Studies on Generation of Software Test Cases by Using Natural Language Processing

March ２０１７

Satoshi Masuda

Studies on Generation of Software Test Cases by Using Natural Language Processing

Graduate School of Systems and Information Engineering

University of Tsukuba

March ２０１７

Satoshi Masuda

Abstract

Software supports our society such as e-mail, online banking system, internet shopping, smartphone, and so on. We require high quality of software and high speed of software development for changes in our businesses and activities. Software testing consumes about 40% of total software life cycle process. However, software testing is not processes efficiently and effectively. Software testing efficiency and effectiveness depends upon the creating test cases. Engineers create test cases from specification documents for software testing. The creating test cases activities depend upon engineer's skills. Hence, test cases are often missed by human skills. In this paper, we focus on generation of test cases from specification documents by using natural language processing.

Software testing often targets natural language specification documents. Creating test cases depends on engineer skills, then automation of creating test cases from natural language specification is important. Logics retrieval is a required technique to automate creating test cases, because once logics are retrieved we can transform them into decision tables and also create test cases from the decision tables. Furthermore, Japanese language structure is different from English. If we target Japanese natural language, a new technique is also required. We propose a Semantic Analysis Technique of Logics Retrieval for Software Testing from Japanese Public Sector's Specification Documents. This technique is a new logics retrieval from harmonization between natural language processing technique and software testing. Applying the analysis technique to total 25 files, 1,218 pages and a million double bytes characters, the precision reached 0.93 to 0.97 and recall reached 0.65 to 0.79.

Decision table testing is a technique to develop test cases from descriptions of conditions and actions in software specification documents. We propose, experiment and evaluate a semantic role labeling technique of conditions and actions for automatic software test cases generation. Our approach uses natural language processing to select sentences from the specification based on syntactic similarity, and then to determine conditions and actions through dependency and case analysis. We got experiment results that precision reached from 0.901 to 0.988, recall reached from 0.946 to 0.974 for different style of descriptions, and the workload was reduced to one-sixth

of manual work. Our results on case studies show the effectiveness of our technique.

Software testing has been one of the important area for software engineering to contribute high quality software. Decision table testing is a general technique to develop test cases from information about conditions and actions from software requirements. Extracting conditions and actions from requirements is the key for efficient decision table testing. We propose, experiment upon, and evaluate the syntactic rules of conditions and actions for automatic software test cases generation. Our approach uses natural language processing to select sentences from the requirements on the basis of syntactic similarity, and then to determine conditions and actions through dependency and case analysis. Experiments revealed that F-measure reached from 0.70 to 0.77 for different style of descriptions. The results on case studies further demonstrate the effectiveness of our technique.

In the early phases of the system development process, stakeholders exchange ideas and describe requirements in natural language. Requirements described in natural language tend to be vague and include logical inconsistencies, whereas logical consistency is the key to raising the quality and lowering the cost of system development. Hence, it is important to find logical inconsistencies in the whole requirements at this early stage. In verification and validation of the requirements, there are techniques to derive logical formulas from natural language requirements and evaluate their inconsistencies automatically. Users manually chunk the requirements by paragraphs. However, paragraphs do not always represent logical chunks. There can be only one logical chunk over some paragraphs on the other hand some logical chunks in one paragraph. In this paper, we present a practical approach to detecting logical inconsistencies by clustering technique in natural language requirements. Software requirements specifications (SRSs) are the target document type. We use k-means clustering to cluster chunks of requirements and develop semantic role labeling rules to derive "conditions" and "actions" as semantic roles from the requirements by using natural language processing. We also construct an abstraction grammar to transform the conditions and actions into logical formulas. By evaluating the logical formulas with input data patterns, we can find logical inconsistencies. We implemented our approach and conducted experiments on three case studies of requirements written in natural English. The results indicate that our approach can find logical inconsistencies.

We apply our technique to generation test case from Unified Modeling

Language (UML). UML is the language of modeling from requirements for software design. UML Testing Profile (UTP) is the definition of the modeling test from requirements analysis for software testing. UTP has Test Architecture, Test Behavior, Test Data, and Time Concepts as the test models. Requirements are described in natural language, and engineers who have modeling skills then manually generate test models. Hence the generation of test models depends upon the engineer's skills, leaving the quality of test models unstable. In this paper, we present automatic generation test models from requirements in natural language by focusing on descriptions of test cases in UTP test behavior. We develop three rules to generate test models from requirements by using natural language processing techniques and experiment with our approach on requirements in language that is considered natural English. Our results in three case studies show the promise of our approach.

We focused on generation of test cases rather than the skills and developed a method for the automatic generation of test cases by using our natural language document analysis techniques which use text parsers for extracting and complementing parameter values from documents. We applied the method to Internet banking system maintenance projects and insurance system maintenance projects. We discuss our method and techniques for automatic generation of test cases and their use in these industry case studies. Our document analysis tool helped automatically generate 95% of the required test cases from the design documents. The work of creating test cases was reduced by 48% in our case studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software supports our society such as e-mail, online banking system, internet shopping, smartphone, and so on. We require high quality of software and high speed of software development for changes in our businesses and activities. Software life cycle process consists of requirement analysis, design, implementation, testing and so on [708]. Software testing accounts for 40% of total software life cycle process [IPA09]. Software testing is also activity of getting better quality of software. However, software testing is not processes efficiently and effectively [IPA09]. Activities of software testing consist of planning, preparing, executing and reporting [715]. The preparing activity of software testing includes creating test cases. Software testing efficiency and effectiveness depends upon the creating test cases. Engineers create test cases from specification documents for software testing. The creating test cases activities depend upon engineer's skills. Hence, test cases are often missed by human skills. There are problems in creating test cases from specification documents. In this paper, we focus on generation of test cases from specification documents by using natural language processing. We discuss studies about:

- rules of creating test cases by using natural language processing at section 3.1
- syntactic analysis as pre-processing as improvement creating test cases for both Japanese and English specification documents at section 3.2 and 3.3
- detecting logical inconsistencies in natural language requirements as

further study at section 4.1

- as applications of our technique, creating test cases from UML (Unified Modeling Language) document at section 5.1 and combine our technique and combinatorial testing at section 5.2

At first, we propose a semantic analysis technique of logics retrieval for software testing from Japanese public sector's specification documents. Japanese language has some different structure from English language. This technique is an approach to apply natural language processing to solve the problems in creating test cases. Harmonization between natural language processing techniques and software testing a key area to determine how we direct to develop test cases. We think this is a design of test architecture. This approach is categorized to the design of test architecture and also this retrieval logics technique is one of the test requirement analysis.

Testing the application software which used in corporate activities is important for the business of the companies that use them. Test cases are generated from the software requirements or design documents related to system testing and user acceptance testing. The requirements are developed and shared among stakeholders and are written in natural languages. Unfortunately, this means that test case generations in software testing techniques[ISO15b] are depend too much upon the person doing the testing having sufficient knowledge and skills to understand the natural language requirements and generate test cases. Under these circumstances, missing important test cases become an issue. One of the solutions to this problem is to write requirements in formal languages, but it is quite difficult for all stakeholders to fully understand the meanings of the requirements in formal languages. Another solution is to analyze natural languages requirements mechanically, which is attractive because the mechanical analysis of requirements does not depend upon personal knowledges or skills.

At second, we target software requirements written in Japanese. We focus on the "conditions" and "actions" in decision table technique[ISO15b], and propose syntactic rules of extracting test cases from software requirements by using techniques of natural language processing and tree kernel techniques. The pre-processing step of our rules is to construct knowledge of base sentences that includes conditions and actions descriptions. We select sentences from requirements by calculating the syntactic similarity [TIM02] between each sentence in the requirements and the base sentences. Our rules target

natural language requirements, so the results may depend on the style of descriptions. We therefore evaluate how our rules apply to other requirements having different description styles. We then work on improving the accuracy improvement of generating conditions and actions from the requirements.

Testing the application software which used in corporate activities is important for the business of the companies that use them. Test cases are generated from the software requirements or design documents related to system testing and user acceptance testing. The requirements are developed and shared among stakeholders and are written in natural languages. Unfortunately, this means that test case generations in software testing techniques[ISO15b] are depend too much upon the person doing the testing having sufficient knowledge and skills to understand the natural language requirements and generate test cases. Under these circumstances, missing important test cases become an issue. One of the solutions to this problem is to write requirements in formal languages, but it is quite difficult for all stakeholders to fully understand the meanings of the requirements in formal languages. Another solution is to analyze natural languages requirements mechanically, which is attractive because the mechanical analysis of requirements does not depend upon personal knowledges or skills.

At third, we target software requirements written in English. We focus on the "conditions" and "actions" in decision table technique[ISO15b], and propose syntactic rules of extracting test cases from software requirements by using techniques of natural language processing and tree kernel techniques. The pre-processing step of our rules is to construct knowledge of base sentences that includes conditions and actions descriptions. We select sentences from requirements by calculating the syntactic similarity between each sentences in the requirements and the base sentences. Our rules target natural language requirements, so the results may depend on the style of descriptions. We therefore evaluate how our rules apply to other requirements having different description styles. We then work on improving the accuracy improvement of generating conditions and actions from the requirements.

At other point of view as forth, logical consistency is the key to raising the quality and lowering the cost of system development. Although formal languages like UML and SysML can be used to describe specifications of documents, stake holders, in the early phases of the system development process, use natural language to exchange ideas, design products, and define requirements because natural language can describe their ideas better. Under such circumstances, the descriptions of the requirements are likely to be

vague and inconsistent. Hence, it is important to find logical inconsistencies at this early stage. Formal languages offer a number of techniques for evaluating the logical consistency of requirements. In verification and validation of the requirements, there are techniques to derive logical formulas from natural language requirements and evaluate their inconsistencies automatically. Here, there are a number of techniques to find logical inconsistencies in natural language requirements. For instance, a framework has been proposed for handling inconsistencies in natural language requirements [GZ05] by using a natural language parser to generate logical formulas. However, users still have to determine how to chunk the requirements into pieces in which to search for logical inconsistencies. For example, they may chunk paragraphs into sentences. However, the logic of these sentences is not independent; that is, in paragraphs written in natural language, logical aspects in one sentence relate to logical aspects in other sentences. This situation suggested to us that we should cluster chunks of requirements. At forth, we present a practical approach to detecting logical inconsistencies by clustering technique in natural language requirements. Software requirements specifications (SRSs) are the target document type. Functionality descriptions of SRS are the main target descriptions. Our approach can be applied on other types of descriptions, however, detecting logical inconsistency for functionality descriptions is the more important. We developed semantic role labeling rules to derive "conditions" and "actions" as semantic roles from requirements by using natural language processing. We also constructed an abstraction grammar to transform the conditions and actions into logical formulas. By evaluating the logical formulas with input patterns, we can find logical inconsistencies.

We implemented a proof-of-concept prototype of our framework. We used the natural language processing parser [MMM06] and conducted dependency analysis on the natural language requirements. Semantic role labeling, abstraction grammar, and evaluation of logical formulas were implemented using our own methods developed using the natural language tool kit [Pro15]. We evaluated our prototype on three case studies: "a detailed system design specification for the coordinated highways action response team (CHART) mapping applications" [Adm03], "business requirements specifications of legal notice publication (eNotification)" [fE12], and "comprehensive watershed management water use tracking (WUT) project software requirements specification" [Dis04].

We have two applications as applying of our research. One is about application for "Automatic Generation of UTP Models from Requirements in

Natural Language" . Model Driven Development (MDD) and Model Based Testing (MBT) are solutions for software quality. Unified Modeling Language (UML) is the language of modeling from requirements for software design [OMG15]. UML Testing Profile (UTP) is the definition of the modeling test from requirements analysis for software testing. UTP has test architecture, test behavior, test data, and time concepts as the test models. The requirements are described in natural language. In current business situations, UTP models are manually generated from requirements. Engineers who have modeling skills then manually generate test models. Hence the generation of test models depends upon the engineer's skills, leaving the quality of test models unstable. Automatic generation of UTP models from requirements is a solution to this problem. We present our approach, Automatic Generation of UTP Models from Requirements in Natural Language. We performed experiments and evaluations with two case studies, and the results show the promise of our approach.

We implemented a proof-of-concept prototype of our approach. We used the English Natural Language Processing parser [DMM08] and dependency analysis on natural language requirements. We developed our own tools using the Python natural language tool kit.

We present automatic generation test models from requirements in natural language by focusing on test case descriptions in UTP test behavior. We develop three rules to generate test models from requirements by using natural language processing techniques and experiment with our approach on requirements in language that is considered natural English. Our results in three case studies show the promise of our approach.

We can use our approach to find vague requirements and provide feedback in the early stages of the system development process. In addition, we will construct new rules and grammar for requirements descriptions. We will contribute to requirement engineering by developing new means to check whether descriptions have vague or inconsistent requirements.

Another application is for "Automatic Generation of Test Cases Using Document Analysis Techniques". Software testing requires high test case coverage [CDFP97] as software becomes large and complex [CKI88]. Currently test cases are most often created manually, so test case coverage depends upon each individual skill [CDPP96]. In software maintenance, software testing consumes 55% of the total software maintenance work [IPA09],[RAF+10]. The problem is how to reduce software testing work while still insuring high quality of software. Some solutions involve software execution automation

tools [CGP+06],[SB10], outsourcing the testing tasks at lower labor rates. Such solutions still depend upon each individual skill in testing software. In contrast, we focused on generation of test cases than the skills and developed a method for the automatic generation of test cases by using our natural language document analysis techniques which use text parsers for extracting and complementing parameter values from documents. We applied the method to Internet banking system maintenance projects and insurance system maintenance projects. In this paper, we discuss our method and techniques for automatic generation of test cases and their use in these industry case studies.

The method targets functional testing for Web application systems. The method uses text parsers to identify parameter values for pairwise testing by using our document analysis tool for the design documents with boundary analysis and defects analysis, thus avoiding the dependencies upon individual skills. The document analysis tool uses natural language processing which is a technique for modeling the logic of the documents and testing the analysis [NTIM11],[NT11],[SPTN10],[TSN+07]. We discuss case studies that demonstrate the method of automatic generation of test cases using document analysis techniques.

Chapter 2

Issues on generation software test cases from specifications in natural language

2.1 Issues on specifications in natural language and generating test cases

Many researchers already tried to apply natural language processing to software development and/or software testing [TN99, KY02]. A related work [Sne07] presented testing against natural language requirements. The approach was to analyze requirements and extract test cases from them. The paper illustrated the approach on an industrial application. The paper did not mention about techniques of natural language processing and the target language is English.

One of other related works [SHE89] presented software development process from natural language specification. That was an approach to solve problems about natural language specification by the process which was defined as "design" and "elaborate". The approach was not to try automatic generation f test cases by using techniques natural language processing. A national standard [CB98] recommend practice for software requirements specifications. The standard describes consideration for producing a good software requirements specification, parts of them and provide templates. Another related work [KKS08] presented measurement of level of quality control activities in software development. The ambiguity definition was described in

the paper. When we target natural language requirements, these two papers are very effective at the point of view, how requirements should be described and how measure them.

About decision table testing, a related work [UMT13] presented an efficient software testing method by decision table verification. That was to verify logics between documents and source codes by comparing each of decision tables which were extracting from documents and codes. They targeted the document was described formal language.

We target Japanese natural language in one of our technique, then we surveyed language research papers. A related work [Kun73] presented differences between Japanese language and English language like as words dependency differences, syntax trees tend to deep on left side and so on.

A related work [Fuk88] presented deriving the differences between English and Japanese on a case study in parametric syntax. That was a model of solution the differences between English and Japanese language toward universal grammar. These papers are very essential to establish Japanese natural language processing.

However, we don't focus on the differences Japanese language, because Japanese natural language processing technique has already solved the differences in their Japanese morphometric analysis and/or statistical analysis. We focus on applying the results of Japanese natural language processing into software testing.

When we target Japanese language, a related work [TKN93] presented retrieve logical structure from Japanese legal documents. Another work [Kat07] also proposed legal engineering [KNS08] and researched them. In legal documents, a related work [Shh12] proposed methodology for designing trustworthy social system by Japanese legal document analysis. These Japanese legal document analyses are helpful for our problem. The big difference between them and us is that they target legal documents which are described more formal than system requirement documents. Legal documents are written by efficient experts who have licenses, not to have ambiguity, comprehensiveness, consistency and so on. System specification documents are written by many people who are not language specialist.

Based on these related works, we propose our semantic analysis technique of logics retrieval for Japanese system specification documents.

There are various problems when it comes to using NLP to extract conditions and actions for decision table testing. A related work [MIH+15] proposed a technique to generate logical relation as "If (CS) is (CE), (AS) is

(AE)." from a natural language requirement by using NLP. This analysis technique retrieves logics, namely, a condition stub (CS), condition entry (CE), action stub (AS) and action entry (AE), from each sentences in requirements specifications. In this technique, however, only one condition and action are generated, not multiple ones. Another approach of extracting conditions and actions is to translate Japanese natural language into logical formula. A related work in this approach demonstrated tool of automatic translation from Japanese natural language into well-formed formulas on an extended predicate logic [TKI12]. This work, however, there are issues that dictionary and translation map between sentences and logical formula are required. Therefore, we put this issue as future work.

We propose our rules to differentiate the deep cases, by surface cases and dependencies that are generated by natural language processing. There are sometimes illogical or vague sentences contained in natural language requirements in industrial examples. For these sentences, we select sentences from the requirements by calculating the structural similarity between each sentences in the requirements and the base sentences. Sentences that define software functions are sometimes listed in the requirements. There are techniques for multiple sentences analysis and contextual analysis[MWM+11]. In this paper, we target on a sentence one by one.

The related work of NLP in software testing has also been discussed. The semantic analysis technique of logics retrieval for software testing from specification documents is a logics retrieval technique derived from harmonization between a natural language processing technique and software testing [MIH+15]. Natural language processing and consistency checking are essential parts of requirement engineering. A formal consistency check of specifications has been written in natural language [AS12]. This "requirement consistency maintenance framework" produces consistent representations. The first part is an automatic translation from the natural language describing the functionalities to a formal logic with an abstraction of time.

Natural language processing and consistency checking are essential parts of requirement engineering. A related work presented a formal consistency check of specifications written in natural language [YCC15]. This "requirement consistency maintenance framework" produces consistent representations. The first part is an automatic translation from the natural language describing the functionalities to formal logic with an abstraction of time. It extends pure syntactic parsing by adding semantic reasoning and support for partitioning input and output variables. The second part uses synthesis

techniques to determine if the requirements are consistent in terms of realizability [YCC15]. Our framework differs from the work [YCC15] as follows: it creates abstraction logic by transforming propositional logic not only time constraints, it uses input data patterns to find logical inconsistencies and perform semantic role labeling. It uses a SAT solver to check the validity and consistency of the logical constraints [HPSS06]. The validity and consistency are really two ways of looking at the same thing and each may be described in terms of syntax or semantics [BHvM09]. It uses combinatorial testing to deal with large numbers of data patterns. Combinatorial Testing (CT) can detect failures triggered by interactions of parameters in the software under test (SUT) with a covering array test suite generated by some sampling mechanisms [NL11]. There is a method for creating test patterns using pairwise selection from the parameter values. The method uses a knowledge base for identifying pair-wise parameter values by using document analysis, boundary analysis and defects analysis [MMT13].

It was described a way to change natural sentences into logical expressions and devised a tool for English [Bos08]. A related work [MIH+15] used natural-language processing to identify the logical pattern "If (A) is (B), (C) is (D)" in sentences, but did not identify conditions or actions. There are also linguistic studies on use cases [SPKB09] and test case generation [WPG+15]. The related work [Sne07] made test cases from requirements described in natural language. This study showed how to interpret specifications as descriptions of inputs and outputs. It didn' t include any morphological analysis or syntactic analysis/analyses using natural language processing. IEEE 830 recommends practices for software requirements specifications [CB98], [ISO15b]. The standard describes the considerations that go into producing a good software requirements specification and provides templates. A related work [KKS08] presented a way of measuring the level of quality control in software development. They defined the notion of ambiguity. A related work [UMT13] presented an efficient software testing method that verifies the logical consistency of the document and source code by comparing decision tables created from them. They targeted documents written in formal language, not natural language.

Various research and industry related works are discussed. We can summarize them by from which artifacts, (e.g., requirements in natural language, UML, use cases, etc.) to generate which artifacts (class diagrams, test cases, etc.), automatically or manually.

An approach to generate test cases from use cases describes a com-

plete process to generate test cases from use cases for web applications. This process also resolves the shortcomings detected in existing approaches [GEMT06]. Automatic test case generation from UML models is a novel approach for generating test cases from UML design diagrams [MFIMA11]. They consider the use case and sequence diagram in our test case generation scheme. Our approach consists of transforming a UML use case diagram into a graph called a use case diagram graph (UDG) and a sequence diagram into a graph called a sequence diagram graph (SDG) and then integrating UDG and SDG to form the System Testing Graph (STG) [MFIMA11].

A survey discusses the test case generation from the UML state machine diagram. It discusses a comparative study of the test case generation techniques from the UML state machine diagram that has been done. That study is an attempt to discover works that have already been done in this field and to realize various aspects of UML state machines that have been considered for generating test cases [SM07]. A related work which targets requirements in natural language is the automatic generation of system test cases from use case specifications [KSKD13]. They develop Use Case Modeling for System Tests Generation (UMTG), an approach that automatically generates executable system test cases from use case specifications and a domain model, the latter including a class diagram and constraints [MFIMA11].

Fundamental research about NLP in model driven development has been conducted. UML and UTP are discussed in model driven development (MDD) and model based testing (MBT). An assist system designed to automatically translate text-based specifications into formal models has been proposed [MOASHL09]. An idea for model driven engineering based on natural language processing has been developed and applied in some case studies. In the requirement engineering research area, a methodology combining models and controlled natural language has been discussed. The methodology combines the advantages of model-based and NL-based documentation by means of a bidirectional multi-step model transformation between both documentation formats. They illustrate the approach by means of an automotive example, explain the particular steps of the model transformation, and present performance results [AS12].

There is research about the transformation of natural language requirements to UML class diagrams. There is an industrial software test case study using UTP [BSBV13]. The case study is transforming from the requirements statements to an intermediary frame-based structured representation using dependency analysis of requirements statements and the grammatical knowl-

edge of Patterns. The knowledge stored in the frame-based structured representation is used to derive class diagrams using a rule-based algorithm. Their approach has generated similar class diagrams as reported in earlier works on the basis of linguistic analysis with either annotation or manual intervention [SSB15].

NLP in software testing has been also discussed. The semantic analysis technique of logics retrieval for software testing from specification documents is a logics retrieval technique derived from harmonization between a natural language processing technique and software testing [MIH+15]. Natural language processing and consistency checking are essential parts of requirement engineering. A formal consistency check of specifications has been written in natural language [AS12]. This "requirement consistency maintenance framework" produces consistent representations. The first part is an automatic translation from the natural language describing the functionalities to formal logic with an abstraction of time.

It extends pure syntactic parsing by adding semantic reasoning and support for partitioning input and output variables. The second part uses synthesis techniques to determine if the requirements are consistent in terms of being realizable. Our approach differs from as follows [AS12]: it creates abstraction logic by transforming propositional logic, not only time constraints; it uses input data patterns to find logical inconsistencies and perform semantic role labeling. It uses a SAT solver to check the validity and consistency of the logical constraints. The validity and consistency are really two ways of looking at the same thing, and each may be described in terms of syntax or semantics.

Natural language processing and checking consistency are essential in requirement engineering. A related work [YCC15] present formal consistency checking over specifications in natural languages. The paper present: a requirement consistency maintenance framework to produce consistent representations. The first part is the automatic translation from natural languages describing functionalities to formal logic with an abstraction of time. It extends pure syntactic parsing by adding semantic reasoning and the support of partitioning input and output variables. The second part is the use of synthesis techniques to examine if the requirements are consistent in terms of realizability [YCC15]. This paper presents the differences from the work [YCC15] as follows, abstraction logic by transforming propositional logic not only time constraints, using input data patterns to find logical inconsistency and semantic role labeling.

2.2 Issues on detecting logical inconsistencies in specifications

SAT solver is a solution for checking logical constraint [HPSS06]. SAT solves about validity and consistency. The validity and consistency are really two ways of looking at the same thing and each may be described in terms of syntax or semantics [BHvM09]. Combinatorial testing is the solution for data patterns. Combinatorial Testing (CT) can detect failures triggered by interactions of parameters in the Software Under Test (SUT) with a covering array test suite generated by some sampling mechanisms [NL11]. There is a method of creating testing pattern for Pair-wise method by using knowledge of parameter values. The method uses knowledge base for identifying pair-wise parameter values by using document analysis to specification documents, boundary analysis and defects analysis [MMT13].

The work [MIH+15] shows the logic to which is "If (A) is (B), (C) is (D)." from a sentence by natural-language processing, but the condition and actions are not identified. There is also linguistic work for use cases [SPKB09] and test case generation [WPG+15]. There is a related work for the study which makes test case from a described required specification by a natural language [Sne07]. The approach which defines classification of how to interpret the specification such as which is description of output or which is description of input as key words. There aren't a morphological analysis and syntactic analysis using analysis technology of natural-language processing, and this can think there are few generalities. There are other related works. Another work offers the way to change from a natural sentence to a logical expression and tool BOXER in English [Bos08].

2.3 Chapter Summary

We raise issues about applying natural language processing to generation of software test cases by referring related works as follows:

- Not generation test cases but only definition a methodology of testing process even by using natural processing.

- Translation to logical formula from Japanese natural language is not available.
- No rules are existing to extract information of test cases from specification documents in natural language.
- Logical inconsistencies exists in specification documents which are develop by stakeholders.

These issues are not solved related works. That is motivation for us to study about generation of software test cases by using natural language processing. Our study can apply to check logical inconsistency on specification documents. We discuss

- rules of creating test cases by using natural language processing at section 3.1
- syntactic analysis as pre-processing as improvement creating test cases for both Japanese and English specification documents at section 3.2 and 3.3
- detecting logical inconsistencies in natural language requirements as further study at section 4.1
- as applications of our technique, creating test cases from UML (Unified Modeling Language) document at section 5.1 and combine our technique and combinatorial testing at section 5.2

Chapter 3

Semantic role labeling for generation test cases

3.1 Semantic Analysis Technique of Logics Retrieval for Software Testing from Specification Documents

3.1.1 Background and Motivation

Formal languages like as UML, SysML are proposed to describe specification of documents. It is already known the techniques of retrieval logics from the documents described these formal languages like as model based testing. However, in the IT industry, especially enterprise system like as back office system, these formal languages are not popular. Stakeholders often use their natural language to exchange their idea, business processes, business rules and other specifications and describe the specifications into documents by their natural language. Under this situation, it has been proposed an approach to apply natural language requirements into system and acceptance testing [Sne07]. This approach is to apply their original processing rules which don't use natural language processing standard techniques like as morphological analysis and structural analysis. Thinking about reuse the research results of natural language processing, it is required another technique which is according with standard natural language processing techniques. In this section, we propose a semantic analysis technique of logics retrieval for software testing from Japanese public sector's specification documents as

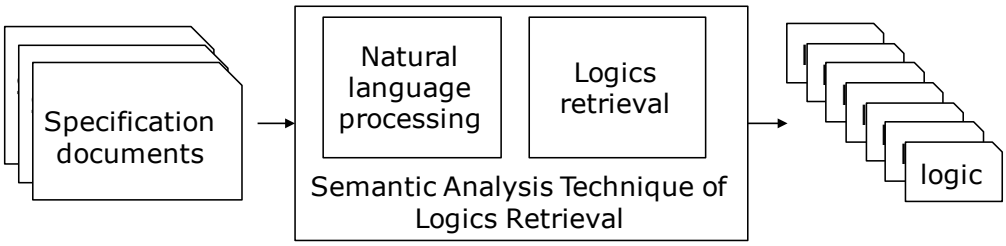

Figure 3.1: The analysis technique retrieves logics from specification documents

figure 3.1.

The research between software testing from specification documents and Japanese natural language processing is insufficient. The work [Sne07] approach is for English language. There are still high demands of IT services including software testing in Japan. Japanese GDP is third of the world and IT businesses are still growing on Japanese language. Then there are opportunities for applying Japanese natural language processing to system development and test. As people know there are differences between Japanese language and English about character, structure and grammar. There has been software development process from natural language process [SHE89]. This is not software testing point of view.

The analysis technique retrieves logics from each sentences in specification documents as condition stub (CS), condition entry (CE), action stub (AS) and action entry (AE). These logic elements come from decision table definition [Ass86]. The logic can be described as (3.1).

$$If(CS)is(CE), (AS)is(AE). \tag{3.1}$$

We define rules and algorithm for logics retrieval. In further research, if we set more rules, the recall ratio will grow up. In this paper, we target the Japanese public sector's specification documents. The reasons are the business size of Japanese public sector's system development and test reached 2.2 billion Japanese yen that was about 20% of total Japanese IT services market [Kis03], and we can get the specification documents of them from the internet.

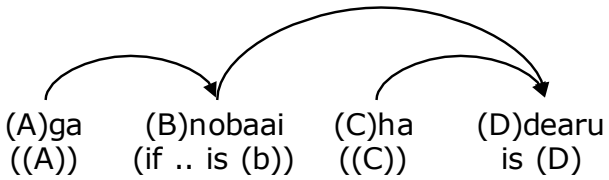

Figure 3.2: A condition logic dependency

3.1.2 Semantic Analysis Technique of Logics Retrieval

We discuss the analysis technique. First of all, we use sample specification sentence (a) and explain the analysis technique intuitively. When we have a sentence (a), let us follow for retrieving logics steps from the sentence. The sentence (a) shows (a)-1 Japanese sentence described by alphabet characters and (a)-2 English sentence which means as same as the Japanese sentence for both languages.

(a) Sample specification sentence 1. Japanese: *Miraini betsuno detaga haitteitabaai, sono jitennno tyokuzennwo shuuryoubitosurukoto.*

2. English: *If another data exists in a future field, set a date just before the data as end date.*

The objective of the analysis technique is to retrieve a logic as figure 3.2 from this sentence. Figure 3.2 shows a structure of logic as "if (A) is (B) then (C) is (D)" in English, and "(A) ga (B) nobaai (C) ha (D) dearu" in Japanese. This logic describes the branch logic of specification. Branch logics are important for test cases. When we retrieve the branch condition, we can put each branch elements into decision table elements and transform into test cases.

Table 3.1 shows a result of Japanese natural language processing techniques which includes morphological analysis and dependency analysis for sample sentence. By using this result

Figure 3.3 shows the steps to retrieve the logic from sentences as follows:

- Require: Sentences have been morphological analyzed and dependency parsed as Table 3.1.
- Step 1: Search (B) words by keyword pattern matching. In this paper we set a rule that (B) Words have "baai" in Japanese which means "if" in English.

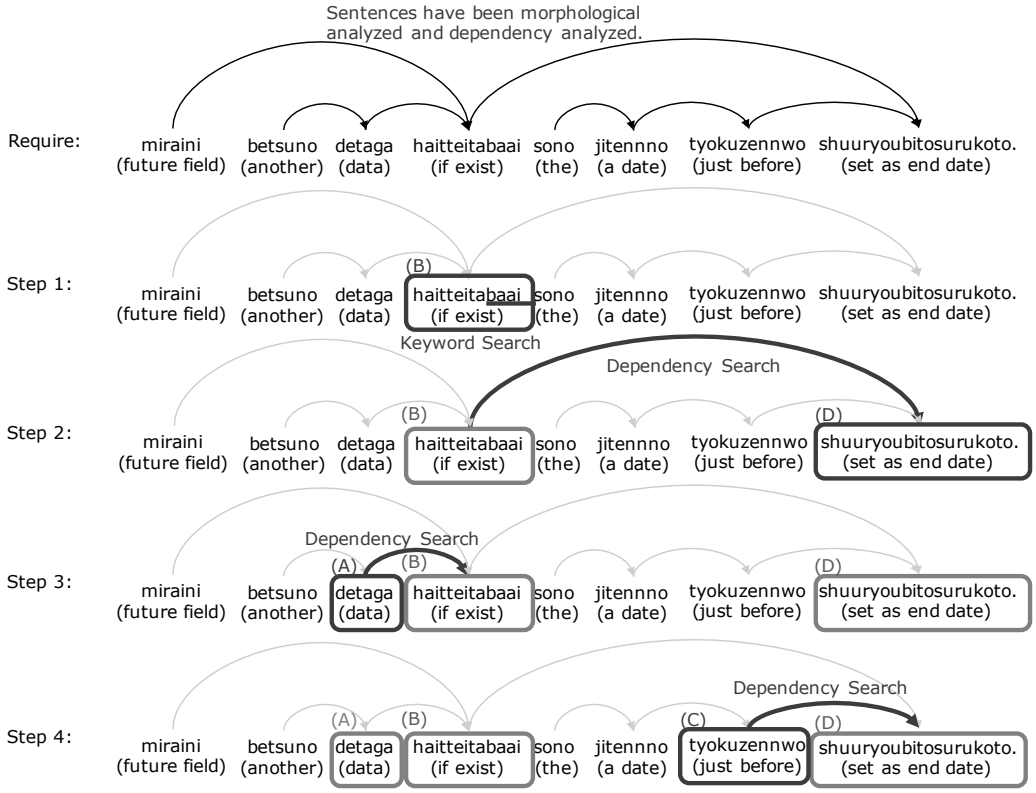

Figure 3.3: Steps to retrieve logics from sentences

- Step 2: Search (D) words which (B) depend on.
- Step 3: Search nearest (C) word which depends on (D)
  - Nearest word has strongest dependency with the target word.
- Step 4: Search nearest (A) word which depend on (B)
  - As same as step 3, nearest word has strongest dependency with the target word.

Then we can get the condition logic from the sentence as "if (A) is (B), (C) is (D)". The definition of decision table [Ass86] is as Figure 3.4.

Once we retrieve a condition logic, we can map the logic into decision table as Table 3.2.

Table 3.1: Structure of Input Data Which is The Results of Japanese Morphological Analysis and Dependency Analysis

i	Pm(i) [Japanese (English)]	Depm(i)
1	miraini (future field)	4
2	betsuno (another)	3
3	detaga (data)	4
4	haitteitabaai (if exist)	8
5	sono (the)	6
6	jitenno (a date)	7
7	tyokuzennwo (just before)	8
8	shuuryoubitosurukoto (sets end date)	T

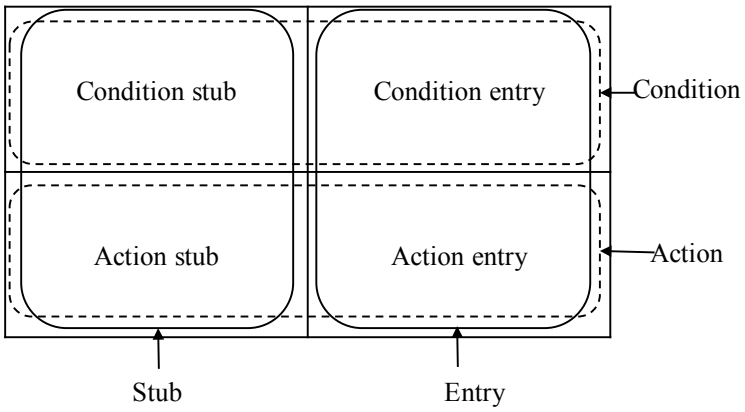

Figure 3.4: Decision tabel definition [Ass86]

Table 3.2: Logic Mapping Into Decision Table Definition

Condition stub (A)	Condition entry (B)
Action stub (C)	Action entry (D)

Decision tables are used in many decision support domains, such as business decision supporting systems, software engineering or system analysis (or evaluation). Decision tables are a simple and important powerful tool to provide reasoning in a compact form [TM07]. After then, we define the analysis technique formally as follows:

- All target documents are D_{All}.
- Each sentences are D_1, D_1, $\cdots D_m$.
  - $D_{All} = \{ D_1, D_1, \cdots D_m \}$
- Dependency structure of phrase P which come from morphological analysis and dependency paring.
  - $D_m = \{ P_m(1), P_m(2), \cdots P_m(n) \}$
  - Dependency patterns of D_m are $\{ Dep_m(1), Dep_m(2), \cdots Dep_m(n-1) \}$. $Dep_m(i)$ is phrase number which the phrase $P_m(i)$ depends on.
- K is the set of key words which search for condition words
  - $K = \{ K_1, K_2, \cdots K_l \}$
  - In this paper, K=K_1="baai"(if).
- Terminal symbol of sentences is T.
- According with decision table definition, condition logic descriptions are:
  - Condition stub is CS
  - Condition entry is CE
  - Action is AS
  - Action entry is AE
- Formulas of retrieving condition logic are:
  - $FI(K, P_m(i))$. $P_m(i)$ is the first phrase which includes K. This $P_m(i)$ is defined as CE.

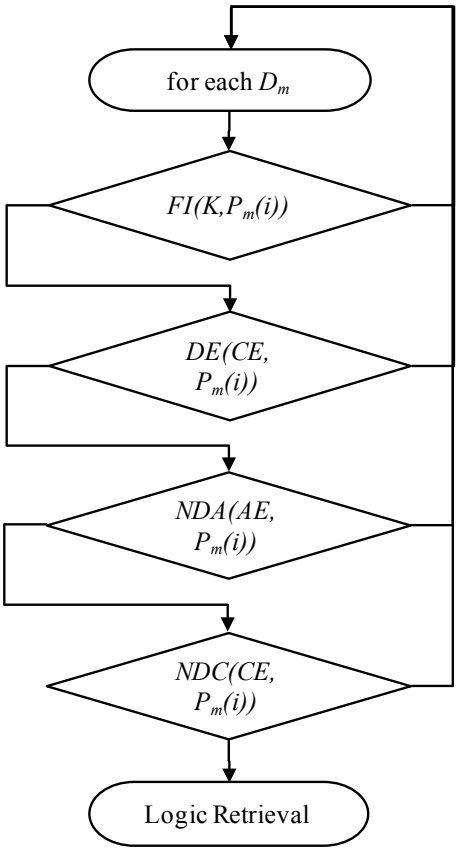

Figure 3.5: Flowchart of The analysis technique

- *DE(CE, $P_m(i)$).* $P_m(i)$ is phrase which *CE* has dependency on, and end of phrases. This $P_m(i)$ is defined as *AE*.
- *NDA(AE, $P_m(i)$).* $P_m(i)$ is the nearest phrase which has dependency on *AE*. This $P_m(i)$ is defined as *AS*.
- *NDC(CE, $P_m(i)$).* $P_m(i)$ is the nearest phrase which has dependency on *CE*. This $P_m(i)$ is defined as *CS*.

Flowchart of the analysis technique is as Figure 3.5.

3.1.3 Experiments

We experimented the analysis technique by creating algorithm, developing programs and applying Japanese specification documents. Algorithm 1 shows how to retrieve condition logic from documents. We have implemented the analysis technique algorithm on Perl program. We also used Japanese morphological analysis and dependency analysis. The analysis technique is independent from the technique and tools of morphological analysis and dependency analysis.

Algorithm 1 The analysis technique algorithm

Require: Input: documents which have been morphological analyzed and dependency parsed

Ensure:

```
1:  for all D_m do
2:      for all P_m(i) do
3:          if P_m(i) = K1 then
4:              CE_i = i
5:              P_m(i) = CE
6:              if P_m(Dep_m(i)) ∈ D_m and P_m(Dep_m(i)) ∈ T then
7:                  AE_i = i
8:                  P_m(Dep_m(i)) = AE
9:              end if
10:         end if
11:     end for
12:     for all P_m(i) do
13:         if Dep_m(i) = CE_i and max(i) then
14:             P_m(Dep_m(i)) = AS
15:         end if
16:         if Dep_m(i) = AE_i and max(i) then
17:             P_m(Dep_m(i)) = CS
18:         end if
19:     end for
20: end for
```

Table 3.1 shows the structure of input data which is results of morphological and dependency analysis. The i is assigned number for each words,

$P_m(i)$ is phrase of the results of morphological analysis, and $Dep_m(i)$ is the results of dependency analysis. Table 3.3 shows the document list of this experiment. The documents are opened to public on the Internet. The documents come from almost Japanese government and cities as public sector. Japanese government order their software and systems to public in order for IT services company can bid the order. Then the documents are system specification documents that IT services company can estimate the function and calculate their costs. In this experiments, we grouped documents from A to F. We use the group A and B to initial experiment to verify our analysis technique and tuning it. Once we fixed the analysis technique algorithm and Perl program, we applied the program from group C to F.

Software testing experts who has 20 years software testing experience evaluate the results of the analysis technique. The points of evaluation are if they retrieve logics manually they compare the analysis technique results and their manual ones by comprehensiveness, correctness of phrases, and can be transform into decision table. Table 3.4 shows the results of the analysis technique versus evaluation by positive and negative. For example, when the analysis technique retrieves condition logic and software testing experts the logic is good, then the retrieval logic is count for positive-positive. Another example, when the analysis technique excludes a sentence but software testing experts the logic evaluate the sentences must be retrieved, then the logic is count for negative-positive.

Table 3.5 shows the results of experiments using recall and precision factors. The recall and precision are calculated by equitation (2) and (3) using (a) to (c) on table 3.5. As table 3.5 shows, the precision reached 0.93 to 0.97 and recall reached 0.65 to 0.79. That means the analysis technique can retrieve condition logic.

$$Precision = \frac{(a)}{((a) + (b))} \tag{3.2}$$

$$Recall = \frac{(a)}{((a) + (c))} \tag{3.3}$$

3.1.4 Evaluations

We observe about reasons of negative evaluation of the results as (b), (c) and (d). In case (b), the analysis technique is positive and evaluation is

Table 3.3: Target Documents List

Document Groups	Pages	Characters (Double Bytes)	File Size (Bytes)	Create date
A	93	74559	1702400	2010/9/16
A	17	3283	4555776	2014/3/18
A	14	9502	60416	2008/6/1
A	15	11182	51712	2014/7/1
A	76	63908	1008129	2003/3/1
B	10	6825	50688	2014/6/1
B	2	779	97280	2007/11/14
B	22	14560	499712	2008/4/1
B	17	11211	54272	2006/9/1
B	19	6844	79360	2010/5/1
B	9	7844	140800	2014/1/1
C	157	152568	1995521	2012/11/1
C	56	55738	609407	2005/7/1
C	73	70167	2775933	2011/7/6
D	11	9060	37888	2012/4/1
D	41	38527	653312	2012/4/1
D	327	296451	6210032	2012/4/1
E	25	20364	880640	2009/12/2
E	8	6717	77312	2012/4/1
E	7	9852	313871	2014/4/1
E	10	8035	72192	2013/5/1
F	10	4419	90624	2011/10/1
F	8	5012	10223	2013/4/1
F	173	188919	1172389	2007/4/1
F	18	26466	618214	2013/4/1

Table 3.4: Results The Analysis Technique Vs. Evaluation

The analysis technique	Positive		Negative	
Evaluation	Positive (a)	Negative (b)	Positive (c)	Negative (d)
A	31	1	15	2
B	15	1	4	3
C	43	2	17	4
D	62	5	33	21
E	35	1	19	6
F	107	8	40	26

Table 3.5: Results of Recall and Precision

Document Groups	A	B	C	D	E	F
Precision	0.97	0.94	0.96	0.93	0.97	0.93
Recall	0.67	0.79	0.72	0.65	0.65	0.73

negative. When there are parentheses, alphabet and other special characters in phrases which have dependency to CE, the morphological analysis engine may determine the characters are individual phrases. In this case, if the logic is retrieved the phrases has not understandable. They are just parentheses characters. In the case (c) that the analysis technique negative and evaluation positive, there are some reasons. First, morphological analysis did not analyze it as expected. For example, when there is a comma "," just before CE the phrase including condition keywords, the analysis engine determines there are no dependency to the CE. Second example is also a morphological analysis. When next phrase of CE is a gerund (in English -ing noun), the analysis engine determines CE depends on the gerund as a verb that is not collect dependency. Japanese morphological analysis and dependency analysis depends on which method they implement the analysis engines.

The analysis technique can use any engine if they derive phrases from sentences and output the dependency among phrases. The (c) categorized sentences are compound sentences, complex sentences, listing sentences and so on. Current algorithm ignores with these kind of sentences because the condition phrase dependency cannot be determined. In the case of (d) that the analysis technique results negative and evaluation also results negative. The sentences categorized in this (d) are original descriptions are incorrect.

The (d) cases are candidates for our future works to feedback of manual description on specification documents. When experts didn't understand the logics of the sentences, it must be something incorrect in the sentences. Then it is important for us to investigate (d) case for future work.

3.2 Semantic role labeling for automatic software test cases generation

3.2.1 Applying Japanese Natural Language Processing Techniques to Decision Table Testing

(a) Decision Table Testing Technique

Decision tables consists of conditions and actions. The conditions and actions are deep cases of the requirements in the natural language processing. In this paper, we propose our rules to differentiate the deep cases from surface cases and to identify the dependencies that are generated by natural language processing. Decision table testing uses a model of the logical relationships between conditions and actions for the test item [ISO15a] in the form of a decision table[ISO15b]. Generating appropriate conditions and actions for this decision table testing technique, it depends largely upon personal knowledges and skills to generate conditions and actions. Our approach is to generate conditions and actions mechanically by natural languages processing. Multiple conditions and actions mean there are a lot of restrictions, for example, if a condition has a particular value, another condition or action must have that some particular value. Generating such restrictions is required to access other domain knowledge. As this generation is outside of scope of this paper, it remains topic of future research.

(b) Natural Language Processing

Natural language processing (NLP) techniques include parsing, morphological analysis, and so on and are used in the analysis of software requirements. There are four steps in NLP:

1. Morphological analysis: This is to parse a sentence to words and tag their parts-of-speech.

2. Dependency analysis: This is to determine dependencies of the words that have been parsed as the result of morphological analysis.
3. Semantic analysis: This is to determine the semantics of the words and phrases.
4. Context analysis: This is to perform analysis over multiple sentences.

The relationship between words is called case and there are surface case and deep case. Surface case is determined by syntactic and deep case is determined by semantic. When a sentence has a case between noun phrase and verb phrase, it is semantic role labeling to determine which case [Oku10].

A related work [TTMM10] demonstrated their technique that they determined semantic role from thesaurus of predicate argument structure for Japanese verbs. In this chapter, our semantic role labeling is determine the cases are "action" or "condition" in decision table testing technique when a sentence has a case between noun phrase and verb phrase. For examples, Japanese surface case has "wo-case", "ga-case". Both cases represent objective cases. Our semantic role labeling is to determine the cases are "action" or "condition" by dependency and evaluation rules of the dependency.

3.2.2 Semantic Role Labeling for Extracting Conditions and Actions

(a) Overview of the Extracting

Figure 3.6 shows our syntactic rules of extracting test cases from software requirements. As the first step, it calculates syntactic similarity between base sentence structure and each sentences in the requirements. Sentences are selected as suitable for extracting conditions and actions. The accuracy of the extraction is raised by this selection. The base sentence structure means parts and dependencies of the sentence. There are two types of sentences, suitable and not suitable for extracting conditions and actions. Suitable sentences progress go to the next step and non-suitable ones are sent back to the requirements developers. This selection process is useful for the developers to improve the descriptions. As the second step, it extracts such as conditions and actions from the results of dependency analysis and case analysis as shown in Figure 3.6.

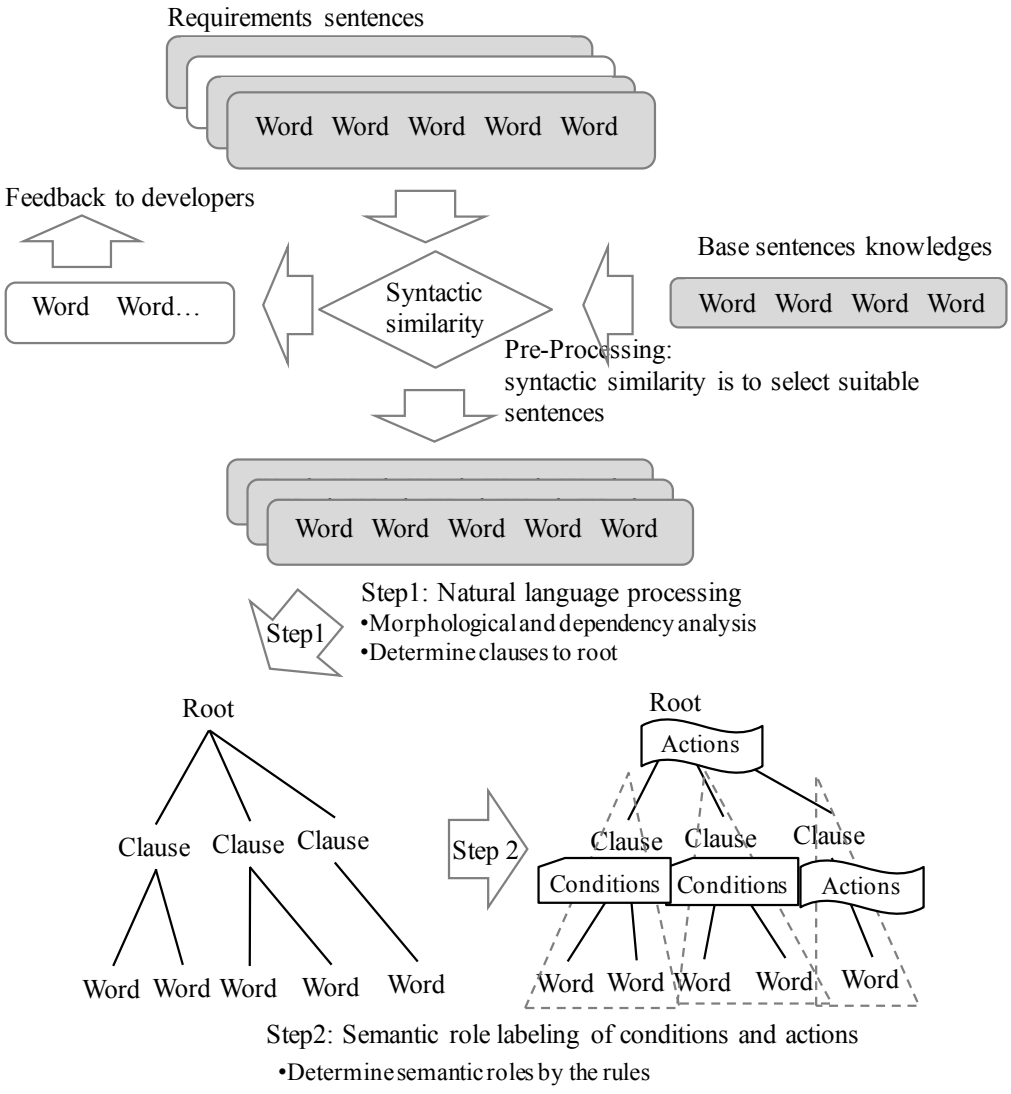

Figure 3.6: Extracting conditions and actions

(b) Syntactic Similarity

The objective of this syntactic similarity is to select suitable sentences for extracting conditions and actions by comparing base sentence structure and each sentences in requirements. In this paper, we use the base sentence as “If the age is more than 20, the entrance fee 1000 yen will be displayed.” (「年齢が２０歳以上の場合、入場料１０００円を表示する。」 in Japanese) as an example. This sentence is suitable sentence for extracting conditions as “If the age is more than 20,” and actions as “the entrance fee 1000 yen will be displayed.” Figure 3.7 shows the base sentence. The base sentence knowledge is a collection of the base sentences. The knowledge is described in a tree structure consisting of parts and their dependencies that have conditions and actions. In this paper, we define the base sentence is as Figure 3.8. BNF descriptions of the base sentence are as follows:

<base sentence>::=

(

(<common noun><case-marking particle>)

(<numeral><noun counter word><noun suffix><particles>)

(<adverbial noun><comma>)

)

(

(<Sa-hen noun>)

(<common noun>)

(<numeral><noun counter word><case-marking particle>)

)

(<Sa-hen noun><Sa-hen verb><period>)

Sentences that are similar to the base sentence, such as “If the number is more than ten, the charge will be three dollars.”, can be selected by calculating the syntactic similarity between them.

We use tree kernel technique[CMB11] to calculate syntactic similarity. Tree kernel is a well-known technique to calculate the similarity of a syntax

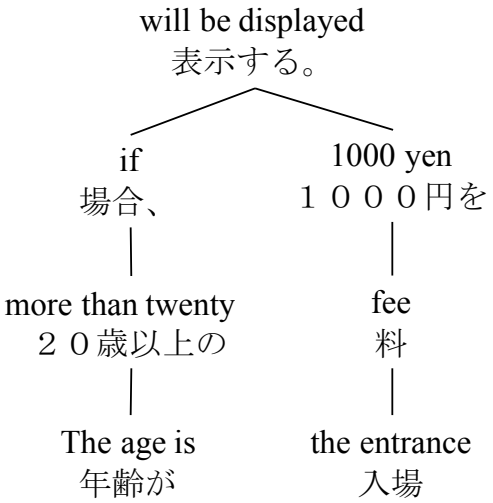

Figure 3.7: The dependencies in the base sentence

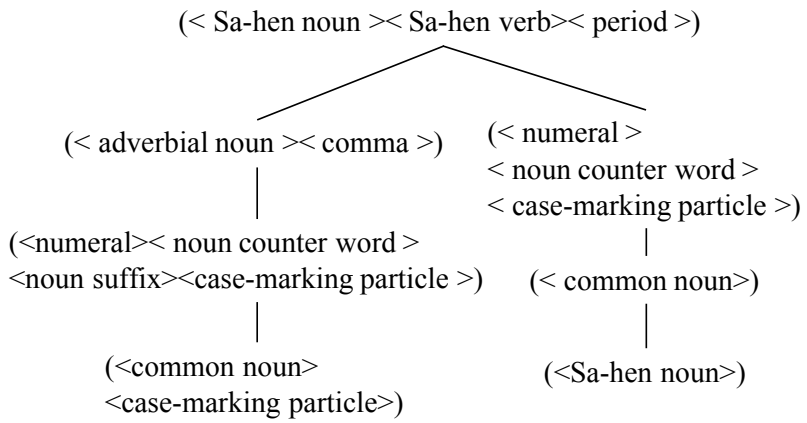

Figure 3.8: Dependency in the base sentence as parts

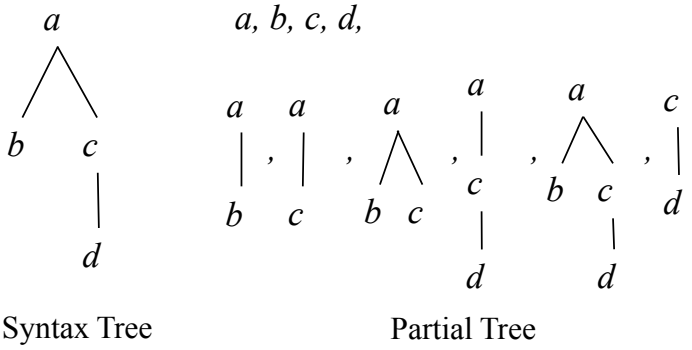

Figure 3.9: Syntax tree and subtrees on tree kernel

tree. Figure 3.9 shows how we calculate the numbers of subtrees between the syntax trees by separating subtrees from the syntax tree [TIM02]. The formula 3.4 shows the cosine similarity that is used to calculate inner products in their tree kernels [CMB11] [OW13].

$$\cos\theta = \frac{k(Ti, Tj)}{\sqrt{k(Ti, Ti)k(Tj, Tj)}} \tag{3.4}$$

Ti, Tj are syntax trees. k is kernel of them. θ is an angle of Ti, Tj .

(c) Extraction of Conditions and Actions

We assume that requirements are described semi-formally in order to share the requirements information in the stakeholders. Examples of requirements descriptions are given below.

- “If the number of stocks got below the criteria, we order the products.” (「在庫量が設定値を下回った場合発注を行う.」in Japanese)
- “If bookings are conflict, error messages are displayed.” (「重複予約が発生した場合, その内容をエラーメッセージとして表示する.」in Japanese)

These sentences of requirements have phrases which depend upon terminal phrase as “If ～, then ～.” Phrases in a sentence depend upon the following phrase except phrases which depend upon terminal phrase [ZO95].

Hence, there are relations between phrases depending upon terminal phrase and the terminal phrase. In this paper, we define that action meanings are action phrases and the others are condition phrases in phrases which depend upon terminal phrase. These relations are extracted from the results of case analysis by natural language processing. In the results of cases analysis, we label ga-case and wo-case in Japanese grammar have "action" meanings by the definition of their case. We also label wo-case as "action" in test cases even if wo-case do not have action meaning when wo-cases depend upon transitive verb.

The case analysis method [KKed] describe there are "continuous cases" and "in-phrase cases" The "continuous cases" are results of cases analysis for modifier of a non-inflectable words. The "in-phrase cases" are determined to compound noun except head basic phrases. Therefore, we label "continuous cases" and "in-phrase cases" as action, because they depends upon the terminal phrase. We label the other cases as condition.

Semantic role labeling for extracting conditions and actions is defined as follows:

- S is the sentence.
- i is the number of each words parsed from S.
- $m(i)$ is a word parsed from S.
- T is the number of root of S.
- $m(T)$ is the root of S.
- $D(i)$ is the word number of $m(i)$ depending on a word.
- $C(i)$ is the case information of $m(i)$ depending on a word.
- $B(i)$ is a clause constructed from all words depending on $m(i)$ as the clause ending word.
- Cn indicates "conditions" and Ac indicates "actions" as attributes of B.
- R is a set of extracting test case rules.
  - Ra is the subset of action labeling rules.

* *Ra1*: Terminal phrases.
* *Ra2*: The cases are wo-case, ga-case, continuous case, in-phrase case as action.

– *Rc* is the complementary subset of condition labeling rules.
* *Rc1*:not *Ra*.

Algorithm 2 shows the algorithm of extracting conditions and actions.

Algorithm 2 Algorithm of extraction conditions and actions

Input: results of morphological, syntactic and semantic analysis
1: Search terminal phrase *m(T)*
2: Search phrases *m(i)*:*D(i)*=*T* which depend upon *m(T)*
3: for all *m(i)*:*D(i)*=*T*
4: Create phrases *B(i)* which have *m(i)* as a terminal phrase.
5: if *C(i)* include *Ra*
6: then *B(i)* are labeled “action”
7: else *B(i)* are labeled “condition”
8: end if
9: end for

Figure 3.10 shows an example of extracting conditions and actions.
An example (1)：

- “When life insurance contracts are executed renewal or to add more options after January 1st 2012 even if the contracts are performed before December 31 2011, the new law category of tax deduction is applied to the whole contracts after the date of renewal.”

  (「平成 23 年 12 月 31 日以前に締結した契約であっても，平成 24 年 1 月 1 日以後に更新・特約中途付加などを行った場合は，異動日以後，契約全体に対して新制度の控除区分が適用されます」in Japanese) (Editing based on [NTAed])

There are four cases and phrases which depend upon the terminal phrase “is applied” in this example sentence by the results of morphological and syntactic analysis in Japanese. The phrase “When” has not-yet case, the phrase “after” has non-yet-case, the phrase “to” has complex predicative modifier, and the phrase “category” has ga-case. By labeling rule *Rc1*, *Ra1*,

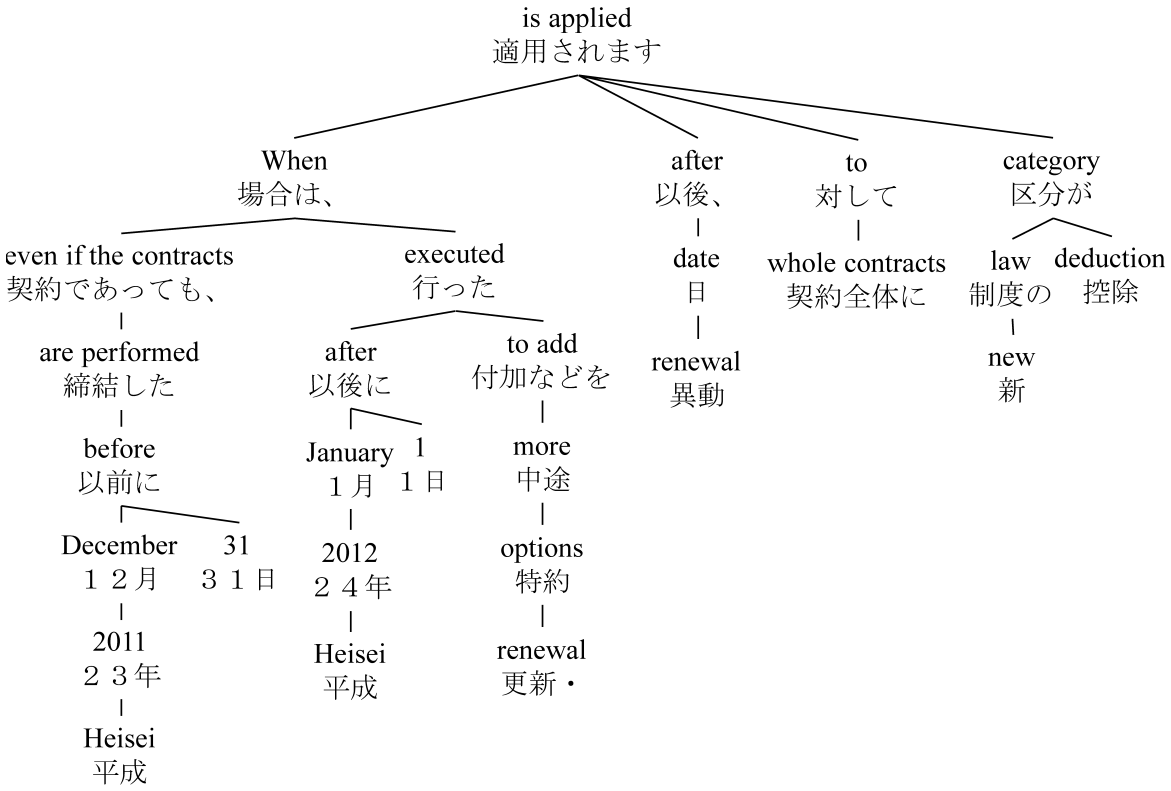

Figure 3.10: Dependency tree of an example sentence

Ra2, the three phrases including “When” , “after” and “to” are labeled as condition. The two phrase including “category” and “is applied” are labeled as action. We label condition and action same as an example sentence.

3.2.3 Experiments

(a) Implementation

We implemented for a proof-of-concept prototype of our approach. We used a Japanese natural language processing parser [KK] and dependency analysis on natural language requirements. We used tree kernel algorithm [TIM02] for comparing syntactic similarity. We developed our own tools using the Python natural language tool kit [Pro15] for the extracting test cases and evaluating the results.

Our technique does not depend upon a specific implementation of natural language processing. The results of dependency and case analysis, however, sometimes different by their dictionary of language and analysis algorithm. We will future work about experiments and evaluation for the differences.

We experimented with two cases: one that has no with syntactic similarity pre-processing step and one that does. We define the threshold of selection sentences by syntactic similarity pre-processing as 0.30 from the typical value in NLP Table 3.6 shows the validity this threshold by comparing the syntactic similarity and software testing results by experts.

We examined two writing styles of documents as our experiments. One of the writing style of documents is use case description for system design documents. This type includes 871 sentences. We call this use case description type documents is document type A[1]. Another writing style of documents is functional description on procurement documents in public sector as fire station system design document [YCed]. This type of documents are available on the internet. This type includes 711 sentences. We call this use case description type documents is document type B. Table 3.7 shows a list of the document type B.

Examples of sentences in each type of documents is as follows:

- Example sentences of document type A:
  - “The system get files specified by users and open attachment

[1]Documents type A is not published due to their confidentiality

Table 3.6: Comparing the syntactic similarity and software testing by experts

Requirements sentences examples	Experts results	Syntactic Similarity
If the age is more than 20, the entrance fee 1000 yen will be displayed. (年齢が２０歳以上の場合、入場料１０００円を表示する)。	OK	1.000
If the number is more than 10, the delivery fee 500 yen will be charged. (個数が１０個以上の場合、配送料５００円を徴収する)。	OK	1.000
If the logon user is not in the target company, the system display that the user is not valid. (ログオン・ユーザーが対象会社ではない場合、システムは、対象ユーザーではない旨を表示する。)	OK	0.594
If the user is more than 20 years old and female, the fee 500 yen will be displayed. (年齢が２０歳以上で女性の場合、５００円を表示する。)	OK	0.563
Users login and input intranet id and password. (ユーザーは、ログインし、イントラネットＩＤおよびパスワードを入力する。)	OK	0.500
If the delivery date is within 3 days, the additional fee 800 yen will be charged and other workers will be arranged. (納期が３日未満の場合、追加料８００円を徴収し、係員が手配をする。)	OK	0.313
If the person is replaced at that time, user will use “Maint-Person” and need to change the person who issue it. (その際に担当が代わる場合は、「担当メンテ」を使用し、発行の担当者名を変更する必要あり。	NG	0.292
Display a list of patients information who are processed in the day (doctors mainly use them in their examination room). (当日受付確認処理された患者情報一覧を表示すること (主に医師が診察室で使用する)。	NG	0.271
Manage schedule (ordered time, accepted time, blood sampling time, etc.). (時間管理が出来ること (オーダー時間、受付時間、採血時間等)	NG	0.200
When it terminates with nothing execution → Termination. (何もせずに終わる場合→終了。)	NG	0.125

Table 3.7: List of document type B

Title of document (in Japanese)	Author (in Japanese)
System operation manual of hiring management system (求人情報管理システム操作マニュアル)	JS Izumo, Shimane prefecture (JS 出雲島根県)
System design document of fire station operation support system (消防業務支援システム基本設計書)	Yokohama city (横浜市)
Specification of vaccination management system development entrustment (予防接種業務支援システム開発業務委託仕様書)	Yokohama city (横浜市健康福祉局健康安全課)
Specification of Kyoto city municipal hospital information system (京都市立病院総合情報システム仕様書)	Kyoto city municipal hospital (京都市立病院)
Specification of Kyoto prefecture municipal library (tentative) information system (京都府立新総合資料館（仮称）統合情報システム 図書系システム機能要求仕様書)	Kyoto prefecture (京都府)
Attachment1: System design overview document of pension management system v1.0 (別紙 1 年金業務システム 基本設計書 第 1.0 版 概要)	Ministry of health, labor and welfare (厚生労働省)
Specification of criminal information web system development (ＷＥＢ公開型犯罪情報システム開発等業務仕様書)	Hiroshima prefecture police department (広島県警察本部)
Specification of Kofu city community support system (甲府市地域包括支援センター支援システム仕様書)	Kofu city (甲府市)
Specification of Kochi prefecture municipal library and Kochi city municipal library information management system development entrustment (高知県立図書館・高知市民図書館新図書館情報システム等基本設計委託業務仕様書)	Kochi prefecture and Kochi city (高知県と高知市)
Specification of new library information management system development entrustment (新図書館情報システム構築等委託業務仕様書)	Kochi prefecture and Kochi city (高知県と高知市)
Specification of Kochi city building evaluation system (RFP) (高知市家屋評価システム仕様書 (RFP))	Finance department, Kochi city (高知市財務部資産税課)
Specification of nutrition management system software (『栄養管理システムソフトウェア一式』仕様書)	National rehabilitation center for persons of disables(国立障害者リハビリテーションセンター)
Requirement definitions of Saga city school attendance support system (佐賀市　学齢簿・就学援助システム開発要件定義書)	Saga city (佐賀市)
Requirement definitions of middle area of Saga city community support system (佐賀中部広域連合地域包括支援センターシステム要件定義書)	Union of middle area of Saga city (佐賀中部広域連合)
Specification (draft) of development and transition for national-owned property information management system (国有財産総合情報管理システムに係る設計・開発及び移行業務 一式 仕様書（案）)	Financial bureau, Ministry of finance (財務省理財局管理課)
System design document of Yamagata prefecture cooperate system (山梨県共同システム基本設計書)	Office union of Yamagata prefecture municipality (山梨県市町村総合事務組合)
Specification of Akita prefecture municipal hospital financial and accounting system development entrustment (地方独立行政法人市立秋田総合病院財務会計システム構築業務委託仕様書	Akita city welfare and health department (秋田市福祉保健部病院法人移行準備室)
Specification of Matsue city water supply and sewage system development requirement (松江市上下水道局水道施設管理マッピングシステム構築業務要求仕様書	Matsue city water supply and sewage system bureau (松江市上下水道局)
Specification (common) of Yaizu city information system integration (焼津市内部情報系システム整備事業　要求仕様書（共通）)	Yaizu city (焼津市)
Specification of Statistical first-aid management system (救急統計管理システム仕様書)	Fire department, Fuefuki city (笛吹市消防本部)
System design document of common reception system (汎用受付システム基本設計書)	Shimane prefecture (島根県)
Work of OSS open laboratory system development and integration (first step) (OSS オープン・ラボ システム開発・構築作業 (第 1 次強化))	Information promotion agency (独立行政法人情報処理推進機構)
Specification of stroke brain images transfer system (脳卒中遠隔画像伝送システム仕様書)	Cooperation of Noto city stroke (能登脳卒中地域連携協議会)

Table 3.8: The number of sentences in cases before and after syntactic similarity pre-processing

syntactic similarity pre-processing	Document type A	Document type B
Before	871	711
After	844	667

files.”「システムは，申請者の指定したファイルを取得し，添付資料を開く.」

– “If the number of input items is not correct, the system display that the number is not correct. ”「入力された項目数が正しくない場合，システムは，項目数が正しくない旨を表示する.」

- Example sentences of document type B:

  – “If the system have not been processed more than the configured duration time, the user will be logged out (automatic log out function)”「あらかじめ設定した時間以上システムが操作されていないと判断される場合は，当該利用者を自動的にログアウトすること(自動ログアウト機能).」

  – “(In case of requests and required to add the number to the list of mandatory for management)Company list will be extracted and displayed”「(依頼に基づく場合、管理上必須一覧の項目に番号が追加されることが必要な場合等) 会社等一覧表を抽出，出力する.」

3.8 shows number of sentences before and after syntactic similarity pre-processing for both type A and B.

(b) Precision and Recall

A decision table testing expert evaluated the results of labeling for conditions and actions. True Positive (TP) refers to clauses that the expert evaluated as having conditions that are labeled as conditions by our rules and to clauses that the expert evaluated as having actions that are labeled as actions by our rules. False Positive (FP) refers to clauses that the expert did not evaluate as having conditions but that are labeled as conditions by our rules and to clauses that the expert did not evaluate as having actions but that are labeled as actions by our rules. False Negative (FN) refers to clauses that the expert

evaluated as conditions but that are not labeled as conditions by our rules and to clauses that the expert evaluated as having actions but that are not labeled as actions by our rules. True Positive (TP) refers to clauses that the expert evaluated as having conditions that are labeled both conditions and actions by our rules. False Positive (FP) refers to clauses that the expert evaluated as having actions that are labeled both conditions and actions by our rules.

We count each number of evaluation results. Equations (3.5), (3.6), and (3.7) show Precision(P), Recall(R) and F-Measure(F). The evaluation does not include the sentences that are not selected by syntactic pre-processing.

$$Precision = \frac{TP}{(TP + FP)} \tag{3.5}$$

$$Recall = \frac{TP}{(TP + FN)} \tag{3.6}$$

$$F = 2 \times Precision \times \frac{Recall}{(Precision + Recall)} \tag{3.7}$$

Table 3.9 and 3.10 shows the evaluation results. In the case of "Before" syntactic similarity pre-processing, the precision results ranged from 0.888 to 0.985, the recall results from 0.944 to 0.973, and the F results from 0.923 to 0.964.

A related work [TNY10] has the precision results 0.815, recall results 0.946 on Japanese use case descriptions. Our technique results more than the related work. The related work [TNY10] is as same as our technique on extracting condition and action from documents in natural language. The related work is different from our technique about labeling rules on condition and action. Labeling rules of the related work are position of words for the keyword as "if". Labeling rules of our technique are from dependency and case analysis. These labeling rules of our technique are new comparing the related work.

In the case of "After", the precision results ranged from 0.901 to 0.988, the recall results from 0.946 to 0.974, and the F results from 0.930 to 0.970. Each maximum precision with syntactic similarity pre-processing was better than that with no pre-processing. The after precision of extracting condition in with syntactic similarity preprocessing on document type B is better than before. We will discuss the reason on the evaluation subsection.

Table 3.9: Counts of conditions and actions

Pre-process	Document type	Counts of conditions			Counts of actions		
		TP	FP	FN	TP	FP	FN
Before	A	1,545	102	43	1,634	25	96
	B	1,113	140	47	1,215	19	72
After	A	1,489	99	39	1,614	20	92
	B	1,039	114	43	1,187	18	56

Table 3.10: Precision-Recall-F

Pre-process	Document type	Counts of conditions			Counts of actions		
		Precision	Recall	F	Precision	Recall	F
Before	A	0.938	0.973	0.955	0.985	0.945	0.964
	B	0.888	0.959	0.923	0.985	0.944	0.964
After	A	0.938	0.974	0.956	0.988	0.946	0.966
	B	0.901	0.960	0.930	0.985	0.955	0.970

(c) Pre-processing by Syntactic Similarity

In this paper, we define the syntactic similarity as cosine similarity by inner products on a tree kernel and the threshold as 0.30 from typical values and our studies on example sentences. For the results of the syntactic similarity pre-processing, Table 3.8 shows the numbers of target sentences selected from 871 to 844 in document type A, from 711 to 667 in document type. The ratio of numbers of non-selected sentences is 3.1% in document type A, 6.2% in document type B. The objective of this syntactic similarity is to select suitable sentences for extracting conditions and actions by comparing base sentence structure and each sentence in the requirements. Table 3.10 shows that syntactic similarity pre-processing delivered equal or better results of precision, recall, and F compared to no syntactic pre-processing. The

precision of document type B achieves 0.901 with the pre-processing than 0.888 at before the pre-processing. These results clearly demonstrate the potential of our approach. As for the threshold, if the value of the threshold is defined as greater than 0.30, more suitable requirements sentences for labeling conditions and actions are selected. The precision will be raised along with the results of the more suitable requirements sentences. However, the number of selected requirements sentences will be less than before. The coverage of requirements sentences will be down at the results of less number of requirements sentences. The requirements sentence that are not selected can be returned to the developer to be updated with information about syntactic similarity. This feedback can raise the coverage of requirements sentences.

(d) Document Type in the Experiment

The results of our technique seem to depend upon the difference between writing styles of documents. We compare and evaluate the difference between type A and B by describing a phrase pattern with case of terminal phrase as follows:

<phrase pattern>::=(<case of terminal phrase>)
| {(<case of terminal phrase>)}<terminal phrase>

The phrase patterns are automatically generated from the results of our technique. For an example, figure 3.10 shows that the first phrase is not-yet-case、the second phrase is non-case, the third phrase is complex predicative modifier, the fourth phrase is ga-case and terminal phrase. The pattern is described as <not-yet-case><non-case><complex predicative modifier><ga-case><terminal phrase>.

Table 3.11 shows top 10 patterns of document type A. Table 3.12 shows top 10 patterns of document type B.

In document type A, it is the most pattern “<predicative modifier><not-yet-case><wo-case><terminal phrase>” This patter's shows a sentence “If ～, ～ do ～ to ～” . An example sentence is “If system cannot get the company information, system display the error message” . There are 177 sentences and account for 20.4% in document type A. Top 10 patterns account for 61.8%.

We discuss about extracting conditions and actions related with the phrase patterns by using “If system cannot get the company information, system display the error message” in document type A. This sentence has

case information as that “If system cannot get the company information” has predicative modifier, “system” has not-yet-case, “the error message” has wo-case and “display” has terminal phrase. Our technique label condition on the phrases: “If system cannot get the company information” and “system”, also label action on “the error message” and “display”. Software testing experts evaluate these labeling are correct.

In document type B, it is the most pattern “<not-yet-case><predicative case><terminal phrase>” This patter's shows a sentence “—∼ is ∼” . The “is” is the terminal phrase in this pattern. An example sentence is “System is completed” . There are 88 sentences and account for 12.6% in document type B. Top 10 patterns account for 39.3%.

We discuss about extracting conditions and actions related with the 10th pattern as “<predicative modifier> <de-case> <predicative case> <terminal phrase>” by using “If comments exist in comment item, a system step is register submissions automatically by putting acceptance information.” in document type B. This sentence has case information as that “If comments exist in comment item” has predicative modifier, “by putting acceptance information” has de-case, “register submissions automatically ” has predicative case and “is” has terminal phrase. Our technique label condition on the phrases: “If comments exist in comment item” and “by putting acceptance information” , also label action on “register submissions automatically ” and “is” . Software testing experts evaluate these labeling are correct.

The document type A has 157 phase patterns and 177 sentences at the top pattern. The document type B has 276 phase patterns and 86 sentences at the top pattern. The document type B has more variety of writing style than the document type A when the variety is defined by the number of phase patterns. The results may be take for granted because the document type B consist of 711 sentences in 23 systems documents and the document type A consist of 871 sentences in 1 system documents. Table 3.10 shows our technique promise effectiveness for variety of writing styles with high precision and recall.

Table 3.11: Top 10 Patterns of Document Type A

Document type	Phrase pattern	Number	Percentage	Cumulative Percent.
A	<predicative modifier> <not-yet-case> <wo-case> <terminal phrase>	177	20.4%	20.4%
	<not-yet-case> <wo-case> <terminal phrase>	95	10.9%	31.3%
	<not-yet-case> <predicative modifier> <wo-case> <terminal phrase>	80	9.2%	40.6%
	<predicative modifier> <wo-case> <terminal phrase>	58	6.7%	47.2%
	<predicative modifier> <not-yet-case> <predicative modifier> <wo-case> <terminal phrase>	43	5.0%	52.2%
	<wo-case> <terminal phrase>	23	2.6%	54.8%
	<wo-case> <predicative case> <terminal phrase>	18	2.1%	56.9%
	<not-yet-case> <kara-case> <wo-case> <terminal phrase>	18	2.1%	59.0%
	<predicative case> <terminal phrase>	12	1.4%	60.4%
	<not-yet-case> <ni-case> <wo-case> <terminal phrase>	12	1.4%	61.8%

Table 3.12: Top 10 Patterns of Document Type B

Document type	Phrase pattern	Number	Percentage	Cumulative Percent.
B	<not-yet-case> <predicative case> <terminal phrase>	88	12.6%	12.6%
	<predicative modifier> <predicative case> <terminal phrase>	70	10.0%	22.6%
	<predicative modifier> <not-yet-case> <predicative case> <terminal phrase>	21	3.0%	25.6%
	<predicative modifier> <wo-case> <terminal phrase>	18	2.6%	28.1%
	<not-yet-case> <wo-case> <terminal phrase>	18	2.6%	30.7%
	<no-case> <in-phrase> <terminal phrase>	17	2.4%	33.1%
	<predicative case> <terminal phrase>	13	1.9%	35.0%
	<ga-case> <predicative case> <terminal phrase>	12	1.7%	36.7%
	<not-yet-case> <predicative modifier> <wo-case> <terminal phrase>	9	1.3%	38.0%
	<predicative modifier> <de-case> <predicative case> <terminal phrase>	9	1.3%	39.3%

Table 3.13: Comparing Precision and Recall with Manual Extraction of Condition and Action

		Precision	Recall	F
Manual extraction	Manual extraction 1	1.000	0.756	0.861
	Manual extraction 2	1.000	0.393	0.564
Our technique	Average	0.949	0.955	0.952

(e) Comparing manual extraction

We discuss precision and recall by comparing manual extraction of condition and action. The base case is a related work [YMT15] as manual extraction. On an experiment in the manual extraction, they extracted 59 from 78 requirements which shall be extracted. We call this result is manual extraction 1. On another experiment in the manual extraction, they extracted 22 from 59 requirements which shall be extracted. We call this result is manual extraction 2. These manual extractions include condition and action to be extracted. Table 3.13 shows Precision, Recall, F-Measure of the manual extraction and our technique. Our technique achieves higher Recall and F-Measure more than the manual extraction, and lower Precision. We discuss about this result on evaluation section.

We compare the work loads of manual extraction and our technique. We execute our experiments on a personal computer which has 2.60GHz CPU, 8GB memory, Japanese natural language processing tool [KK] with implementing script language. In the manual extraction, an engineer who has 20 years experiences about software development and testing, executed to extract condition and action from documents. We compare work loads for extraction of condition and action. We compare between manual extraction and step 1 to step 2 on our technique. Table 3.14 shows the results of workloads of manual extraction and our technique. Our technique achieves one six duration time than manual extraction.

3.2.4 Evaluations

We confirm the results promise our technique achieve higher precision and recall than the related work. We also confirm the result promise the syntactic similarity pre-processing has effectiveness for variety of writing styles as document type B. Test cases are often required almost 100% coverage for

Table 3.14: Comparing Work Loads with Manual Extraction (minutes)

	Manual Extraction		Our technique	
Document type	A	B	A	B
Step 1.Natural language processing	240	210	42	36
Step 2.Extracting condition and action			1	1

functions to be tested in industry actual project. In this paper, when we target system test and/or user acceptance test, the results of precision and recall shows effectiveness of our technique in industry actual project.

We evaluate precision and recall on difference of writing styles. The document type A is formalized with the project writing rules as use case description. The writing style is compatible with our technique. The document type A, for example, has a sentence as "If the logon user is not in the target company, the system display that the user is not valid. " Conditions and actions are written clearly in the sentence. Our technique labeled condition on "If the logon user is not in the target company" and "the system" , and action on "display that the user is not valid" .

We confirm the results promise our technique achieve higher precision and recall on document type B than the related work [TNY10]. We evaluate the precision of condition extraction on document type B before syntactic similarity pre-processing. The main reason is extracting action predicative as condition. The example sentence is "If the system have not been processed more than the configured duration time, the user will be logged out (automatic log out function)" . This sentence has brackets. Our technique may label this bracket phrase as action and the other phrases as condition.

We can make precision and recall better by applying syntactic similarity pre-processing to these writing style sentences. The sentences are not suitable for test cases that results below the criteria of syntactic similarity. It is useful for engineers in actual works to get feedbacks about the sentences which are not suitable for test cases.

We compared phrase patterns by using our technique. The comparing method may apply to calculate variety of descriptions in requirement engineering. We can apply the comparing results to feedback to description rules.

Comparing manual extraction, our technique achieve higher recall, F-measure, but lower precision. The precision of manual extraction is 1.000.

The precision is the result of extracting from small number of functions. We need to evaluate quality of manual extraction, however, the rules of manual extraction can be future rules on our technique. We confirmed one six workloads than manual extraction for extracting condition and action. This workloads reduction can be a motivation for automatic generation of test cases.

The future work is to apply our technique to listing sentences. Specification documents often have listing sentences such as definition of functions. One of the approach is to involve implicit relation analysis to our technique. Another future work is to how we define constraints between condition and action. The approach of this future work is to involve logical formula transformation [TKI12] and dictionary development.

3.3 Syntactic Rules of Extracting Test Cases from Software Requirements

3.3.1 Applying English Natural Language Processing Techniques to Decision Table Testing

(a) Decision Table Testing Technique

Decision tables consists of conditions and actions. The conditions and actions are deep cases of the requirements in the natural language processing. In this paper, we propose our rules to differentiate the deep cases from surface cases and to identify the dependencies that are generated by natural language processing. Decision table testing uses a model of the logical relationships between conditions and actions for the test item in the form of a decision table[ISO15b]. Generating appropriate conditions and actions for this decision table testing technique, it depends largely upon personal knowledges and skills to generate conditions and actions. Our approach is to generate conditions and actions mechanically by natural languages processing. Multiple conditions and actions mean there are a lot of restrictions, for example, if a condition has a particular value, another condition or action must have that some particular value. Generating such restrictions is required to access other domain knowledge. As this generation is outside of scope of this paper, it remains topic of future research.

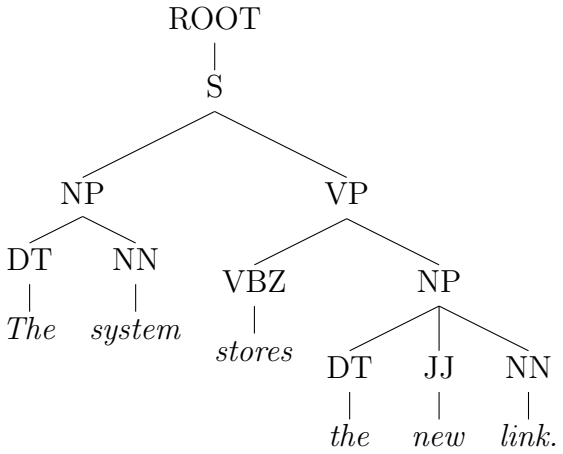

Figure 3.11: Parse tree of "The system stores the new link."

(b) Natural Language Processing

Natural language processing (NLP) techniques include parsing, morphological analysis, and so on and are used in the analysis of software requirements. There are four steps in NLP:

1. Morphological analysis: This is to parse a sentence to words and tag their parts-of-speech.

2. Dependency analysis: This is to determine dependencies of the words that have been parsed as the result of morphological analysis.

3. Semantic analysis: This is to determine the semantics of the words and phrases.

4. Context analysis: This is to perform analysis over multiple sentences.

Figure 3.11 shows the results of morphological and dependency analyses of "The system stores the new link" as a tree. For example, the sentence consists of NP and VP, and NP consists of DT and NN. S: sentence, NP: noun phrase, VP: verb phrase, NN: noun, VBZ: verb behavior, and so on.

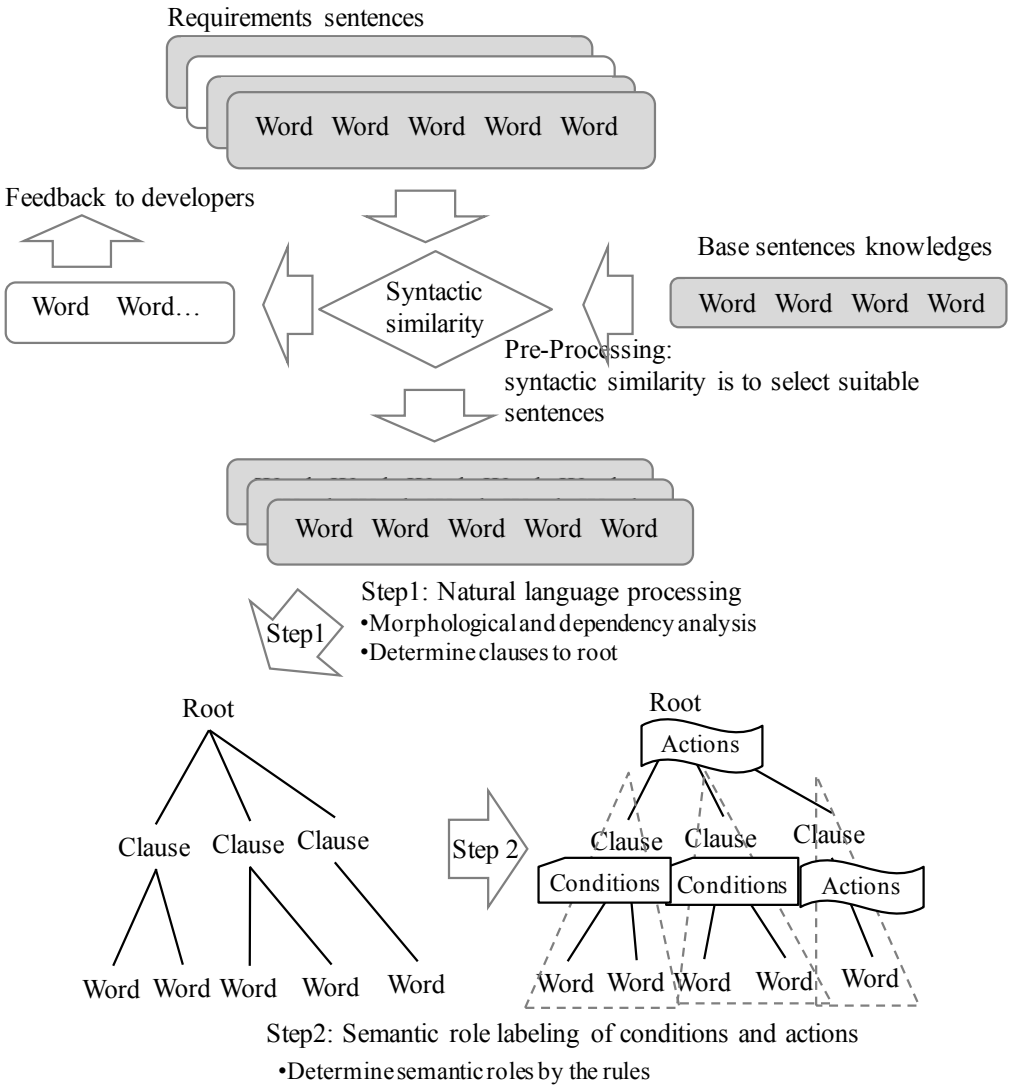

Figure 3.12: Syntactic rules of extracting test cases from software requirements

3.3.2 Syntactic Rules of Extracting Test Cases from Software Requirements

(a) Overview of the Syntactic Rules of Extraction

Figure 3.12 shows our syntactic rules of extracting test cases from software requirements. As the first step, it calculates syntactic similarity between base sentence structure and each sentences in the requirements. Sentences are selected as suitable for extracting conditions and actions. The accuracy of the extraction is raised by this selection. The base sentence structure means parts and dependencies of the sentence. There are two types of sentences, suitable and not suitable for extracting conditions and actions. Suitable sentences progress go to the next step and non-suitable ones are sent back to the requirements developers. This selection process is useful for the developers to improve the descriptions. As the second step, it extracts such as conditions and actions from the results of dependency analysis and case analysis as shown in Figure 3.12.

(b) Syntactic Similarity

The objective of this syntactic similarity is to select suitable sentences for extracting conditions and actions by comparing base sentence structure and each sentences in requirements. In this paper, we use the base sentence as "If the age is more than twelve, the fee will be five dollars." as an example. This sentence is suitable sentence for extracting conditions as "If the age is more than twelve," and actions as "the fee will be five dollars.". Figure 3.14 shows the base sentence and Table 3.15 shows the dependency in the base sentence. The base sentence knowledge is a collection of the base sentences. The knowledge is described in a tree structure consisting of parts and their dependencies that have conditions and actions. In this paper, we define the base sentence is as Figure 3.15. Penn tree descriptions of the base sentence are as Figure 3.13.

Sentences that are similar to the base sentence, such as "If the number is more than ten, the charge will be three dollars.", can be selected by calculating the syntactic similarity between them.

We use tree kernel technique[CMB11] to calculate syntactic similarity. Tree kernel is a well-known technique to calculate the similarity of a syntax tree. Figure 3.16 shows how we calculate the numbers of subtrees between the syntax trees by separating subtrees from the syntax tree, The formula 3.8

<Base sentence structure>::=
(ROOT
  (S
   (SBAR (IN If)
    (S
     (NP (DT the) (NN age))
     (VP (VBZ is)
      (NP
       (QP (JJR more) (IN than) (CD twelve))))))
   (, ,)
   (NP (DT the) (NN fee))
   (VP (MD will)
    (VP (VB be)
     (NP (CD five) (NNS dollars))))
   (. .)))

Figure 3.13: Penn tree descriptions of the base sentence

Table 3.15: The dependencies in the base sentence

From		To	
IN	(if)	CD	(twelve)
DT	(the)	NN	(age)
NN	(age)	CD	(twelve)
VBZ	(is)	CD	(twelve)
JJR	(more)	IN	(than)
IN	(than)	CD	(twelve)
CD	(twelve)	NNS	(dollars)
DT	(the)	NN	(fee)
NN	(fee)	NNS	(dollars)
MD	(will)	NNS	(dollars)
VB	(be)	NNS	(dollars)
CD	(five)	NNS	(dollars)
NNS	(dollars)	ROOT	

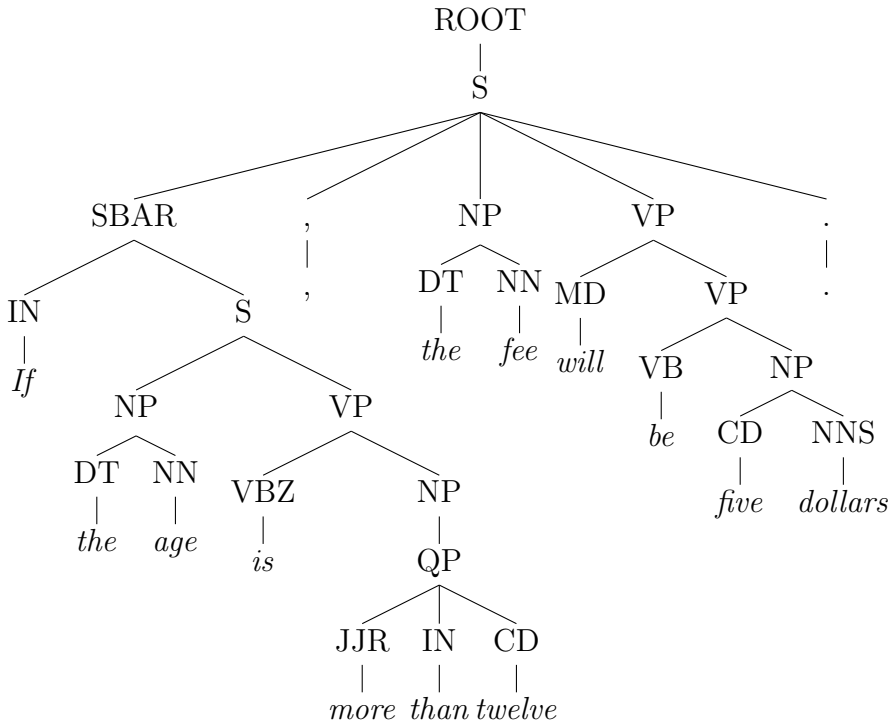

Figure 3.14: Parse the base sentence

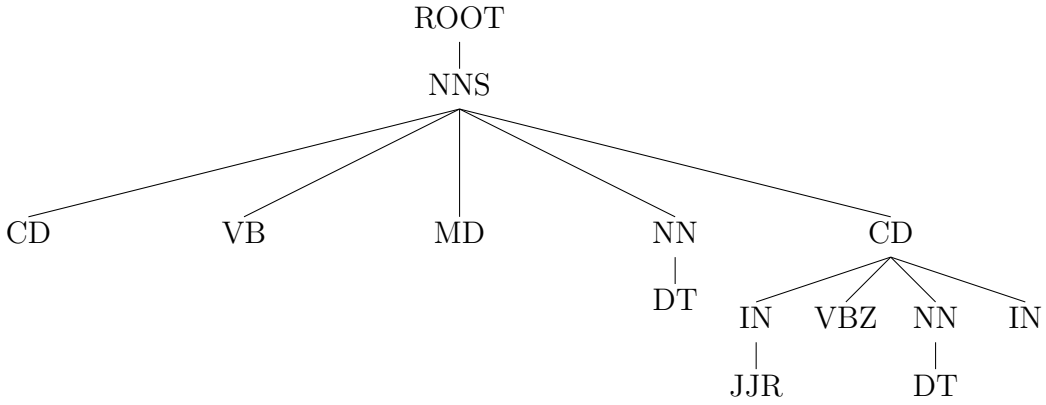

Figure 3.15: Dependency in the base sentence as parts

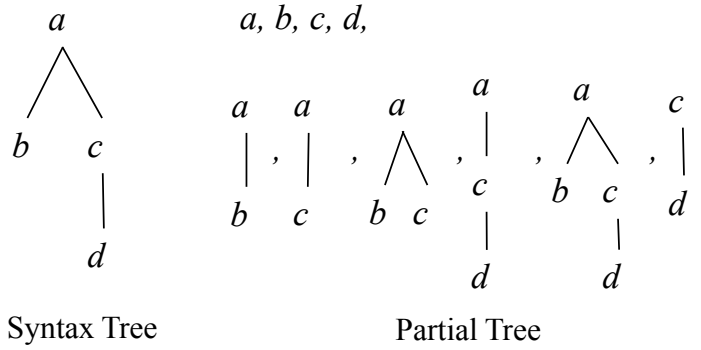

Figure 3.16: Syntax tree and subtrees on tree kernel

shows the cosine similarity that is used to calculate inner products in their tree kernels.

$$\cos\theta = \frac{k(Ti, Tj)}{\sqrt{k(Ti, Ti)k(Tj, Tj)}} \tag{3.8}$$

Ti, Tj are syntax trees. k is kernel of them. θ is an angle of Ti, Tj .

(c) Extraction of Conditions and Actions

We assume requirements are described formally to some extent for sharing information among stakeholders. Examples of requirements descriptions are

given below. These sentences have clauses that depend upon the root word. The meanings of dependency types are defined in [DMM08] as follows:

- prep: prepositional modifier. The prepositional modifier of a verb, adjective, or noun is any prepositional phrase that serves to modify the meaning of the verb, adjective, noun, or even another preposition.
- cop: copula. A copula is the relation between the complement of a copular verb and a copular verb.
- dobj: direct object. The direct object of a VP is the noun phrase which is the (accusative) object of the verb.
- nsubj: nominal subject. A nominal subject is a noun phrase which is the syntactic subject of a clause.
- nsubjpass: passive nominal subject. A passive nominal subject is a noun phrase that is the syntactic subject of a passive clause.
- aux: auxiliary. An auxiliary of a clause is a non-main verb of the clause.
- auxpass: passive auxiliary. A passive auxiliary of a clause is a non-main verb of the clause that contains the passive information.
- neg: negation modifier. The negation modifier is the relation between a negation word and the word it modifies.
- root: root. The root grammatical relation points to the root of the sentence.

The root is defined in [DMM08] as "The root grammatical relation points to the root of the sentence". In other words, the root is the main word in a sentence. In this paper, we define clauses that have condition words that are dependent upon a root word as condition clauses, and clauses that have action words that are dependent upon a root word as action clauses. Both condition and action clauses are labeled on the basis of the results of dependency analysis in NLP. The extraction rules are that when dependency types are one of "prep", "cop", "dobj", "nsubj", "nsubjpass", "aux", "auxpass", "neg", or "root", the words are labeled "action".

The extracting test case is defined as follows:

- S is the sentence.

- i is the number of each words parsed from S.
- $m(i)$ is a word parsed from S.
- T is the number of root of S.
- $m(T)$ is the root of S.
- $D(i)$ is the word number of $m(i)$ depending on a word.
- $C(i)$ is the case information of $m(i)$ depending on a word.
- $B(i)$ is a clause constructed from all words depending on $m(i)$ as the clause ending word.
- Cn indicates "conditions" and Ac indicates "actions" as attributes of B.
- R is a set of extracting test case rules.
  - Ra is the subset of action labeling rules.
- $Ra1$: $C(i)$ is one of "prep", "cop", "dobj", "nsubj", "nsubjpass", "aux", "auxpass", "neg", "root".
  - Rc is the complementary subset of condition labeling rules.
- $Rc1$:not $Ra1$

The algorithm 3 shows the algorithm of extracting test cases.

Algorithm 3 Extracting test cases algorithm

Require: Input: Results of parsing, dependency analysis
Ensure:
1: Search ROOT word
2: Search $m(i)$ in case $D(i)=T$
3: **for all** F **do**or all $m(i)$:$D(i)=T$
4: construct $B(i)$ with $m(i)$ as the clause ending word
5: label $B(i)$ as "actions" in the case of $C(i) \subseteq Ra$
6: label $B(i)$ as "conditions" in the case of $C(i) \subseteq Rc$
7: **end for**

3.3.3 Experiments

(a) Implementations

We implemented for a proof-of-concept prototype of our approach. We used an English natural language processing parser [MMM06] and dependency

analysis on natural language requirements. We developed our own tools using the Python natural language tool kit [Pro15] for the extracting test cases and evaluating the results. We experimented the prototype on three case studies of natural language requirements.

- CHART: The purpose of this design document is to provide implementation details that form the basis for the software coding. The details presented in this design fit within the high level approach documented in the high level design document [Adm03]. We used sections 2-1, 2-2 and 2-3 describing general system functionalities.

- eNotification: The purpose of this document is to define the electronic transmission of a data exchange involving between a party that has to get a legally required notice, e.g., a public procurement notice, published by a journal or newspaper [fE12]. We used section 5-1-1 of the business requirements statements.

- WUT: The Water Use Tracking (WUT) System's system requirements specifications is a collection of artifacts that were developed separately during the implementation phase of the project [Dis04]. We used the functional requirements section (4-1-1-1) of the document.

We experimented with two cases: one that has no with syntactic similarity pre-processing step and one that does. We define the threshold of selection sentences by syntactic similarity pre-processing as 0.30 from the typical value in NLP[NDM11][SNK14].

Table 3.16 shows the validity this threshold by comparing the syntactic similarity and software testing results by experts.

We examined three case studies of natural language requirements. Table 3.17 shows the number of sentences in two cases: one that does not have a syntactic similarity pre-processing step and one that does.

(b) Precision and Recall

A decision table testing expert evaluated the results of labeling for conditions and actions. True Positive (TP) refers to clauses that the expert evaluated as having conditions that are labeled as conditions by our rules and to clauses that the expert evaluated as having actions that are labeled as actions by our rules. False Positive (FP) refers to clauses that the expert did not evaluate

Table 3.16: Comparing the syntactic similarity and software testing by experts

Requirements sentences examples	Experts results	Syntactic Similarity
If the age is more than twelve, the fee will be five dollars.	OK	1.00
If the age is less than eighteen, the charge will be ten dollars.	OK	0.94
The map will be generated with the background map layers in the same fashion as the Intranet map.	OK	0.46
If the cmd parameter is ViewEvents, the mapping application initializes to "Zoom in" mode and displays the event list in the "data" frame.	OK	0.32
When a user selects one theme for display, the map frame sends the request to server to retrieve the background map.	NG	0.27
The Publisher informs the submitter, that publication is refused, if the notice is not eligible for publication for reasons.	NG	0.21
Requests from a client may go to different servers, causing the session information to be inconsistent.	NG	0.20
The Notice Summary may contain a reference to the regulation the notice applies to.	NG	0.19

Table 3.17: The number of sentences in cases before and after syntactic similarity pre-processing

Syntactic Pre-process	CHART	eNOT	WUT
Before	204	144	110
After	166	123	91

as having conditions but that are labeled as conditions by our rules and to clauses that the expert did not evaluate as having actions but that are labeled as actions by our rules. False Negative (FN) refers to clauses that the expert evaluated as conditions but that are not labeled as conditions by our rules and to clauses that the expert evaluated as having actions but that are not labeled as actions by our rules. True Positive (TP) refers to clauses that the expert evaluated as having conditions that are labeled both conditions and actions by our rules. False Positive (FP) refers to clauses that the expert evaluated as having actions that are labeled both conditions and actions by our rules.

We count each number of evaluation results. Equations (3.9), (3.10), and (3.11) show Precision(P), Recall(R) and F-Measure(F). The evaluation does not include the sentences that are not selected by syntactic pre-processing.

$$Precision = \frac{TP}{(TP + FP)} \tag{3.9}$$

$$Recall = \frac{TP}{(TP + FN)} \tag{3.10}$$

$$F = 2 \times Precision \times \frac{Recall}{(Precision + Recall)} \tag{3.11}$$

Table 3.18 and 3.19 shows the evaluation results. In the case of "before" syntactic similarity pre-processing, the precision results ranged from 0.73 to 0.85, the recall results from 0.56 to 0.67, and the F results from 0.67 to 0.72.

In the case of "after" , the precision results ranged from 0.73 to 0.87, the recall results from 0.58 to 0.73, and the F results from 0.70 to 0.77. Each maximum precision with syntactic similarity pre-processing was better than that with no pre-processing.

(c) Pre-processing by Syntactic Similarity

In this paper, we define the syntactic similarity as cosine similarity by inner products on a tree kernel and the threshold as 0.30 from typical values and

Table 3.18: Counts of conditions and actions

Pre-process	Requirements	Counts of conditions			Counts of actions		
		TP	FP	FN	TP	FP	FN
No	CHART	61	11	48	57	14	40
	eNotification	16	6	9	16	6	9
	WUT	14	4	7	14	4	7
Yes	CHART	48	7	35	46	10	30
	eNotification	14	3	5	14	5	6
	WUT	10	2	4	10	3	4

Table 3.19: Precision-Recall-F

Pre-process	Requirements	"Condition"			"Action"		
		Precision	Recall	F	Precision	Recall	F
Yes	CHART	0.85	0.56	0.67	0.80	0.59	0.68
	eNotification	0.73	0.64	0.68	0.73	0.64	0.68
	WUT	0.78	0.67	0.72	0.78	0.67	0.72
Yes	CHART	0.87	0.58	0.70	0.82	0.61	0.70
	eNotification	0.82	0.73	0.77	0.73	0.69	0.71
	WUT	0.83	0.71	0.77	0.75	0.71	0.73

our studies on example sentences. For the results of the syntactic similarity pre-processing, Table 3.17 shows the numbers of target sentences selected from 204 to 166 in CHART, from 144 to 123 in eNotification, and from 110 to 91 in WUT. The ratio of numbers of non-selected sentences is 19% in CHART, 15% in eNotification, and 17% in WUT. The objective of this syntactic similarity is to select suitable sentences for extracting conditions and actions by comparing base sentence structure and each sentence in the requirements. Table 3.19 shows that syntactic similarity pre-processing delivered equal or better results of precision, recall, and F compared to no syntactic pre-processing. These results clearly demonstrate the potential of our approach. As for the threshold, if the value of the threshold is defined as greater than 0.30, more suitable requirements sentences for labeling conditions and actions are selected. The precision will be raised along with the results of the more suitable requirements sentences. However, the number of selected requirements sentences will be less than before. The coverage of requirements sentences will be down at the results of less number of requirements sentences. The requirements sentence that are not selected can be returned to the developer to be updated with information about syntactic similarity. This feedback can raise the coverage of requirements sentences.

3.3.4 Evaluations

Our approach achieved F results ranging from 0.70 to 0.77, as shown in Table 3.19. Ideally, test cases from the requirements may be required to have 100% coverage of the functions in order to be tested. In this paper, however, our approach is useful when we target system testing or user acceptance testing.

Our approach can also be applied to several description types of requirements. We tested our approach with three requirements documents, CHART, eNotification, and WUT, each of which have different description types. Each of their F values was from 0.70 to 0.77, which demonstrates that our framework can be applied for each type.

Table 3.17 shows the numbers of target sentences selected from 204 to 166 in CHART, from 144 to 123 in eNotification, and from 110 to 91 in WUT. The ratio of the numbers of non-selected sentences is 19% in CHART, 15% in eNotification, and 17% in WUT. These results show that our approach can be applied to several different description types of requirements. They also show that it is difficult to extract conditions and actions from 15% to 19% of the requirements. Conditions and actions descriptions are function

descriptions in the requirements. Therefore, from 15% to 19% of the requirements descriptions should be improved from the functions descriptions point of view. These results also demonstrate the potential of our approach.

The threshold of syntactic similarity is defined as 0.30 in this paper. We confirmed that this threshold is valid by checking our example sentences, as shown in Table 3.16. Table 3.19 also shows the results of precision, recall, and F after syntactic similarity pre-processing. The F value of each of them ranged from 0.70 to 0.77. These results also show the potential of our approach.

3.4 Chapter Summary

We proposed the analysis technique, a semantic analysis technique of logics retrieval for software testing from Japanese public sector's specification documents concept, technique and presented the results of experiments. The result was that the precision reached 0.93 to 0.97 and recall reached 0.65 to 0.79. That showed the analysis technique worked for retrieving condition logics. We confirmed the analysis technique could retrieve logics from Japanese natural language specification documents. When we target to retrieve condition logics, the number of keywords for conditions limited. Then the analysis technique works for retrieval conditions from specification documents.

This result is the starting point to research about harmonization between natural language processing and software testing. The related work [Mat12] also proposed the benefits of retrieving logics from specification documents into decision tables and surveyed the opportunities of the future of decision table. The analysis technique can detect logic ambiguity of specification documents and feedback measurements for document quality. The measurements of specification documents are proposed by a related work [KKS08] as quality metrics, for examples, document defect density, document reusability and so on. The analysis technique can feedback to how we write manually specification documents precisely. The more correct we can describe logic on specification documents in advance, the less workload to fix of incorrect logic.

Decision table testing is a technique to develop test cases from descriptions of conditions and actions in software specification documents. For Japanese language, propose, experiment and evaluate a semantic role labeling technique of conditions and actions for automatic software test cases gener-

ation. Our approach uses natural language processing to select sentences from the specification based on syntactic similarity, and then to determine conditions and actions through dependency and case analysis. We got experiment results higher precision and recall for different style of descriptions, and the workload was reduced to one-sixth of manual work. Our results on case studies show the effectiveness of our technique. We will research about feedback for engineer to write documents easier understand by improving our technique.

For English language, we proposed, experimented, and evaluated a technique for extracting the conditions and actions of test cases for automatic software test case generation. Our approach uses natural language processing to select sentences from the specifications on the basis of syntactic similarity and then determines the conditions and actions through dependency and case analysis. Experimental results showed that F-measure reached from 0.70 to 0.77 for different styles of description. Our results on case studies demonstrate the effectiveness of our technique. For our future work, we will extend our approach to give feedback to developers so that they can improve their requirements descriptions.

Chapter 4

Detecting Logical Inconsistencies in Requirements

4.1 Detecting Logical Inconsistencies by Clustering Technique in Natural Language Requirements

4.1.1 Detecting Logical Inconsistencies by Clustering Technique in Natural Language Requirements

(a) An example for illustrating the approach

In this section, we present a practical approach to detecting logical inconsistencies by clustering technique in natural language requirements (See Figure 4.1).

Figure 4.2 presents an example that we will use to discuss our approach.

The example requirement, **REQ-ex**, is: "*If the programs start at the same time, the program listed first in the menu has priority.*"

The results of the parsing and dependency analysis are that "*has*" is the root word. The root is defined in [DMM08] as "*The root grammatical relation points to the root of the sentence*" . In other words, root is a main word in the sentence. Continuing the example results of the parsing and dependency analysis, "*start*" depends on "*has*" (root), and "*programs*" and "*time*" depend on "*start*" . The word, "*programs*" is substantive

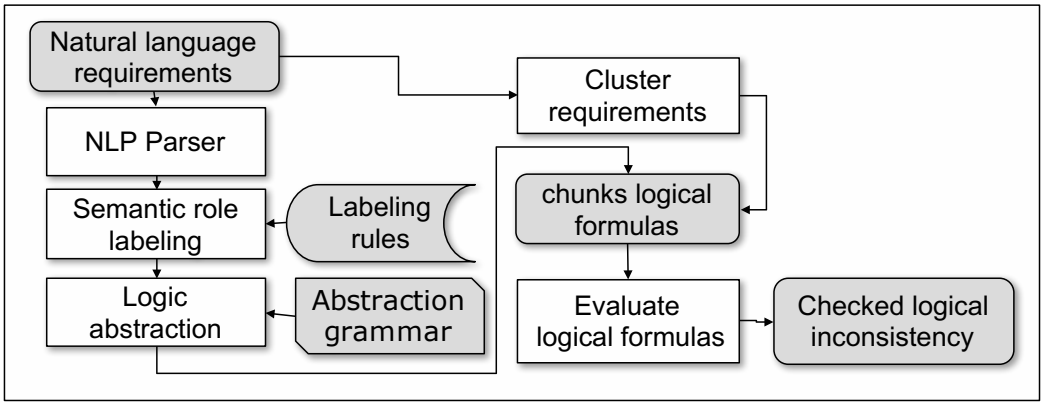

Figure 4.1: Framework for detecting logical inconsistencies by clustering technique in natural language requirements

Example requiremnt:

" If the programs start at the same time, the program listed first in the menu has priority. "

Parsing and dependency apalysis:

"(If) (the programs) (start) (at the same time), ROOT (the program listed first in the menu) (has) (priority)."

Semantic role labeling:

condition action

Logical abstraction:

substantive complement ROOT

transform into propositional logic:

P1 **&** ***P2*** **->** ***P3***

transform equivalent proposition logic:

-P1 **|** ***-P2*** **|** ***P3***

Evaluate logical formulas by input data patterns:

Patterns	***P1***	***P2***	***P3***	Results of (***-P1|-P2|P3***)
1	True	True	True	TRUE
2	True	False	True	TRUE
3	True	True	False	FALSE
4	True	False	False	TRUE
5	False	True	True	TRUE
6	False	False	True	TRUE
7	False	True	False	TRUE
8	False	False	False	TRUE

Figure 4.2: Example

according to our logical abstraction grammar, and hence, "*programs*" is a propositional variable of the clause that can be represented as, for example, *P1*. The word, "*time*" is a complement according to our logical abstraction grammar; accordingly, "*time*" is a propositional variable, represented as *P2*, for example. The word, "*has*" is the root of the sentence; accordingly, "*has*" is also a propositional variable, represented as *P3*, for example. Logical abstraction grammar comes from the definitions of the meaning of the type of dependency in the parser [DMM08]. Here we get the logical formula: *P1* & *P2* → *P3*. "&" , "|" , and "→" mean "and" , "or" , and "imply" . To evaluate this logical formula, it is first translated by negation into *-P1 | -P2 | P3*. The symbol "-" means "not" , and *P1*, *P2* and *P3* take values of true or false. In order to evaluate the negated logical formula, we input data patterns shown in Figure 4.2. The results of the evaluation reveal a logical inconsistency: pattern 3, i.e., *P1*=True, *P2*=True and *P3*=False, returns false (*-P1 | -P2 | P3*).

(b) Clustering natural language requirements

We use the *k*-means clustering algorithm to cluster chunks of natural language requirements. Figure 4.3 illustrates *k*-means clustering and clustering paragraphs. *k*-means clustering is to partition n data points into k clusters [KMN+02] and also is used to cluster natural languages. When a paragraph of requirements is defined as a data in *k*-means clustering, each clusters can be determined by similarity between the paragraphs at the results of *k*-means clustering. Paragraphs are candidate chunks, because the ordinary writing style is to separate different topics into paragraphs, sections. There are vague separations of paragraphs in requirements. Then we focus on similarity between paragraphs by automatic calculation such as *k*-means clustering.

For instances, figure 4.3 illustrates when we apply *k*-means to paragraphs from paragraph 1.1.1 to 2.1.4, we can get from cluster 1 to cluster 4 each similarities by *k*-means algorithm. In this case, Sub A-1 and Sub A-2 have similarity of their descriptions as cluster 1. Then the scope of detecting logical inconsistency the cluster 1 is more suitable than scopes of Sub A-1 and Sub A-2. Other paragraphs are also clustered into each cluster from 2 to 4. In this example, there are four scopes of detecting logical consistency.

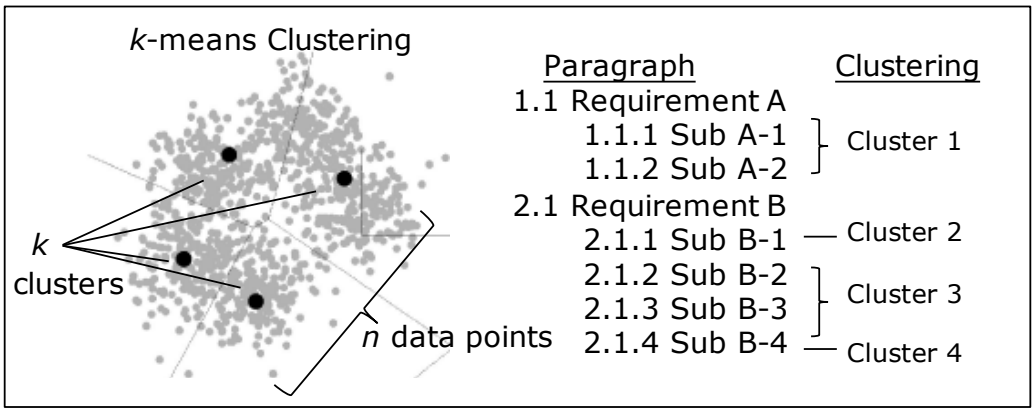

Figure 4.3: Clustering natural language requirements from Figure. 3 in [KMN+02]

(c) Parsing and dependency analysis

A natural language processing (NLP) parser assigns numbers to the words and punctuation in the natural language requirements. For example, **REQ-ex** is parsed into "`If-1 the-2 programs-3 start-4 at-5 the-6 same-7 time-8 ,-9 the-10 program-11 listed-12 first-13 in-14 the-15 menu-16 has-17 priority-18 .-19`". Here, integers are joined to each word or punctuation mark with a hyphen.

The NLP parser also analyzes the dependencies among the words. Table 4.1 shows the results of the dependency analysis of the example. Each parser defines a set of dependency types. We used the NLP parser [DMDS+14].

The meanings of dependency types are defined in [DMM08] as follows:

- prep: prepositional modifier. A prepositional modifier of a verb, adjective, or noun is any prepositional phrase that serves to modify the meaning of the verb, adjective, noun, or even another preposition.
- cop: copula. A copula is the relation between the complement of a copular verb and the copular verb.
- dobj: direct object. The direct object of a VP is the noun phrase which is the (accusative) object of the verb.

Table 4.1: Dependency Analysis

Dependency Type(*1)	From	To
root	ROOT-0	has-17
mark	start-4	If-1
det	programs-3	the-2
nsubj	start-4	programs-3
advcl	has-17	start-4
case	time-8	at-5
det	time-8	the-6
amod	time-8	same-7
nmod	start-4	time-8
det	program-11	the-10
nsubj	has-17	program-11
acl	program-11	listed-12
advmod	listed-12	first-13
case	menu-16	in-14
det	menu-16	the-15
nmod	listed-12	menu-16
dobj	has-17	priority-18

- nsubj: nominal subject. A nominal subject is a noun phrase which is the syntactic subject of a clause.
- nsubjpass: passive nominal subject. A passive nominal subject is a noun phrase which is the syntactic subject of a passive clause.
- aux: auxiliary. An auxiliary of a clause is a non-main verb of the clause.
- auxpass: passive auxiliary. A passive auxiliary of a clause is a non-main verb of the clause which contains the passive information.
- neg: negation modifier. The negation modifier is the relation between a negation word and the word it modifies.
- root: root. The root grammatical relation points to the root of the sentence.

(d) Semantic Role Labeling

The purpose of semantic role labeling in this paper is to label "actions" and "conditions" in a sentence. Even though requirements are written in natural language, the style of their descriptions will likely be formalized to some extent. Requirements are used to inform stakeholders or force him/her to act in accordance with them. Hence, their descriptions consist of condition sub clauses and action sub clauses, for example, "If (a) is (b), then (c) is (d)" , where clauses (a) to (d) represent words and phrases. At this point, we label the requirements with semantic roles. The labeling is defined as follows:

- **S** is a sentence.
- **i** is a number of each words parsed from **S** .
- **m(i)** is a word parsed from **S** .
- **T** is the number of root of **S** .
- **m(T)** is the root of **S** .
- **D(i)** is the word number of **m(i)** depending on.
- **C(i)** is the type description of **m(i)** depending on.
- **B(i)** is a clause constructed from all words depending on **m(i)** as the clause ending word.
- **Cn** indicates "conditions" and **Ac** indicates "actions" as attributes of **B**.
- **R** is a set of semantic role labeling rules.
  - **Ra** is the subset of action labeling rules.
- **Ra1**: **C(i)** is one of "prep" , "cop" , "dobj" , "nsubj" , "nsubjpass" , "aux" , "auxpass" , "neg" , "ROOT" .
  - **Rc** is the complementary subset of condition labeling rules.
- **Rc1**:not **Ra1**

Algorithm 4 Semantic role labeling algorithm

Require: Input: Results of parsing, dependency analysis **Output**: "actions" and "conditions" clauses

Ensure:

1: Search ROOT word
2: Search **m(i)** in case **D(i)=T**
3: **for all** F **do**or all **m(i)**:**D(i)=T**
4: construct **B(i)** with **m(i)** as the clause ending word
5: label **B(i)** as "actions" in the case of **C(i)** $\subseteq$ **Ra**
6: label **B(i)** as "conditions" in the case of **C(i)** $\subseteq$ **Rc**
7: **end for**

These definitions are used to interpret the requirements that have been parsed and whose dependencies have been analyzed as in Table 4.1. Algorithm 4 shows the semantic role labeling algorithm. The first step is to search for the root word. The next step is to search for all the words that depend on the root word. Clauses are constructed from the dependencies on the root word, and are labeled according to the "actions" and "conditions" rules. The rules depend on the language and the requirements. We have constructed ones for English and Japanese. Moreover, each language has its own natural language parser and dependency types. **Ra1** is a rule that when dependency types are one of "prep", "cop", "dobj", "nsubj", "nsubjpass", "aux", "auxpass", "neg", and "ROOT" the words are labeled "action".

All of the dependency types identify semantic label as actions when their dependence is on the root word. On the other hand, an adverbial clause modifier, i.e., "advcl" is a typical dependency type for labels identified as conditions. An adverbial clause modifier of a VP or S is a clause modifying the verb [DMM08]. The output consists of clauses labeled as "actions" and "conditions".

Figure 4.4 shows how the example requirement is assigned semantic role labels. The results of the parsing and dependency analysis is that "has" is **m(T)**, the root word. The words, "priority", "program" and "start" have dependency to "has". For instance, the type of dependency from "priority" to "has" is "dobj"; thus, "priority" is determined as part of the action from rule **Ra1**. "program" and all of its dependent words are also determined as part of the action from **Ra1**. The word, "start", is not determined as

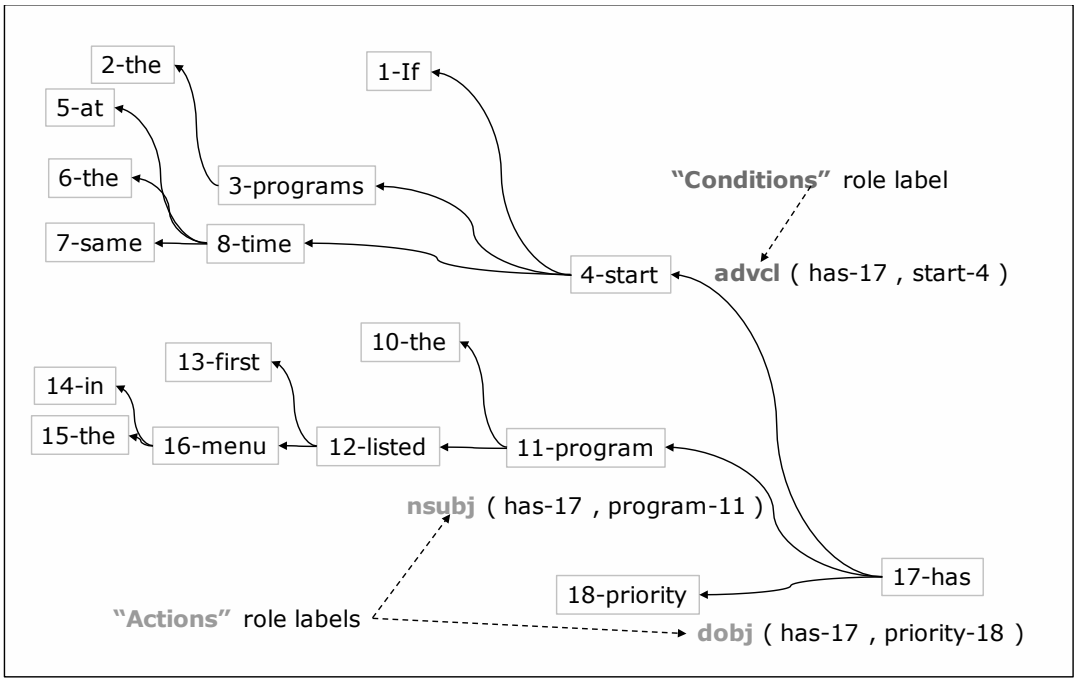

Figure 4.4: Example of semantic role labeling

part of **Ra1**; thus, it is determined as part of the condition from rule **Rc1**.

(e) Abstraction Grammar of the Structured English

Figure 4.5 lists the abstraction grammar of the structured English. The highlighted characteristics of this grammar are as follows: action and condition sub clauses, root word, and propositional variables. By applying this grammar, requirements can be translated into logical abstractions. A similar grammar is proposed in [YCC15]. It supports present, future and passive tenses with correct syntax according to English grammar. In this paper, we focus on action and condition sub clauses first. After action and condition sub clauses are determined, propositional variables are determined in the root word, substantive, and complement. The root word is determined by dependency analysis and is defined as a propositional variable representing

the action clause. First-order logic is one way to translate natural language into logical formulas, and some translation techniques use first order logic; however, their results are too complex for the purpose of finding logical inconsistencies. In first order logic translation, every word is translated into a function. By contrast, proposition logic is sufficient for finding logical inconsistencies. That is why we choose to use propositional variables and logic. A substantive is defined as a propositional variable representing substantive phrases in the condition sub clause. The complement is defined as a propositional variable representing complement phrases in the condition sub clause. Propositional logic formulas are then defined using these variables. Note the current grammar only supports "if" and "when" subordinators, and the case studies' requirements only have these subordinators.

The grammar was constructed from the description style of the requirements in the same way as the semantic role labeling. Requirements are formal at some level. They are used to inform stakeholders and force them to act accordingly. Hence, their style consists of condition sub clauses and action sub clauses.

Now let us show how the grammar works in more detail by using the sample sentence as an example. The sentence consists of two sub clauses, "If the programs start at the same time" and "the program listed first in the menu has priority" The root word "has" is identified by dependency analysis. The clause "the program listed first in the menu has priority" is an *action_subclause*, and the root word "has" is identified as a propositional variable. The other clause has the subordinator "if" ; this clause is identified as a *condition_subclause*. The *condition_subclause* is divided up into *(subordinator).(clause)* and *(subject).(predicate)* as the abstract grammar description. Finally, the subject is translated into substantives, and the word "programs" is identified as a propositional variable representing propositional logic. In the same way, the word "time" is identified as a propositional variable representing propositional logic.

(f) Evaluate logical formulas

Logical formulas are generated by using the abstraction grammar, for example, $P \,\&\, Q \rightarrow R$ and $S \mid Q \,\&\, T \rightarrow U$. In order to evaluate all logical formulas to find inconsistencies, the logical formulas are translated into their negation and the product of the negations is taken, for example, *(-P | -Q | R)* & *(-S* & *-Q | -T | U)*. In this example, the propositional variables are *P, Q, R, S,*

sentence	::=	*(action_subclause,)*.(, condition_subclause)**
action_subclause	::=	*clauses. (ROOT(proposional variable))*
condtion_subclause	::=	*subclause*
subclause	::=	(*subordinator*).(*clauses*)
clauses	::=	(*clause*).[, (*conjunction*).(*clause*)]
clause	::=	(*subject*).(*predicate*)
subject	::=	*substantives*
substantives	::=	(*substantive*).[(*conjunction*).(*substantives*)]
substantive	::=	*propositional variable*
predicates	::=	[*modality*].*predicate*
predicate	::=	*verb* | (*be*).*participle* | (*be*).(*complement*)
participle	::=	(*verb*).(ed) | (*verb*).(ing)
complement	::=	*propositional variable*
modality	::=	shall |should |will |would |can |could |must
subordinator	::=	if | when
conjunction	::=	and | or

Figure 4.5: Abstraction Grammar

T and U. Q appears in each formula. Then, patterns of TRUE or FALSE for all propositional variables are generated as input data of logical formulas to check whole logical formulas. If the result of some pattern is false, there are logical inconsistencies in the particular data patterns and the requirements sentence is considered inconsistent.

We used combinatorial testing to create input data patterns for evaluation of the logical formulas. The SAT solver is a program for checking logical constraints [HPSS06],[ES04],[HPS05]. The solver checks validity and consistency. Validity and consistency are really two ways of looking at the same thing and each may be described in terms of syntax or semantics [BHvM09]. SAT also has semantic versions of validity and consistency that are defined in terms of the concept of structure [BHvM09]. However, it only checks satisfiability as to whether there is at least one data pattern that solves the logical formula. That means it is insufficient for our objective of finding logical inconsistencies in data patterns and raising a false flag on all of the logical formulas. We must also evaluate the logical formulas using all combinations of input data patterns or by using combinatorial testing.

(g) Input data patterns

We evaluated the logical formulas by using combinatorial testing (CT) instead of checking all patterns of variables because the number of patterns is so large. Given n propositional variables, the number of patterns is 2 to the n-th power. For example, there are 85 propositional variables in section 2.2 of the CHART case study. The number of these variables combinations is 2^{82}, i.e., $4.8*10^{24}$. This is not a practical number of data patterns in which to find logical inconsistencies using the present computer resources. CT is the solution for this problem. CT can detect failures triggered by interactions of parameters in the software under test (SUT) with a covering array test suite generated by some sampling mechanism. CT has the following characteristics [NL11]: (1) it creates test cases by selecting values for parameters and by combining these values to form a covering array; (2) it uses a covering array as the test suite; (3) not every parameter of SUT can trigger a fault, and some faults can be exposed by testing interactions among a small number of parameters; (4) being a specification-based testing technique, CT requires no knowledge about the implementation of SUT; (5) tests can be automatically generated, which is a key to CT gaining in popularity. Characteristic (3) is not suitable for our framework. That is, even if the results of evaluation for

logical formulas have no inconsistency by using data which CT produced, that is not a proof of no inconsistency. Thus, we only use CT as a way of creating input data patterns and finding inconsistencies, not as a means of guaranteeing there are none. Our approach uses pairwise selection [NL11] which is a CT technique.

4.1.2 Experiments

We implemented for a proof-of-concept prototype of our framework. We used the Natural Language Processing parser [MMM06] and dependency analysis on natural language requirements. We developed our own tools using the Python natural language tool kit [Pro15] for the semantic role labeling, abstraction grammar, and evaluation of the logical formulas.

We experimented the prototype on three case studies of natural language requirements.

- CHART: The purpose of this design document is to provide implementation details that form the basis for the software coding. The details presented in this design fit within the high level approach documented in the high level design document [Adm03]. We used sections 2-1, 2-2 and 2-3 describing general system functionalities.

- eNotification: The purpose of this document is to define the electronic transmission of data exchanged between a party that has to get a legally required notice, e.g. a public procurement notice, published by a journal or newspaper [fE12]. We used section 5-1-1 of the business requirements statements.

- WUT: The Water Use Tracking (WUT) System's system requirements specifications is a collection of artifacts that were developed separately during the implementation phase of the project [Dis04]. We used the functional requirements section (4-1-1-1).

The experiments used two types of chunk: paragraph one chunked by paragraph number and clustered one chunked by the k-means algorithm. In order to compare the effects of varying number of paragraphs and number of clusters, we set different numbers of paragraphs on the number of clusters. We set the number of clusters in each case study to two and got the following clusters:

- CHART cluster 1 (C1) includes sections 2-1 and 2-2-1, and cluster 2 (C2) includes sections 2-2-2 to 2-3.
- eNotification (eNot) C1 includes functions numbering from 1 to 19, whereas C2 includes those from 20 to 28.
- WUT C1 includes functions 1 to 6, C2 includes functions 7 to 10.

Table 4.2 shows the chunked-by-paragraph results and the chunked-by-clustering results in case study CHART. The columns show the number of each artifact's instances. Table 4.3 shows the chunked-by-paragraph results and the chunked-by-clustering results in case study eNotification (eNot). Table 4.4 shows the chunked-by-paragraph results and the chunked-by-clustering results in case study WUT.

The number of requirements is the number of sentences used in the experiments. Each requirement is transformed into logical formulas and propositional variables. The following is an example of a requirement and corresponding logical formula in the CHART C1 case study: Requirement: "*If a layer is due for update, the client's browser will initiate remote scripting request to retrieve a new VML layer and replace the current layer.*" Logical formula: *(-layer — -update — initiate)* The words "layer", "update" and "initiate" are just representative words of the propositional logic transformed from the requirements, as in Figure 4.2. This formula is to be evaluated with input data patterns. For example, when the input data pattern is true, true and false, the logical formula is false.

The data input patterns were created by CT selection of true and false for each propositional variable. After evaluating the logical formulas by using these input data patterns, we calculated the inconsistency hit ratio (number of inconsistencies divided by number of input patterns).

4.1.3 Evaluations

The inconsistency hit ratios in Tables 4.2 show that our clustering approach could find more logical inconsistencies than paragraph chunking in the CHART case study. Our clustering approach found 19 total inconsistencies for which the inconsistency hit ratio was 0.719. By contrast, the chunked-by-paragraph

[1]Patterns (pairwise)
[2]Number of inconsistencies calculated from number of input patterns

Table 4.2: CHART case study

	Paragraphs			Clustering	
	2-1	2-2	2-3	C1	C2
Number of requirements	43	70	10	96	27
Number of logical formulas	33	85	18	95	41
Number of proposit. variables	43	82	13	106	32
Number of input patterns[1]	8	8	8	16	8
Number of inconsistencies	4	4	4	15	4
Total number of inconsistencies			12		19
Inconsistency hit ratio[2]			0.583		0.719

Table 4.3: eNot case study

	Paragraphs	Clustering	
		C1	C2
Number of requirements	46	37	9
Number of logical formulas	36	33	8
Number of proposit. variables	55	46	9
Number of input patterns[1]	8	8	8
Number of inconsistencies	4	4	3
Total number of inconsistencies	4		7
Inconsistency hit ratio[2]	0.500		0.438

Table 4.4: WUT case study

	Paragraphs	Clustering	
		C1	C2
Number of requirements	99	90	9
Number of logical formulas	83	78	12
Number of proposit. variables	64	56	8
Number of input patterns[1]	8	8	4
Number of inconsistencies	4	4	2
Total number of inconsistencies	4		6
Inconsistency hit ratio[2]	0.500		0.500

approach found 12 total inconsistencies from a total of 24 data inputs, for which the inconsistency hit ratio was 0.583. Table 4.3 shows the eNotification case study. Our clustering approach found 7 total inconsistencies more than 4 total inconsistencies by chunked-by-paragraph. The eNotification case study had an inconsistency hit ratio of 0.438 chunked by clustering and 0.500 for chunked by paragraph. Table 4.4 shows the WUT case study. Our clustering approach found 6 total inconsistencies more than 4 total inconsistencies by chunked-by-paragraph. The WUT case study had an inconsistency hit ratio of 0.500 for both chunked by paragraph and chunked by clustering.

The clustering approach found more total numbers of logical inconsistencies in all case studies. Even when it has fewer chunks than paragraph approach in CHART case study, the clustering approach found more numbers of logical inconsistencies. These results promise for usefulness of our approach in detecting logical inconsistencies.

Each case study has its own sentence style. CHART has a paragraph style, and its sentences have subjects and predicates. The function requirements are described in a number of sentences. eNotification has a list style: each function requirement is described in one sentence. WUT has an imperative style; its function description sentences do not have subjects but do have predicates. The number of propositional variables is an indication of the difference between these sentence styles. In CHART and eNotification, the propositional variables outnumber the requirements. In WUT, however, the number of propositional variables is smaller than the number of requirements.

Even though the requirements consisted of only dozens of sentences, there were a huge number of propositional variable combinations. For an example, CHART C1 has 2^{106}, i.e., $8.1 * 10^{31}$, combinations of variables. In the requirements, stakeholders describe their desires in terms of scenarios that cover up a huge amount of the total logic. Underneath them are vast arrays of logical combinations. To find logical inconsistencies from the vast arrays of logical combinations, our approach uses pairwise selection and the SAT solver to create a feasible amount of input data. When we tried only pairwise (true, false) selection on each propositional variable, the evaluations were mostly false because the logical formulas were so complex. Thus we used the SAT solver to select "base" input data patterns with which the logical formulas produce true. After that, we used pairwise selection on the base patterns.

In order to reproduce the experiments according with our framework described, we discuss tools, steps and execution time. We used Minisat which is one of state-of-the-art SAT solver [ES04], [VAF10]. We used Stanford NLP

parser [DMM08], [DMDS+14] for parsing sentences in requirements. We implemented semantic role labeling, abstraction grammar, and evaluation of logical formulas by Python with the natural language tool kit [Pro15]. We used Minisat for SAT solver, Allpairs for pairwise tool [Bac16] in creating feasible amount of input data. The execution time of each programs from seconds to minutes. We implemented as we mentioned, however, some of inconsistency checking processes have manual execution, e.g. copy files, start programs, etc. Then, total time of checking process depends on case studies. Total time of checking was about one hour at maximum so far.

In the experiments, we fixed the number of clusters created by the k-means clustering, because we wanted to compare the number of paragraphs of requirements and number of clusters of requirements. There is another clustering algorithm that calculate the number of clusters automatically like as x-means. The x-means is an extending k-means and has a new algorithm that quickly estimates k [PM00]. When this kind of automatic clustering algorithm cluster chunks of requirements by results of morphological and dependency analysis of them, our framework would be able to use it.

There is another framework for handling inconsistencies in natural language requirements [GZ05]; like ours, it uses an NLP parser and transforms requirements into logical formulas. The difference is that framework does not use chunks of requirements and transforms sentences into first order logic in order to find logical inconsistencies. Our approach clusters chunks of requirements and uses propositional logic. We will discuss the other related work in the next section.

4.2 Chapter Summary

We presented a practical approach to detecting logical inconsistencies by clustering technique in natural language requirements. The method uses k-means clustering to cluster chunks of the requirements and labeling rules to derive “conditions” and “actions” as semantic roles from the requirements by using natural language processing. We also constructed an abstraction grammar to transform the conditions and actions into logical formulas. By evaluating the logical formulas with input data patterns, we can find logical inconsistencies. We experimented with this approach on three case studies of requirements written in natural English. The results indicate that our approach can find logical inconsistencies.

In the future, we will use our framework to find vague requirements and provide feedback in early stage of the system development process. In addition, we will construct new rules and grammar for requirements descriptions. We will contribute to requirement engineering by developing new means to check whether descriptions have vague or inconsistent requirements.

Chapter 5

Applications of our technique

5.1 Automatic Generation of UTP Models from Requirements in Natural Language

5.1.1 Background and Approach

(a) UML Testing Profile

The UTP is a standardized language based on OMG's UML. UML is for designing, visualizing, specifying, analyzing, constructing, and documenting the artifacts commonly used. UTP is required for various testing approaches, in particular model-based testing (MBT) approaches. MBT specifications expressed with the UML Testing Profile are independent to any methodology, domain, or type of system [SSB15]. MBT is a software testing approach by using techniques and methodologies in Model Driven Development (MDD). Requirements are transferred into models by using UML on MDD. Requirements are also transferred into UTP models by using UTP. Modeling can contribute to show all the necessary functions, share the design of software, and reduce the ambiguity of design descriptions. Hence, UML and UTP are used to design, develop, and test mission critical software.

Figure 5.1 shows the structure in UTP. In [WPG+15], UTP is made as follows:

- Domain-independent test modeling
  - Test basis

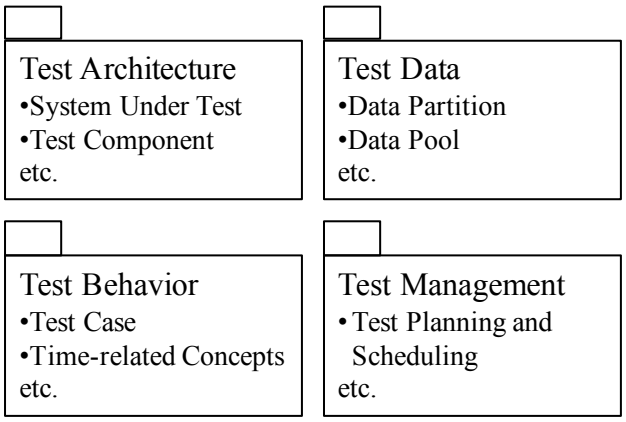

Figure 5.1: UTP definition overview

  - Test specifications

- Test case specifications
  - Abstract/concrete vs. logical/technical
- Test data specifications
- Test deployment
- Test result visualization
- Combination with other profiles

Figure 5.2 shows UTP test cases [FH14] as an example of test case specifications from the requirements "Users click the link as LinkNew.jsp" in natural language. The data of the sequence diagram in the UTP test cases consist of classes, actions and attributes, and the order of actions [OMG14]. In figure 5.2, the class is "Users". At activity 01 in figure 5.2, the action is "click", and the attributes are "link" and "LinkNew.jsp".

(b) Natural Language Processing

NLP Techniques include parsing, morphological analysis, and so on. NLP techniques are used in the analysis of software documents. There are four steps in NLP:

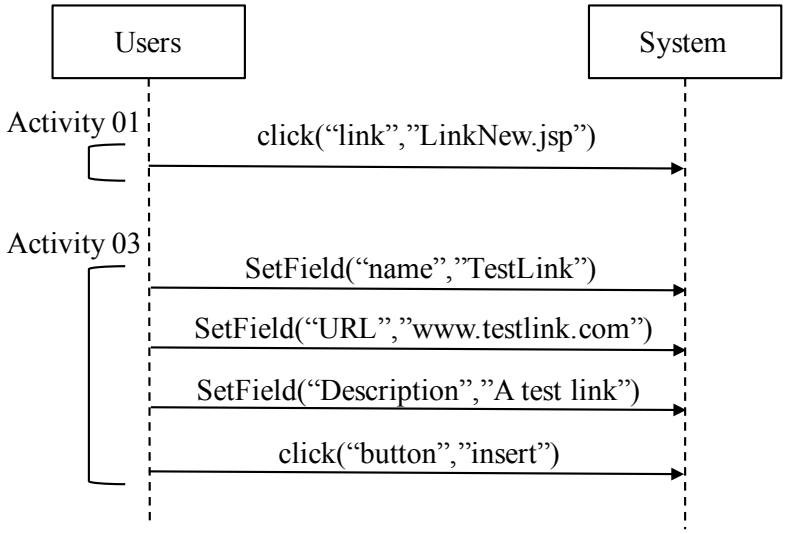

Figure 5.2: Example UTP test cases for editing the figure in [FH14]

1. Morphological analysis: This is to parse a sentence to words and tag their parts-of-speech.
2. Dependency analysis: This is to determine dependencies of the words that have been parsed as the results of morphological analysis.
3. Semantic analysis: This is to determine the semantics of the words and phrases.
4. Context analysis: This is to perform analysis over multiple sentences.

Figure 5.3 shows the results of morphological analysis and dependency analysis of “The system stores the new link” as a tree. For example, the sentence consists of NP and VP, and NP consists of DT and NN. S: sentence, NP: noun phrase, VP: verb phrase, NN: noun, VBZ: verb behavior, and so on.

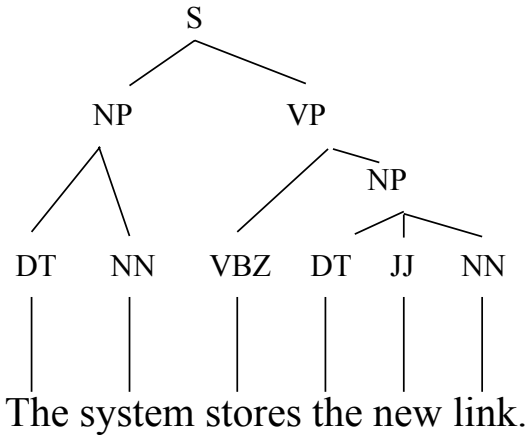

Figure 5.3: Parse tree of "The system stores the new link."

(c) Approach

We present our approach, Automatic Generation of UTP Models from Requirements in Natural Language as shown figure 5.4.

When we define the following:

- $\boldsymbol{S}$ are sentences of the requirements in natural languages,
- $\boldsymbol{U(cl, ac, ar)}$ are activities of the sequence diagram in UTP test cases which consist of classes (cl), actions (ac), and attributes (at), and
- $\boldsymbol{G}$ are generation rules from $\boldsymbol{S}$ (requirements) into $\boldsymbol{U}$ (classes, actions, and attributes).

Our approach is to define $\boldsymbol{G}$ in equation (5.1).

$$U(cl, ac, at) = G \times S \tag{5.1}$$

Our approach is to develop the generation rules on the basis of class generation rules from requirements [KSKD13] as a related work.

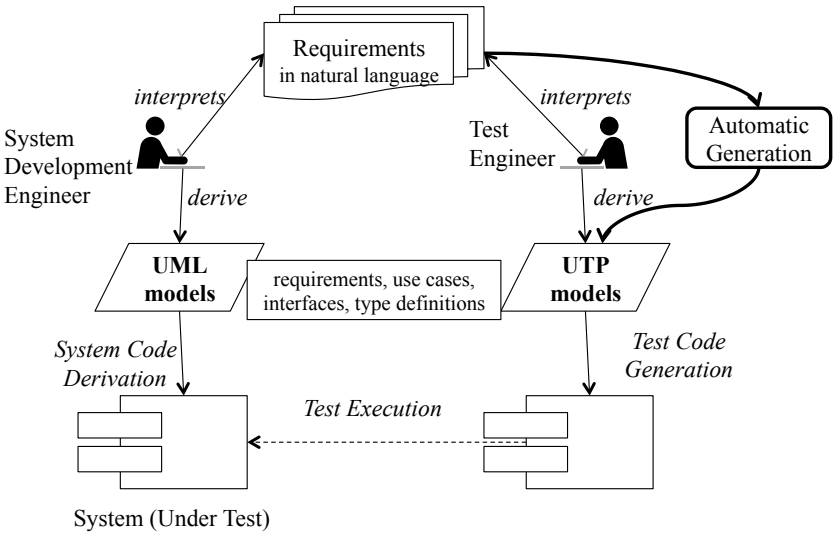

Figure 5.4: Generation of UTP from requirement by editing the figure in [OMG14]

5.1.2 Automatic Generation of UTP Models from Requirements

(a) Overview

Figure 5.5 shows a flow of the automatic generation of UTP from requirements in natural language. From the first step of the flow, we get parsed text, parts-of-speech, and dependency from requirements by using natural language processing techniques. In the next steps, we generate UTP models from them by applying rules of generation.

(b) Rules of Generation

In related work [FH14], the data of class diagrams are generated from requirements by using natural language processing. The rules are that classes be generated from noun words, that the class attributes be generated from the adjectives of the noun words, and that the class actions be generated from the verbs of the noun words. When we generate the UTP models, the definition of the class diagrams is the same as the related work. We then can reuse the rules in our generation technique. In UTP models, however,

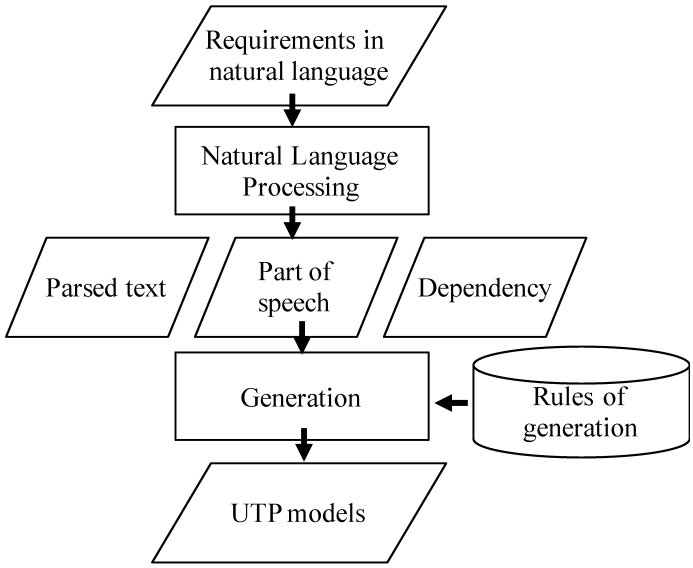

Figure 5.5: Automatic generation of UTP models from requirements in natural language

messages of the sequences are necessary. In order to get messages of the sequences, it is necessary to get actions between the classes. We develop new rules to get action information between classes. The approach toward the rules is that the verb between the names of classes is the actions between the classes.

UTP model generation rules are:

- Rule #1: Class generation rule
  a. Subject is generated to class
  b. Verb is generated to action
  c. Complement is argument
  d. Structure of text as tree bank expression as (5.2). The asterisk in the parenthesis means any part-of-speech. When there are requirements texts, which have this structure,
     * NN1 is generated to class
     * VBZ is generated to action

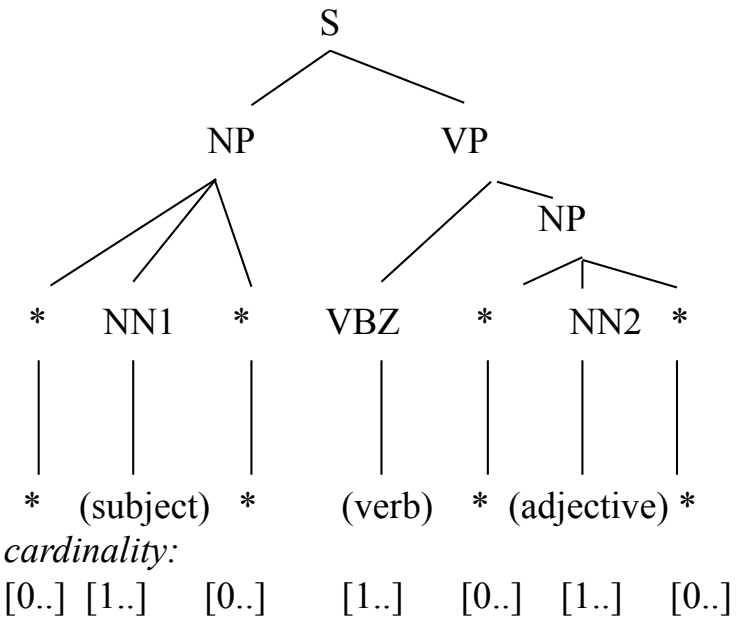

Figure 5.6: Parse tree of a generation rule

∗ NN2 is generated to attribute

$$S(NP((*)(NN1)(*))VP(VBZ)(NP((*)(NN2)(*))) \tag{5.2}$$

- Rule #2: Messages between classes generation rule

  a. When NN1 and NN2 have already been determined as classes, VBZ is the message from NN1 to NN2

- Rule #3: Order of sequence is equal to order of description in the requirements

Figure 5.6 shows parsed tree of (5.2).

5.1.3 Experiments

(a) Implementation

We implemented a proof-of-concept prototype of our approach. We used the English Natural Language Processing parser [DMM08] and dependency

Algorithm 5 Generation of UTP models from requirements in natural language

Require: Input: documents which have been morphologically analyzed and dependency parsed
PT: Parsed Tree in requirements
R1: Generation rule #1
RS: Generation rule #1 structure of text as pattern
R2: Generation rule #2

Ensure:

```
for all PT do
    mat ← return(search RS forall PT)
    if mat == TRUE then
        mat2 ← return(search NN forall PT by R1)
        if mat2 == TRUE then
            determine state and store the NN as "subject" (NN1)
            mat3 ← return(search VBZ forall PT by R1)
            if mat3 == TRUE then
                determine the VBZ as "verb"
                mat4 ← return(search NN forall PT by R1)
                if mat4 == TRUE then
                    determine and store the NN as "adjective"(NN2)
                end if
            end if
        end if
    end if
end for
for all PT do
    mat ← return(search RS forall PT)
    if mat == TRUE then
        mat2 ← return(search NN forall PT by R1)
        if mat2 == TRUE then
            determine and store the NN as "subject"(NN1)
            mat5 ← return(search the NN in NN1 and NN2 by R2)
            if mat5 == TRUE then
                mat3 ← return(search VBZ forall PT by R1)
                if mat3 == TRUE then
                    mat4 ← return(search NN forall PT by R1)
                    if mat4 == TRUE then
                        mat6 ← return(search the NN in NN1 and NN2 by R2)
                        if mat4 == TRUE then
                            determine the VBZ is "message"
                        end if
                    end if
                end if
            end if
        end if
    end if
end for
```

Table 5.1: Requirements in CHART system [Adm03].

SEQ#	Requirements of the section 2-1 in [Adm03].
1:	The Listener provides a conduit between the CHART II application and the Mapping software.
2:	The Listener detects CHART II CORBA events and writes the appropriate data to the Mapping database as events come in.
3:	The existing Listener, called the CHARTWeb Listener, already listens for CORBA events from CHART II pertaining to Traffic Events, DMSs, and TSSs.
4:	They also have a "lollipop" interface icon extending up from them, as sometimes the grey does not show up in printed copies.
5:	The class diagram shows a threesome of classes for each of the object types to be handled.
6:	The Module is the top-level class for each object type.
7:	The Module sets up the PushReceiver class to receive CORBA events from the CHART II Event Service pertaining to the appropriate object type, and upon receipt of these CORBA events the PushReceiver calls the appropriate helper methods of the DatabaseHelper to make the appropriate updates to the web database.
8:	Each resource has a unique ID by which it is referred to in future CORBA Events.
9:	Note that resources are not routinely deleted when a resource departs the scene; the Departure TimeStamp will be updated and the record will be left intact.
10:	When a traffic event is finally deleted, its associated resources will be deleted as well.
11:	The message indication will indicate only whether there is a non-default message on the HAR or not.
12:	It will not provide an indication of what the message is, since it is an audio message.
13:	This is because of the non-guaranteed nature of CORBA events, which raises the possibility that the local database may get out of synch with the CHART II database.
14:	A refresh for a particular class of objects will be completed in one database transaction, so that in the likely case that nothing has changed, there will be no "flicker" of activity detectable through the web database.
15:	Event processing for all object types will be made more robust, in that if a CORBA Event is received pertaining to an update for a device or traffic event that does not exist in the Web database, the Listener will attempt to actively collect the missing item data from the appropriate CHART II service and then proceed with the update if necessary.

Table 5.2: Template for the Use Case in [FH14].

Name	UC-01. Add new link
Main sequence	1. The user selects the option: add a new link. 2. The system selects the "top" category and shows the form to introduce the information of a link (SR-02). 3. The user introduces information of the new link and presses the insert button. 4. The system stores the new link.
Errors and alternatives	4. If the link name or URL link is empty, the system shows an error message and asks for the value again.
Post condition	The new link is stored into the system.

analysis [DMDS+14] on natural language requirements. We developed our own tools using the Python natural language tool kit [Pro15] in accordance with the algorithm 5.

(b) Experiments

We evaluate our prototype in two case studies:

- Requirements: "UC-01. Add new link" [GEMT06]. Table 5.2 shows the requirements. We call this case study GEN.
- Section 2-1 of requirements: "a detailed system design specification for the coordinated highways action response team (CHART) mapping applications" [Adm03]. Table 5.1 shows the requirements. We call this case study CHART.

GEN is a template use case in related work [GEMT06]. CHART is the requirements of the coordinated highways action response team.

We also evaluate the required time to generate UTP models from requirements in natural language. We compare a manual approach to our automatic approach. The manual approach is executed by a software testing expert who has 21 year's experience in software development and testing. The automatic approach is executed by the proof-of-concept prototype of our approach.

5.1.4 Evaluations

We have evaluated the results of the automatically generated UTP models by software testing experts' reviews. The evaluation methods for each class, action, and attribute are as follows:

- If the experiments generate classes, actions, and attributes, and the experts review results that shall be generated, the evaluation is True Positive (TP).
- If the experiments generate classes, actions, and attributes, and the experts review results that shall not be generated, the evaluation is False Positive (FP).
- If the experiments do not generate classes, actions, and attributes, and the experts review results that shall be generated, the evaluation is False Negative (FN).

We count the number of TP, FP, and FN. We calculate Precision, Recall, and F-measure as (5.3), (5.4), and (5.5).

$$Precision = \frac{TP}{(TP + FP)} \tag{5.3}$$

$$Recall = \frac{TP}{(TP + FN)} \tag{5.4}$$

$$F - Measure = 2 \times Precision \times \frac{Recall}{(Precision + Recall)} \tag{5.5}$$

Table 5.3 shows the expert's evaluation of the results of the GEN and CHART case studies. Table 5.4 shows the experiment results of the GEN and CHART case studies.

Comparing minimum values of precision and recall at GEN and CHART experiments, the values at GEN are greater than the values at CHART. The requirements of GEN are originally described for engineers to derive UTP models manually, so this result shows that our automatic UTP models generation technique can re-produce people's derivations work. As a result of CHART experiments, it is also greater than or equal to 0.75, except for the values of attributes in rule #1. This also shows promise for our technique. The reason for 0.56 of rule #1 about attributes during the CHART experiments is the difference in the text tree between structures as an equation

Table 5.3: Expert Evaluation of Results

Case study			**Number of generated**	**False Positive**	**False Negative**
GEN	**Rule #1**	*Class*	4	0	1
		Action	3	1	1
		Attribute	3	1	1
	Rule #2	*Message*	1	0	0
CHART	**Rule #1**	*Class*	13	3	3
		Action	12	4	4
		Attribute	9	7	7
	Rule #2	*Message*	6	2	1

Table 5.4: Experiment Results

Case study			**Precision**	**Recall**	**F-Measure**
GEN	**Rule #1**	*Class*	1.00	0.80	0.89
		Action	0.75	0.75	0.75
		Attribute	0.75	0.75	0.75
	Rule #2	*Message*	1.00	1.00	1.00
CHART	**Rule #1**	*Class*	1.00	0.80	0.89
		Action	0.75	0.75	0.75
		Attribute	0.75	0.75	0.75
	Rule #2	*Message*	1.00	1.00	1.00

Table 5.5: Required time comparison (minutes)

Activities	**Manual**		**Our approach**	
	GEN	CHART	GEN	CHART
Requirements analysis	28	62	1	1
UTP models deriving	8	33	1	1
Total	36	95	2	2
Reduction			94.4%	97.9%

(5.2) and CHART's text structure. Attributes of UTP models should be generated from NN2 in our generation rule; however, there are more complex structures such as multiple NP in CHART's requirements. Another reason is due to the writing style of the case study. The sentences are simply written as subject (NP) and verb (VP) and are continued with more conditions and actions for other information in the sentences.

Our approach is more effective for simple sentences in the requirements. Expression 5.2 is based on a simple sentence. Our approach is also effective for compound sentences. Compound sentences consist of two or more simple sentences with coordinating conjunctions; for example, *and*, *or*, *but*, and so on. Compound sentences have the same structure as simple sentences. We can apply our approach to each simple sentence in compound sentences. It is, however, difficult to apply our approach to complex sentences. Complex sentences consist of two or more simple sentences with subordinating conjunctions; for example, *when*, *if*, *while*, and so on. In complex sentences, the subject, verb, or compliment is missing from each of the simple sentences. Complex sentences do not have the same structure as simple sentences. Therefore, it is difficult to apply rule #1 of our approach to complex sentences.

Table 5.5 shows comparison results between the manual approach and our approach in required time to generate UTP models. Activities of generation UTMP models were divided into requirements analysis and derived UTP models. Experiment environments have a 2.60 GHz CPU, 8GB memory, and use a natural language tool kit [Pro15]. The manual approach is executed by a software testing expert who has 21 years experiences in software development and testing. Our approach reduces time from 94.4% to 97.9% corresponding with the manual generation of UTP models. This required time does not include the implementation time of our approach. The implementation time is not continuous work. We therefore ignore the implementation time in this evaluation. The results of the required time reduction show the effectiveness of our approach.

We determined that our technique could generate UTP models automatically from requirements in natural language with improvements in the text tree. As for future research, we will develop more generation rules and apply more actual industry case studies.

5.2 Automatic Generation of Test Cases Using Document Analysis Techniques

5.2.1 Motivations in Automated Creation of Software Testing Cases

Software testing verifies and validates software as being consistencies with the requirements and design specifications. Effective software testing can improve software quality, but it is expensive. As software becomes larger and complex, the costs of software testing can rise exponentially. While both the time for delivery and the work of software maintenance are must be reduced, one key is to improve the efficiency of a software testing. The activities of software testing consist of designing, refining and executing test cases [Uet13]. The activities of testing design and refinement are both important, because they impact the effectiveness and efficiency of software testing. Test cases are created during the test design work, so if too few test cases are created, then the functions of the software are not tested sufficiently and the defects will remain undetected in the software [Uet13].

In this paper, we target functional tests of Web application in software maintenance for the automatic generation of test cases using document analysis technique. Currently, test cases are usually created in three steps. Testing engineers (S1) read the specifications manually, (S2) identify the input parameters and values for the Web screens input parameters, and (S3) apply specialized techniques such as boundary analysis using their expert knowledge. The quality and coverage of the resulting test cases depends upon the skills of the testing engineers, leading to these potential problems [MMT13]: * Improper understanding of the specifications. Misreading or overlooked documents, basically due to human errors can result in some parameters and values for the test cases being not properly identified. * Missing parameters and values. In design documents, the types of the parameters are described as character strings, numerical values, dates and so on. Testing engineers create test values of the parameters by using their own knowledge. * Insufficient combinatorial test cases. It is difficult to manually create combinatorial test cases, again depending upon individual skills in understanding the required test coverage. These kinds of problems motivate automatic generation of software test cases.

5.2.2 Creating Test Cases by Using Document Analysis Techniques

In this section, we discuss about our method to address these problems, reducing the need to depend upon individual skills and using document analysis techniques instead.

(a) Overview

Figure 5.7 shows our approach to automatic generation of test cases from the design documents. The document analysis tool reads and analyzes the design documents. Examples of design documents include data specifications for screens, screens design specifications, and event process specifications. By using text parsers to analyze the documents, the analysis tool can output parameters and values with boundary analysis, event conditions, events and expected results. Then our pairwise testing tool uses the parameters and values to create combinatorial test cases [STBF11]. We applied our method to Japanese documents in this paper.

The document analysis tool divides its results for the test cases into two parts. The first part is the test data and conditions and the second part is the expected results. The test data and conditions consist of parameters and values, event conditions and events. The expected results are the expected results of execution with the test data and conditions and are described at the bottom of the test cases matrix.

(b) Problems in automatic generation of test cases from documents

In document-based automatic test case generation, we have two problems: (P1) How to automatically extract the parameter values, such as test conditions and test values, from document set; (P2) How to infer the test conditions and test data that are not explicitly described. The descriptions in the documents have different formats in each project and usually contain natural language descriptions for which information extraction is not easy. So we need flexible information extraction techniques to extract the parameter values from the documents. In addition, the target documents do not explicitly mention all of the required parameter values. This is because such documents are not intended to describe how to test the system. The informa-

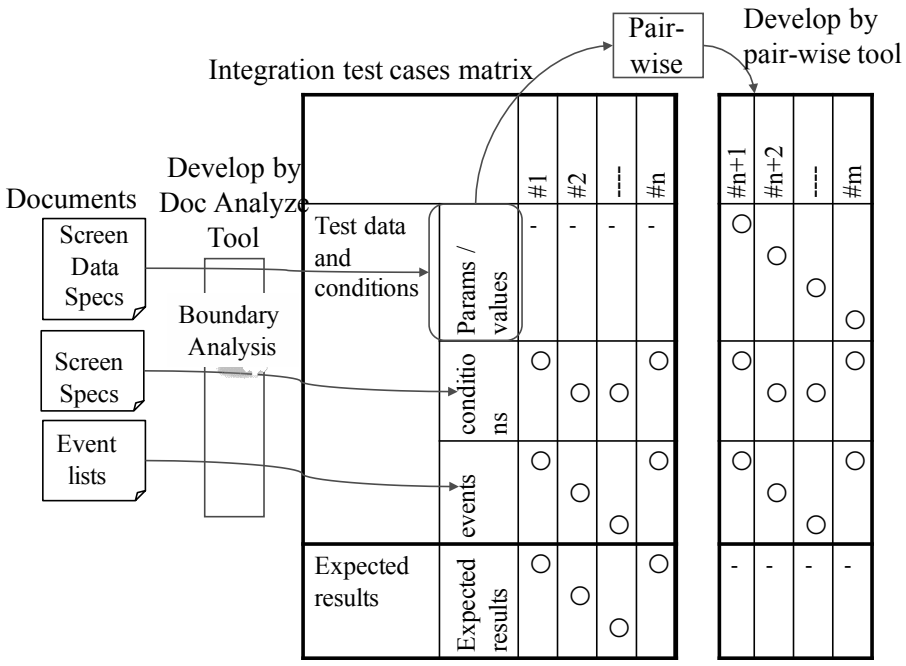

Figure 5.7: Automatic creating test cases by using document analysis techniques

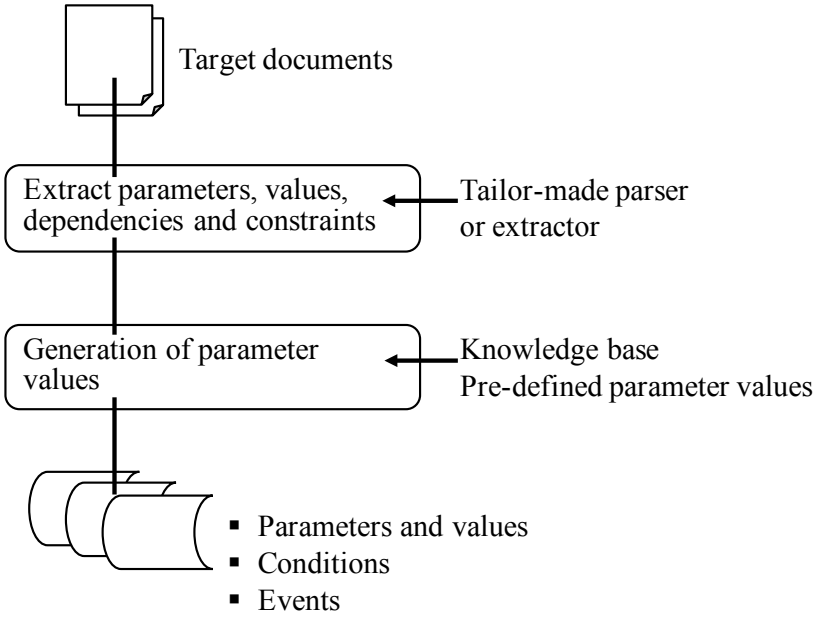

Figure 5.8: Overview of automatic extracting parameters and values from design documents

tion that is not described explicitly needs to be complemented or extracted by knowledge about testing and about the systems under test. The extracted information needs to be automatically integrated into the test cases. This also requires special knowledge, such as how to combine parameters and values, which must be derived from the system under test. For these problems, we identify technologies and mechanism that support the automation in each test case generation phase, such as text parser and knowledge for extracting test values.

This figure is an overview of the automatic extraction of parameters and values from the design documents:

As shown in Figure 5.8, there are two main phases corresponding to the above two problems facing automatic test case generation. We discuss each phase next two sub sections:

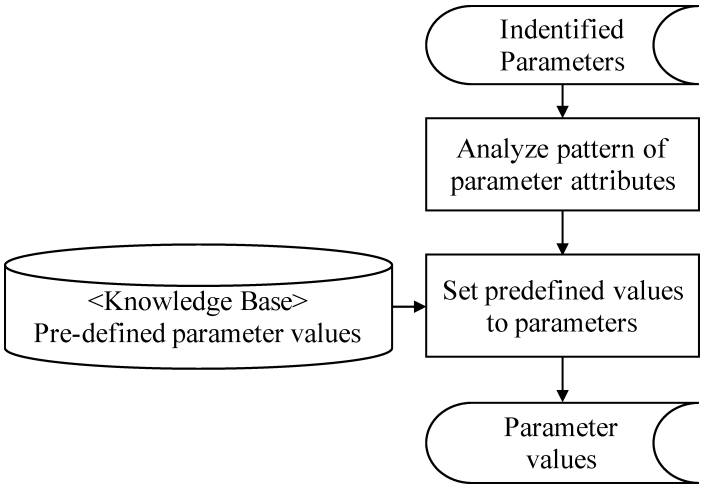

Figure 5.9: Generation of parameter values flow

(c) Generation of Parameter Values

Figure 5.9 shows the generation of parameter values flow. We need a flexible generation of parameter values to support various kinds of documents and description formats. Recent document analysis and parsing techniques [NTIM11],[INT12] support such flexible information extraction. Document modeling and format checking [NTIM11] support the information extraction for various user-defined document-structures and our text parsers combination system [INT12] makes it easy to create various information extraction parsers at the text description level. For examples, by using the system [INT12] and combining some formal or natural language parsers, we can flexibly create text parsers that extract the parameters, conditions, values and some dependencies from the text descriptions in target documents.

The knowledge of information which is used for identification candidates of parameter values is information that accumulated the results of the boundary analysis pattern from the attributes of the factor in description of specifications. In this paper, we use information which is the results from identify parameter values by patterning the attributes information according with patterns in table 5.6.

Document analysis tool identify parameter values by suing boundary analysis and apply predefined values from the results of analysis about the

Table 5.6: Knowledge Pattern of Parameter Values

Pattern No.	Pattern of parameter values
A1	Character or numeric
A2	Maximum number of characters
A3	Double bytes characters or not
A4	Single byte character or not

Table 5.7: Example for Pattern of 4-digit Integer Parameter Values

Pattern No.	Pattern of parameter values	Objectives of testing
B1	-1	To test error process for negative values
B2	0	To test error process for zero values
B3	1 and 9999	To test normal process for 4-digit integer
B4	10000	To test error process for 5-digit integer

Table 5.8: Example for Pattern of 6-digit Integer Parameter Values

Pattern No.	Pattern of parameter values	Objectives of testing
C1	ABCDEFG	To test 7-digit characters
C2	ABCDEF	To test 6-digit characters
C3	ABCDE	To test 5-digit characters
C4	アイウエオ	To test 5-digit double bytes Katakana
C5	123	To test numeric

attributes pattern, such as a character string, a numerical value and length, and a digit number, and so on. This enables not to depend on individual skill, the parameter values can be identified a level of coverage and used for test cases. Comparing create test cases by manually, this way makes test cases have better quality and more efficient in creation of test cases.

For examples about predefined values in the case of the attribute of the integer of 4-digit, by an experienced person's knowledge, parameter values are created by the patterns showed in table 5.7. The pattern number from B1 to B4 come from the results of boundary analysis and B5 comes from the knowledge of problem information. The document analysis tool creates the predefined values according with the attributes of values.

Another example about predefined vales in the case of the attribute of the character string of 6 digits, by using the knowledge of experienced person the predefined parameter values are created in a similar manner as table 5.8. The pattern number from C1 to C4 come from the results of boundary analysis and C5 and C6 comes from the knowledge of problem information.

(d) Problems in automatic generation of test cases from documents

In the last step, the pairwise testing tool creates combinatorial test cases from the identified parameters and values. So we can create the combinatorial pairwise test cases not depending upon individual skill. The test cases have a level of quality as high as pairwise combination, and the work of generation test cases are reduced by using the tool.

Each technique in our method is known techniques, such as text parser, boundary analysis, pairwise testing and so on. Our method is, however, unique for combining of them to create test cases automatically from design documents. Our method also solves the problems which are miss-reading of the documents, lacks of identification of parameter values, and insufficient of a combination test cases. Comparing manual test cases as ideal test case [INT12], our document analysis tool helped automatically generate 95% of the required test cases from the design documents.

5.2.3 Experiments

We applied our method to testing for Web applications in internet banking system and insurance system. The case studies are functional testing by

inputting test data from Web screens. The Web application documents are Web screens parameter specifications which describe attributes of parameter on Web screens input data, Web screens design documents which describe screens layout and precondition of events, and an event lists which describe the post processes of the events by clicking the buttons. In order to compare activities and work between manual way of creating test cases and our method way of automatic generation of test cases, each activity of creating testing cases are listed in Table 5.9. The list of activities shows activity number, name of activities, breakdown activities and comparing manual way and our method.

We applied our method to the following four case studies:

- Case 1: The generation test cases from an Internet Banking system project.
- Case 2: The generation test cases from another Internet Banking system project.
- Case 3: The generation test cases from Insurance system the input data of a batch job.
- Case 4: The generation test cases from Insurance system the screens of Web application

Table 5.10 shows the results of work. Each of case studies is compared between manual way and our method. The work in the generation test cases by our method was reduced from 23% to 48%. So our method improved work of generation test cases about twice as much as manual way.

5.2.4 Evaluations

We devised a method of automatic generation of test cases by using document analysis technique and applied the method to several case studies. The work for the generation of test cases was reduced by up to 48%. We demonstrated that our new method improved the effectiveness creating test

[1]About the workload of the case 3 and 4, the work of manual method is from interviews and the work of our method are calculated out on paper

[2]The case 3 did not have event description of a button, a link, etc. for batch input data.

[3]M(h): Manual results(hours), O(h): Our method results(hours)

Table 5.9: Creating Test Cases Activities Comparison

Act No.	Activities	Break down activities	Manual way	Our method
1-1	Create test data and variation	Analyze document	Analyze manually	Analyze by tools
1-2		Create test data	Identify manually	Identify by tools
1-3		Create parameter values	Create manually	Create by tools
2-1	Identify preconditions	Analyze document	Analyze manually	Analyze by tools
2-2		Identify preconditions	Identify manually	Identify by tools
3-1	Identify event conditions	Analyze document	Analyze manually	Analyze by tools
3-2		Identify event conditions	Identify manually	Identify by tools
4-1	Create test cases	Create test cases which verify conditions	Create manually	Create test cases by tools.
4-2		Create combinatorial test cases.	Create manually	Create by pairwise tools
5	Review test cases	Review test cases	Review manually	Review manually

Table 5.10: Workload Comparison of Test Case Generation

	Case1		Case2		Case3[1]		Case4[1]	
No.	M(h)[3]	O(h)[3]	M(h)	O(h)	M(h)	O(h)	M(h)	O(h)
1-1	5	1	5	1	2	1	1	1
1-2	12	1	5	1	3	1	2	1
1-3	10	4	10	3	4	2	2	1
2-1	5	1	5	1	2	1	1	1
2-2	13	5	15	4	6	3	3	2
3-1	8	1	5	1	-[2]	-[2]	1	1
3-2	15	5	15	4	-[2]	-[2]	3	2
4-1	62	42	60	50	10	8	4	4
4-2	38	20	40	30	10	7	4	3
5	15	15	10	10	3	3	1	1
Tot.	183	95	170	105	40	26	22	17
Reduction of work	48%		38%		35%		23%	

cases and also studied how we could apply our method to actual cases. If we can automate the review of the test cases, then the improvement will be larger. In the actual case studies, all of the design documents were not formalized according to the document standards, because the participants had been allowed to freely describe their design specifications. Notwithstanding, our text parser was able to read their document and create up to 95% of the required test cases for satisfactory testing.

5.3 Chapter Summary

In this chapter, I discuss two applications of our technique. We presented automatic generation test models from requirements in natural language by focusing on descriptions of test cases in UTP test behavior. We developed three rules to generate test models from requirements by using natural language processing techniques and experimented with our approach on requirements in language that is considered natural English. Our results for three case studies show the promise of our approach.

We will use our approach to find vague requirements and provide feedback in the early stage of the system development process. In addition, we will construct new rules and grammar for requirements descriptions. We will

contribute to requirement engineering by developing new means to check whether descriptions have vague or inconsistent requirements.

In this paper, we discussed a method of automatic generation testing cases by using document analysis techniques. We also discussed four case studies which demonstrated the efficiency of our approach in creating high-coverage test cases by using the document analysis tool. The method targets functional testing of the interaction screens for Web application systems. The method uses text parsers that identify input parameters and their acceptable values by using document analysis on the design documents, thus avoiding dependencies upon individual skills.

There are still some problems with how we extract the information from the design documents. Future work includes applying on other actual cases in order to get learn how we can configure the text parsers for natural languages. We are also studying and developing document models of the design documents to help formalize the documents. We will use our framework to find vague requirements and provide feedback in early stage of the system development process. In addition, we will construct new rules and grammar for requirements descriptions. We will contribute to requirement engineering by developing new means to check whether descriptions have vague or inconsistent requirements.

Chapter 6

Conclusion

We discussed studies about:

- rules of creating test cases by using natural language processing at section 3.1
- syntactic analysis as pre-processing as improvement creating test cases for both Japanese and English specification documents at section 3.2 and 3.3
- detecting logical inconsistencies in natural language requirements as further study at section 4.1
- as applications of our technique, creating test cases from UML (Unified Modeling Language) document at section 5.1 and combine our technique and combinatorial testing at section 5.2

We proposed the analysis technique, a semantic analysis technique of logics retrieval for software testing from Japanese public sector' s specification documents concept, technique and presented the results of experiments. The result was that the precision reached 0.93 to 0.97 and recall reached 0.65 to 0.79. That showed the analysis technique worked for retrieving condition logics. We confirmed the analysis technique could retrieve logics from Japanese natural language specification documents. When we target to retrieve condition logics, the number of keywords for conditions limited. Then the analysis technique works for retrieval conditions from specification documents.

This result is the starting point to research about harmonization between natural language processing and software testing. Matsuodani2012 [Mat12]

also proposed the benefits of retrieving logics from specification documents into decision tables and surveyed the opportunities of the future of decision table. The analysis technique can detect logic ambiguity of specification documents and feedback measurements for document quality. The measurements of specification documents are proposed by Kim2008 as quality metrics, for examples, document defect density, document reusability and so on. The analysis technique can feedback to how we write manually specification documents precisely. The more correct we can describe logic on specification documents in advance, the less workload to fix of incorrect logic.

Decision table testing is a technique to develop test cases from descriptions of conditions and actions in software specification documents. We propose, experiment and evaluate a semantic role labeling technique of conditions and actions for automatic software test cases generation. Our approach uses natural language processing to select sentences from the specification based on syntactic similarity, and then to determine conditions and actions through dependency and case analysis. We got experiment results higher precision and recall for different style of descriptions, and the workload was reduced to one-sixth of manual work. Our results on case studies show the effectiveness of our technique. We will research about feedback for engineer to write documents easier understand by improving our technique.

We proposed, experimented upon, and evaluated a technique for extracting the conditions and actions of test cases for automatic software test case generation. Our approach uses natural language processing to select sentences from the specifications on the basis of syntactic similarity and then determines the conditions and actions through dependency and case analysis. Experimental results showed that F-measure reached from 0.70 to 0.77 for different styles of description. Our results on case studies demonstrate the effectiveness of our technique. For our future work, we will extend our approach to give feedback to developers so that they can improve their requirements descriptions.

We presented a practical approach to detecting logical inconsistencies by clustering technique in natural language requirements. The method uses k-means clustering to cluster chunks of the requirements and labeling rules to derive “conditions” and “actions” as semantic roles from the requirements by using natural language processing. We also constructed an abstraction grammar to transform the conditions and actions into logical formulas. By evaluating the logical formulas with input data patterns, we can find logical inconsistencies. We experimented with this approach on three case studies

of requirements written in natural English. The results indicate that our approach can find logical inconsistencies.

We presented automatic generation test models from requirements in natural language by focusing on descriptions of test cases in UTP test behavior. We developed three rules to generate test models from requirements by using natural language processing techniques and experimented with our approach on requirements in language that is considered natural English. Our results for three case studies show the promise of our approach.

We will use our approach to find vague requirements and provide feedback in the early stage of the system development process. In addition, we will construct new rules and grammar for requirements descriptions. We will contribute to requirement engineering by developing new means to check whether descriptions have vague or inconsistent requirements.

In this paper, we discussed a method of automatic generation testing cases by using document analysis techniques. We also discussed four case studies which demonstrated the efficiency of our approach in creating high-coverage test cases by using the document analysis tool. The method targets functional testing of the interaction screens for Web application systems. The method uses text parsers that identify input parameters and their acceptable values by using document analysis on the design documents, thus avoiding dependencies upon individual skills.

There are still some problems with how we extract the information from the design documents. Future work includes applying on other actual cases in order to get learn how we can configure the text parsers for natural languages. We are also studying and developing document models of the design documents to help formalize the documents. We will use our framework to find vague requirements and provide feedback in early stage of the system development process. In addition, we will construct new rules and grammar for requirements descriptions. We will contribute to requirement engineering by developing new means to check whether descriptions have vague or inconsistent requirements.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Prof. Kazuhiko Tsuda for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

I would like to thank the rest of my thesis committee: Prof. Kenichi Yoshida, Prof. Setsuya Kurahashi, Assoc. Prof. Kazuki Katagishi, and Assoc. Prof. Chika Yoshida for their insightful comments and encouragement, but also for the hard question which incented me to widen my research from various perspectives.

My heartfelt appreciation to Dr. Tohru Matsuodani for letting me in this research and sincere encouragement.

I gratefully acknowledge the work of past and present members of our laboratory and I would like to thank my working company for my study support, which I am working for.

Last but not the least, I would like to thank my family for supporting me spiritually throughout writing this thesis and my life in general.

References

[708] ISO/IEC JTC 1/SC 7. Iso/iec 12207:2008 systems and software engineering – software life cycle processes. *ISO/IEC JTC 1/SC 7*, 2008.

[715] ISO/IEC/IEEE JTC 1/SC 7. Software and systems engineering — software testing — part 2:test processes. *ISO/IEC/IEEE JTC 1/SC 7*, 2015.

[Adm03] Maryland State Highway Administration. Detailed system design specification for the coordinated highways action response team (chart) mapping applications. *http://www.chart.state.md.us/*, 2003.

[AS12] M. Aggarwal and S. Sabharwal. Test case generation from uml state machine diagram: A survey. In *Computer and Communication Technology (ICCCT), 2012 Third International Conference on*, pp. 133–140, Nov 2012.

[Ass86] Japan Industry Standard Association. Jis x 0125-1986 decision table. *Japan Industry Standards*, 1986.

[Bac16] James Bach. Allpairs test case generation tool (version 1.2.1). 2016.

[BHvM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, Vol. 185. ios press, 2009.

[Bos08] Johan Bos. Wide-coverage semantic analysis with boxer. In *Proceedings of the 2008 Conference on Semantics in Text Processing*, pp. 277–286. Association for Computational Linguistics, 2008.

[BSBV13] A. Bagnato, A. Sadovykh, E. Brosse, and T. E. J. Vos. The omg uml testing profile in use–an industrial case study for the future internet testing. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pp. 457–460, March 2013.

[CB98] IEEE Computer Society. Software Engineering Standards Committee and IEEE-SA Standards Board. Ieee recommended practice for software requirements specifications. Institute of Electrical and Electronics Engineers, 1998.

[CDFP97] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The aetg system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, Vol. 23, No. 7, pp. 437–444, Jul 1997.

[CDPP96] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, Vol. 13, No. 5, pp. 83–88, Sep 1996.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pp. 322–335, New York, NY, USA, 2006. ACM.

[CKI88] Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Commun. ACM*, Vol. 31, No. 11, pp. 1268–1287, November 1988.

[CMB11] Danilo Croce, Alessandro Moschitti, and Roberto Basili. Structured lexical similarity via convolution kernels on dependency trees. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, pp. 1034–1046, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[Dis04] Southwest Florida Water Management District. Comprehensive watershed management water use tracking project

software requirements specification. *http://open.sjrwmd.com*, 2004.

[DMDS+14] Marie-Catherine De Marneffe, Timothy Dozat, Natalia Silveira, Katri Haverinen, Filip Ginter, Joakim Nivre, and Christopher D Manning. Universal stanford dependencies: A cross-linguistic typology. In *Proceedings of LREC*, pp. 4585–4592, 2014.

[DMM08] Marie-Catherine De Marneffe and Christopher D Manning. Stanford typed dependencies manual. Technical report, Technical report, Stanford University, 2008.

[ES04] Niklas Een and Niklas Sorensson. An extensible sat-solver. In *Theory and applications of satisfiability testing*, pp. 502–518. Springer, 2004.

[fE12] The United Nations Economic Commission for Europe. Business requirements specifications of legal notice publication. *http://www1.unece.org*, 2012.

[FH14] Markus Fockel and Jorg Holtmann. A requirements engineering methodology combining models and controlled natural language. In *Model-Driven Requirements Engineering Workshop (MoDRE), 2014 IEEE 4th International*, pp. 67–76. IEEE, 2014.

[Fuk88] Naoki Fukui. Deriving the differences between english and japanese: A case study in parametric syntax. *English Linguistics*, Vol. 5, No. 0, pp. 249–270, 1988.

[GEMT06] Javier Jesus Gutierrez, Maria Jose Escalona, Manuel Mejias, and Jesus Torres. An approach to generate test cases from use cases. In *Proceedings of the 6th international conference on Web engineering*, pp. 113–114. ACM, 2006.

[GZ05] Vincenzo Gervasi and Didar Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 14, No. 3, pp. 277–330, 2005.

[HPS05] Brahim Hnich, Steven Prestwich, and Evgeny Selensky. Constraint-based approaches to the covering test problem. In *Recent Advances in Constraints*, pp. 172–186. Springer, 2005.

[HPSS06] Brahim Hnich, Steven D Prestwich, Evgeny Selensky, and Barbara M Smith. Constraint models for the covering test problem. *Constraints*, Vol. 11, No. 2-3, pp. 199–219, 2006.

[INT12] F. Iwama, T. Nakamura, and H. Takeuchi. Constructing parser for industrial software specifications containing formal and natural language description. In *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1012–1021, June 2012.

[IPA09] IPA.METI. Report of industry actual survey for embedded software in 2009. 2009. in Japanese.

[ISO15a] ISO/IEC/IEEE. Software and systems engineering — software testing — part 1:concepts and definitions. *ISO/IEC/IEEE JTC 1/SC 7*, pp. 70–72, 2015.

[ISO15b] ISO/IEC/IEEE. Software and systems engineering — software testing — part 4:test techniques. *ISO/IEC/IEEE JTC 1/SC 7*, pp. 70–72, 2015.

[Kat07] Takuya Katayama. Legal engineering-an engineering approach to laws in e-society age. *Procedure of the 1st Intl. Workshop on JURISIN*, 2007.

[Kis03] Shuhei Kishimoto. Government procurement system and it system -who the raised or it general constructor. *Finance*, Vol. 38, No. 12, pp. 37–47, 2003.

[KK] Sadao Kurohashi and Daisuke Kawahara. Resources for natural language. in Japanese.

[KKS08] CJ Kim, S-M Kim, and K-W Song. Measurement of level of quality control activities in software development [quality control scorecards]. In *Convergence and Hybrid Information Technology, 2008. ICHIT'08. International Conference on*, pp. 763–770. IEEE, 2008.

[KKed] Sadao Kurohashi and Daisuke Kawahara. Feature list given by knp. 2016 Accessed. in Japanese.

[KMN+02] Tapas Kanungo, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. An efficient k-means clustering algorithm: Analysis and implementation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, Vol. 24, No. 7, pp. 881–892, 2002.

[KNS08] Yusuke Kimura, Makoto Nakamura, and Akira Shimazu. Trasnformation logical expression from legal documents which have listing and references. *Assocaiton for natural Language Processing*, Vol. 14th Annual Conference, pp. 612–615, 2008.

[KSKD13] Oliver Keszocze, Mathias Soeken, Eugen Kuksa, and Rolf Drechsler. Lips: An ide for model driven engineering based on natural language processing. In *Natural Language Analysis in Software Engineering (NaturaLiSE), 2013 1st International Workshop on*, pp. 31–38. IEEE, 2013.

[Kun73] Susumu Kuno. The structure of the japanese language. *Cambridge, MA:MIT Press.*, 1973.

[KY02] Taku Kudoh and Matsumoto Yuhji. Japanese dependency analysis using cascaded chunking. *Journal of Information Processing Society of Japan*, Vol. 43, No. 6, pp. 1834–1842, 2002.

[Mat12] Tohru Matsuodani. Application of decision table to software logic with designing and testing. *in Reliability Enginerring Association of Japan*, Vol. 34, No. 6, pp. 397–404, 2012.

[MFIMA11] Wendland Marc Florian, Schieferdecker Ina, Schacher Markus, and Metzger Armin. Uml testing profile tutorial. *MBT User Conference*, 2011.

[MIH+15] Satoshi Masuda, Futoshi Iwama, Nobuhiro Hosokawa, Tohru Matsuodani, and Kazuhiko Tsuda. Semantic analysis technique of logics retrieval for software testing from specification documents. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pp. 1–6. IEEE, 2015.

[MMM06] M. Marneffe, B. Maccartney, and C. Manning. Generating typed dependency parses from phrase structure parses. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC-2006)*, Genoa, Italy, May 2006. European Language Resources Association (ELRA). ACL Anthology Identifier: L06-1260.

[MMT13] Satoshi Masuda, Tohru Matsuodani, and Kazuhiko Tsuda. A method of creating testing pattern for pair-wise method by using knowledge of parameter values. *Procedia Computer Science*, Vol. 22, pp. 521–528, 2013.

[MOASHL09] M. Mussa, S. Ouchani, W. Al Sammane, and A. Hamou-Lhadj. A survey of model-driven testing techniques. In *Quality Software, 2009. QSIC '09. 9th International Conference on*, pp. 167–172, Aug 2009.

[MWM+11] Junta Mizuno, Yotaro Watanabe, Kohji Murakami, Kentaro Inui, Yuji Matsumoto, et al. Organizing agreeing and conflicting opinions based on semantic relation recognition. *Journal of Information Processing*, Vol. 52, No. 12, pp. 3408–3422, 2011. in Japanese.

[NDM11] Tuan Doc Nguen, Bollegara Danushika, and Ishizuka Mitsuru. Exploiting relational similarity between entity pairs for latent relational search. *Information Processing Society Japan*, Vol. 52, No. 4, pp. 1–13, 2011.

[NL11] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, Vol. 43, No. 2, pp. 11:1–11:29, February 2011.

[NT11] T. Nakamura and H. Takeuchi. Document quality verification tool. *ProVision*, Vol. 69, pp. 78–79, 2011. In Japanese.

[NTAed] National-Tax-Agency. No.1140 life insurance deduction. 2015 Accessed. in Japanese.

[NTIM11] T. Nakamura, H. Takeuchi, F. Iwama, and K. Mizuno. Enabling analysis and measurement of conventional software development documents using project-specific formalism. In

2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement, pp. 48–54, Nov 2011.

[Oku10] Manabu Okumura. *Basic of Natural Language Processing*. CORONA PUBLISHING CO.,LTD., 2010. in Japanese.

[OMG14] OMG. Uml testing profile version 1.2. 2014.

[OMG15] OMG. Unified modeling language version 2.5. 2015.

[OW13] Yusuke Oda and Shigeru Wakabayashi. Method of similarity quantification between program codes. *Jounal of Kobe City College of Technology*, Vol. 51, pp. 103–108, 2013. in Japanese.

[PM00] Dan Pelleg and Andrew W. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pp. 727–734, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[Pro15] NLTK Project. Natural language toolkit. *http://www.nltk.org/*, 2015.

[RAF+10] D Reifer, J Allen, B Fersch, B Hitchings, J Judy, W Rosa, and D Saltojanes. Total cost of software maintenance workshop. *approved for public release, review by AMRDEC, public affair office, FN4344, at"http://"csse. usc. edul csse/.. ISoftware% 20maintenance*, Vol. 20, , 2010.

[SB10] M. Sharma and S. C. B. Automatic generation of test suites from decision table - theory and implementation. In *2010 Fifth International Conference on Software Engineering Advances*, pp. 459–464, Aug 2010.

[SHE89] Motoshi Saeki, Hisayuki Horai, and Hajime Enomoto. Software development process from natural language specification. In *Proceedings of the 11th international conference on Software engineering*, pp. 64–73. ACM, 1989.

[Shh12] Akira Shhimazu. Legal engineering: Methodology for designing trustworthy social systems -legal document analysis-. *IEICE Foundamentals Review*, Vol. 5, pp. 320–328, 2012.

[SM07] M. Sarma and R. Mall. Automatic test case generation from uml models. In *Information Technology, (ICIT 2007). 10th International Conference on*, pp. 196–201, Dec 2007.

[Sne07] Harry M Sneed. Testing against natural language requirements. In *Quality Software, 2007. QSIC'07. Seventh International Conference on*, pp. 380–387. IEEE, 2007.

[SNK14] Takase Sho, Okazaki Naoki, and Inui Kentaro. Clustering of relation pattern by a method of high speed similarity calculation. *The Association for Natural Language Processing Annual Conference*, Vol. 20, No. 4, pp. 1–4, 2014.

[SPKB09] Avik Sinha, Amit Paradkar, Palani Kumanan, and Branimir Boguraev. A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pp. 327–336. IEEE, 2009.

[SPTN10] A. Sinha, A. Paradkar, H. Takeuchi, and T. Nakamura. Extending automated analysis of natural language use cases to other languages. In *2010 18th IEEE International Requirements Engineering Conference*, pp. 364–369, Sept 2010.

[SSB15] R. Sharma, P.K. Srivastava, and K.K. Biswas. From natural language requirements to uml class diagrams. In *Artificial Intelligence for Requirements Engineering (AIRE), 2015 IEEE Second International Workshop on*, pp. 1–8, Aug 2015.

[STBF11] Itai Segall, Rachel Tzoref-Brill, and Eitan Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pp. 254–264, New York, NY, USA, 2011. ACM.

[TIM02] Tetsuro Takahashi, Kentaro Inui, and Yuji Matsumoto. Methods for estimating syntactic similarity. *Information Processing Society of Japan Natural Language Processing*, Vol. NL-150-7, , 2002. in Japanese.

[TKI12] Shunsuke Takayanagi, Atsushi Kamijo, and Tsutomu Ishikawa. Translator from japanese sentences to well-formed formulas on an extended predicate logic:conv. *Transactions of the Japanese Society for Artificial Intelligence*, Vol. 27, No. 5, pp. 271–280, 2012. in Japanese.

[TKN93] Kikuo Tanaka, Ichiro Kawazoe, and Hajime Narita. Standard structure of legal provisions - for the leagal knowledge processing by natural language -. *Information Processing Society of Japan Natural Language Processing*, Vol. 93, No. 79, pp. 79–86, 1993.

[TM07] Yan Tang and Robert Meersman. On constructing semantic decision tables. In *Database and Expert Systems Applications*, pp. 34–44. Springer, 2007.

[TN99] Kazuhiko Tsuda and Masami Nakamura. The extraction method of the word meaning class. In *Knowledge-Based Intelligent Information Engineering Systems, 1999. Third International Conference*, pp. 534–537. IEEE, 1999.

[TNY10] Hironori Takeuchi, Taiga Nakamura, and Takahira Yamaguchi. Use case analysis using text analytics. *Technical report of IEICE. KBSE*, Vol. 110, No. 305, pp. 55–60, 2010. in Japanese.

[TSN+07] H. Takeuchi, L. V. Subramaniam, T. Nasukawa, S. Roy, and S. Balakrishnan. A conversation-mining system for gathering insights to improve agent productivity. In *The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE 2007)*, pp. 465–468, July 2007.

[TTMM10] Koichi Takeuchi, Suguru Tsuchiya, Masato Moriya, and Yuuki Moriyasu. Construction of argument structure analyzer toward searching same situations and actions. *IEICE technical report. Natural language understanding and models of communication*, pp. 1–6, 2010. in Japanese.

[Uet13] Keiji Uetsuki. Research about a design technique of software testing by using decision table. *University of Tsukuba*, pp. 1–6, 2013. in Japanese.

[UMT13] Keiji Uetsuki, Tohru Matsuodani, and Kazuhiko Tsuda. An efficient software testing method by decision table verification. *International Journal of Computer Applications in Technology*, Vol. 46, No. 1, pp. 54–64, 2013.

[VAF10] Bernardo C. Vieira, Fabrício V. Andrade, and Antônio O. Fernandes. A modular cnf-based sat solver. In *Proceedings of the 23rd Symposium on Integrated Circuits and System Design*, SBCCI '10, pp. 198–203, New York, NY, USA, 2010. ACM.

[WPG+15] Chunhui Wang, Fabrizio Pastore, Arda Goknil, Lionel Briand, and Zohaib Iqbal. Automatic generation of system test cases from use case specifications. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pp. 385–396, New York, NY, USA, 2015. ACM.

[YCC15] Rongjie Yan, Chih-Hong Cheng, and Yesheng Chai. Formal consistency checking over specifications in natural languages. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pp. 1677–1682. EDA Consortium, 2015.

[YCed] Yokohama-City. System design document of fire station support system. 2014 Accessed. in Japanese.

[YMT15] Tsuyoshi Yumoto, Tohru Matsuodani, and Kazuhiko Tsuda. A study on an approach for analysing test basis using i/o test data patterns. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pp. 1–6. IEEE, 2015.

[ZO95] Yujie Zhang and Kazuhiko Ozeki. Statistical property of distance between modifier and modified phrases and its application to dependency analysis of japanese sentences. *IEICE technical report. Natural language understanding and models of communication*, Vol. 95, No. 429, pp. 61–68, 1995. in Japanese.