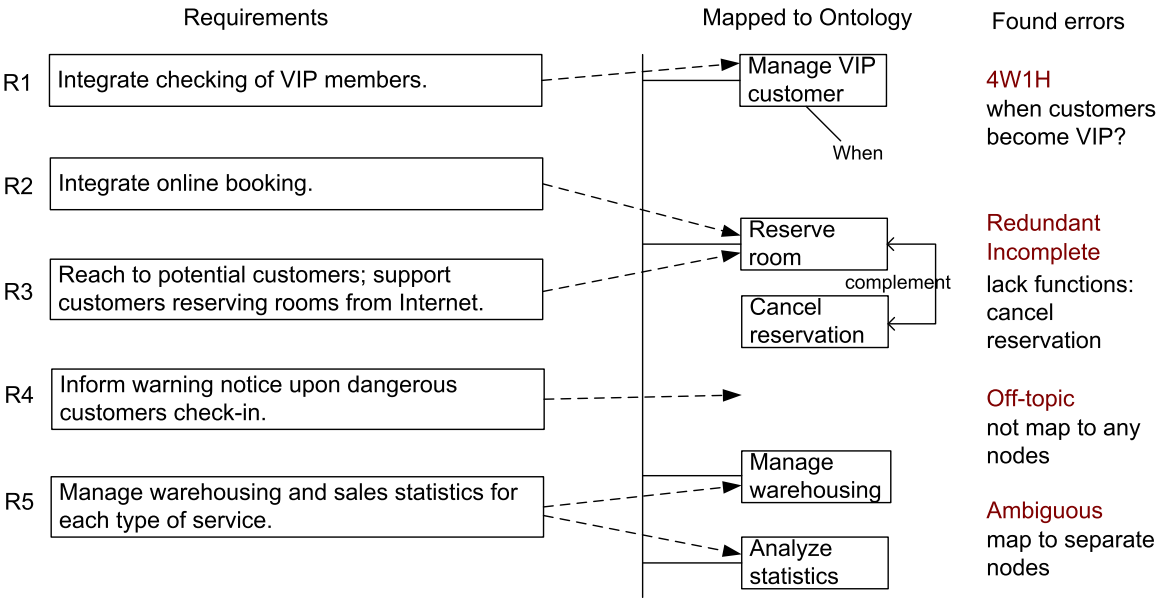


Figure 5.7 Examples of errors in some requirements of hotel management system.

Table 5.9 summarizes the metrics of recall and precision of results from two subjects. In major types of errors, the results from subject with method have more correctly detected errors than the results from the subject without method. However, in the precision metrics of redundant errors and ambiguous errors, the results from the subject with method have lower values than those from the subject without method. The reasons are that the method uses assumptions about redundant and ambiguous errors: two requirements mapping to a same function in ontology might have a similar meaning so they might be redundant;

a requirement mapping to two different nodes in ontology might have two meanings so it might be ambiguous. The improvement of these assumptions is left as topics in future research.

Table 5.9 Number of errors found by two subjects in Experiment 3.

	Recall		Precision	
	w/o. method	w. method	w/o. method	w. method
Incomplete	3 / 9	7 / 9	3 / 4	7 / 7
Inconsistent	0 / 1	0 / 1	0 / 0	0 / 0
Redundant	2 / 3	2 / 3	2 / 3	2 / 4
Ambiguous	2 / 6	4 / 6	2 / 2	4 / 5
Off-topic	0 / 1	1 / 1	0 / 2	1 / 2
4W1H errors	2 / 10	8 / 10	2 / 2	8 / 8

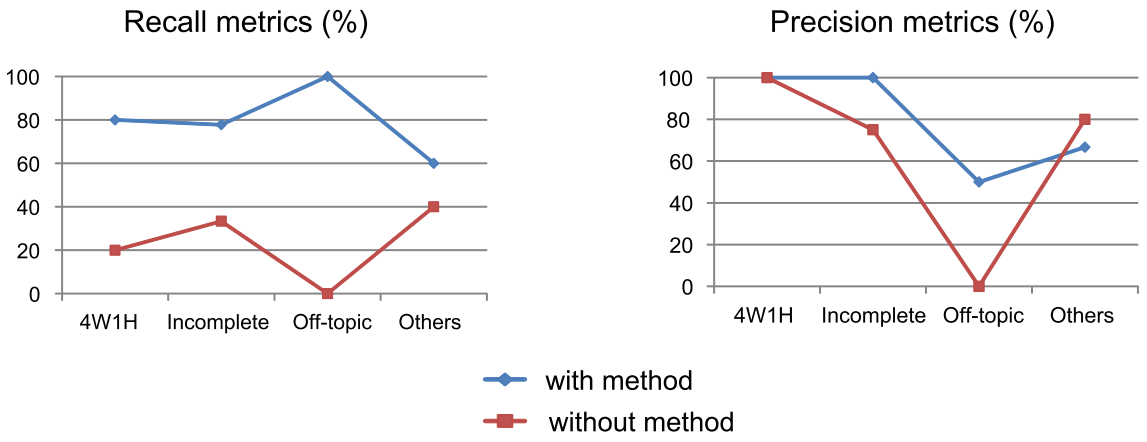


Figure 5.8 Recall and precision metrics of results in Experiment 3.

Fig. 5.8 displays the distribution of recall and precision metrics in scale of 100% and Table 5.10 shows the averages of these metrics. From the graph and the table, we see that the recall and precision of result by the subject with method are better than these metrics of results by the subject without method.

In summary, two experiments' results suggest that our verification method has positive effect on detection of errors in software requirements.

Table 5.10 Averages of recall and precision metrics in Experiment 3 (higher is better).

	with method	without method
Recall	79.4%	35.1%
Precision	79.1%	63.7%

5.3 Two Experiments with More Participants

There are only two subjects participates in Experiment 2 and Experiment 3. In order to yield more objective evaluation of our ontology-based method, we conducted two other experiments with participation of a larger number of subjects. The subjects in the experiments were 15 graduate students. All the students study information science and have been trained with errors types in requirements specification and introduced with ontology-based method. Fifteen subjects were divided into two groups: one group had seven people (group S), the other group had eight people (group T). In each experiment, two groups were asked to check the same SRS and submit results, but one group used ontology and the other group worked freely. The working method of two groups exchanged in the second experiment: the group using ontology in the first experiment will work freely in the second experiment and vice versa.

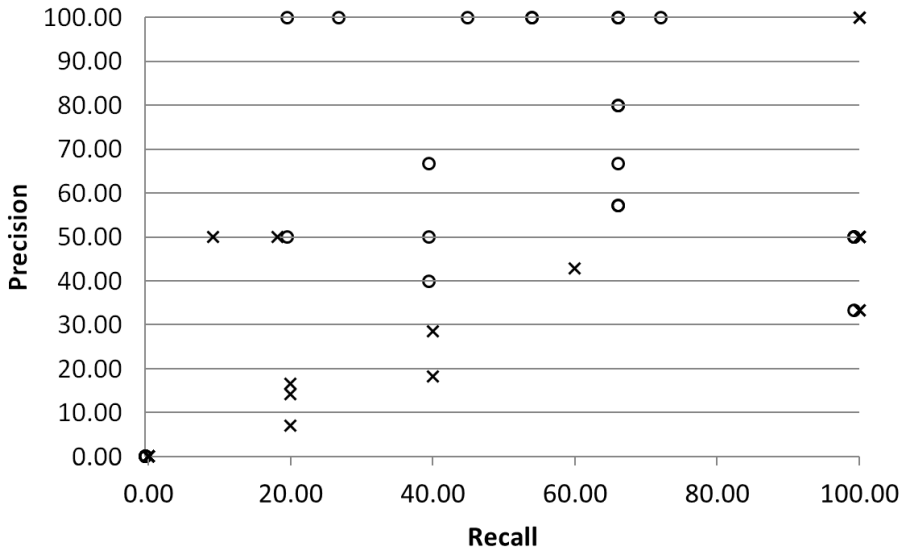
5.3.1 Experiment 4

Overview of the experiment

In Experiment 4, two groups of subjects checked a SRS for a software "manage working schedule" of a company. The SRS includes 15 requirements and is listed in Fig. B1 of Appendix B. Subjects were allowed 40 minutes to check the SRS. In this experiment, the group S was provided with requirements ontology of managing schedule (listed in Table BI of Appendix B). When receiving checking results from subjects, a requirements specialist reviewed their results to examine whether each reported error was correctly detected error. The list of correctly detected errors is displayed in Fig. B2 of Appendix B.

Results and analysis of results

Table 5.11 summarizes the number of errors the two groups of subjects detected using two metrics: recall and precision. We denote subjects in the first group as S1–S7 and subjects in the second group as T1–T8. The detected errors are classified into four types: 4W1H errors, incomplete (incomp.) errors, off-topic errors, and other errors (inconsistency, redundancy, ambiguity). In Table 5.11, the recall and precision metrics of the incomplete errors and 4W1H errors by the group using ontology were higher than those by the group not using ontology. Because subjects in the group working freely did not use ontology, so they did not have knowledge in application domain to detected incomplete errors and 4W1H errors. However, in the results of off-topic errors and other errors, the results by the group using ontology were not better than results by group working freely. The reasons are that the ontology-based method only detects potential problems in SRS, and human will check finally whether they are errors. If the subjects made wrong mappings requirements to ontology or made wrong evaluation of potential errors, the results quality would be decreased.



o: with ontology; x: without ontology

Figure 5.9 Recall and precision metrics of results in Experiment 4.

To illustrate the recall and precision metrics in Table 5.11 more visually, we calculate these metrics values in 100% scale and show in Fig. 5.9. The figure displays that in major cases, the precision metrics and recall metrics of the results by group using ontology were

Table 5.11 Number of errors detected in Experiment 4.

Group	Subject	Precision Metrics				Recall Metrics			
		4W1H	Incomp.	Off-topic	Others	4W1H	Incomp.	Off-topic	Others
with ontology	S1	4 / 5	0 / 3	1 / 2	0 / 4	4 / 6	0 / 11	1 / 1	0 / 5
	S2	4 / 6	6 / 6	0 / 1	0 / 1	4 / 6	6 / 11	0 / 1	0 / 5
	S3	4 / 4	6 / 6	1 / 2	2 / 5	4 / 6	6 / 11	1 / 1	2 / 5
	S4	4 / 5	5 / 5	1 / 2	1 / 1	4 / 6	5 / 11	1 / 1	1 / 5
	S5	4 / 7	8 / 8	1 / 3	1 / 2	4 / 6	8 / 11	1 / 1	1 / 5
	S6	4 / 4	3 / 3	0 / 1	2 / 3	4 / 6	3 / 11	0 / 1	2 / 5
	S7	4 / 7	0 / 1	0 / 0	2 / 4	4 / 6	0 / 11	0 / 1	2 / 5
without ontology	T1	0 / 0	0 / 4	1 / 2	2 / 11	0 / 6	0 / 11	1 / 1	2 / 5
	T2	0 / 0	0 / 0	1 / 3	0 / 4	0 / 6	0 / 11	1 / 1	0 / 5
	T3	0 / 0	2 / 4	1 / 2	0 / 13	0 / 6	2 / 11	1 / 1	0 / 5
	T4	0 / 0	1 / 2	1 / 3	1 / 14	0 / 6	1 / 11	1 / 1	1 / 5
	T5	0 / 0	0 / 0	1 / 2	1 / 7	0 / 6	0 / 11	1 / 1	1 / 5
	T6	0 / 0	0 / 0	1 / 2	3 / 7	0 / 6	0 / 11	1 / 1	3 / 5
	T7	0 / 0	0 / 0	1 / 1	2 / 7	0 / 6	0 / 11	1 / 1	2 / 5
	T8	0 / 0	0 / 0	1 / 1	1 / 6	0 / 6	0 / 11	1 / 1	1 / 5

Precision: the number of correctly detected errors / the number of detected errors.

Recall: the number of correctly detected errors / the number of errors.

Table 5.12 Averages of recall and precision metrics in Experiment 4 (higher is better).

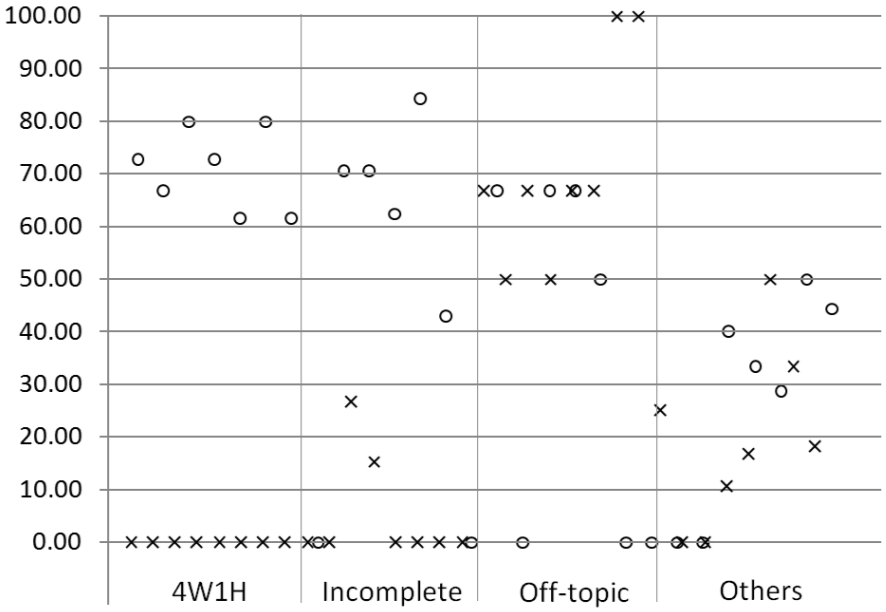
	with ontology	without ontology
Recall	45.8%	32.1%
Precision	54.7%	21.7%

higher than metrics of the results by group working freely. The averages of the recall metrics and precision metrics of the results by the group using ontology, as shown in Table 5.12, were 45.8% and 54.7%, respectively, higher than those average metrics values by the group not using ontology which were 32.1% and 21.7%, respectively.

Using precision metrics and recall metrics separately is not always appropriate to show performance of an errors retrieval method. When recall tends to increases, precision would be decreased, and vice versa [1]. Harmonic mean is another metrics that combined both

recall and precision [18]. Harmonic mean is computed as in Equation 5.1. It aslo have value from 0 to 1; the bigger it is, the better the method is.

$$HarmonicMean = \frac{2}{\frac{1}{Recall} + \frac{1}{Precision}} \tag{5.1}$$



o: with ontology; x: without ontology

Figure 5.10 Harmonic mean metrics in Experiment 4.

Fig. 5.10 displays the distribution of harmonic mean metrics in scale of 100%, and Table 5.13 shows the averages of this metrics in the experiment. The averages of harmonic mean of 4W1H errors, incomplete errors, and other errors of results by the group with ontology were higher than those by the group without ontology. However, the average of harmonic mean of off-topic errors of results by the group working freely was higher. The reason was that some subjects in the group using ontology had made wrong mapping or evaluation of potential off-topic errors, and it downgraded the quality of results. On all type of errors, the group using ontology has average of harmonic mean of 45.4% which was higher than that 23.8% by the group working freely. It shows that in major cases in this experiment, using of ontology had positive effect in checking requirements specification.

Table 5.13 Averages of harmonic mean metrics in Experiment 4 (higher is better).

	with ontology	without ontology
4W1H	70.7%	0%
Incomplete	47.3%	5.3%
Off-topic	35.7%	70.8%
Others	28.1%	19.2%
All	45.4%	23.8%

5.3.2 Experiment 5

Overview of the experiment

In the second experiment, two groups of subjects checked SRS for a "test editor" software which included 13 requirements (listed in Fig. C1 of Appendix C). Two groups exchanged their working method: the group using ontology in the first experiment (group S) worked freely in the second experiment, and the group working freely in the first experiment (group T) used ontology in the second experiment. Subjects were also allowed 40 minutes to check the SRS. The group T was provided with requirements ontology for test editor software (shown in Table CI of Appendix C).

Results and analysis of results

Table 5.14 summarizes the number of errors that the two groups of subjects detected in two metrics: recall and precision. The table shows that in most cases the results by group using ontology had more correctly detected errors than the results by group working freely (Correctly detected errors in this experiment are listed in Fig. C2 of Appendix C).

To illustrates the results more visually, Fig. 5.11 shows the distribution of the values of recall and precision metrics in 100% scale, and Table 5.15 lists the averages of these metrics. The figure and the table display that in major cases, precision metrics and recall metrics of the results by group using ontology were higher than metrics of the results by group working freely. The averages of the recall metrics and precision metrics of the results by group using ontology, as shown in Table 5.15, were 39.4% and 76.1%, respectively, higher than those average metrics values by group not using ontology which were 13.7% and 22.3%, respectively.

Fig. 5.12 displays the distribution of harmonic mean metrics in scale of 100%, and

Table 5.14 Number of errors detected in Experiment 5.

Group	Subject	Precision Metrics				Recall Metrics			
		4W1H	Incomp.	Off-topic	Others	4W1H	Incomp.	Off-topic	Others
without ontology	S1	0 / 0	0 / 0	1 / 1	1 / 5	0 / 6	0 / 16	1 / 2	1 / 7
	S2	0 / 0	0 / 0	0 / 1	3 / 6	0 / 6	0 / 16	0 / 2	3 / 7
	S3	0 / 0	1 / 6	0 / 0	1 / 4	0 / 6	1 / 16	0 / 2	1 / 7
	S4	0 / 0	1 / 3	0 / 0	4 / 4	0 / 6	1 / 16	0 / 2	4 / 7
	S5	0 / 0	0 / 0	1 / 2	2 / 4	0 / 6	0 / 16	1 / 2	2 / 7
	S6	0 / 0	0 / 0	1 / 1	2 / 6	0 / 6	0 / 16	1 / 2	2 / 7
	S7	0 / 0	1 / 3	0 / 1	2 / 7	0 / 6	1 / 16	0 / 2	2 / 7
with ontology	T1	1 / 1	3 / 3	1 / 3	2 / 7	1 / 6	3 / 16	1 / 2	2 / 7
	T2	1 / 1	5 / 5	1 / 1	1 / 2	1 / 6	5 / 16	1 / 2	1 / 7
	T3	5 / 5	3 / 3	2 / 3	2 / 2	5 / 6	3 / 16	2 / 2	2 / 7
	T4	0 / 0	2 / 2	1 / 2	2 / 3	0 / 6	2 / 16	1 / 2	2 / 7
	T5	3 / 4	2 / 2	2 / 2	4 / 6	3 / 6	2 / 16	2 / 2	4 / 7
	T6	3 / 4	2 / 2	2 / 3	3 / 5	3 / 6	2 / 16	2 / 2	3 / 7
	T7	5 / 5	2 / 2	0 / 1	1 / 5	5 / 6	2 / 16	0 / 2	1 / 7
	T8	1 / 1	3 / 3	2 / 2	3 / 4	1 / 6	3 / 16	2 / 2	3 / 7

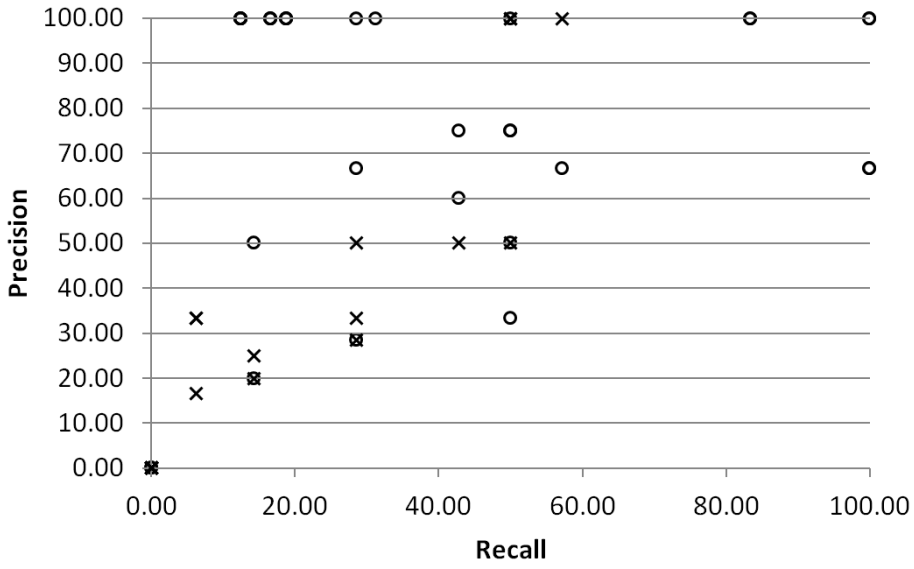
Precision: the number of correctly detected errors / the number of detected errors.

Recall: the number of correctly detected errors / the number of errors.

Table 5.15 Averages of recall and precision metrics in Experiment 5 (higher is better).

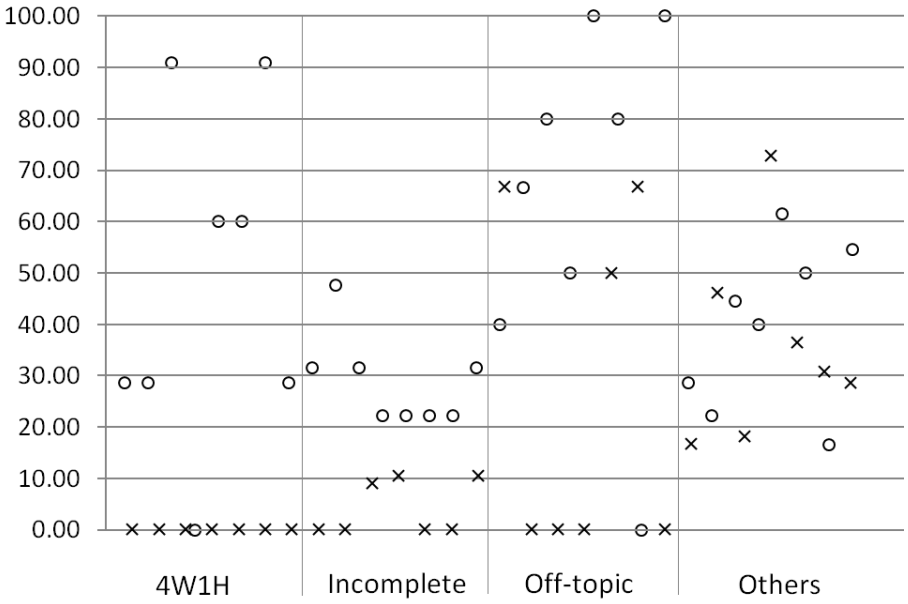
	with ontology	without ontology
Recall	39.4%	13.7%
Precision	76.1%	22.3%

Table 5.16 shows the averages of this metrics. The averages of harmonic mean of 4W1H errors, incomplete errors, off-topic errors and other errors of results by group with ontology were higher than those by group without ontology. On all type of errors, the group using ontology had average of harmonic mean as 45.4% which was higher than that 16.5% by the group working freely. It shows that in major cases in the second experiment, using ontology had positive effect in checking requirements specification.



o: with ontology; x: without ontology

Figure 5.11 Recall and precision metrics of results in Experiment 5.



o: with ontology; x: without ontology

Figure 5.12 Harmonic mean metrics in Experiment 5.

Table 5.16 Averages of harmonic mean metrics in Experiment 5 (higher is better).

	with ontology	without ontology
4W1H	48.4%	0%
Incomplete	28.9%	4.3%
Off-topic	64.6%	26.2%
Others	39.8%	35.6%
All	45.4%	16.5%

5.3.3 Discussion

In major results of both experiments, the numbers of correctly detected errors by the group using ontology were higher than those by the group working freely. On average, the recall, precision, and harmonic mean metrics of the results by group using ontology were higher. It suggests that using ontology-based method can help improve the percentage of correctly detected errors in verification of software requirements.

However, the performance of detection of off-topic errors and other errors (redundancy, inconsistency, ambiguity) by groups with method were not high. It is easy and effective to detect incomplete errors and 4W1H errors by ontology-based verification method. However, it is not easy to detect other types of errors using ontology. The ontology-based verification method will provide suggestions of potential errors, and human will evaluate finally whether these problems are really errors. The method uses assumptions about redundant and ambiguous errors (two requirements mapping to a same function in ontology might have a similar meaning so they might be redundant; a requirement mapping to two different nodes in ontology might have two meanings so it might be ambiguous). The improvement of these assumptions is left as a topic in future research.

It is also not easy to detect off-topic errors. The results of off-topic errors by group using ontology were better in Experiment 5, but the results of the group working freely were better in Experiment 4. The ontology-based verification method detects off-topic errors by finding requirements that are not mapped to any node in requirements ontology. Because these requirements do not correspond to functions in the domain, they might be out of scope of the problem domain. If the subjects made wrong mapping requirements to ontology or wrong evaluation of potential off-topic errors, the quality of results would be decreased.

Harmonic mean is considered as an indication of quality of detected errors. The higher

Table 5.17 Improvement of harmonic mean (%) by using ontology comparing to not using ontology.

Subject	Difference (harmonic mean)				Sign (+, –, .)			
	4W1H	Incomp.	Off-topic	Others	4W1H	Incomp.	Off-topic	Others
S1	72.7	0.0	0.0	-16.7	+	.	.	–
S2	66.7	70.6	0.0	-46.2	+	+	.	–
S3	80.0	61.5	66.7	21.8	+	+	+	+
S4	72.7	52.0	66.7	-39.4	+	+	+	–
S5	61.5	84.2	0.0	-7.8	+	+	.	–
S6	80.0	42.9	-66.7	19.2	+	+	–	+
S7	61.5	-10.5	0.0	15.9	+	–	.	+
T1	28.6	31.6	-26.7	3.6	+	+	–	+
T2	28.6	47.6	16.7	22.2	+	+	+	+
T3	90.9	4.9	13.3	44.4	+	+	+	+
T4	0.0	6.8	0.0	29.5	.	+	.	+
T5	60.0	22.2	33.3	44.9	+	+	+	+
T6	60.0	22.2	13.3	0.0	+	+	+	.
T7	90.9	22.2	-100.0	-16.7	+	+	–	–
T8	28.6	31.6	0.0	36.4	+	+	.	+

Difference: harmonic mean (with ontology) – harmonic mean (without ontology)

Sign: (+) if Difference > 0 ; (–) if Difference < 0 ; (.) if Difference = 0

Total: 42 signs (+) ; 9 signs (–) ; 9 signs (.)

the harmonic mean is, the better the quality is. We subtract the harmonic mean metrics when using ontology to the harmonic mean metrics when working freely by all subjects and display the difference values in Table 5.17. The signs of the difference values are also shown in the right side of Table 5.17: sign (+) if the difference > 0; sign (–) if the difference < 0; sign (.) if the difference = 0. In total, there are 42 signs (+), 9 signs (–), and 9 signs (.). It means that by using ontology, the harmonic mean metrics are increased in 42 cases and decreased in 9 cases. Of the 9 cases of decreasing the harmonic mean, the major are in off-topic errors and other errors columns. However, in sign columns for off-topic and other errors in Table 5.17, the numbers of sign (+) are still bigger than the numbers of sign (–): six signs (+) and three signs (–) in off-topic column; nine signs (+) and five signs (–) in

other errors column. The signs of differences of harmonic mean between using ontology and working freely show that using ontology-based method increases the harmonic mean in major cases.

In these two experiments, subjects with method used ontology but verified requirements by hand. They mapped requirements to ontology manually and reasoned errors manually, so they might make wrong mapping requirements to functions in ontology, might make mistakes in reasoning errors. The reasoning for errors sometimes requires recursive reasoning which is difficult to do manually. Those are the reasons that the recall metrics in these two experiments were lower than recall metrics in the previous three experiments. Nevertheless, subjects in the previous three experiments were supported by verification tool which allows automatically parsing of requirements, automatically mapping requirements to ontology using thesaurus, and reasoning for errors recursively.

To confirm the above reasons, we used verification tool to check SRSs in Experiment 4 and Experiment 5 again. We retrieved all reported problems in the SRSs by the verification tool. We considered these problems as potential errors. These potential errors were then evaluated by the same list of correctly detected errors in experiments. Recall and precision metrics are computed and displayed in Table 5.18. The table suggests that if we used verification tool and utilized all reported problems by the tool, the recall metrics could be much higher than verification by hand.

Table 5.18 Recall and precision metrics of results by applying verification tool on SRSs in Experiment 4 and Experiment 5.

	verify SRS in Exp. 4 by tool	verify SRS in Exp. 5 by tool
Recall	91.3%	87.1%
Precision	70.0%	67.5%

In summary, the above analysis of results shows that ontology-based verification method has positive effect in Experiments 4 and Experiment 5. On average our method helps increase the number of correctly detected errors in software requirements. Based on errors found, requirements engineers will discuss with stakeholders to revise and improve the quality software requirements.

5.4 Limitations and Threats to Validity of the Evaluation Experiments

This section will discuss several issues of above evaluation experiments and possible solutions to them in future research.

5.4.1 Limitations

The evaluation by above five experiments still has some limitations as follows.

1. Recall metrics were not high. Table 5.19 summarizes recall metrics of results by several subjects in applying ontology-based verification method and verification tool. The table shows that students yielded lower recall metrics value than software engineers, and both results of them still had less than 80% of total errors in SRSs (the quantitative target at the end of Sect. 1.1).

   Table 5.19 Recall metrics of results by different subjects with method.

	Recall	Subjects
Experiment 1	71.8%	students
Experiment 2	78.9%	software engineer
Experiment 3	79.4%	software engineer
Experiment 4	91.3%	tool (by ourselves)
Experiment 5	87.1%	tool (by ourselves)

   Solution: By improving some aspects: the correctness and completeness of requirements ontology, the skill of subjects in using ontology-based method, we expect that the quantitative target could be attained .

2. In some experiments, the capability of the ontology-based method in detection of redundancy, ambiguity, and off-topic errors was not high. Reasons included incompleteness of the requirements ontologies and imperfectness of the pre-defined rules.

   Solution: By improvement of the ontologies and the reasoning rules, the capability of detection of such errors would be increased. Through errors detected outside requirements ontology, we can add lack information to requirements ontology for later usage.

3. The experiments focused on evaluation of errors detection capability but did not evaluate rules definition capability. In the experiments, subjects have not been asked to define new rules.

   Solution: Evaluation of rules definition should be studied in the future.

5.4.2 Threats to validity

There are some threats to validity of the above comparative experiments as follows.

1. Most of experiments used recall and precision metrics to evaluate results. These metrics might be not appropriate in some cases, because we do not know there are how many total of errors [1]. However, we have to accept a fact that "Testing shows the presence, not the absence of bugs" (Dijkstra) [7]. We evaluated the experiments results with the correct results which we can know.

   Solution: Use expert opinions and repeated verification of SRS to learn how many total of errors exist in SRS.

2. How is about the scalability of our method? In the experiments, size of SRS and ontology was small (less than 50 requirements and 200 ontology nodes). In real software projects, the number of requirements and ontology nodes would be huge.

   Solution: With large SRS and ontology, we can use verification tool to input large SRS/ ontology/ rules files and do verification automatically.

3. Experiments compared results by ontology-based method with results by working freely, but experiments have not compared results by our method with results by other related researches.

   Solution: Conduct experiments to compare our method with other related works.

5.5 Chapter Summary

Experiments' results suggest that our ontology-based verification method has positive effect on detection of errors in elicited requirements or requirements specification. The results by subjects with method were often better than results by subjects without method. Averagely, ontology-based verification method in experiments helps increasing recall metrics more than 10%. That increment of recall metrics is increment of performance by each subject using ontology-based method compared to himself/herself not using the method.

In the above experiments, the working method of groups without ontology was similar to inspections and reviews (a common method in requirements quality assurance). Therefore, at some level, experiments compared our ontology-based verification method with a likely common method in requirements quality assurance.

The efficiency of using ontology-based verification method depends on the completeness and correctness of domain ontology. Therefore, after built, requirements ontology needs to be checked and improved frequently. The verification of the correctness of requirements ontology will be covered in the subsequent chapters.

This page intentionally left blank.

Chapter 6

Rule-based Verification Method of Requirements Ontology

On using ontology to support requirements engineering, quality of elicited requirements and requirements specification depends on quality of ontology. Therefore, after construction, ontology needs to be checked for correctness. Though there are many researches about ontology-based requirements elicitation (Sect. 2.2), there is no research about verifying quality of requirements ontologies (ontologies which supports requirements engineering). In this chapter, we present a verification method of the correctness of requirements ontology in order to improve the quality of the ontology.

Some researches provided methods to detect errors in OWL ontologies [21, 60]. In our opinion, requirements ontology should be described in OWL format—a common knowledge description language. Therefore, requirements ontology can be verified by methods which support reasoning with OWL such as in [21, 60]. In addition, some parts of our method can also be implemented using existing tools [37, 41, 65, 66]. However, we do not focus on implementation ways; instead, our research focuses on support requirements engineers to improve the quality of requirements ontology.

In concern for improvement of quality of requirements ontology, Huy and Ohnishi proposed a verification method of the correctness of requirements ontology using three types of rules [16]. A rule language and prototype system in Japanese was provided, but evaluation was not done. In this thesis, we extend Huy and Ohnishi's work [16] with development of verification tool based on English (this chapter) and providing evaluation with the tool (next chapter).

6.1 A Verification Method of Requirements Ontology

Requirements ontology is a representation of domain knowledge about requirements, so requirements ontology should be consistent with domain rules. Domain rules specify constraints and conditions that requirements belonging to each domain should follow. The increasing size of requirements ontology makes it difficult to guarantee the correctness of it (For example, in a team where each person develops a part of ontology, and then they merge ontology parts together). A verification method of requirements ontology and a supporting tool are a solution for this problem.

In this section, we propose a method of verification of the completeness and the consistency of a requirements ontology by checking the consistency between the requirements ontology and a rule description that specify the correctness of the ontology. Firstly, we describe our rule language to specify the correctness of a requirements ontology.

6.1.1 Rules description language

Requirements ontology consists of: functional requirements, attributes of functional requirements, and relations among functional requirements. Therefore, there are several types of errors that might occur with requirements ontology: (1) lack of functional requirements, (2) lack or wrong descriptions of attributes, (3) lack or wrong relations. To verify the correctness of requirements ontology, we provide three types of rules: attributes rules, relations rules, and inference rules. The first type of rule statement is to specify the correctness of attributes of functional requirements in an ontology. The second type is to specify the correctness of relations between functional requirements. The last type is to specify inference of relations between functional requirements.

The first type of rules (attribute rules) specifies the correctness of attributes of functional requirements. The grammar of the first type of rules is shown below. Labels of bold fonts correspond to reserved words, while labels of italic fonts correspond to identifiers. "[A|B|C]" means selection of items.

Attribute-name **[should| should not] be [where| who| why| when| how] information of** *function*

For example, a rule description, "Students should be who information of register courses" (#1) (examples of rules are shown in Table 6.1), specifies agent of the functional requirements of the registration of courses. With this type of rules we can detect both wrong attributes and lack of attributes of functional requirements in an ontology.

Table 6.1 Examples of rules.

R#	Rule	Rule type
#1	Students should be who information of register courses	first type
#2	There exists a relation of complement between borrow books and return books	second type
#3	If complement(X, Y) and complement(Y, Z) then ADD complement(X, Z)	third type
#4	If complement(X, Y) and supplement(X, Y) then REMOVE supplement(X, Y)	third type
#5	Student should not be who information of * scores	first type
#6	There exists a relation of complement between register * and cancel *	second type
#7	There exists a relation of complement between register X and cancel X	second type

The second type of rules (relation rules) specifies the correctness of relations between functional requirements in an ontology. The grammar of the second type of rule is shown below.

There exists a relation of [complement| exclusion| inconsistent| redundancy] between *function1* **and** *function2*

For example, a rule description, "There exists a relation of complement between borrow books and return books" (#2), specifies a complement relation between the two functional requirements "borrow books" and "return books". With this type of rules we can detect wrong relations and lack of relations between functional requirements in an ontology.

The third type of rules (inference rules) specifies conditions and actions on relations among functional requirements. The grammar of the third type of rules is shown below.

If [complement| supplement| exclusion| redundancy] (*variable1, variable2*) **and [complement| supplement| exclusion| redundancy]** (*variable3, variable4*) **then [ADD| REMOVE] [complement| supplement| exclusion| redundancy]** (*variable5, variable6*)

For example, a rule description, "If complement(X, Y) and complement(Y, Z) then ADD complement(X, Z)" (#3), specifies that the complement relation is transitive. An-

other rule description, “If complement(X, Y) and supplement(X, Y) then REMOVE supplement(X, Y)” (#4), means that relation of supplement between two functional requirements which are having relation of complement will be deleted. With this type of rules we can detect lack of relations and remove wrong relations in an ontology.

We can use wildcard symbols in rules. For example, a rule description, “Student should not be who information of * scores” (#5), specifies the wrong agent of the group of functions accessing scores. Another rule description, “There exists a relation of complement between register * and cancel *” (#6), specifies a complementary relation between the two groups of functions: registration of something and cancellation of something. However, rule (#6) might request complementary relation of unrelated functions, e.g., “register account” and “cancel order”. To fix this problem, we can use variables in rules (#5) and (#6). For example, rule (#6) can be rewritten such as “There exists a relation of complement between register X and cancel X” (#7). This revised rule specifies complementary relation between functions “register account” and “cancel account”, between functions “register order” and “cancel order”, and so on.

When rules writers detect an error in a requirements ontology, they can define a new rule to detect similar errors. For example, when users can wrongly register books in a requirements ontology of library system, the rules writers can write a rule such as “Users should not be who information of register *.” (It means that users should not register anything.) To support definition of rules, we provide examples of three types of rules, and these examples are easy to understand how to detect errors in an ontology, then rules writers can revise the examples to make their own rules. Of course, rules writers can also define new rules based on the rules grammars.

6.1.2 Verification of the correctness of requirements ontology

We compare each rule with information in requirements ontology, and check the consistency between them. If consistent, we inform that the requirements ontology satisfies the rule. If inconsistent, we inform that the ontology does not satisfy the rule and counter examples will be shown. The counter examples are useful to correct wrong parts of the ontology. Verification procedures are different for three types of rules.

The verification procedure for the first type of rules is described below, and is illustrated in Fig. 6.1.

1. Check whether functions specified in a rule exist in requirements ontology. If not exist, error message will be given and verification will be ended unsuccessfully.

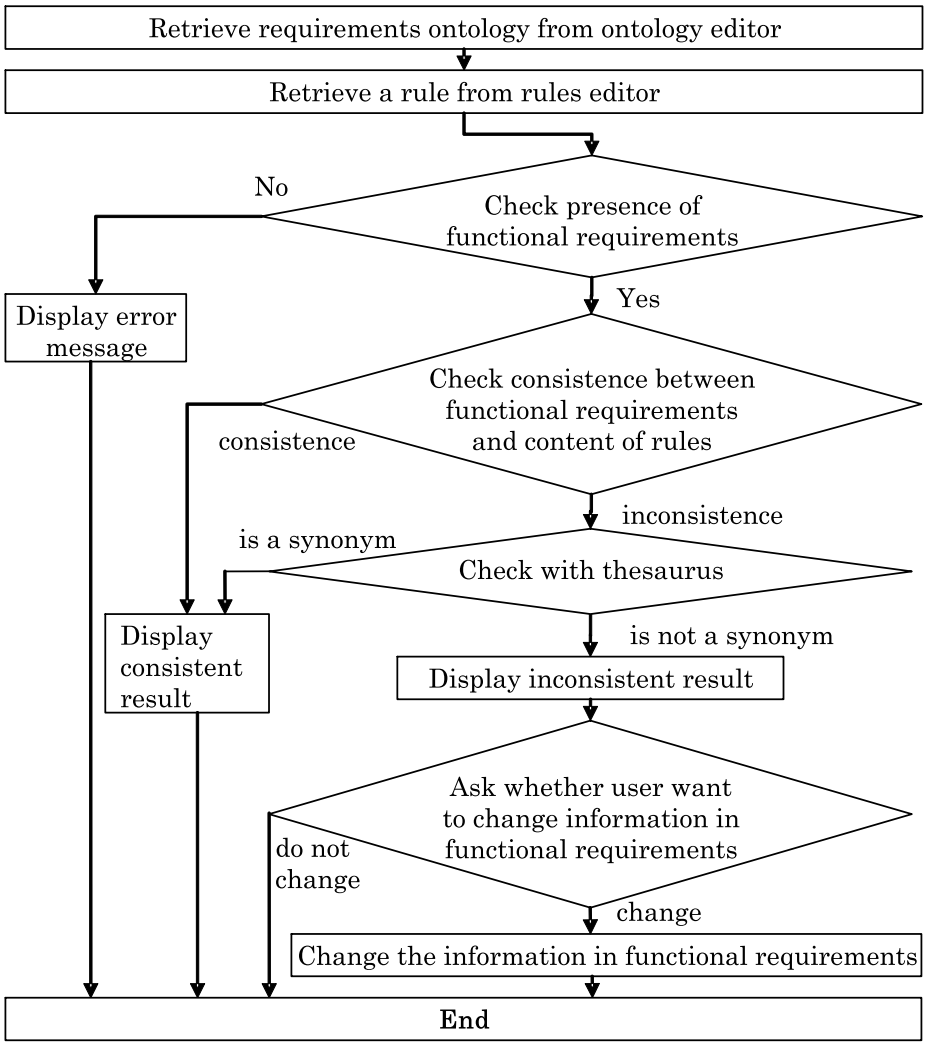


Figure 6.1 Flow of checking first type of rules.

(Flows of checking second & third types of rules are similar)

2. If exist, corresponding attribute of the functional requirements in the ontology will be derived and compared with specified attribute in the rule.

3. If same, verification will be ended successfully with a message: "successfully verified."

4. If different, transform the attribute in the rule using a thesaurus, then compare the transformed attribute with the corresponding attribute in the ontology.

5. If still different, error message and two different attributes will be given. If the rule is

correct and the ontology seems wrong, the ontology will be automatically corrected after user's approval.

As for the second type of rules, the verification procedure is as follows.

1. Check whether two functions specified in a rule exist in requirements ontology. If not exist, error message will be given and verification will be ended unsuccessfully.
2. If the two functions exist, check whether relation between the two requirements in the ontology is same as the relation in the rule.
3. If same, verification will be ended successfully with a message: "successfully verified."
4. If different, error message and two different relations will be given. If the rule is correct and the ontology seems wrong, the ontology will be automatically corrected after user's approval.

The verification procedure of the third type of rules is shown below.

1. Check whether functional requirements satisfied the two relations specified in the conditional part of a rule exist in requirements ontology. If not exist, error message will be given and verification will be ended unsuccessfully.
2. If exist, then list all of the functional requirements which are satisfied with the relation in the part of the rule. If ADD is specified in the rule, relations that do not exist in the ontology will be automatically added after user's approval. If REMOVE is specified in the rule, relations that exist in the ontology will be automatically deleted after user's approval.

6.2 A Supporting Tool for Verification of Requirements Ontology

We have developed a supporting tool for rule-based verification method with the three types of rules above. Fig. 6.2 shows configuration of the verification tool. The requirements ontology editor and the verification tool co-operate through a shared memory. The requirements ontology editor builds an ontology tree in memory and the verification tool can

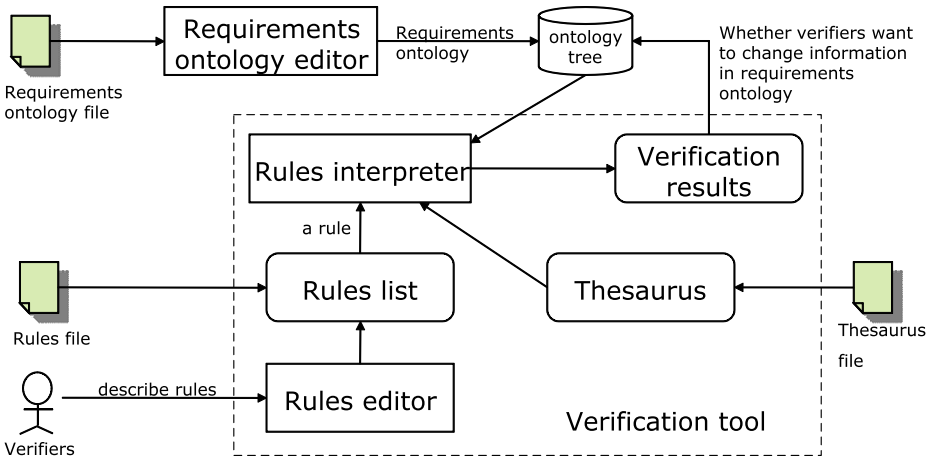


Figure 6.2 Configuration of the verification tool.

access that ontology tree to traverse through the tree; retrieve functions, attributes; and updates functions and attributes if users approve. The verification tool was written with Java on an Eclipse 3.5.2, and was one person-month product; the number of source code lines was 3,652. The verification tool provides six functions, namely to say, (1) input/saving of rule description files, (2) editing thesaurus, (3) editing rule descriptions and rule attributes, (4) analyzing rules, (5) retrieving rules, and (6) verifying requirements ontologies with rules. All implementations of the above functions are newly developed.

We will explain more about processing of rules by the rules interpreter (a module in Fig. 6.2). Firstly, a selected rule is classified into one of the three types of rules. Secondly, functions, attributes, and variables are extracted from the rule with regular expressions. Thirdly, the rules interpreter traverses through requirements ontology tree to compare functions, attributes in the tree with functions, attributes, and variables in the rule. The traversal and comparison are separated for each type of rules: for example, steps for the first type of rules are as in Fig. 6.1. The rules interpreter was written with Java, in 1,396 lines of code.

Fig. 6.3 shows verification with a type 1 rule. Attributes of a functional requirement specified in a rule will be compared with attributes of the requirements in the applied ontology. In Fig 6.3, the highlighted rule in the left box states that agent of the group of function of "* scores" should not be "student". In the right box, verifier shows a verification result that the ontology is inconsistent with the rule, because in the ontology, the function "Change scores" has a wrong Who attribute such as "student". The verification tool also suggests to delete that wrong information from ontology.

Fig. 6.4 shows a verification of a type 3 rule. In this figure, the rule specifies that if relation of supplement between x and y exists and relation of supplement between y and

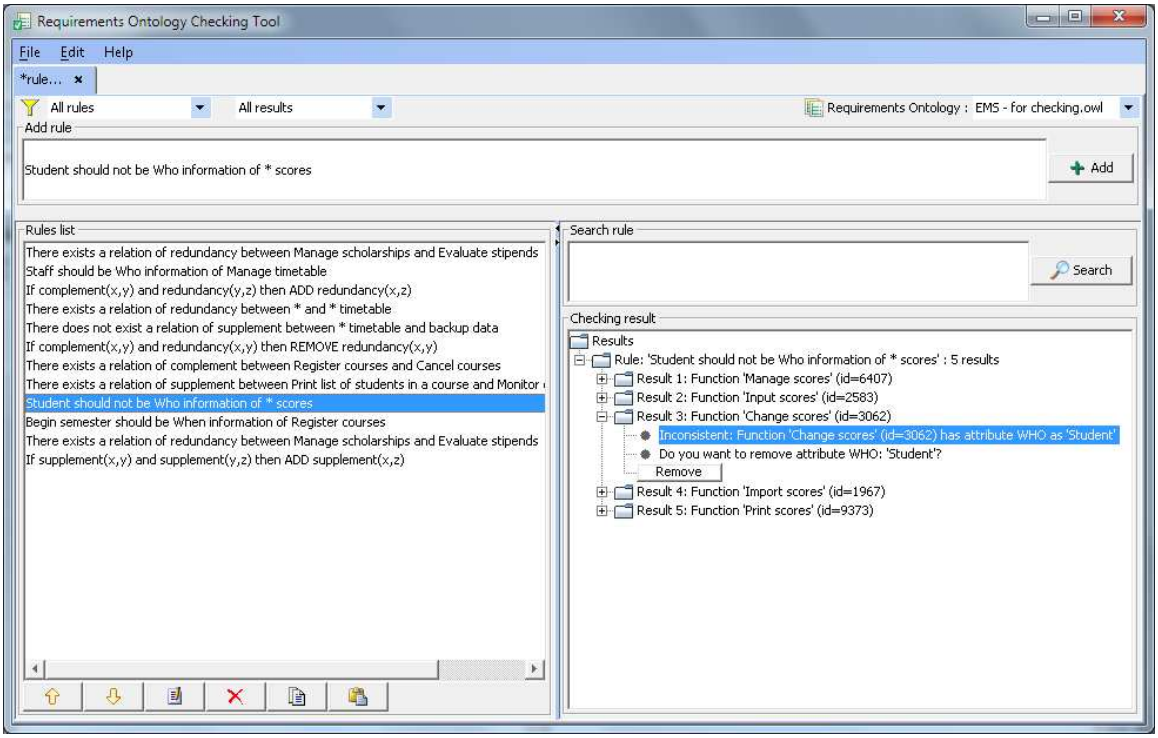


Figure 6.3 Verification of who information of a functional requirement.

z exists, then relation of supplement between x and z should exists in the ontology. In the right-side area of the window, components of ontology that meet the rule are displayed.

6.3 Chapter Summary

We proposed a rule-based verification method of the correctness of requirements ontology. We have developed a supporting tool for ontology verification on the basis of the method and illustrated the behaviors of the tool with examples. In the following chapter, we will evaluate this rule-based verification method through experiments.

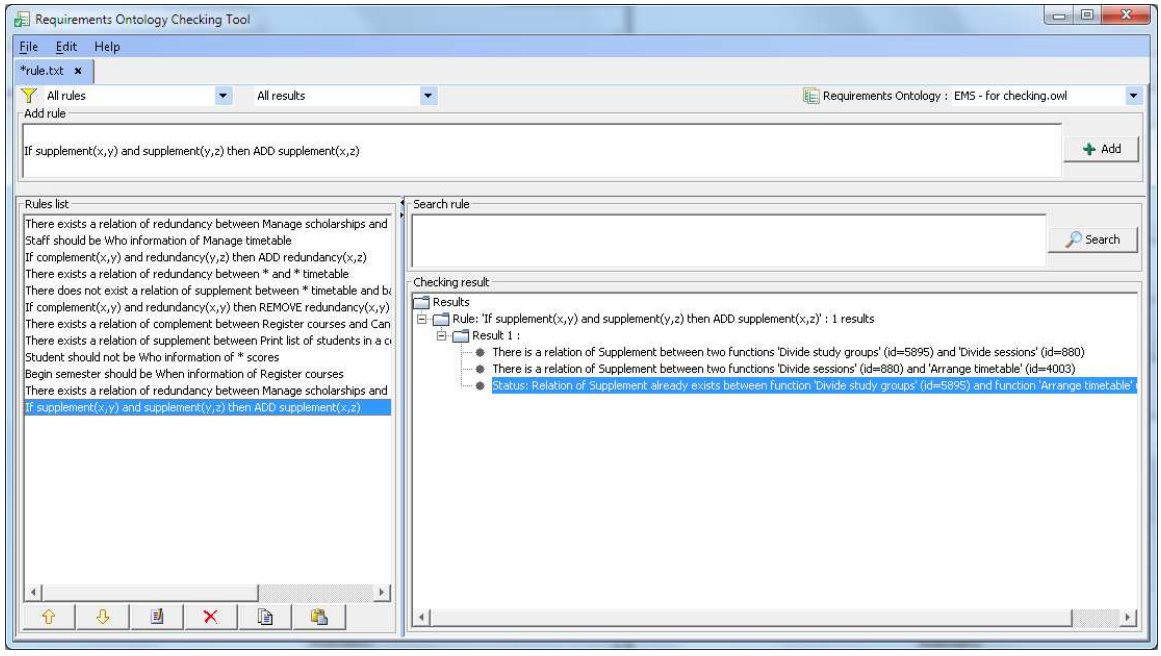


Figure 6.4 Verification with inference rule

This page intentionally left blank.

Chapter 7

Evaluation of Rule-based Verification Method of Requirements Ontology

To evaluate the effect of using verification tool to support rule-based verification method of requirements ontology, we conducted two comparative experiments: Experiment 6 and Experiment 7. The subjects in the experiments were four students who studied information science. All subjects have been trained with errors types in requirements ontology and introduced with rule-based verification method. In each experiment, subjects were asked to check a same requirements ontology and submit results. Four subjects were divided into two groups, each group had two people; one group checked by tool (with tool) and the other group checked by hand (without tool). The group checking by hand was provided with ontology editor which allows to view functions structure, search for functions, and display attributes of functions. The group checking by tool was provided with ontology editor, and verification tool which allows to define checking rules and to verify automatically using rules. Both groups were provided with samples of checking rules for corresponding domains (pre-defined rules). The working method of two groups exchanged in the two experiments: the group using tool in the Experiment 6 checked by hand in the Experiment 7 and vice versa.

The major difference between using tool and not using tool is the capability to detect similar errors with a known error. All of the subjects know the correctness and the incorrectness of requirements ontology. For example, indispensable functional requirements should exist in ontology, wrong attributes of a functional requirement should not be specified, and wrong relationships should not be specified, and so on. Subjects with tool check requirements ontology and when they found an error, they can write a rule to detect similar errors. For example, if user is wrongly specified as an actor of a function of management

of rental object, they specify a rule that actor of functions of managing something should be staff. Then the tool can detect similar errors if exist. By contrast, subjects without tool check ontology and when they found an error, they correct the error, but they cannot easily detect similar errors without the tool. This is the major difference between subjects with/without tool. In other words, subjects with tool can easily detect similar errors using the tool, but subjects without tool detect similar errors with effort.

Similar to Chapter 5, we also used three metrics to compare the results: recall, precision, and harmonic mean.

7.1 Experiment 6

In this experiment, two groups of subjects checked a requirements ontology for "hotel reception desk software". The requirements ontology included 80 function nodes; subjects were allowed 30 minutes to check and find errors in the requirements ontology. Subjects were requested to find errors such as: lacking or wrong 4W1H information, lacking or wrong relations between functional requirements.

Table 7.1 summarizes the recall and precision metrics of the results by two groups. We denote subjects as A, B, C, and D. Table 7.1 shows that all of the recall metrics and precision metrics of the results by the group with tool are higher than those by the group without tool.

Table 7.1 Precision and recall metrics in Experiment 6.

Group	Subject	Precision		Recall	
		Metrics	%	Metrics	%
with tool	A	11 / 15	73.33%	11 / 26	42.31%
	B	13 / 16	81.25%	13 / 26	50.00%
without tool	C	6 / 13	46.15%	6 / 26	23.08%
	D	8 / 20	40.00%	8 / 26	30.77%

Precision: the number of correctly detected errors / the number of detected errors.

Recall: the number of correctly detected errors / the number of errors.

7.2 Experiment 7

In Experiment 7, subjects checked a requirements ontology for "asset management software". The requirements ontology included 51 function nodes; subjects were also allowed 30 minutes to check the requirements ontology. Table 7.2 summarizes the recall and precision metrics of the results by two groups. The table shows that all of the recall metrics and precision metrics of the results by the group with tool are higher than those by the group without tool, though two groups have exchanged working method.

Table 7.2 Precision and recall metrics in Experiment 7.

Group	Subject	Precision		Recall	
		Metrics	%	Metrics	%
without tool	A	9 / 14	64.29%	9 / 34	26.47%
	B	8 / 14	57.14%	8 / 34	23.53%
with tool	C	11 / 14	78.57%	11 / 34	32.35%
	D	14 / 16	87.50%	14 / 34	41.18%

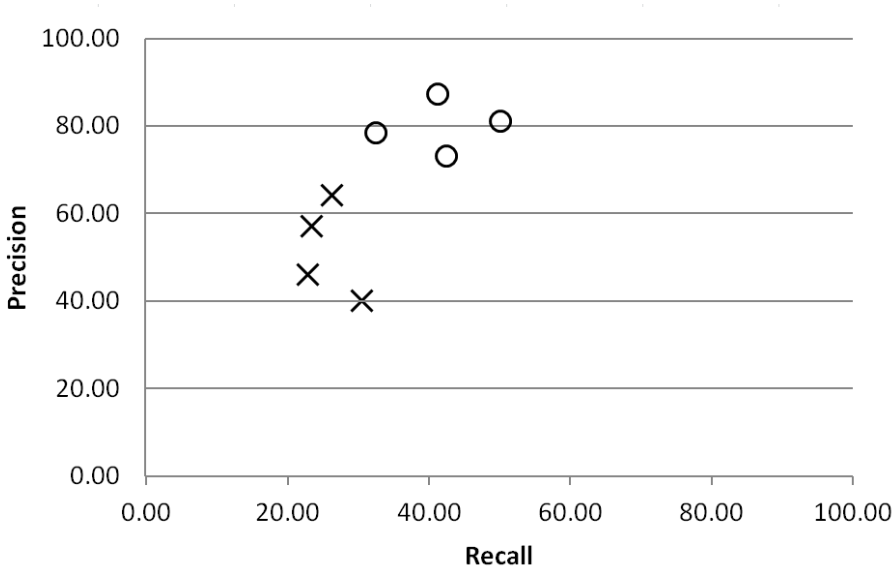
Precision: the number of correctly detected errors / the number of detected errors.

Recall: the number of correctly detected errors / the number of errors.

7.3 Discussion

To illustrate the recall and precision metrics more visually, we show these values in 100% scale in Fig. 7.1. The figure displays that in all cases in the two experiments, the precision metrics and recall metrics of the results by group with tool were higher than metrics of the results by group without tool. The averages of the recall metrics and precision metrics of the results by using tool, as shown in Table 7.3, were 41.46% and 80.16%, respectively, higher than those average metrics by not using tool which were 25.96% and 51.90%, respectively.

The recall metrics of results by subjects using tool were not high (the average was 41.46%). To find the reason, we classify the correctly detected errors into three sources: (1) errors detected by pre-defined rules, (2) errors detected by definition of new rules, (3) errors detected by hand. Fig. 7.2 shows the portions of sources of errors found by subjects with method. Majority of errors were detected by using pre-defined rules, and only minority of errors were detected by definition of new rules and by hand (subject C in



o: with tool; x: without tool

Figure 7.1 Distribution of precision metrics and recall metrics in two experiments.

Table 7.3 Averages of metrics in two experiments (higher is better).

	with tool	without tool
Recall	41.46%	25.96%
Precision	80.16%	51.90%

Experiment 7). On the errors detected by pre-defined rules, the subjects with tool also did not utilize all the errors that the tool could detect. It was because of their unfamiliarity with rule-based verification method and tool, and because of limited time. The verification tool only suggested potential issues, and subjects evaluated whether these issues were errors. In the future, if subjects could master rule-based verification method and have more time to define new rules, the recall metrics would be higher.

There were errors detected by subjects with tool but not detected by subjects without tool. For example, in Experiment 6 about "hotel reception desk software", there was an error of lacking of supplement relation between two functions: "split group receipt" and "process guest check out." The group using tool detected this error by a rule of the third type: "If complement(x, y) and supplement(y, z) then ADD supplement(x, z)" (The reasoning was as follows: Because there existed a relation complement(split group receipt,

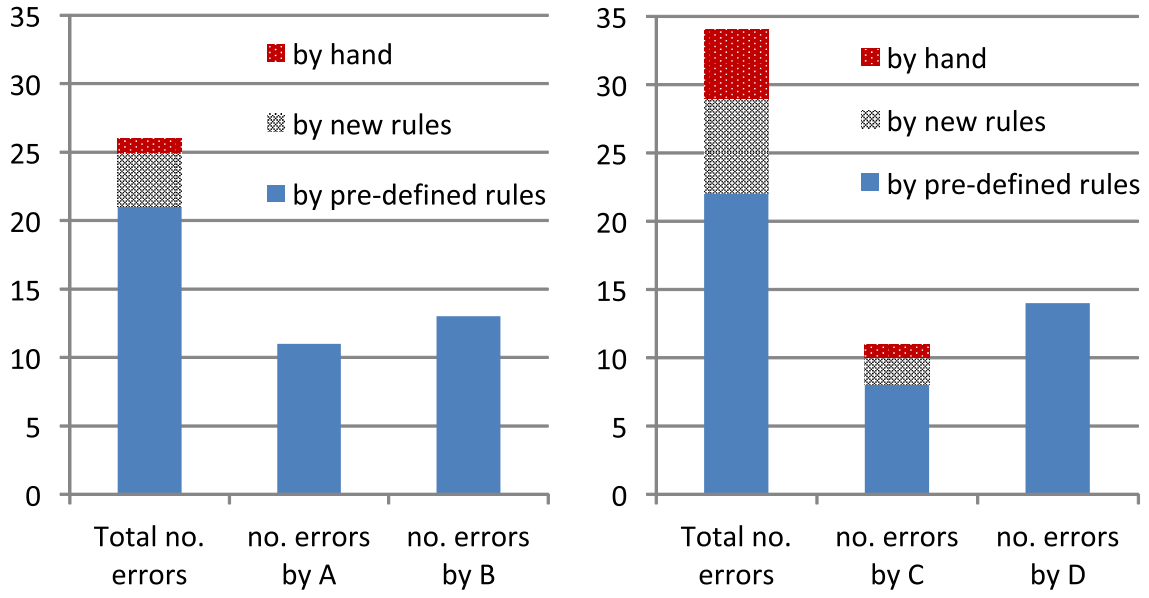


Figure 7.2 The number of errors found by subjects with tool

merge group receipt) and a relation supplement(merge group receipt, process guest check out), so we should add a relation supplement(split group receipt, process guest check out)). However, not having support by reasoning engine of the verification tool, subjects without tool did not detect this error. Another example was that in Experiment 7 about "asset management software," subject C (using tool) defined a new rule of the first type: "System should be who information of * session parameters." Using this rule, subject C detected an error of wrong who attribute of function "set session parameters" in the requirements ontology, which was previously described as "staff." Nevertheless, subjects without tool did not detect this error.

On the other hand, there were errors which were not detected by tool, but detected by hand. For example, in Experiment 7, there existed an error of lacking of supplement relation between functions "view unused assets" and "view list of assets." This error was detected by the group without tool, but was not detected by the group with tool. Though this error can be detected with rule-based verification method by definition of a rule of the second type, subjects with tool did not find this error by mistake. In addition, detection of this error required knowledge of the domain "asset management software." One solution is to provide a customization mechanism or rules to automatically apply the customized rules

to ontologies in future work.

Table 7.4 summaries the number of pre-defined rules and new rules which were defined by subjects using tool. Mostly pre-defined rules were used, but three new rules were described by subject C in Experiment 7 (The three new rules are shown in Table 7.5). One reason that new rules were not defined so much was that subjects were allowed a short time in each experiment (30 minutes). All the three new rules by subject C were syntactically and semantically correct. In definition of new rules, subjects were guided with examples of rules, and they used these examples to construct new rules.

Table 7.4 The number of pre-defined rules and the no. of new rules defined by subjects with tool.

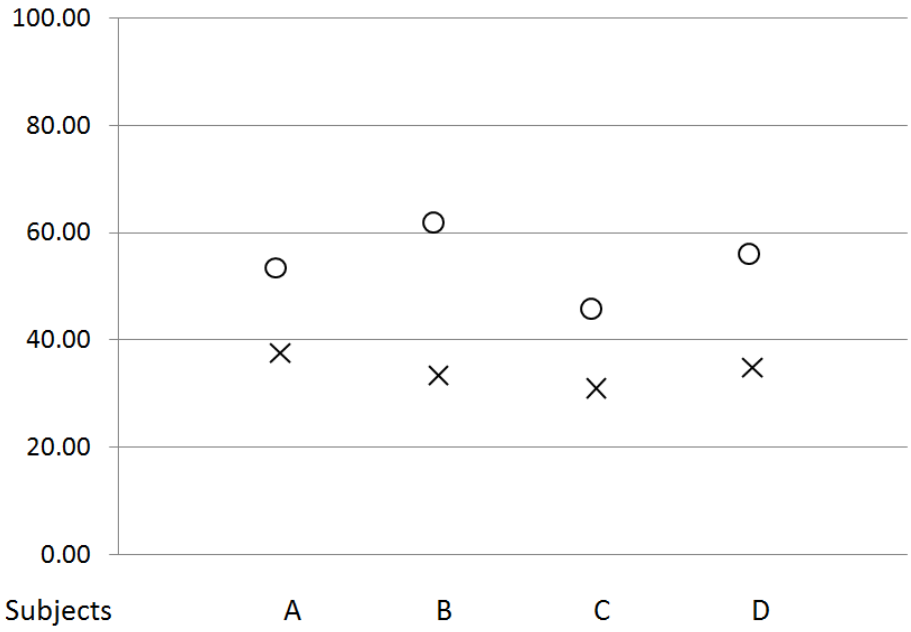
	Rules sources	no. of rules
Exp. 1	pre-defined rules	17
	new rules by A	0
	new rules by B	0
Exp. 2	pre-defined rules	18
	new rules by C	3
	new rules by D	0

Table 7.5 New rules defined by subject C.

R#	Rule	Rule type
#RC1	System should be who information of * session parameters	first type
#RC2	There exists a relation of supplement between Print X and Create X	second type
#RC3	There exists a relation of complement between Allocate * and Update *	second type

Fig. 7.3 displays the distribution of harmonic mean metrics in scale of 100%. It shows that in results of all subjects, the harmonic mean metrics by using tool were higher than the harmonic mean metrics by not using tool.

In summary, the recall, precision, and harmonic mean metrics of results by using tool were higher than those of results by not using tool. It suggests that using the verifica-



o: with tool; x: without tool

Figure 7.3 Harmonic mean metrics in two experiments.

tion tool with checking rules has positive effect in verification of requirements ontology, and helps improve the percentage and total of correctly detected errors in verification of requirements ontology. Based on errors found, requirements engineers will revise and improve the quality requirements ontology.

However, similar to Experiments 1 to 5, Experiments 6 and 7 still have some limitations and threats to validity as described in below sub-sections.

7.3.1 Limitations

1. Recall metrics were not high. In Experiments 6 and 7, recall metrics were not higher than 50%. Reasons included the facts that subjects with method described a few new rules and they did not utilize all the errors that the tool could detect.

   Solution: By applying rules customization, and improving skill of subjects in using the method in future experiments, we expect that recall metrics would be increased.

2. Subjects with method described a few new rules (only three). Reasons might be difficulties in specification of rules and limited time for each experiment (30 minutes).

Solution: Apply rules customization mechanism to help subjects describing rules more easily.

7.3.2 Threats to validity

1. The experiments had limited numbers of subjects (only four). The numbers of subjects were not high enough for satisfactory statistics.

   Solution: By conducting experiments with more subjects in the future, we can yield more reliable statistics results to evaluate our method.

2. All subjects in the experiments were students. It raises a question of effectiveness of our method in real projects of ontology development.

   Solution: Conduct experiments of rule-based verification method with other subjects than students to evaluate the effect of the method.

7.4 Chapter Summary

The experiments show that our rule-based verification method enables to detect incorrect components of requirements ontologies and improve the quality of requirements ontologies. Using a requirements ontology of good quality, we can elicit or verify requirements with good quality. In the future, we plan to provide a customization mechanism of rules to apply the rules to ontologies. The rule-based verication method of requirements ontology has potential to support verification of other ontologies.

Chapter 8

Conclusion

8.1 Thesis Summary

This thesis focused on ontology-based approach for improving quality of software requirements document. Using requirements ontology model derived from previous researches [35, 43], an ontology-based verification method of software requirements was proposed. The verification mechanism uses rules for detection of errors in SRS. Rules are classified into domain independent rules which are described in advance, and user definition rules which are described latter by analysts. Through reasoning with rules, inaccurate or insufficient requirements can be detected. Reasoning results are then used to revise requirements document. We have developed an ontology-based verification tool to support our method. The tool helps analysts to detect errors in software requirements using ontology and rules, and generate questions to suggest revision of requirements.

We have conducted five comparative experiments to evaluate the effect of ontology-based verification method. In each experiment, a group of analysts used our method and another group did not. Results of experiments suggested that on average our ontology-based method has positive effect in verification of software requirements. Our method helped increasing the number of correctly detected errors in SRS, thus contributed to improvement of quality of SRS.

On using requirements ontology to support requirements engineering including requirements verification, quality of output requirements depends on quality of requirements ontology. Therefore, we also proposed a rule-based verification method of the correctness of requirements ontology with three types of rules. We have developed a supporting tool for ontology verification on the basis of the method and illustrated the behaviors of the tool with examples.

The other two experiments on verification of requirements ontology show that our rule-based method enables to detect incorrect components of requirements ontologies and improve the quality of requirements ontologies. Using a requirements ontology of good quality, we can improve quality of software requirements.

8.2 Contributions

The thesis contributed to improving quality of software requirements specification in following ways:

1. Ontology-based verification method of software requirements was proposed. Requirements ontology and rules are used to find errors in software requirements. Requirements ontology is a type of domain knowledge about software requirements. Rules contain general knowledge (domain independent rules) and domain knowledge (user definition rules). Therefore, ontology-based verification method utilizes domain knowledge and general knowledge to establish a reasoning mechanism for detection of errors in SRS and then improving quality of software requirements. Our method can be used to verify requirements which are stated in natural languages.

2. Ontology-based verification method was not only proposed as a prototype but it was also provided with detail guidelines and supportive implementation of each step in verification (Chapter 4).

3. A distinction of our ontology-based verification method is that existing researches support detection of errors in SRS with general rules, but our method supports detection of errors with user definition rules. Another key point of our method is automatic verification of natural languages SRS: automatic parsing of requirements sentences, automatic mapping of sentences to ontology, automatic reasoning of errors, automatic generation of questions for revising requirements. These automations help shortening verification time of SRS (discussed in sub-Sect. 5.1.3).

4. Application of ontology-based verification method does not cost much resource. In experiments (in Chapter 5), it took about 90 minutes to train participants about the method. The subjects understood the method quickly and could apply it instantly.

5. Our ontology-based verification method does not only verify quality of requirements specification, but it can also verify quality of initial requirements or elicited requirements. Fig. 8.1 illustrates the fact that our method can be applied in early stages

of requirements engineering such as elicitation or evaluation as a contribution to the field.

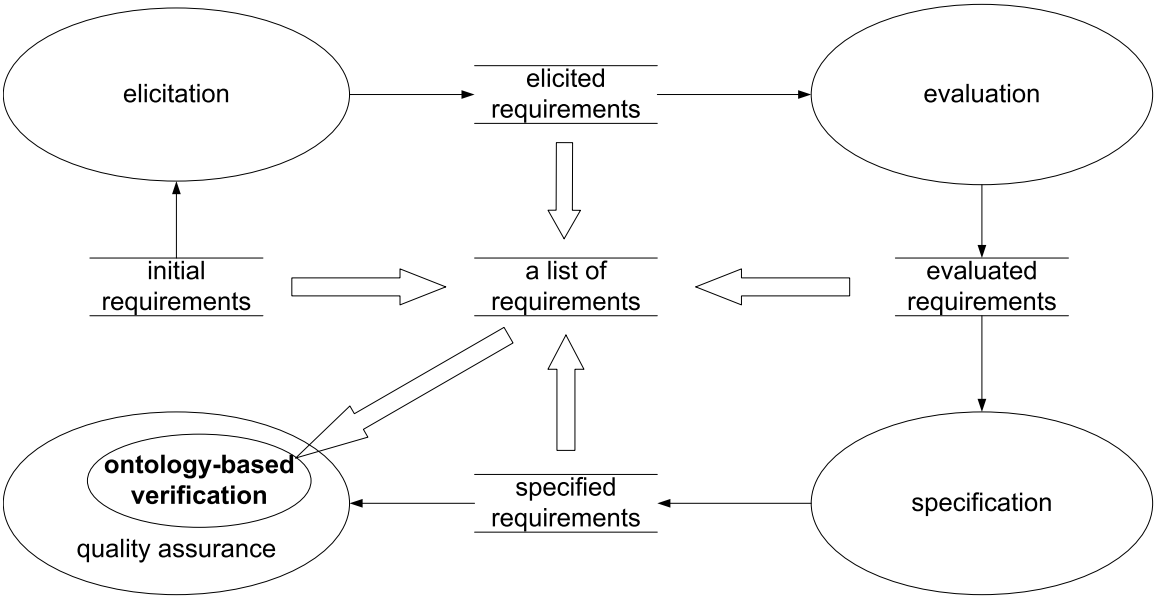


Figure 8.1 A contribution of our research to requirements engineering.

6. Not only ontology-based verification method contributes to good quality of software requirements, but correctness of requirements ontology also does. For that purpose, a rule-based verification method for guarantee the correctness of requirements ontology was proposed.

8.3 Future Works

There still exists several issues in our current research such as: weakness in detection of some specific types of errors, and difficulty for users of the methods to specify new rules. Future works of this research includes the following topics to solve them.

1. Improve method of detection of errors in software requirements.
   In some cases of specific types of errors such as: inconsistency, redundancy, and ambiguity, the effectiveness of the ontology-based verification method was still limited (Chapter 5). The improvement of method to detect these errors are left as a future research topic.

2. Improve processing of requirements sentences.
   In implementation, we can extract 4W1H information from requirements sentences but we still cannot classify types of 4W1H easily (Sub-sect. 4.6.1). Requirements also can be mapped to ontology using other clues (4W1H information), instead of only verb and nouns as in our current method. Therefore, we can study more on how to extract information from requirements sentences and how to map them to ontology nodes exactly.

3. Study more on revision of requirements.
   In our ontology-based verification method of software requirements, we focused on detection of errors. Though our method supports generate questions for revision of requirements, that feature was only used in Experiment 1 (in Chapter 5). In the future, we can study more on how to revise requirements based on verification results.

4. Verify full sentence meaning of requirements statements.
   In this thesis, we verify software requirements using verbs, nouns, and 4W1H information in requirements sentences. Is it possible to verify full meanings of requirements sentences? Suppose that requirements are stated in controlled languages (Controlled languages use limited words and simple grammar). We can translate the requirements to first-order logic and conduct model checking. This method can be used to check full meanings of requirements statements, and can be applied in critical systems.

5. Conduct larger experiments.
   Experiments in this thesis have evaluated the effect of our methods, but they were still on a small scale. In the future, we plan to evaluate our methods through larger experiments (in number of participants, size of SRS and ontology). User definition rules provide an extension to ontology-based verification method, but it was not used in the experiments, so we also would like to apply this extension in future experiments. In addition, since sometimes we do not know there are how many total of errors [1], we expect to apply a better evaluation than recall and precision in future experiments.

6. Study more on rule-based verification method of requirements ontology.
   We proposed rule-based verification method of requirements ontology and evaluated the method. But in the experiments (in Chapter 7), subjects described only a few new rules (mostly pre-defined rules were used), and the percentage of correctly detected

errors was not high (less than 50%). In the future, we plan to propose a rules customization mechanism to help subjects describing rules. In addition, the rule-based verification method of requirements ontology has potential to support verification of other ontologies.

7. Use data ontology in verification.
   Data ontology represents a hierarchy and relations of objects in a system. Data ontology has potential usefulness in requirements engineering because it represents domain knowledge including data structures in the application domain. The research in this thesis focused on usage of requirements ontology, but it still does not utilize data ontology. In the future, we can study on usage of data ontology to support requirements elicitation or verification.

8. Other research directions.
   Our current requirements ontology model is not capable of representing time order/sequence of events/activities, so it cannot help detection of errors relating to sequence of requirements. In the future, we expect to extend requirements ontology model so it can represent sequence of functional requirements, e.g., integration of a temporal logic approach to requirements ontology model.

   We claimed that requirements should be written using natural languages, but diagrammatic notations are common in nowaday SRS. We can study on whether requirements ontology can support detection of errors in data-flow diagram and UML diagrams.

   We use rules and ontology in reasoning errors in software requirements, but we have concerned only about correctness of requirements ontology. Therefore, the correctness of rules might also be a topic in future research.

This page intentionally left blank.

Bibliography

[1] R. A. Baeza-Yates and B. Ribeiro-Neto, "Retrieval evaluation," in *Modern Information Retrieval*. Addison-Wesley, 1999, p. 81.

[2] A. Bao, L. Yao, W. Zhang, and J. Yuan, "Approach to the formal representation of owl-s ontology maintenance requirements," in *Proc. 9th Int. Conf. on Web-Age Information Management (WAIM '08)*, July 2008, pp. 56–61.

[3] B. W. Boehm, "Software engineering economics," *Software Engineering, IEEE Transactions on*, vol. SE-10, no. 1, pp. 4–21, Jan 1984.

[4] B. W. Boehm, "Verifying and validating software requirements and design specifications," *IEEE Software*, vol. 1, no. 1, pp. 75–88, 1984.

[5] B. W. Boehm, Ed., *Software Risk Management*. Piscataway, NJ, USA: IEEE Press, 1989.

[6] K. Breitman and J. do Prado Leite, "Ontology as a requirements engineering product," in *Proc. 11th IEEE Int. Requirements Engineering Conf. (RE'03)*, Sept. 2003, pp. 309–319.

[7] J. Buxton and B. Randell, Eds., *Software Engineering Techniques, Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy*, 1970.

[8] C. Coral, R. Francisco, and P. Mario, *Ontologies for Software Engineering and Software Technology*. Berlin, Heidelberg: Springer-Verlag, p. 3, 2006.

[9] G. Dobson, S. Hall, and G. Kotonya, "A domain-independent ontology for non-functional requirements," in *Proc. IEEE Int. Conf. on e-Business Engineering (ICEBE'07)*, Oct. 2007, pp. 563–566.

[10] D. Gildea and D. Jurafsky, "Automatic labeling of semantic roles," *Comput. Linguist.*, vol. 28, no. 3, pp. 245–288, Sep. 2002.

[11] R. B. Grady, "Using domain ontology as domain knowledge for requirements elicitation," in *Proc. Applications of Software Measurement Conference*, 1999, pp. 227–239.

[12] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowl. Acquis.*, vol. 5, no. 2, pp. 199–220, Jun. 1993.

[13] N. Guarino, *Formal Ontology in Information Systems: Proceedings of the 1st International Conference June 6-8, 1998, Trento, Italy*, 1st ed. Amsterdam, The Netherlands: IOS Press, 1998.

[14] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Syst. J.*, vol. 41, no. 1, pp. 4–12, Jan. 2002.

[15] V. Hirankitti and V. T. Xuan, "A meta-logical approach for reasoning with semantic web ontologies," in *Proc. 4th IEEE Int. Conf. on Computer Sciences: Research, Innovation and Vision for the Future*, 2006, pp. 229–236.

[16] B. Q. Huy and A. Ohnishi, "A verification method of the correctness of requirements ontology," in *Proc. of the 10th Joint Conference on Knowledge-Based Software Engineering (JCKBSE 2012)*, 2012, pp. 1–10.

[17] IEEE, "Recommended practice for software requirements specifications," *IEEE Std 830-1998*, 1998.

[18] W. S. Jr, R. Burgin, and P. Howell, "Performance standards and evaluations in ir test collections: Cluster-based retrieval models," *Information Processing and Management*, vol. 33, no. 1, pp. 1–14, 1997.

[19] H. Kaiya and M. Saeki, "Using domain ontology as domain knowledge for requirements elicitation," in *Proc. 14th IEEE Int. Requirements Engineering Conf.*, ser. RE '06, 2006, pp. 186–195.

[20] K. Kakimoto, "Development of requirements ontology construction support tool from software documents," Ritsumeikan Univeristy, graduation thesis, 2014 (in Japanese).

[21] A. Kalyanpur, B. Parsia, E. Sirin, and J. A. Hendler, "Debugging unsatisfiable classes in owl ontologies," *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, no. 4, pp. 268–293, 2005.

[22] J. C. Kelly, J. S. Sherif, and J. Hops, "An analysis of defect densities found during software inspections," *Journal of Systems and Software*, vol. 17, no. 2, pp. 111–117, 1992.

[23] M. Kitamura, R. Hasegawa, H. Kaiya, and M. Saeki, "A supporting tool for requirements elicitation using a domain ontology," in *Software and Data Technologies*. Springer Berlin Heidelberg, 2009, vol. 22, pp. 128–140.

[24] R. Kluge, T. Hering, R. Belter, and B. Franczyk, "An approach for matching functional business requirements to standard application software packages via ontology," in *Proc. 32nd IEEE Int. Conf. on Computer Software and Applications (COMPSAC '08)*, Aug. 2008, pp. 1017–1022.

[25] G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Techniques*. Wiley, p. 9, 1998.

[26] Ref. 25, p. 10.

[27] Ref. 25, p. 6.

[28] Ref. 25, p. ix.

[29] P. Kroha, R. Janetzko, and J. E. Labra, "Ontologies in checking for inconsistency of requirements specification," in *Proc. 2009 Third Int. Conf. on Advances in Semantic Processing*, ser. SEMAPRO '09, 2009, pp. 32–37.

[30] S. Lauesen, *Software Requirements: Styles and Techniques*. Addison-Wesley, p. 390, 2002.

[31] Ref. 30, p. 378.

[32] L. Liu, Q. Liu, C. hung Chi, Z. Jin, and E. Yu, "Towards a service requirements ontology on knowledge and intention," in *Proc. 6th Int. Conf. on Quality Software (QSIC'06)*, Oct. 2006, pp. 452–462.

[33] T. Morgan, in *Business Rules and Information Systems: Aligning It with Business Goals*. Boston, USA: Addison-Wesley, 2002, ch. 3, pp. 59–100.

[34] P. Naur and B. Randell, Eds., *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany*, 1969.

[35] A. Ohnishi and J. Kato, "Evaluation of requirements elicitation method using ontology," IEICE, Tech. Rep. vol. 106, no. 382, pp. 25–30, Nov. 2006 (in Japanese).

[36] OpenNLP library, http://opennlp.sourceforge.net/, accessed: 2011-10-25.

[37] Pellet: OWL 2 Reasoner for Java, http://clarkparsia.com/pellet/, accessed: 2012-10-14.

[38] R. Pressman, *Software Engineering: A Practitioner's Approach*, 6th ed. McGraw-Hill, p. 6, 2005.

[39] Ref. 38, p. 142.

[40] Ref. 38, p. 772.

[41] Protege, http://protege.stanford.edu/, accessed: 2013-5-14.

[42] R. G. Ross, "Rulespeak – templates and guidelines for business rules," *Business Rules Journal*, vol. 2, no. 5, 2001.

[43] R. Sakamoto, "Development of ontology editor for requirements elicitation," Ritsumeikan Univeristy, graduation thesis, 2007 (in Japanese).

[44] R. Sharman, R. Kishore, and R. Ramesh, Eds., *Ontologies: A Handbook of Principles, Concepts and Applications in Information Systems*. Springer New York, p. xi, 2007.

[45] I. Sommerville, *Software Engineering*, 9th ed. Addison-Wesley, p. 7, 2010.

[46] I. Sommerville and P. Sawyer, *Requirements Engineering: A good practice guide*. Wiley, p. 5, 1997.

[47] Ref. 46, p. 190.

[48] Ref. 46, p. 201.

[49] Statistical parsing of English sentences, http://www.codeproject.com/Articles/~12109/Statistical-parsing-of-English-sentences, accessed: 2011-12-10.

[50] R. Studer, V. R. Benjamins, and D. Fensel, "Knowledge engineering: Principles and methods," *Data Knowl. Eng.*, vol. 25, no. 1-2, pp. 161–197, Mar. 1998.

[51] T. Sugiura, "Ontology creation tool for support requirements elicitation," Ritsumeikan Univeristy, master thesis, 2007 (in Japanese).

[52] SWI-Prolog, http://www.swi-prolog.org/, accessed: 2012-6-15.

[53] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, p. 187, 2009.

[54] Ref. 53, p. 145.

[55] Ref. 53, p. 211.

[56] Ref. 53, p. 213.

[57] Ref. 53, p. 33.

[58] Ref. 53, p. 34.

[59] Ref. 53, p. xxi.

[60] H. Wang, M. Horridge, A. L. Rector, N. Drummond, and J. Seidenberg, "Debugging owl-dl ontologies: A heuristic approach," in *International Semantic Web Conference*, vol. 3729, 2005, pp. 745–757.

[61] K. Wiegers, *Software Requirements*, 2nd ed. Microsoft Press, p. 263, 2003.

[62] K. Wiegers, *More About Software Requirements: Thorny Issues and Practical Advice: Thorny Issues and Practical Advice*. Microsoft Press, p. 281, 2009.

[63] Ref. 62, p. 4.

[64] Wiktionary: A free-content multilingual dictionary, http://en.wiktionary.org/~wiki/verify, accessed: 2014-5-10.

[65] World Wide Web Consortium (W3C), "OWL Web Ontology Language Overview," http://www.w3.org/TR/owl-features/, accessed: 2012-1-16.

[66] World Wide Web Consortium (W3C), "SWRL: A Semantic Web Rule Language," http://www.w3.org/Submission/SWRL/, accessed: 2011-11-25.

[67] J. Xiang, L. Liu, W. Qiao, and J. Yang, "Srem: A service requirements elicitation mechanism based on ontology," in *Proc. 31st IEEE Int. Conf. on Computer Software and Applications (COMPSAC '07)*, vol. 1, July 2007, pp. 196–203.

[68] T. Yoshimura, "Development of creation support tool of requirements ontology from software documents," Ritsumeikan Univeristy, master thesis, 2012 (in Japanese).

[69] W. Zhang, H. Mei, and H. Zhao, "A feature-oriented approach to modeling requirements dependencies," in *Proc. 13th IEEE Int. Requirements Engineering Conf. (RE'05)*, Sept. 2005, pp. 273–282.

[70] X. Zhu and Z. Jin, "Detecting of requirements inconsistency: an ontology-based approach," in *Proc. 5th Int. Conf. on Computer and Information Technology (CIT'05)*, Sept. 2005, pp. 869–875.

[71] L. Zong-Yong, W. Zhi-Xu, Z. Ai-Hui, and X. Yong, "The domain ontology and domain rules based requirements model checking," *International Journal of Software Engineering and Its Applications*, vol. 1, no. 1, pp. 89–100, 2007.

[72] L. Zong-yong, W. Zhi-xue, Y. Ying-ying, W. Yue, and L. Ying, "Towards a multiple ontology framework for requirements elicitation and reuse," in *Proc. 31st IEEE Int. Conf. on Computer Software and Applications (COMPSAC '07)*, vol. 1, July 2007, pp. 189–195.

List of Publications

Journals

1. Dzung, D., Ohnishi, A., "Ontology-based Checking Method of Requirements Specication," *IEICE Transactions on Information and Systems*, vol. E97-D, no. 5, pp. 1028–1038, 2014.

2. Dzung, D., Huy, B., Ohnishi, A., "Rule-based Verification Method of Requirements Ontology," *IEICE Transactions on Information and Systems*, vol. E97-D, no. 5, pp. 1017–1027, 2014.

International Conferences

1. Dzung, D., Ohnishi, A., "Evaluation of Ontology-based Checking of Software Requirements Specification," *Proc. 37th Int. Computer Software & Applications Conf. (COMPSAC'13)*, July 2013, pp.425–430.

2. Dzung, D., Ohnishi, A., "A Verification Method of Elicited Software Requirements using Requirements Ontology," *Proc. 19th AsiaPacic Software Engineering Conference (APSEC'12)*, Dec. 2012, pp.553–558.

3. Dzung, D., Ohnishi, A., "Ontology-based Reasoning in Requirements Elicitation," *Proc. 7th IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM'09)*, Nov. 2009, pp. 263–272.

Technical Reports

1. Dzung, D., Ohnishi, A., "Detecting Errors in Initial Requirements: An Ontology-based Approach," *IEICE Tech. Rep.*, vol. 109, no. 150, pp. 7–12, July 2009.

2. Dzung, D., Ohnishi, A., “Checking of Requirements using Requirements Ontology,” *IEICE Tech. Rep.*, vol. 108, no. 326, pp. 25–30, Nov. 2008.

Appendix A: Reference data of Experiment 1

Final list of functional requirements of QMS system found by subject S3

1. *Upon arriving at city hall, citizens get tickets at ticket machine.*
2. *After finish processing with a citizen, staffs call next citizen in.*
3. *The LED displays at staffs counters show the calling ticket.*
4. *The central LED display shows three recent called tickets.*
5. *The speaker informs the citizen turn.*
6. *Staffs can call any ticket number from queue.*
7. *If citizens do not come, staffs call them again.*
8. *Staffs can use microphone to call citizen directly.*
9. *QMS system displays guideline how to use the system for citizens.*
10. *Citizens can reserve tickets in advance through internet. They will be informed the time that they should be at the city hall.*
11. *Citizens can reserve tickets through telephone or SMS.*
12. *Citizens can choose priority order if deserved.*
13. *Start session: Staffs login the QMS system; the system then establish working environment for staffs.*
14. *Staffs can view list of tickets in queues.*
15. *Staffs can filter queuing tickets for their counters.*
16. *Staffs will transfer citizens to other counters if the citizens have chosen wrong services codes.*
17. *Staffs can move the absent ticket to the end of queue.*
18. *Staffs can cancel citizens turn if after calling for certain of time no citizen comes in.*
19. *Staffs can stop transactions at their counters for certain of time.*

20. *When staffs log off QMS system, it will save log history of them.*
21. *Administrators can view list of staffs.*
22. *Add a staff*
23. *Edit a staff*
24. *Delete a staff*
25. *Administrators can view list of counters.*
26. *Add a counter*
27. *Edit a counter*
28. *Delete a counter*
29. *Administrators can view list of services for citizens at city hall.*
30. *Add a service*
31. *Edit a service*
32. *Delete a service*
33. *Administrators assign services to be processed at corresponding counters.*
34. *Administrators assign staffs to sit at corresponding counters.*
35. *Administrators can update chains of services. A chain of services includes an order of counters that citizens must follow to finish a transactions.*
36. *Administrators can view reports*
37. *Statistics on number of citizens per services*
38. *Statistics on transactions time*
39. *Statistics on waiting time of citizens*
40. *Administrators can view log of using system: users, functions, date time, actions, and results.*
41. *Administrators can set up system parameters.*
42. *Backup system*
43. *Restore system*
44. *Administrators can delete tickets in queue.*
45. *At the beginning of working day, all existing tickets in queue will be deleted, and new tickets are issued starting from 1.*
46. *Stop queue: stop issuing tickets*
47. *Resume queue: continue issuing tickets*

Appendix B: Reference data of Experiment 4

R1. Manage schedule of managers.

R2. Manage schedule of departments.

R3. Manage schedule of staffs.

R4. Staffs have to log in system with accounts in order to view schedule.

R5. Staffs can search schedule of the company and schedule of other people.

R6. A staff can update schedule of other staffs if the schedule has changed.

R7. Staffs can update schedule of their departments.

R8. Only secretary can create schedule of managers.

R9. Only managers can delete schedule of departments.

R10. A staff can view working schedule of other staffs.

R11. Managers can view working schedule of staffs.

R12. We want to display our working schedule in a screen at the entrance of our building.

R13. The screen will show some special news for that day.

R14. A secretary will input schedule of the company.

R15. Staff can create working events of his/her department.

Figure B1 SRS of a managing schedule software for a company.

Table BI Requirements ontology of managing schedule software.

Function	Relation	4W1H
F.1 View schedule		
F.1.1 View weakly schedule	c F.1.2	How: this weak by default
F.1.2 View daily schedule	s F.1.1	How: current day by default
F.1.3 View event detail	s F.1.2	How: current day by default
F.2 Manage schedule of manager		
F.2.1 View schedule of manager	i F.1	
F.2.2 Search schedule of manager		
F.2.3 Create schedule of manager	c F.2.5	Who: secretary, manager
F.2.4 Update schedule of manager	s F.2.3	Who: secretary, manager
F.2.5 Delete schedule of manager	c F.2.3	Who: administrator
F.3 Manage schedule of department		
F.3.1 View schedule of department	i F.1	
F.3.2 Search schedule of department		
F.3.3 Create schedule of department	c F.3.5	Who: secretary, manager
F.3.4 Update schedule of department	s F.3.3	Who: secretary, manager
F.3.5 Delete schedule of department	c F.3.3	Who: administrator
F.4 Manage schedule of staff		
F.4.1 View schedule of staff	i F.1	
F.4.2 Search schedule of staff		
F.4.3 Create schedule of staff	c F.4.3	Who: that person
F.4.4 Update schedule of staff	s F.4.3	Who: that person
F.4.5 Delete schedule of staff	c F.4.3	Who: administrator
F.5 Manage system		
F.5.1 Log in system	c F.5.2	
F.5.2 Log out system	c F.5.1	
F.5.3 Create user account	s F.5.1	Who: administrator
F.5.4 Assign user right	s F.5.3	Who: administrator

Abbreviation of relations: c - complement, s - supplement, i - inheritance

I. 4W1H errors (lacking or wrong description of functional requirements):
   - R6: (wrong Who) Requirements: “A staff” vs. Ontology: “that person”
   - R7: (wrong Who) Requirements: “Staffs” vs. Ontology: “secretary, manager”
   - R8: (wrong Who) Requirements: “Only secretary” vs. Ontology: “secretary, manager”
   - R9: (wrong Who) Requirements: “Only managers” vs. Ontology: “administrator”
   - R14: (wrong Who) Requirements: “A scretary” vs. Ontology: “secretary, manager”
   - R15: (wrong Who) Requirements: “Staff” vs. Ontology: “secretary, manager”

II. Incomplete errors (lacking functions):
   - F.1 View schedule
   - F.1.1 View weakly schedule
   - F.1.2 View daily schedule
   - F.1.3 View event detail
   - F.2.4 Update schedule of manager
   - F.2.5 Delete schedule of manager
   - F.4.3 Create schedule of staff
   - F.4.5 Delete schedule of staff
   - F.5.2 Log out system
   - F.5.3 Create user account
   - F.5.4 Assign user right

III. Redundant errors (duplicate requirements):
   - R8 and R14 express the same function “create schedule of manager”
   - R10 and R11 express the same function “view schedule of staff”

IV. Ambiguous errors (multiple-meaning requirements):
   - R5 expresses three functions: “search schedule of manager”, “search schedule of department” and “search schedule of staff”
   - R12 expresses two functions: “view schedule of manager” and “view schedule of department”
   - R14 expresses two functions: “create schedule of manager” and “create schedule of department”

V. Off-topic errors (out of scope requirements):
   - R13 is not a function of managing schedule software.

Figure B2 Correctly detected errors in SRS of managing schedule software.

This page intentionally left blank.

Appendix C: Reference data of Experiment 5

R1. We want a "Test Editor" which allows creation of tests; each test includes a set of multiple-choice quizzes.

R2. The Test Editor exports a test to HTML file.

R3. A test file in HTML can run independently in web browsers.

R4. The tool allows declaring parameters of test: number of quizzes, with suggestions or without suggestions, limited time or unlimited time.

R5. Teachers can insert quizzes to a test.

R6. Teachers can create a crossword puzzle with the Test Editor.

R7. Teachers can add new quizzes to an exported test file in HTML.

R8. Teachers can mix quizzes in a test.

R9. Teachers can create quizzes bank with the tool.

R10. The tool can generate a test from quizzes bank.

R11. Teachers can reuse a previous test and permute quizzes to make several tests.

R12. The tool can generate different tests with a same number of quizzes.

R13. Teachers can export quizzes from quizzes bank and share with other teachers.

Figure C1 SRS of a test editor.

Table CI Requirements ontology of test editor.

Function	Relation	4W1H
F.1 Create a new quiz		
F.1.1 Add an answer	c F.1.2	
F.1.2 Remove an answer	c F.1.1	
F.1.3 Change answers order		
F.1.4 Insert image into quiz		
F.1.5 Insert equation into quiz		
F.2 Create quizzes bank	s F.3.2, F.3.3	
F.2.1 Insert a quiz to bank	c F.2.2; i F.1	
F.2.2 Remove a quiz from bank	c F.2.1	
F.2.3 Import quizzes from file	c F.2.4	
F.2.4 Export quizzes to file	c F.2.3	
F.3 Create a test		
F.3.1 Declare test parameters		How: tests with suggestions allow unlimited time
F.3.2 Generate a test from bank		How: randomly
F.3.3 Insert a quiz from bank		
F.3.4 Insert a quiz directly	i F.1	Where: begin of test, end of test, after certain quiz
F.3.5 Remove a quiz from test		How: select the quiz number to delete
F.3.6 Mix quizzes in a test		How: mix randomly, mix by groups of quizzes
F.3.7 Mix answers choices		
F.3.8 Save a test	s F.3	
F.3.9 Export a test	s F.3.8	How: to HTML, Word files
F.3.10 Print a test	s F.3	

Abbreviation of relations: c - complement, s - supplement, i - inheritance

I. 4W1H errors (lacking or wrong description of functional requirements):
   - R2: (lack How) Ontology: “How: to HTML, Word files”
   - R4: (lack How) Ontology: “How: tests with suggestions allow unlimited time”
   - R5: (lack Where) Ontology: “Where: begin of test, end of test, after certain quiz”
   - R8: (lack How) Ontology: “How: mix randomly, mix by groups of quizzes”
   - R10: (lack How) Ontology: “How: randomly”
   - R12: (lack How) Ontology: “How: randomly”

II. Incomplete errors (lacking functions):
   - F.1 Create a new quiz
   - F.1.1 Add an answer
   - F.1.2 Remove an answer
   - F.1.3 Change answers order
   - F.1.4 Insert image into quiz
   - F.1.5 Insert equation into quiz
   - F.2.1 Insert a quiz to bank
   - F.2.2 Remove a quiz from bank
   - F.2.3 Import quizzes from file
   - F.3.3 Insert a quiz from bank
   - F.3.4 Insert a quiz directly
   - F.3.5 Remove a quiz from test
   - F.3.7 Mix answers choices
   - F.3.8 Save a test
   - F.3.10 Print a test
   - Create hints for answer choice

III. Redundant errors (duplicate requirements):
   - R8 and R11 express the same function “mix quizzes in a test”
   - R10 and R12 express the same function “generate a test from bank”
   - R11 and R12 express the same function “mix quizzes in a test”

IV. Inconsistent errors (contradict requirements):
   - R1 contradicts with R6: “multiple-choice quizzes” and “crossword puzzle”

V. Ambiguous errors (multiple-meaning requirements):
   - R5 expresses two functions: “insert a quiz from bank”, “insert a quiz directly”
   - R7 expresses two functions: “insert a quiz from bank” and “insert a quiz directly”
   - R12 expresses two functions: “generate a test from bank” and “mix quizzes in a test”

VI. Off-topic errors (out of scope requirements):
   - R3: “run in web browsers” is not a function of test editor software.
   - R6: “create a crossword puzzle” is not a function of test editor software, since “each test includes a set of multiple-choice quizzes.”

Figure C2 Correctly detected errors in SRS of test editor.