IEICE TRANS. INF. & SYST., VOL.E99–D, NO.9 SEPTEMBER 2016

PAPER Special Section on Knowledge-Based Software Engineering

Detecting Logical Inconsistencies by Clustering Technique in Natural Language Requirements

Satoshi MASUDA[†a)], ***Nonmember***, **Tohru MATSUODANI**[††], ***and*** **Kazuhiko TSUDA**[†††], ***Members***

SUMMARY 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.

key words: *requirement analysis, natural language processing, clustering*

1. Introduction

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. Logical consistency is the key to raising the quality and lowering the cost of system development. 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 [24] 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. In this paper, we present a practical approach to detecting logical inconsistencies by clustering technique in natural language requirements (See Fig. 1). 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.

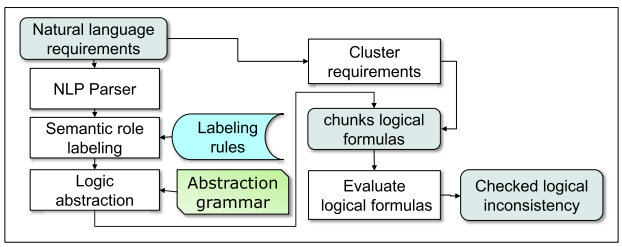

Fig. 1 Framework for detecting logical inconsistencies by clustering technique in natural language requirements

We implemented a proof-of-concept prototype of our framework. We used the natural language processing parser [4] 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 [17]. We evaluated our prototype on three

Manuscript received January 13, 2016.
Manuscript revised May 30, 2016.
Manuscript publicized July 6, 2016.

[†]The author is with IBM Research - Tokyo, Tokyo, 103–8511 Japan.
[††]The author is with Debug Engineering Research Laboratory, Tama-shi, 206–0022 Japan.
[†††]The author is with University of Tsukuba, Tokyo, 112–0012 Japan.
a) E-mail: smasuda@jp.ibm.com

DOI: 10.1587/transinf.2015KBP0005

case studies: "a detailed system design specification for the coordinated highways action response team (CHART) mapping applications" [14], "business requirements specifications of legal notice publication (eNotification)" [15], and "comprehensive watershed management water use tracking (WUT) project software requirements specification" [16].

The paper is structured as follows. We present our approach in Sect. 2, starting with a simple example illustrating its flow followed by a description of each step of the approach, i.e., the parsing and dependency analysis, semantic role labeling, logical abstraction grammar, evaluation of the logical formulas, and creation of the input data patterns. Section 3 discusses the experiments and the results. A summary of related work is presented in Sect. 4. We conclude the paper in Sect. 5.

2. Detecting Logical Inconsistencies by Clustering Technique in Natural Language Requirements

2.1 An example for Illustrating the Approach

Figure 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 [5] 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 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 compliment 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 [5]. Here we get the logical formula: ***P1 & P2*** $\rightarrow$ ***P3***. "&", "|", and "$\rightarrow$" 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 the input data patterns shown in Fig. 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***).

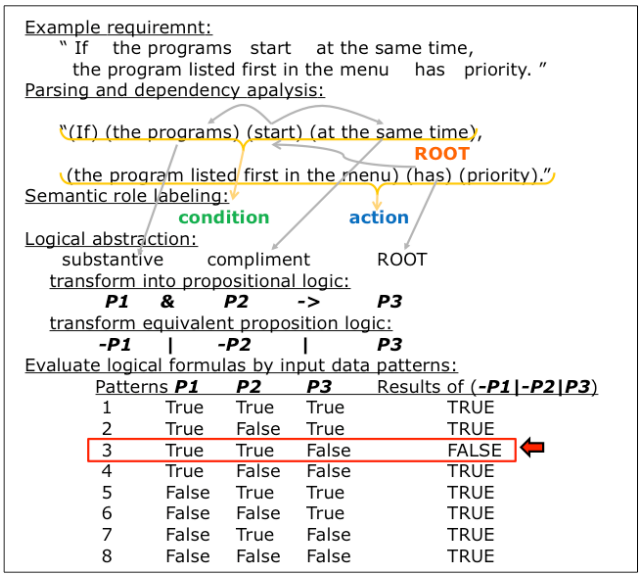

Fig. 2 Example

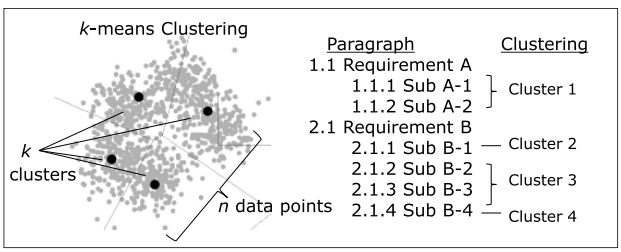

Fig. 3 Clustering natural language requirements (from Fig. 3 in [25])

2.2 Clustering Natural Language Requirements

We use the *k*-means clustering algorithm to cluster chunks of natural language requirements. Figure 3 illustrates *k*-means clustering and clustering paragraphs. *k*-means clustering is to partition *n* data points into *k* clusters [25] 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 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.

2.3 Parsing and Dependency Analysis

A natural language processing (NLP) parser assigns num-

Table 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

bers 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 1 shows the results of the dependency analysis of the example. Each parser defines a set of dependency types. We used the NLP parser [18].

The meanings of dependency types are defined in [5] 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.
- 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.

Semantic role labeling algorithm

Input: Results of parsing, dependency analysis
Output: "actions" and "conditions" clauses
1: Search ROOT word
2: Search **m(i)** in case **D(i)=T**
3: For 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)**⊆**Ra**
6: label **B(i)** as "conditions" in the case of **C(i)**⊆**Rc**
7: end for
8:

Fig. 4 Semantic role labeling algorithm

- 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.

2.4 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**

Below Fig. 4 shows the algorithm of semantic role labeling.

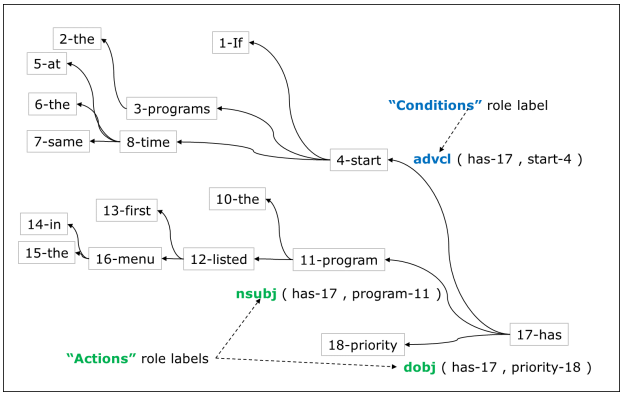

Fig. 5 Example of semantic role labeling

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

Fig. 6 Abstraction grammar

These definitions are used to interpret the requirements that have been parsed and whose dependencies have been analyzed as in Table 1. Figure 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 [5]. The output consists of clauses labeled as "actions" and "conditions".

Figure 5 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 part of **Ra1**; thus, it is determined as part of the condition from rule **Rc1**.

2.5 Abstraction Grammar of the Structured English

Figure 6 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 [1]. 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.

2.6 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, 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 [19], [22], [23]. 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 [21]. SAT also has semantic versions of validity and consistency that are defined in terms of the concept of structure [21]. It, however, 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.

2.7 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 Sect. 2.2 of the CHART case study. The number of combinations of these variables 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 [20]: (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 [20] which is a CT technique.

3. Experiments and Evaluations

3.1 Experiments

We implemented for a proof-of-concept prototype of our framework. We used the Natural Language Processing parser [4] and dependency analysis on natural language requirements. We developed our own tools using the Python natural language tool kit [17] for the semantic role labeling, abstraction grammar, and evaluation of the logical formulas.

We tested 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 [14]. 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 [15]. 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 [16]. We used the functional requirements section (4-1-1-1).

The experiments used two types of chunk: paragraph chunks chunked by paragraph number and clustered chunks 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 2 shows the chunked-by-paragraph results and

Table 2 CHART case study

	Paragrahs (CHART)			Clustering (CHART)	
	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 (*a)	8	8	8	16	8
Number of inconsistencies	4	4	4	15	4
Total number of incosistencies			**12**		**19**
Incosistency hit ratio (*b)			**0.583**		**0.719**

*a. Patterns (pairwise)
*b: Number of inconsistencies calculated from number of input patterns

Table 3 eNot case study

	Paragraphs (eNot)	Clustering (eNot)	
		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 (*a)	8	8	8
Number of inconsistencies	4	4	3
Total number of incosistencies	**4**		**7**
Incosistency hit ratio (*b)	**0.500**		**0.438**

*a. Patterns (pairwise)
*b: Number of inconsistencies calculated from number of input patterns

Table 4 WUT case study

	Paragraph (WUT)	Clustering (WUT)	
		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 (*a)	8	8	4
Number of inconsistencies	4	4	2
Total number of incosistencies	**4**		**6**
Incosistency hit ratio (*b)	**0.500**		**0.500**

*a. Patterns (pairwise)
*b: Number of inconsistencies calculated from number of input patterns

the chunked-by-clustering results in case study CHART. The columns show the number of instances of each artifact. Table 3 shows the chunked-by-paragraph results and the chunked-by-clustering results in case study eNotification (eNot). Table 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 fs 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 Fig. 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).

3.2 Evaluations

The inconsistency hit ratios in Table 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 approach found 12 total inconsistencies from a total of 24 data inputs, for which the inconsistency hit ratio was 0.583. Table 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 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 in Sect. 2, we discuss tools, steps and execution time. We used Minisat which is one of state-of-the-art SAT solver [22], [26]. We used Stanford NLP parser [5], [18] 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 [17]. We used Minisat for SAT solver, Allpairs for pairwise tool [27] 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 [28]. 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 [24]; 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. Related Work

Natural language processing and consistency checking are essential parts of requirement engineering. Yan2015 presented a formal consistency check of specifications written in natural language [1]. 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 [1]. Our framework differs from Yan2015 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 [19]. 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 [21]. 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 [20]. 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 [3].

Bos2007 describes a way to change natural sentences into logical expressions and devised the BOXER tool for English [6]. Masuda2015 used natural-language processing to identify the logical pattern "If (A) is (B), (C) is (D)" in sentences [2], but did not identify conditions or actions. There are also linguistic studies on use cases [10] and test case generation [13]. Sneed2007 made test cases from requirements described in natural language [11]. 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 [7], [8]. The standard describes the considerations that go into producing a good software requirements specification and provides templates. Kim2008 [9] presented a way of measuring the level of quality control in software development. They defined the notion of ambiguity. Uetsuki2013 presented an efficient software testing method [12] 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.

5. Conclusion

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.

References

[1] R. Yan, C.-H. Cheng, and Y. Chai, "Formal consistency checking over specifications in natural languages," Proceedings of the 2015 Design Automation Test in Europe Conference Exhibition, Dresden, Germany, no.452, pp.1677–1682, 2015

[2] S. Masuda, F. Iwama, N. Hosokawa, T. Matsuodani, and K. Tsuda, "Semantic analysis technique of logics retrieval for software testing from specification documents," Proceedings of IEEE Eighth International Conference on IEEE Software Testing, Verification and Validation Workshops (ICSTW), Graz, Austria, pp.1–6, 2015

[3] S. Masuda, T. Matsuodani, and K. Tsuda, "A Method of Creating Testing Pattern for Pair-wise Method by Using Knowledge of Parameter Values," Procedia Computer Science, Fukuoka, Japan, vol.22, no.k13is-063, pp.521–528, 2013

[4] M.-C. De Marneffe, B. MacCartney, and C.D. Manning, "Generating typed dependency parses from phrase structure parses," Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC-2006), Genoa, Italy, no.L06-1260, pp.449–454, 2006

[5] M.-C. De Marneffe and C.D. Manning, "Stanford typed dependencies manual," The Stanford Natural Language Processing Group, http://nlp.stanford.edu/downloads/dependencies_manual.pdf, accessed Sept. 20. 2015

[6] J. Bos, "Wide-coverage semantic analysis with boxer," Association for Computational Linguistics Proceedings of the 2008 Conference on Semantics in Text Processing, Stroudsburg, USA, pp.277–286, 2008

[7] 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

[8] International Organization for Standardization (ISO)/ International Electrotechnical Commission (IEC)/ Institute of Electrical and Electronics Engineers (IEEE), Software and systems engineering - Software testing - Part 4: Test techniques, ISO/IEC/IEEE JTC 1/SC 7, pp.70–72, 2015.

[9] C.J. Kim, S.-M. Kim, and K.-W. Song, "Measurement of Level of Quality Control Activities in Software Development [Quality Control Scorecards]," Proceedings IEEE Convergence and Hybrid Information Technology 2008 (ICHIT'08), Daejeon, Korea, pp.763–770, 2008

[10] A. Sinha, A. Paradkar, P. Kumanan, and B. Boguraev, "A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases," Proceedings IEEE/IFIP International Conference on IEEE Dependable Systems & Networks 2009 (DSN'09), Toulouse, France, pp.327–336, 2009.

[11] H.M. Sneed, "Testing against natural language requirements," Proceedings in IEEE Quality Software 2007 QSIC'07 Seventh International Conference on, Portland, USA, pp.380–387, 2007.

[12] K. Uetsuki, T. Matsuodani, and K. Tsuda, "An efficient software testing method by decision table verification," Proc. International Journal of Computer Applications in Technology, vol.46, no.1, pp.54–64, 2013.

[13] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, "Automatic generation of system test cases from use case specifications," Proceedings ACM of the 2015 International Symposium on Software Testing and Analysis, Baltimore, USA, pp.385–396, 2015.

[14] Maryland State Highway Administration, "Detailed System Design Specification for the Coordinated Highways Action Response Team (CHART) Mapping Applications," http://www.chart.state.md.us/downloads/readingroom/ChartIntranetMapping/Detailed_System_Design_Specification_Final.doc, accessed Sept. 20. 2015

[15] The United Nations Economic Commission for Europe (UNECE), "Business Requirements Specifications of Legal Notice Publication," http://www1.unece.org/cefact/platform/download/attachments/45449366/eNotification_LNP_BRS_20120914.v_1_0_1.doc, accessed Sept. 20. 2015

[16] Southwest Florida Water Management District, "Comprehensive Watershed Management Water Use Tracking Project Software Requirements Specification," http://open.sjrwmd.com/ep/docs/Elaboration/Software%20Requirements%20Specification.doc, accessed Sept. 20. 2015

[17] Natural Language Tool Kit (NLTK) Project, "Natural Language Toolkit," http://www.nltk.org/, accessed Sept. 20. 2015

[18] M.-C. De Marneffe, T. Dozat, N. Silveira, K. Haverinen, F. Ginter, J. Nivre, and C.D. Manning, "Universal Stanford Dependencies: A cross-linguistic typology," Proceedings of the Nineth International Conference on Language Resources and Evaluation (LREC-2014), Reykjavik, Iceland, pp.4585–4592, 2014.

[19] B. Hnich, S.D. Prestwich, E. Selensky, and B.M. Smith, "Constraint models for the covering test problem," Constraints, vo.11, no.2, pp.199–219, July 2006.

[20] C. Nie and H. Leung, "A survey of combinatorial testing," ACM Computing Surveys (CSUR), vol.43, no.2, pp.11:1–11:29, Feb. 2011.

[21] A. Biere, M. Heule, and H. van Maaren, Handbook of satisfiability, IOS Press, Amsterdam, 2009.

[22] N. Een and N. Sorensson, "An extensible SAT-solver," Theory and Applications of Satisfiability Testing, Lecture Notes in Computer Science, vol.2919, pp.502–518, 2004.

[23] B. Hnich, S. Prestwich, and E. Selensky, "Constraint-based approaches to the covering test problem," Recent Advances in Constraints, vol.3419, pp.172–186, 2005.

[24] V. Gervasi and D. Zowghi, "Reasoning about inconsistencies in natural language requirements," ACM Transactions on Software Engineering and Methodology (TOSEM), vo.14, no.3, pp.277–330, 2005.

[25] T. Kanungo, D.M. Mount, N.S. Netanyahu, C.D. Piatko, R. Silverman, and A.Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," IEEE Transactions on Pattern Analysis and Machine Intelligence, vo.24, no.7, pp.881–892, 2002.

[26] B.C. VIEIRA, F.V. Andrade, and A.O. Fernandes, "A Modular CNF-based SAT Solver," Proceedings of the 23rd symposium on Integrated circuits and system design: ACM, pp.198–203, 2010

[27] B. James, "ALLPAIRS Test Case Generation Tool (Version 1.2.1)," Satisfice, Inc., http://www.satisfice.com/ tools.shtml, access May 23, 2016

[28] D. Pelleg and A.W. Moore, "X-means: Extending K-means with Efficient Estimation of the Number of Clusters," Proceedings of the Seventeenth International Conference on Machine Learning, pp.727–734, 2000.

Satoshi Masuda received his B.S. in Mechanical Engineering from Saitama University in 1991. In 1991, he joined IBM Japan Ltd. and engaged software engineering in enterprise application systems. His technical expertise ranges from software testing to application architecture. In 2014, he transferred to IBM Research - Tokyo. His research area is software engineering. He is a senior member of The Information Processing Society Japan and a member of IEEE Computer Society.

Tohru Matsuodani received his Ph.D. from Tsukuba University in 2005. He is a part-time lecturer in Hosei University. He is a member of NEC, where he has developed hardware and operating systems. He has been chief executive officer of the Debug Engineering Laboratory. He is a member of The Information Processing Society Japan and the IEEE computer society.

Kazuhiko Tsuda has been working as a Professor at the Graduate School of Business Sciences, University of Tsukuba, Tokyo, Japan since 1998. He received his B.S. and Ph.D. in Engineering from Tokushima University in 1986 and 1994, respectively. He was with Mitsubishi Electric Corporation during 1986 to 1990, and with Sumitomo Metal Industries Ltd. during 1991 to 1998. His research interests include natural language processing, database, information retrieval, algorithm and human-computer interaction. He is a member of IEEE Computer Society, The Information Processing Society of Japan and The Institute of Electronics, Information and Communication Engineers.