IEICE TRANS. INF. & SYST., VOL.E97–D, NO.5 MAY 2014

PAPER Special Section on Knowledge-Based Software Engineering

Ontology-Based Checking Method of Requirements Specification

Dang Viet DZUNG[†a)], *Nonmember and* **Atsushi OHNISHI**[†b)], *Member*

SUMMARY This paper introduces an ontology-based method for checking requirements specification. Requirements ontology is a knowledge structure that contains functional requirements (FR), attributes of FR and relations among FR. Requirements specification is 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 and correctly and efficiently revise requirements. To support this method, an ontology-based checking tool for verification of requirements has been developed. Finally, the requirements checking method is evaluated through an experiment.

key words: *requirements verification, requirements ontology, checking requirements specification*

1. Introduction

Requirements specifications should meet quality demands such as: correctness, completeness, consistency, unambiguity [1]. Traditional methods to verify requirements specification are: review, prototyping, model validation, and requirements testing [2]. These techniques require reviewing requirements documents manually, or building models or prototypes of system and checking them, or generating test cases from requirements. With the increasing complexity of nowaday systems and the diversity of application domain knowledges, it is hard to check requirements documents or to build prototypes manually.

Formal methods are effective in requirements verification since they can be automated by tools, but these techniques demand that requirements must be specified formally [3]. It is difficult for stakeholders to learn formal languages in order to understand specification documents. The translation from a natural language to a formal language requires a lot of work-load and is hardly implemented automatically. Although there are many ways to write requirements, using natural languages is still the best for stakeholders. Even diagrammatic presentations still require stakeholders to learn some notations, and not all customers of software systems are willing to learn them.

How to verify requirements specification which are written in natural languages? The format of the requirements documents 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 system being developed (section 3 in IEEE std 830-1998 [1]). The semantic checking of FR demands domain understanding from analysts. Even if requirements analysts are experts in the application domain, there are still problems with checking large requirements specification documents due to complexity of natural languages.

Is it possible to support requirements verification with a technique that utilizes domain knowledge and does not require writing requirements specification in a special notation other than natural languages? Our research aims to use requirements ontology, a representation of domain knowledge, to check the semantics of requirements specification. Problems to be solved in the research are:

1. What is the model of requirements ontology?
2. How to use requirements ontology to detect errors in requirements specification?
3. Which tools should be developed to support the method?
4. How to evaluate the efficiency of the method?

Ontology-based checking method works by comparing requirements specification with FR ontology and using rules to revise requirements, and a tool for this method has been developed [4]. In order to evaluate the efficiency of the method, we conducted a comparative experiment, in which one group of requirements analysts used our method and the other group did not. We expected that by using the method, the number of errors detected in requirements would be more, and the quality of final requirements specification would be improved.

The paper is organized as follows. The next section will describe our requirements ontology model and the method of checking requirements, and illustrate rules and reasoning process. After that, Sect. 3 introduces requirements checking tool that we have developed, then Sect. 4 describes a comparative experiment. Section 5 discusses related researches and compares with our work, and finally, Sect. 6 arrives at a conclusion.

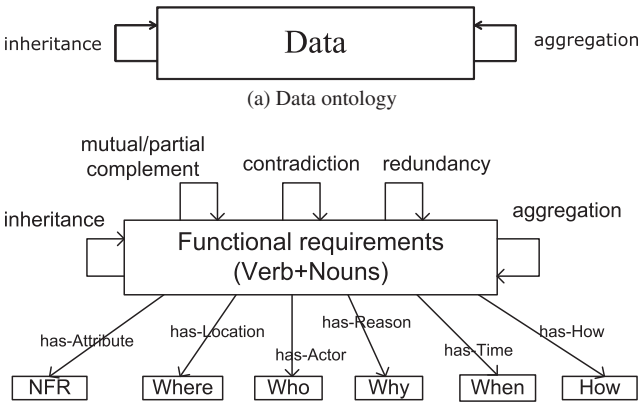
2. Ontology-Based Checking Method

This section firstly introduces requirements ontology model, then it describes a method to detect errors in software requirements specification (SRS) by using reasoning rules.

Manuscript received June 24, 2013.
Manuscript revised October 23, 2013.

[†]The authors are with the Department of Computer Science, Ritsumeikan University, Kusatsu-shi, 525–8577 Japan.
a) E-mail: dungdv@selab.is.ritsumei.ac.jp
b) E-mail: ohnishi@cs.ritsumei.ac.jp

DOI: 10.1587/transinf.E97.D.1028



(a) Data ontology

(b) FR ontology

Fig. 1 Meta-model of requirements ontology.

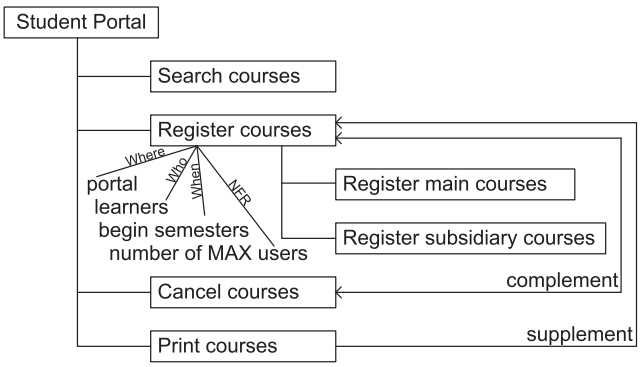


Fig. 2 A part of the FR ontology for a university portal site.

2.1 Requirements Ontology Model

Requirements ontology is a representation of domain knowledge to support requirements engineering. We have two types of requirements ontology: data ontology and FR ontology, as shown in Fig. 1. Data ontology represents a hierarchy and relations of objects in a system, and FR ontology is those of functions in the system. The two ontology are useful in requirements engineering because they represent domain knowledge including data structures and features in the application domain.

FR ontology represents: (1) a functional hierarchy of a certain software system, (2) relations among FR, and (3) attributes of FR. From users' point of view, FR of a system to be built 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 a system. Attributes of FR include: agent (who), location (where), time (when), reason (why), method (how), and non-functional requirements (NFR). FR does not have "what" attribute because "what" is FR itself: FR is specified with verb and its objects which imply action or function, and "what" attribute in this context of requirements ontology also means action or function, so we omit the "what" attribute. Relations among nodes in ontology include: complement, supplement, aggregate, inherit, contradict, and redundant.

Figure 2 shows an example of a part of requirements ontology for a university's portal site. This ontology represents a case where a portal for students has several functions such as "search courses, register courses, cancel courses" and so on. The function register courses has sub-functions: register main courses, and register subsidiary courses. The function "register courses" and the function "cancel courses" have complement relation, so they are always together. If a software allows students to register courses, it should allow students to cancel courses. However, the software might have the function "print courses" since it is supplement to the function "register courses." Figure 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."

Requirements ontology, as in Fig. 2, can be built by some ways. One way is to analyze manual documents of existing software systems in the application domain. 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. This information can be used to build requirements ontology. Another way is to construct a hierarchy of functions and attributes of functions directly by interviewing with domain experts. However, the detail of development of requirements ontology is out of scope of this paper. We suppose that requirements ontology already exists, and use it for checking quality of requirements specification.

2.2 Checking Requirements Using Ontology

In our checking method, software requirements are described as a list of sentences. Each sentence is parsed to get a list of a verb and nouns, then the list is compared and mapped to a node in FR ontology with the help of a thesaurus. For examples, a requirement "Student can register for courses online" is parsed to get a list of a verb and a noun as {V: register, N: courses}, and then mapped to node "register courses" in ontology model, as illustrated in Fig. 3.

After mapping, we use relations among nodes in ontology to find errors in SRS. In Fig. 3, the complement relation between the two functions "register courses" and "cancel courses" suggests that the function cancelling courses should be added to requirements. If only the function "register courses" is present without the function "cancel courses", the requirements list is considered incomplete. Similarly, through supplement relation, the function "print courses" is optionally added. For optional functions, requirements team can discuss with customers whether adding them to requirements specification. After doing so, the requirements list will be more complete.

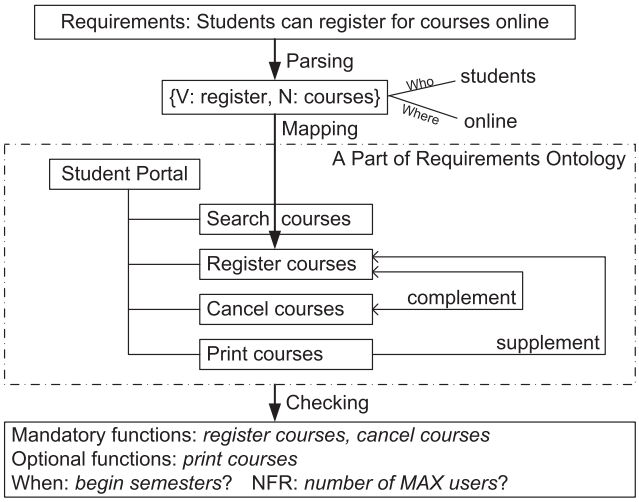


Fig. 3 An example of checking requirements with ontology.

In order to check 4W1H attributes, 4W1H phrases are extracted from sentences in SRS and then compared with 4W1H of the mapped nodes in ontology. For example, after mapping the requirements sentence as in Fig. 3, 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. Requirements also should specify 4W1H attributes if corresponding ontology nodes have them. For example, 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.

We summarize the method of checking requirements specification as four main steps as follows:

1. Parse requirements specification;
2. Map requirements to nodes on ontology;
3. Check requirements using ontology and rules;
4. Interpret results, suggest revising requirements.

In step 3 above, rules are necessary for checking 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 checking method: are people who want to check requirements specification documents; they can be requirements elicitors, analysts, managers, or stakeholders. In this paper, we refer to users of the checking method briefly as analysts.
- Domain independent rules: are applied to requirements belonging to any domains. Domain independent rules are provided in advance.

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().

Fig. 4 Predicates used in rules.

- User definition rules: are specified by analysts. User definition rules include domain specific rules and system specific rules.
- Domain specific rules: are used to check requirements belonging to each specific domain.
- System specific rules: are used to check specific requirements documents of a specific system.

2.3 Rules Definition

Figure 4 introduces some notations to describe rules, e.g., the fact that a function X is in the requirements specification is denoted by inSRS(X).

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 FR already in specification, rules will help to find incomplete, inconsistent, or redundant requirements.

Figure 5 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, but the function "print courses" 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 the 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,

(1) MO(R ↦ X) → inSRS (X)
(2) complement(X,Y) → complement(Y,X)
(3) contradict(X,Y) → contradict(Y,X)
(4) redundant(X,Y) → redundant(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) aggregate(X,Y) ∧ inSRS(Y) → maybeInSRS(X)
(9) inherit(X,Y) ∧ inSRS(X) → shouldbeInSRS(Y)
(10) inherit(X,Y) ∧ inSRS(Y) → maybeInSRS(X)
(11) contradict(X,Y) ∧ MO(R ↦ X) ∧ MO(S ↦ Y) → maybeInconsistent(R,S)
(12) redundant(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)

Fig. 5 Rules for checking requirements.

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.

Rules from (16) to (27) are for checking 4W1H attributes [5]. 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 checking when, where, why, how, and NFR attributes are similar to rules (16) and (17).

Using this checking method, 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 checking, new requirements can be found and can be added to SRS. Analysts can start again another round of checking 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.

<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)
<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

Fig. 6 A grammar for extension of rules.

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. 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. 6. This grammar is in BNF form and uses pre-defined predicates in Fig. 4; it allows to define two common types of business rules: integrity rule and derivation rule [6], [7]; 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'

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:

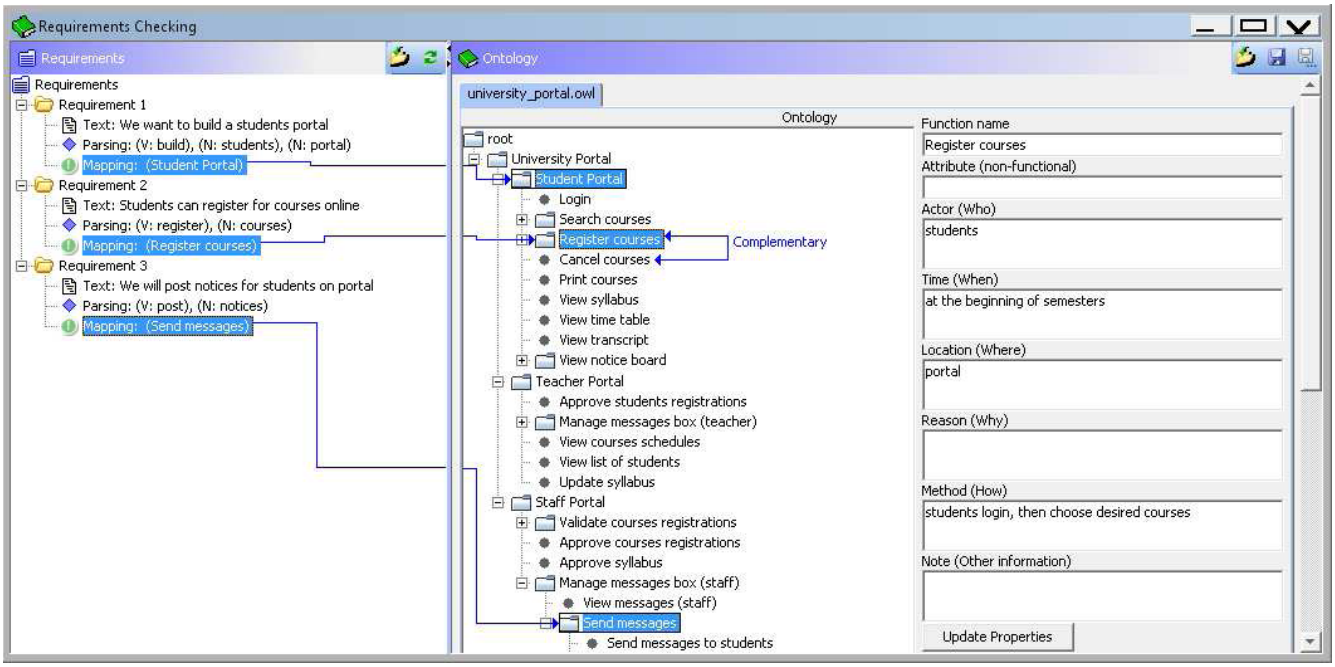


Fig. 7 Screenshot of requirements checking tool.

(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 checks a SRS should not participate in preparing requirements documents, and he/she wants to assure that the SRS satisfy domain properties, so he/she can describe domain properties such as user definition rules for verification of SRS. In case an analyst is a domain expert, he/she still may need supportive rules to check a large requirements specification.

This section have described requirements ontology, rules, and checking method to detect errors in requirements specification. However, with a large ontology and requirements specification, the checking process will require a lot of efforts from analysts. Therefore, a tool is needed to support this method. The next section will introduce the ontology-based checking tool that have been developed.

3. The Requirements Checking Tool

This checking 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 suggestion to revise requirements. Based on these suggestions, analysts will work with customers to revise the requirements list. The tool supports four main steps in ontology-based checking method: (1) Parse requirements specification, (2) Map requirements to nodes on ontology, (3) Check the requirements using ontology and rules, and (4) Interpret the checking results. We will describe these functions in detail in what follows.

Software requirements are often stated in natural language, 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[†] to parse requirements specification. The requirements should be stated using plain language, so it is easier to parse and process. Figure 7 illustrates the working of the checking tool. The left side depicts the requirements

[†]OpenNLP library http://opennlp.apache.org/

for building a portal for students to register courses; the right side is a part of the ontology of EMS. In the left part there are parsing results, e.g., 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:notice}, so that the list to represent the requirements is {post, notice}.

The parsing results in terms of lists of verb and nouns are then used to map requirements to nodes in ontology. 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 [4]. For example, in Web Portal thesaurus that we built, the word 'message' is synonymous with the word 'notice', and the word 'post' is synonymous with the word 'send'. Therefore, we map the list {V:post, N:notice} to ontology node "send messages", as in Fig. 7. In case of too many matches because of using thesaurus, analysts can choose which requirements map to which nodes in ontology.

After mapping requirements to nodes in ontology, we use rules to detect errors in SRS. Our requirements ontology is stored in OWL files (Web Ontology Language), but the reasoning power of OWL language is limited [8], so we use Prolog for checking requirements. The Prolog execution will return the results of mandatory requirements, optional requirements, redundant requirements, and inconsistent requirements. In order to detect off-topic requirements, we use the Prolog command to find requirements which are not mapped to nodes in ontology.

From checking results, analysts propose mandatory requirements and optional requirements to customers. Analysts also pose to them questions concerning inconsistent requirements and redundant requirements. The checking tool can generate questions automatically from checking results by using questions templates. For example, if the tool detects mandatory requirements as "cancel courses", then it generates questions such as: "Do you agree to add the function 'cancel courses' to requirements list?" Similarly, if the tool finds two redundant functions such as: "view message" and "show message", then it generates questions as follows: "The two functions 'view message' and 'show message' are redundant. Which one do you want to remove from requirements?" 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 checking again, until they find no errors in requirements specification.

4. Experiment

In this experiment, we assigned two groups of subjects to working on a same list of requirements, but one group used our method and the other did not. We compare the results of two groups in the number of errors they found, time they used, and the number of FR in the revised list of requirements.

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.

Fig. 8 Initial requirements of QMS system.

4.1 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. 8. Among the four subjects, two subjects worked freely and two subjects were provided with ontology and checking 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.

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 checking tool, whereas subjects S3 and S4 did use them. To illustrate our checking method, we will explain the initial steps of requirements checking by subject S3 in the next sub-section.

4.2 Example of Checking Requirements by Subject S3

Subject S3, after receiving the initial requirements in Fig. 8, 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. 9. 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 FR of QMS system and recommended to be removed from requirements list.

After checking the first round, subject S3 found eight requirements, as shown in Fig. 10. After revising requirements, he continued with a second round of checking. Fi-

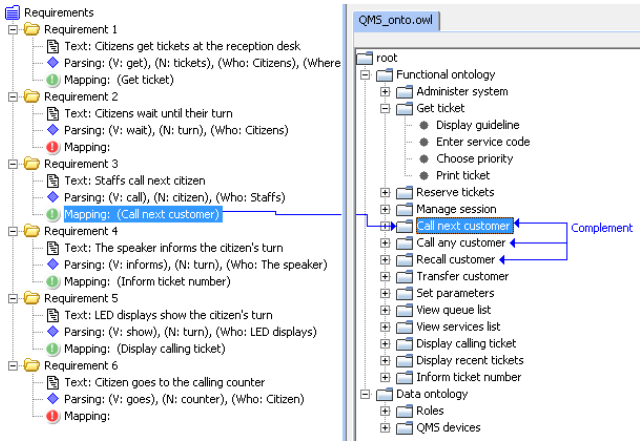


Fig. 9 Mapping initial requirements to QMS ontology.

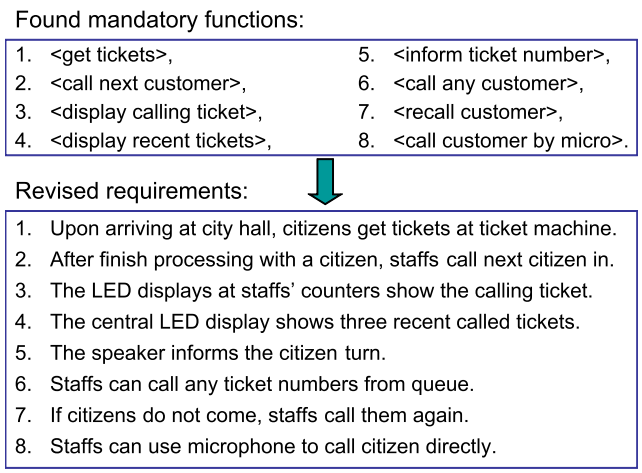


Fig. 10 A part of requirements list elicited by subject S3.

nally, after four rounds he could find total 47 FR. The final requirements list is in the appendix. Following sections will compare used time and results by two groups of subjects and discuss reasons why they had that results, and evaluate the effect of using the checking method in improving quality or requirements specification.

4.3 Time Used to Check Requirements

As described in Sect. 2.3, 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 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 forth round (The number of requirements that each subject found in each round are shown in Sect. 4.5).

We summarize time that each subject used in each round in Table 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 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 checking 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 checking 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 checking tool to generate questions, asked customers directly, then entered answers to the checking 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.

4.4 Errors Found by Two Groups of Subjects

Table 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 1). However, in average time used on each error, subjects without method spent more time than

Table 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 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

Table 4 The number of questions.

	without method		with method	
	S1	S2	S3	S4
Add FR proposals	12	11	63	67
Remove FR proposals	0	0	4	2
4W1H questions	14	25	61	45

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.

Fig. 11 A part of questions from group with method and answers.

Table 5 The number of FR found by two groups of subjects.

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

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 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.

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 checking tool to generate questions. Table 4 lists the number of questions that the customer received from two groups. There were three types of questions received from subjects: proposals to add FR, proposals to remove FR, and questions about descriptions of FR such as 4W1H information. We see that subjects in the group with method generated much more proposal questions to add FR than the other two subjects did. In particular, the group without method did not ask any question to remove FR. 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 11 lists some questions from the group with method, and answers from the customer. Questions five and six are proposals to add requirements; questions 16, 17, and 25 are descriptions of requirements. The checking 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 NFR 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.

4.5 The Number of FR Found by Two Groups of Subjects

Table 5 summarizes the number of FR 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.

Figure 12 shows the intersection of FR lists by two groups. That figure only represents some FR shortly in terms of verb and nouns (The complete FR list are in the appendix). Six FR that found by all subjects (shown in the left text box of Fig. 12) are common functions of QMS system. Therefore, every subject could detect them; even subjects S1 and S2 did not use ontology and checking tool. The 30

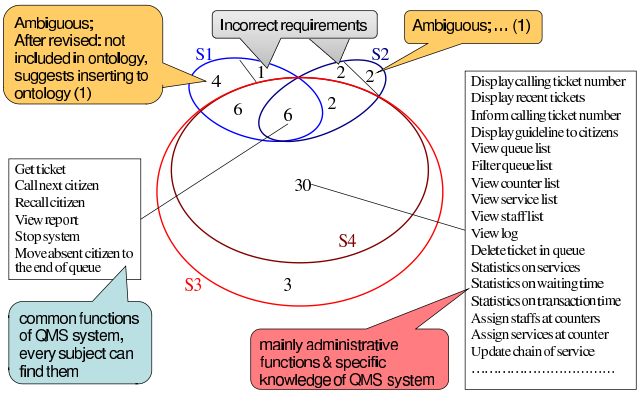


Fig. 12 Intersection of FR lists by two groups of subjects.

FR that the group with method found, but the group without method did not find—in the right text box of Fig. 12—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 FR that the group without method found, but the group with method did not. Of the nine FR, three were incorrect and needed to be eliminated from specification. The other six FR were ambiguous, and needed to be revised. The reason that the group with method did not find these FR is that they were not included in the QMS requirements ontology, so the subjects did not reason out them. It suggests inserting these FR to requirements ontology for later use.

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

Though both subjects S3 and S4 used the 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 checking tool supports automatically parsing requirements, mapping to ontology, checking, and generating questions for revising requirements. It suggests that the checking 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 experiments.

Using checking method, analysts can detect only errors about non-conformance between requirements and ontology. It does not support detection of requirements which are outside ontology. The efficiency of using checking method depends on the completeness and correctness of requirements ontology. Therefore, after built, requirements ontology needs to be checked [9], and improved frequently.

The rules definition is an extension adding capability to the checking method. However, in the experiment, the group using checking method did not define new rules. We expect to evaluate effect of rules extension in future study.

Though having some preliminary results but the experiment scale was still small. We expect to improve and evaluate it by larger experiments in future research.

5. Related Works

There are several related works in requirements engineering using ontology. Some works focused on early stage—the construction of ontology [10]–[12], and some others supported requirements elicitation [13]–[19], and some works related to checking the quality of requirements [20]–[22].

Some works focused on the construction of ontology. Breitman and Leite [10] used language extended lexicon (LEL) to represent terms and phrases in application language, and then proposed framework to construct ontology. Similarly, Zhang, Mei, and Zhao [11] provided feature diagram to represent domain knowledge; Bao et al. [12] proposed maintenance framework for domain ontology with focus on formal representation of process changes.

A number of researchers have explored the usage of ontology to support requirements elicitation, notably Kaiya, Saeki, Kitamura and colleagues [13], [14] proposed ontology-based requirements elicitation method; their method measures the quality of elicited requirements. Kluge et al. [15] described business requirements and software characteristics in terms of ontology, somewhat similar to our method of FR representation by ontology. Their method helps business people to compare and match their FR to functions of available commercial software products, and choose a suitable one. The method by Zong-yong and colleagues [16] divides ontology into multiple stages: domain ontology, task ontology, and application ontology. 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 [17] proposed NFR ontology to be used to discover NFR. Our ontology model also contains NFR and also supports elicitation of those requirements. Xiang et al. [18] divided initial requirements to a list of goal; each goal is narrowed down to a list of sub-goals. Using relation among goals, they can refine initial requirements. Similarly, Liu and colleagues [19] used ontology model consisted of actor, goal, task to do reasoning, but their approach is more formal compared to our method. The above researches focus on elicitation using ontology, but their methods do not support detection of 4W1H errors, redundant errors, and off-topic errors in SRS; especially their methods do not allow user definition rules as our method.

Some literatures related to detection of errors in requirements. Zong-yong and colleagues [20] represented requirements model as UML diagrams and checked the model using domain ontology and rules. Zhu and Jin [21] proposed method to detect inconsistency in requirements using state transition diagram. Kroha and colleagues [22] transform static parts of UML diagrams to ontology and find inconsistency. Our checking method mainly focus on incompleteness (the lacking of requirements might cause big losses in latter phases), but it can detect other errors such as: incorrectness, inconsistency, redundancy. The above three researches focus on checking diagrammatic elements in requirements specification, but they do not focus on improvement of quality of requirements which are stated in natural languages.

6. Conclusion

This paper has illustrated a checking method for detecting errors in requirements specification. The method focuses on FR and supports to find mandatory requirements, optional requirements, inconsistent requirement, and redundant requirements. The checking results are then used to revise requirements documents.

The checking tool supports reasoning about errors and generating questions to suggest revision of requirements. The comparative experiment, in which one group of analysts used checking method and the other did not, suggests that using the method can improve the quality of requirements specification.

Our method can be extended in some directions. Using the parsing technique as in our tool to parse software manual documents, we can obtain information to build ontology. Requirements also can be mapped to ontology using other clues, instead of only verbs and nouns as in our method. The method supports checking with FR ontology, but it still does not utilize data ontology. The checking method can be more formal. If requirements are specified in controlled languages (Controlled languages use limited words and simple grammar), they could be translated into first-order logic and conducted model checking. This method can be used to check the full meaning of requirements specification, and can be applied in critical systems.

References

[1] IEEE, "Recommended practice for software requirements specifications," IEEE Std 830-1998, p.i, 1998.
[2] G. Kotonya and I. Sommerville, Requirements Engineering: Processes and Techniques, p.87, Wiley, 1998.
[3] A. van Lamsweerde in Requirements Engineering: From System Goals to UML Models to Software Specifications, p.202, Wiley, 2009.
[4] D.V. Dzung and A. Ohnishi, "Ontology-based reasoning in requirements elicitation," Proc. 7th IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM'09), pp.263–272, Nov. 2009.
[5] D.V. Dzung and A. Ohnishi, "A verification method of elicited software requirements using requirements ontology," Proc. 19th Asia Pacific Software Engineering Conference (APSEC 2012), pp.553–558, Dec. 2012.
[6] T. Morgan, Business Rules and Information Systems: Aligning It with Business Goals, ch. 3, pp.59–100, Addison-Wesley, Boston, USA, 2002.
[7] R.G. Ross, "Rulespeak – Templates and guidelines for business rules," Business Rules Journal, vol.2, no.5, http://www.BRCommunity.com/a2001/b066.html, 2001.
[8] V. Hirankitti and V.T. Xuan, "A meta-logical approach for reasoning with semantic web ontologies," Proc. 4th IEEE Int. Conf. on Computer Sciences: Research, Innovation and Vision for the Future, pp.229–236, 2006.
[9] B.Q. Huy and A. Ohnishi, "A verification method of the correctness of requirements ontology," Proc. 10th JCKBSE, ed. M. Virvou and S. Matsuura, Frontiers in Artificial Intelligence and Applications, vol.240, pp.1–10, IOS Press, 2012.
[10] K. Breitman and J. do Prado Leite, "Ontology as a requirements engineering product," Proc. 11th IEEE Int. Conf. on Requirements Engineering (RE'03), pp.309–319, Sept. 2003.
[11] W. Zhang, H. Mei, and H. Zhao, "A feature-oriented approach to modeling requirements dependencies," Proc. 13th IEEE Int. Conf. on Requirements Engineering (RE'05), pp.273–282, Sept. 2005.
[12] A. Bao, L. Yao, W. Zhang, and J. Yuan, "Approach to the formal representation of owl-s ontology maintenance requirements," Proc. 9th Int. Conf. on Web-Age Information Management (WAIM '08), pp.56–61, July 2008.
[13] H. Kaiya and M. Saeki, "Using domain ontology as domain knowledge for requirements elicitation," Proc. 14th IEEE Int. Requirements Engineering Conference, RE '06, pp.186–195, 2006.
[14] M. Kitamura, R. Hasegawa, H. Kaiya, and M. Saeki, "A supporting tool for requirements elicitation using a domain ontology," in Software and Data Technologies, pp.128–140, Springer Berlin Heidelberg, 2009.
[15] R. Kluge, T. Hering, R. Belter, and B. Franczyk, "An approach for matching functional business requirements to standard application software packages via ontology," Proc. 32nd IEEE Int. Conf. on Computer Software and Applications (COMPSAC '08), pp.1017–1022, Aug. 2008.
[16] L. Zong-yong, W. Zhi-xue, Y. Ying-ying, W. Yue, and L. Ying, "Towards a multiple ontology framework for requirements elicitation and reuse," Proc. 31st IEEE Int. Conf. on Computer Software and Applications (COMPSAC '07), pp.189–195, July 2007.
[17] G. Dobson, S. Hall, and G. Kotonya, "A domain-independent ontology for non-functional requirements," Proc. IEEE Int. Conf. on e-Business Engineering (ICEBE'07), pp.563–566, Oct. 2007.
[18] J. Xiang, L. Liu, W. Qiao, and J. Yang, "Srem: A service requirements elicitation mechanism based on ontology," Proc. 31st IEEE Int. Conf. on Computer Software and Applications (COMPSAC '07), pp.196–203, July 2007.

[19] L. Liu, Q. Liu, C. hung Chi, Z. Jin, and E. Yu, "Towards a service requirements ontology on knowledge and intention," Proc. 6th Int. Conf. on Quality Software (QSIC'06), pp.452–462, Oct. 2006.
[20] 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.
[21] X. Zhu and Z. Jin, "Detecting of requirements inconsistency: An ontology-based approach," Proc. 5th Int. Conf. on Computer and Information Technology (CIT'05), pp.869–875, Sept. 2005.
[22] P. Kroha, R. Janetzko, and J.E. Labra, "Ontologies in checking for inconsistency of requirements specification," Proc. 2009 Third Int. Conf. on Advances in Semantic Processing, SEMAPRO '09, pp.32–37, 2009.

Appendix: Final List of FR 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 citizen's 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*

Dang Viet Dzung received B. of Engineering degree from Hanoi University of Science and Technology in 2003, and M. of Engineering degree from Ritsumeikan University in 2009. He was a Researcher at Vietnam National University from 2003 to 2007, and from 2009 to 2011. Currently he is a doctor course student at Ritsumeikan University. His current research interests include requirements engineering, and ontology-based verification. Dzung is a student member of IEEE.

Atsushi Ohnishi received B. of Engineering, M. of Engineering, and Dr. of Engineering degrees from Kyoto University in 1979, 1981, and 1988, respectively. He was a Research Associate of Kyoto University from 1983 to 1989 and an Associate Professor of Kyoto University from 1989 to 1994. Since 1994 he has been a Professor at Departmentof Computer Science, Ritsumeikan University. He was a visiting scientist at UC Santa Barbara, California, U.S.A. from 1990 to 1991 and also a visiting scientist at Georgia Institute of Technology, Georgia, U.S.A. in 2000. His current research interests include requirements engineering, object oriented analysis, and software design techniques. Dr. Ohnishi is a member of IEEE Computer Society, ACM, IEICE, Information Processing Society (IPS) Japan, and Japan Society for Software Science and Technology (JSSST).