Regular Paper

Automated Generation of Extraction Rules for Japanese Functional Requirements

Maiko Onishi[†‡], Kozo Okano[‡]

[†]Graduate School of Humanities and Sciences, Ochanomizu University, Japan
[‡]Faculty of Engineering, Shinshu University, Japan

Abstract - In software development, extracting relevant information from the functional requirements of specifications is crucial for ensuring consistency and reliability. Traditional methods of information extraction are often burdened with extensive annotation requirements, which can be both time-consuming and error-prone. To address this issue, we applied NLP techniques to streamline the extraction of information from functional requirements, aiming to enhance software development efficiency. We developed an information extraction system and an automatic rule generation process to extract preconditions, actions, and resulting states from Japanese functional requirements without requiring specialized knowledge and with minimal annotation. Experimental results using benchmark requirement documents demonstrated that our proposed method achieved an F1-score of 80%. This approach lays the foundation for automating the extraction of functional requirements, improving the consistency and reliability of requirement documents, and enabling more efficient software development.

Keywords: software engineering, natural language processing, requirements specification, information extraction

1 INTRODUCTION

The application of Natural Language Processing (NLP) technology to functional requirements in a specification document represents a significant advancement in streamlining software development processes. Functional requirements in a specification document detail the specific functionalities and behaviors that the system should provide to meet user needs. These requirements define:

- ***Preconditions*** required before the system can perform a specific action.
- ***Actions*** that the system carries out when certain conditions are met.
- ***Resulting State*** of the system after the actions have been completed.

They serve as guidelines for developers in designing and implementing the system. However, functional requirements may contain contradictions, potentially causing confusion in the development process. For instance, if the specification document provides inconsistent information about how a certain functionality should operate, developers may struggle to determine which information to trust. Contradictory information may also arise across different sections or parts of the document, leaving developers uncertain about what is correct and potentially impacting the progress of the project. According to [1], these types of contradictions pose serious challenges for developers and require time and effort to resolve. While requirements specifications written in natural language offer flexibility and expressiveness, they are also prone to contradictions, which can lead to inconsistencies and misunderstandings in requirements. Furthermore, once the requirements specification is created, the development process progresses based on it. However, if defects arise in this upstream process, they can have a significant impact on downstream stages. To prevent such issues, it is desirable to identify and resolve contradictions in functional requirements.

To identify inconsistencies in functional requirements, model checking is often regarded as a useful tool. Furthermore, it is anticipated that using NLP technology to efficiently create formal specifications for a system will reduce the cost of performing model checking. Formal specifications need to be prepared in advance for model checking, and methods have been proposed to efficiently create these specifications from requirements documents written in natural language [2–4].

Among the requirements documents, functional requirements are particularly useful for creating formal specifications related to behavior. Therefore, if NLP technology can be employed to automatically extract information from functional requirements, it could reduce the cost of creating behavior-related formal specifications. Amidst the growing attention to the application of NLP in software development, there is a shortage of appropriate training corpora for deep learning in requirements engineering [5]. It is crucial for information extraction to be applicable to general language patterns without incurring the cost of domain adaptation, considering the diverse range of requirement specification documents created across various fields [6].

However, research into extracting information from functional requirements for creating formal specifications is still underdeveloped, and information extraction from Japanese lags behind that from English. One reason is that it is challenging to precisely define whether the events and states included in functional requirements correspond to *Preconditions*, *Actions*, or *Resulting State*. Even if a precise definition were established, it would require specialized knowledge from the information extraction operators, increasing their workload.

In this study, we developed information extraction rules that consider both syntactic structure and semantics to extract *Preconditions* and either *Actions* or *Resulting State* from Japanese functional requirements. Additionally, we imple-

mented an automatic rule generation process to enable information extraction without requiring specialized knowledge from users. Although this paper focuses on Japanese, it employs NLP techniques that are also applicable to English, allowing for the possibility of extending the approach to English as well.

We created 186 binary relations for the events and states extracted from 46 functional requirements of the *Wadai-Futto pot Ver. 7* [7], which is used as a benchmark for Japanese requirements documents. We annotated these relations to specify which of the pairs correspond to *Preconditions*, *Actions*, or *Resulting State*, such as *Preconditions—Actions* or *Preconditions—Resulting State*. As reported in [6], the majority of functional requirements are contained within a single sentence. Therefore, in the initial stages of our methodology, we excluded functional requirements that span multiple sentences. We reported descriptive statistics on the generated extraction rules and then compared the experimental results between extraction solely based on the rules and application of a filter using similarity of binary relationships after applying the rules.

The structure of the following sections is as follows. Section 2 provides a concrete example of how model checking can be used to efficiently correct erroneous functional requirements. The formal specifications used in model checking are created by leveraging the events, states, *Preconditions*, *Actions*, and *Resulting State* defined by the functional requirements. This demonstrates how these elements can be utilized as components of the functional requirements. Section 3 will summarize the main research on efficient ideas for creating formal specifications, elements extracted from functional requirements, and deep learning-based and rule-based extraction methods. Next, Section 4 outlines the procedure for extracting events and states through syntactic and semantic analysis. The method for identifying the relationship between the two extracted events/states is explained in Section 5.1. Additionally, Section 5.2 describes how the extraction rules can be automatically generated with minimal annotation by the workers. In Section 6, we will report statistics on the extraction rules created based on this method. We will then describe the experimental results comparing extraction solely based on the rules with those after applying a filter. Finally, in Section 7, we will summarize this study and discuss future prospects.

2 DETECTING INCONSISTENCIES USING MODEL CHECKING

In software and system development, it is not uncommon for functional requirements written in documents to contain errors. Especially when requirements are written in natural language, there is a possibility that they may include writing mistakes. If such errors go undetected during the design phase and the system implementation proceeds, they can later become serious issues. This research presents a method for identifying errors in functional requirements using Model Checking. Model Checking is a technique that formally describes the behavior of a system and automatically detects design errors through verification using logical formulas. In this study, we use PAT (Process Analysis Toolkit) [8][1] for Model Checking. By using PAT, it is possible to describe the system behavior as a formal model and perform verification based on formulas. This section demonstrates how errors in functional requirements can be identified by applying Model Checking with PAT, using specific erroneous requirements as examples.

2.1 CSP Model

To illustrate an example of applying Model Checking, let us consider the following functional requirements documented for a simple button-operated door model:

- The door remains closed while the button is pressed.
- The door remains open when the button is not pressed.

These descriptions contain an error. Normally, when the button is pressed, the door is expected to open, and when not pressed, the door is expected to close. However, the specifications above describe a contradictory behavior, where the door remains closed when the button is pressed.

The correct specification should be as follows:

- The door remains **open** while the button is pressed.
- The door remains **closed** when the button is not pressed.

To model this erroneous functional requirement, we use CSP (Communicating Sequential Processes)[9, 10]. CSP is a formal method for modeling system behavior, which is adopted in PAT.

In CSP, an "event" refers to a basic action that constitutes the behavior of the system. For example the following four events can be defined:

- $press$: The event of pressing the button.
- $release$: The event of releasing the button.
- $close$: The event of the door closing.
- $open$: The event of the door opening.

These events are treated as instantaneous or atomic. It is possible to treat pressing and releasing the button as a single event, but since we should focus on each action separately, we will not do so. On the other hand, while it is possible to divide the moment the door starts to open and the moment it finishes opening, these moments are not the focus, so we treat them as a single event.

In CSP, we focus on the order in which events occur, without directly considering causal relationships. For instance, whether the press action is done by a person or by a machine is not considered; it is simply treated as an event.

Additionally, there are three important concepts in CSP notation concerning processes:

- **Prefix**: $x \rightarrow P$ means that event x occurs first, followed by the execution of process P. This is a right-associative operation.

[1] A system that integrates model editing, simulation, and model checking functionality.

- **Recursion**: $P = x \rightarrow y \rightarrow P$ means that after event x occurs, event y will occur, and then the process will return to P. In other words, when x occurs, y occurs.
- **General Choice**: $P \,\Box\, Q$ means that the environment will choose whether to execute P or Q as the first event. If both P and Q can occur, it becomes nondeterministic, which is why we start with events like $press$ and $release$, which cannot occur simultaneously.

To define the two states of the button, we have:

- $PRESSED$: The button is in the pressed state.
- $RELEASED$: The button is in the released state.

These are defined as CSP patterns as follows:

$$PRESSED = release \rightarrow open \rightarrow RELEASED$$
$$RELEASED = press \rightarrow close \rightarrow PRESSED$$

Based on these definitions, the entire behavior of the button can be modeled as:

$$BUTTON = (press \rightarrow close \rightarrow PRESSED) \\ \Box\, (release \rightarrow open \rightarrow RELEASED)$$

The CSP model using PAT is shown in Fig. 1. Furthermore, Fig. 2 shows the constraint graph generated from Fig. 1. Visualizing the model is an important technique for efficiently identifying and correcting errors.

In this way, by using CSP, we can formally describe the system's behavior. To assist in the creation of this CSP model, it is useful to extract the *Preconditions*, *Actions*, and *Resulting State* related to the behavior of the button from the sentences. Using states and events, "The door remains closed while the button is pressed" can be broken down as follows:

- *Preconditions*: $press$
- *Actions*: $close$
- *Resulting State*: $PRESSED$

"The door remains open when the button is not pressed" can be broken down as follows:

- *Preconditions*: $release$
- *Actions*: $open$
- *Resulting State*: $RELEASED$

It can be seen that these correspond to the CSP model of the button. In general, not all elements are present in the sentences, so some completion is required. However, this study does not address the completion process. As will be discussed in Section 6, the distinction between *Actions* and *Resulting State* is not made, and the focus is on extracting *Preconditions* and either *Actions* or *Resulting State*.

2.2 Property Expression and Model Checking

When performing model checking, not only the model but also a verification formula is required. A verification formula defines the conditions to check whether the system behaves as expected. Based on the correct behavior of the button, we can create a verification formula.

The verification formula describes the expected behavior where the door opens while the button is pressed. This property is expressed using Linear Temporal Logic (LTL) as follows:

$$\#assert\ BUTTON \models G(press \rightarrow X\ open);$$

This is a sample code for model checking in PAT, where G (globally) indicates that the specified condition must always hold, and X (next) means that the condition must be true in the next state. This formula states that whenever the button is pressed, the door should be open in the next state. However, in the incorrect model shown in Fig. 1, this verification formula does not hold.

As a result of model checking, PAT determines that this verification formula is invalid and visually presents a counterexample, as shown in Fig. 3. In the counterexample, pressing the button leads to the door closing, which contradicts the expected behavior described by the verification formula.

To resolve this contradiction, the functional requirements related to the "closed" and "open" actions need to be swapped. Through this process, incorrect descriptions become clearly evident, making their correction straightforward. Model checking allows us to identify and correct inconsistencies present in the text.

3 RELATED WORK

While functional requirements are commonly described in natural language, the inherent noise and ambiguity of natural language can make analysis challenging. To circumvent this issue and facilitate the transformation of functional requirements into formal specifications, a method involves describing the requirements in Controlled Natural Language (CNL), a language with restricted sentence structures [11–13]. The aim of these studies is to provide a formal foundation for model-based testing through natural language parsing and automatic generation of test cases. Since models containing ambiguity can lead to the creation of incorrect test cases, CNL, which specifies a single interpretation, plays a crucial role. Those applying extraction rules, as seen in [12], typically work with syntax trees. The extracted elements are organized into frame-based semantic representations, which are then mapped to internal formal models. The effectiveness of a small set of rule-based extraction rules is attributed to the restricted sentence structures in CNL.

On the other hand, describing functional requirements in natural language offers the advantage of high expressiveness. While CNL may require extensions to handle new expressions, natural language inherently does not. However, a drawback arises in the increased complexity of extraction rules

$$BUTTON = (press \rightarrow close \rightarrow PRESSED) \,\square\, (release \rightarrow open \rightarrow RELEASED);$$
$$PRESSED = (release \rightarrow open \rightarrow RELEASED);$$
$$RELEASED = (press \rightarrow close \rightarrow PRESSED);$$

Figure 1: A sample code for modeling a button function to open and close a door on a PAT using CSP.

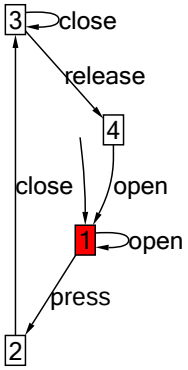

Figure 2: The constraint graph automatically generated from the CSP in Fig. 1.

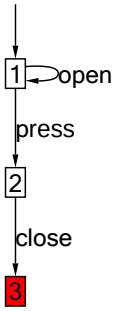

Figure 3: The counterexample output from conducting model checking.

compared to CNL when extracting elements from sentences. In [3], examples of extraction rules targeting Japanese sentences are provided. Extraction rules are applied when specific syntactic structures within the sentence are identified. The information to be extracted consists of condition clauses and action clauses necessary for creating intermediate models. In [4], a wider range of sentence structures is addressed by increasing the patterns of syntactic structures, allowing for the extraction of both condition and action clauses. Both [4] and [14] adopt a bottom-up syntax parsing using the algorithm based on Context-Free Grammar (CFG), similar to previous studies. They utilize syntactic parsing rules and dependency parsing results to determine syntactic structure patterns. A noted challenge is the increased complexity of syntactic parsing rules.

Research focusing on the detection and extraction of causal relations in the field of requirements engineering [5, 6, 15] considers functional requirements as a type of causal relation. They have analyzed a large dataset of 14,983 sentences from requirements specifications to investigate the frequency, forms, and complexity of causal relations, revealing that approximately 28% of the sentences contain causal information. Additionally, they conducted a case study exploring the correlation between the occurrence of causal relations and the requirements lifecycle, demonstrating the positive impact of detecting causal relations on the requirements process. Additionally, [15] indicates that it is insufficient to identify key phrases that signify causal relations solely based on the vocabulary within a sentence.

4 EXTRACTION OF EVENTS AND STATES

Here is the process for extracting events and states described in the functional requirements document, using a uniform method without distinguishing between events and states. First, as illustrated in Fig. 4, the procedure involves converting sentences into a tree structure that considers both syntactic and semantic structures through syntactic parsing, semantic parsing, and tree generation. Following this, we identify nodes within the tree structure that represent events or states, and extract information from the subtrees with these nodes as their roots.

For grammatical analysis, we adopt Combinatory Categorial Grammar (CCG). One advantage of using CCG is its ability to perform syntactic and semantic parsing simultaneously. In CCG parsing, each word or phrase is assigned a category, and these categories combine according to grammatical rules to determine the overall structure of the sentence. In semantic analysis, each category corresponds to a semantic representation, which is combined according to the syntactic rules. This approach allows for the semantic linking of multiple lexical items that constitute events or states.

For syntactic parsing with CCG, we utilize depccg[2] [16] and select janome[3] as the Japanese morphological analyzer within the tool. This parser outputs CCG derivation trees based on the Japanese CCGBank [17].

In semantic parsing, the process of outputting a logical formula from CCG derivation trees based on semantic templates and lexical items follows the ccg2lambda[4] [18, 19] framework. First, meanings are assigned to all leaf nodes of the CCG derivation tree. This assignment is processed based on the following semantic templates and lexical items:

- **Semantic templates** that have matching conditions for syntactic features and syntactic categories.
- **Lexical items** that have matching conditions not only for syntactic features and syntactic categories but also

[1] https://github.com/masashi-y/depccg
[2] https://github.com/mocobeta/janome
[3] https://github.com/mynlp/ccg2lambda

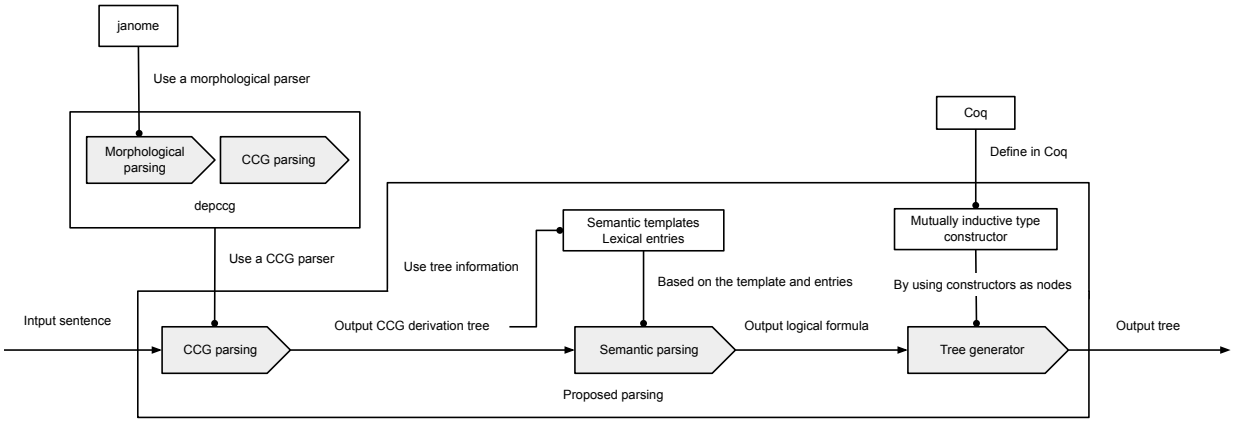

Figure 4: The procedure involves converting sentences into a tree structure that considers both syntactic and semantic structures

for outputs of morphological analysis such as surface forms, base forms, and part-of-speech tags.

The method for constructing these is followed as described in [19, 20]. Next, semantic composition rules are applied top-down from the leaf nodes, calculating the meanings assigned to the remaining nodes. When no further rules can be applied, the final calculated meaning is outputted.

The logical formulas output by the proposed system include functions called constructors, which are used to create instances of data types. These constructors define how data types are structured and assign specific values to them. The proposed data types are $\mathsf{Tree_{np}}$, which represents the structure of noun phrases, and $\mathsf{Tree_s}$, which represents the structure of sentences. Mutually inductive type declarations and constructor definitions are implemented in Coq [21] as follows:

$$
\begin{aligned}
&\mathsf{Parameter\ \ Entity : Type.}\\
&\mathsf{Parameter\ \ Event : Type.}\\
&\mathsf{Parameter\ \ Glue_{np} : Type.}\\
&\mathsf{Parameter\ \ Glue_s : Type.}\\
&\mathsf{Parameter\ \ Tense : Type.}\\
&\mathsf{Parameter\ \ Aspect : Type.}\\
&\\
&\mathsf{Inductive\ \ Tree_{np} : Type :=}\\
&\quad \mathsf{|\, Nil_{np} : Tree_{np}}\\
&\quad \mathsf{|\, En : Entity \to Tree_{np}}\\
&\quad \mathsf{|\, Np : Tree_{np} \to list\ Glue_{np} \to Tree_{np} \to Tree_{np}}\\
&\quad \mathsf{|\, Adn : Tree_s \to Tree_{np} \to Tree_{np}}\\
&\mathsf{with\ \ Tree_s : Type :=}\\
&\quad \mathsf{|\, Ev : Tree_{np} \to Tree_{np} \to Tree_{np} \to Tree_{np}}\\
&\qquad \mathsf{\to list\ Event \to list\ Aspect \to list\ Tense \to Tree_s}\\
&\quad \mathsf{|\, Adv_{np} : Tree_{np} \to list\ Glue_s \to Tree_s \to Tree_s}\\
&\quad \mathsf{|\, Adv_s : Tree_s \to list\ Glue_s \to Tree_s \to Tree_s}\\
&\quad \mathsf{|\, Hyp : Tree_s \to list\ Glue_s \to Tree_s \to Tree_s.}\\
&\\
&\mathsf{Parameter\ \ Closure : Tree_{s_t} \to Prop.}
\end{aligned}
$$

The explanations for each are given below:

- **Type** Entity represents nouns.
- **Type** Event represents actions or states expressed by verbs.
- **Type** $\mathsf{Glue_{np}}$ represents particles used to connect noun phrases.
- **Type** $\mathsf{Glue_s}$ represents conjunctions or particles used to connect sentences or clauses.
- **Type** Tense represents the syntactic tense of verbs.
- **Type** Aspect is a syntactic type used to represent the nature or progress of actions or states as expressed by the verb.
- **Constant** $\mathsf{Nil_{np}}$ represents a non-existent noun phrase when a verb is not associated with nominative, accusative, or dative cases.
- **Constructor** En applies to nouns.
- **Constructor** Np applies to noun phrases.
- **Constructor** Adn applies when modifying nouns.
- **Constructor** Ev generates a term representing an event consisting of a single verb. It takes as arguments terms of type $\mathsf{Tree_{np}}$ representing noun phrases marked as nominative, accusative, and dative cases. The fourth argument is another term of type $\mathsf{Tree_{np}}$ representing complements linked by a copula.
- **Constructor** $\mathsf{Adv_{np}}$ applies in various contexts, such as when a noun phrase is used adverbially to modify the entire sentence or is inserted in a specific position within the sentence.
- **Constructor** $\mathsf{Adv_s}$ is used to introduce adverbial elements that modify the entire sentence.
- **Constructor** Hyp applies in conditional sentences.

- **Function** Closure takes instances of type $\mathsf{Tree_{np}}$ or $\mathsf{Tree_s}$ as arguments and returns them as propositions. The purpose of this function is solely to enable the use of Coq's proof assistance capabilities by converting data types into propositions.

For instances of constant list types, the function used to append a constant to a list is defined as follows:

```
Fixpoint append A : Type (l l': list A) : list A :=
  match l with
  |nil => l'
  |cons a l => cons a (append l l')
  end.
```

For example, consider an instance of the list Event type, which is an argument of the constructor Ev:

```
(append (append (cons ロック (cons 解除
nil))(cons し nil))(append nil nil))
```

In this instance, "ロック"*(lock)*, "解除"*(release)*, and "する"*(do)* are constants of the Event type. The process of appending these constants is not necessarily done by adding them one by one to an empty list nil. The order of appending is influenced by the sequence in which syntactic rules are applied in the CCG derivation tree. However, the order of concatenation does not necessarily affect the meaning. Instead of deriving the meaning of the event compositionally from each individual constant's meaning, it is more natural to assign meaning to the whole phrase "ロック解除する"*(unlock)*. Therefore, by using Coq's proof assistance to execute unfold append and simplify the concatenation of lists, such as:

```
(cons ロック (cons 解除 (cons し nil)))
```

This shows that lists with different concatenation orders can have the same meaning. Besides list Event, other cases where the order of concatenation does not affect meaning include list $\mathsf{Glue_{np}}$ for noun phrases, list Aspect and list Tense for auxiliaries, and list $\mathsf{Glue_s}$ for conjunctions.

After simplifying the list, instances of the data types are converted into tree structures. Nodes in the tree are represented as constructors, with the arguments of these constructors serving as child nodes. The positions of the child nodes correspond to the positions of the arguments.

Once a tree structure like the one shown in Fig. 5 is obtained, subtrees representing events or states are extracted. Since the constructor Ev represents the minimal event, we extract information from subtrees with Ev as the root node. For example, by listing all the leaf nodes within the subtree, we can obtain a textual representation of the event. However, since Ev always includes a predicate, it is not possible to extract event nouns using the same method. Therefore, event nouns are extracted manually.

5 RULE GENERATION

5.1 Pattern Matching for Binary Relations

We will provide a detailed explanation of how to identify the relationship between two events or states using paths within a tree structure. This method involves identifying the shortest path from a source node (starting point) to a target node (endpoint) within the tree structure and mapping this path to the labels of a binary relation. The specific steps of this method are as follows:

1. **The identification and setup of nodes** involves first identifying the nodes within the $\mathsf{Tree_{np}}$ or $\mathsf{Tree_s}$ tree structure that represent events or states, and then setting the source node (starting point) and target node (endpoint) for analysis. In Fig. 5, we identify three event nodes:

   - Ev_1 is Ev that represents "ロックされていたら"*(if it is locked)*.
   - Ev_2 is Ev represents "ロック解除し"*(unlock it)*.
   - Ev_3 is Ev represents "ロックランプを消灯する"*(turn off the lock indicator light)*.

   Based on these nodes, we consider the following three binary relations, where the left node is the source and the right node is the target:

   $$\mathsf{Ev}_1 \to \mathsf{Ev}_2, \quad \mathsf{Ev}_2 \to \mathsf{Ev}_3, \quad \mathsf{Ev}_1 \to \mathsf{Ev}_3$$

2. **The identification of the Lowest Common Ancestor** is crucial for finding the shortest path from the source node to the target node. Specifically, this requires identifying the Lowest Common Ancestor (LCA) of both nodes, which is the most distant ancestor node from the root that is common to both the source and target nodes. This node will serve as the reference point for determining the path. For the three relations in Fig. 5, the LCA is identified as follows:

   - The LCA for Ev_1 and Ev_2 is Hyp.
   - The LCA for Ev_2 and Ev_3 is $\mathsf{Adv_s}$.
   - The LCA for Ev_1 and Ev_3 is $\mathsf{Adv_s}$.

3. **The construction of the shortest path** can be achieved once the LCA is identified, by combining the path from the source node to the LCA and the path from the LCA to the target node. Additionally, tregex patterns are used to represent the shortest paths within the tree structure. Tregex [22] is a pattern language designed for efficiently performing pattern matching on tree structures. Each pattern specifies dominance or sibling relationships between nodes and is used to identify specific patterns within a tree structure. Basic node relationships supported by tregex patterns include:

   - $A <_i B$: Node B is the i-th child of node A

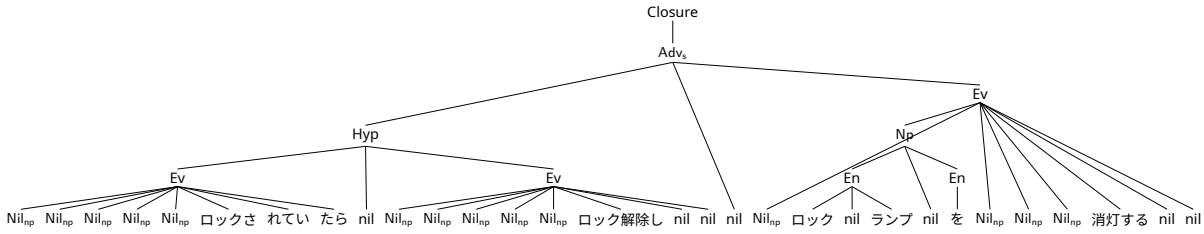

Figure 5: A Tree$_s$ type tree structure of “ロックされていたらロック解除し、ロックランプを消灯する”*(If it is locked, unlock it and turn off the lock indicator light)*.

- $A >_i B$: Node A is the i-th child of node B

By combining these patterns, it is possible to gain a detailed understanding of the relationships between nodes within the tree structure and to identify relationships using paths. For the relations shown in Fig. 5, the shortest paths are as follows:

- The path from Ev_1 to Ev_2 is represented by the tregex pattern $(\mathsf{Ev}_1 >_1 (\mathsf{Hyp} <_3 \mathsf{Ev}_2))$.
- The path from Ev_2 to Ev_3 is represented by the tregex pattern $(\mathsf{Ev}_2 >_3 (\mathsf{Hyp} >_1 (\mathsf{Adv_s} <_3 \mathsf{Ev}_3)))$.
- The path from Ev_1 to Ev_3 is represented by the tregex pattern $(\mathsf{Ev}_1 >_1 (\mathsf{Hyp} >_1 (\mathsf{Adv_s} <_3 \mathsf{Ev}_3)))$.

4. **The mapping to binary relation labels** involves mapping the obtained shortest path to the labels of a binary relation through annotations. For the paths shown in Fig. 5, the relations are labeled as follows:

   - $\mathsf{Ev}_1 \rightarrow \mathsf{Ev}_2$: *Preconditions—Actions*
   - $\mathsf{Ev}_2 \rightarrow \mathsf{Ev}_3$: *Actions—Actions*
   - $\mathsf{Ev}_1 \rightarrow \mathsf{Ev}_3$: *Preconditions—Actions*

   The rule consists of the relationship between nodes, including surrounding information such as the source node, target node, and the shortest path represented by tregex patterns, with labels assigned to the relationship. When applying the rule to automatically assign labels, the condition is that the shortest path between nodes must match.

5.2 Annotations

To automatically extract information from functional requirements, it is necessary to prepare extraction rules in advance. We describe a method for creating effective extraction rules with minimal annotation, even for users without expertise in syntactic or semantic analysis. The system performs the following steps automatically:

1. **The syntactic and semantic parsing** generates a tree structure of type $\mathsf{Tree_{np}}$ or $\mathsf{Tree_s}$ from the input data.

2. **The extraction of event and state nodes** involves identifying and extracting nodes that correspond to events or states from the tree structure.

3. **The identification of the shortest path** determines the shortest path between the source node and the target node.

Following these steps, the user performs the following task:

- **The manual annotation** involves the user manually annotating the relationships between the source and target node pairs.

For this task, a simple tool is used that allows users to visually verify the output of common processing. The tool clearly indicates the source and target nodes for each data item and provides visual support, allowing the user to determine whether the items correspond to functional requirements. This method makes the annotation process efficient. After annotation, the assigned labels are mapped to tregex patterns, and extraction rules are accumulated accordingly.

In this way, users are only involved in the annotation phase and do not need specialized knowledge, facilitating the automatic generation of extraction rules.

6 METHOD APPLICATION AND RELATION RESULTS

We report the results of applying this method to practical data, evaluating the extraction of events and states, annotation, and identification of binary relations.

6.1 Sentence Unification Process

We conducted experiments using 46 functional requirements related to the behavior of an electric kettle, as described in *Wadai-Futto pot Ver. 7* [7]. Each functional requirement is written as a single sentence that represents a specification. However, if the original text contained bulleted lists, we replaced them with variables and converted the entire content into a single sentence.

We made this modification because structured sentences or sentences with coordinate conjunctions tend to reduce syntactic parsing accuracy. Since our research does not focus on improving syntactic parsing accuracy, we unified all sentences to avoid these issues. For example, if the original text had a bulleted list like this:

> 以下のいずれかの時、ロック/ロック解除できない。

- 給湯中
- 蓋センサoff

Lock/unlock is disabled in the following cases:

- During water dispensing
- When the lid sensor is off

We converted it into a single sentence as follows:

*a*のいずれかの時、*b*できない。

- *a*: 給湯中, 蓋センサoff
- *b*: ロック/ロック解除

b is disabled when condition *a* occurs.

- *a*: During water dispensing, when the lid sensor is off
- *b*: Lock/unlock

Using this unification rule, we converted all 46 sentences and then extracted events and states.

6.2 Event and State Extraction Results

We conducted experiments on the extraction of events and states. From these sentences, 130 events/states were extracted. Here, we present some notable results of the extraction. In the sentence:

[アイドル中]$_{\text{state}}$に[蓋センサonとなったら]$_{\text{event}}$ [沸騰行為に遷移する]$_{\text{event}}$

[While idle]$_{\text{state}}$ [When the lid sensor is turned on]$_{\text{event}}$ [transition to boiling]$_{\text{event}}$

was successfully extracted with most of the vocabulary in the sentence being identified as events or states. On the other hand, in the sentence:

[カルキ抜きの加熱を終えたら]$_{\text{event}}$[沸騰行為を中止し]$_{\text{event}}$保温行為に[遷る]$_{\text{event}}$

[When dechlorination heating is finished]$_{\text{event}}$ [stop boiling]$_{\text{event}}$ and switch to warming [transition]$_{\text{event}}$

the expected extraction was “保温行為に遷る”*(switch to warming transition)*, but some information was missing.

When counting the number of missing characters, it was found that, as shown in Table 1, there was an average of 6.49 characters missing, which corresponds to an average of 0.17 in terms of the proportion of the text, with a maximum of 0.61. Missing characters are generally not problematic when they are conjunctions. However, if the missing characters are nouns, it indicates that important information related to events or states in the text is absent, which could potentially cause issues.

Table 1: Descriptive Statistics of Missing Character Counts

Metric	Value
Mean	6.49
Standard Deviation	6.94

Table 2: Distribution of labels for the 186 binary relations

Pair	Frequency
Preconditions—Preconditions	26
Preconditions—Actions	92
Preconditions—Resulting State	14
Actions—Actions	45
Actions—Resulting State	2
Resulting State—Resulting State	3
Other	4
Total	186

6.3 Labeling of Binary Relations

From the extracted events and states, 186 binary relations were generated, and we conducted annotation on these relations. The distribution of the labels is shown in Table 2.

By prioritizing the classification of frequently occurring labels, we ensure the utility of the proposed system for the most common cases. Therefore, we retrieve *Preconditions*, *Actions*, and *Resulting State* from *Preconditions—Actions* and *Preconditions—Resulting State*, without distinguishing between the categories of *Actions* and *Resulting State*. The validation method is explained as follows. The data is divided into 69 test data and 117 training data. Tregex patterns created from the training data, combined with annotated labels, are used to accumulate extraction rules. Then, tregex patterns generated from the test data are compared against these extraction rules to determine their applicability. Specifically, based on the label distribution shown in Table 2, the label “CAUSE” is reassigned to the pairs *Preconditions—Actions* and *Preconditions—Resulting State*, while the label “OTHER” is reassigned to all other pairs. The evaluation focuses on accurately classifying these binary labels. Test data are prepared according to the distribution shown in Table 3, and the remaining data are used as training data.

To avoid bias, careful attention was given to the data allocation. The functional requirements in [7] are divided into

Table 3: Distribution of labels for the 186 binary relations

Pair	Label	test data
Preconditions—Preconditions	OTHER	14
Preconditions—Actions	CAUSE	29
Preconditions—Resulting State	CAUSE	12
Actions—Actions	OTHER	6
Actions—Resulting State	OTHER	2
Resulting State—Resulting State	OTHER	3
Other	OTHER	3
Total		69

Table 4: Comparison of Accuracy Metrics for Different Rule Application Methods.

	F1-score	precision	recall
rule only	0.53	0.70	0.57
rule + TED	**0.80**	0.81	0.80

sections by functionality, with sentences within each section tending to have similar syntax. Allocating sentences with similar syntax from the same section to both the test and training datasets could lead to easy predictions using similar rules from the training data, thereby not accurately reflecting the system's performance on truly unknown data. Therefore, test data were created from sections different from those used for training.

6.4 Evaluation of Rule Application Methods

The rules generated from the training data total 92, which means that approximately 80% of the binary relations in the training data were used to create these rules. The generated rules were applied to the test data using two different methods, each with distinct characteristics:

1. **Applying rules only when paths are identical** ensures that when paths are the same, the structure of the target data matches exactly, making the application of the rules expected to be accurate. This method maintains structural consistency in the data and helps avoid incorrect applications.

2. **Applying the rule with the Tree Edit Distance (TED) of paths** quantitatively indicates how different two patterns are. By utilizing the minimum TED for the rule application, the transformation that is closest to the original data is achieved, resulting in a more natural application. The APTED algorithm [23] is used to calculate the TED of paths.

The results of applying each method are shown in Table 4.

It was found that, even without considering the meaning of vocabulary, 80% of the relations between two events/states included in functional requirements could be identified based solely on structure. Allowing the rules to apply to similar paths using the TED, rather than applying them strictly, significantly improved accuracy. The maximum F1-score is around 80% at the TED threshold of 7. According to Fig. 6, using only precondition-action/resulting state labels without the OTHER labels causes the F1-score to drop to about 50%. This shows that using multiple labels is more effective than using a single label.

An example that contributed to the improvement in accuracy is provided. A binary relation labeled as *Preconditions—Actions/Resulting State* was generated from the sentence:

> ロック中に[このボタンを]$_{\text{source}}$100msec以上[押す]$_{\text{source}}$とロックは解除され、[ロックランプを消灯する]$_{\text{target}}$

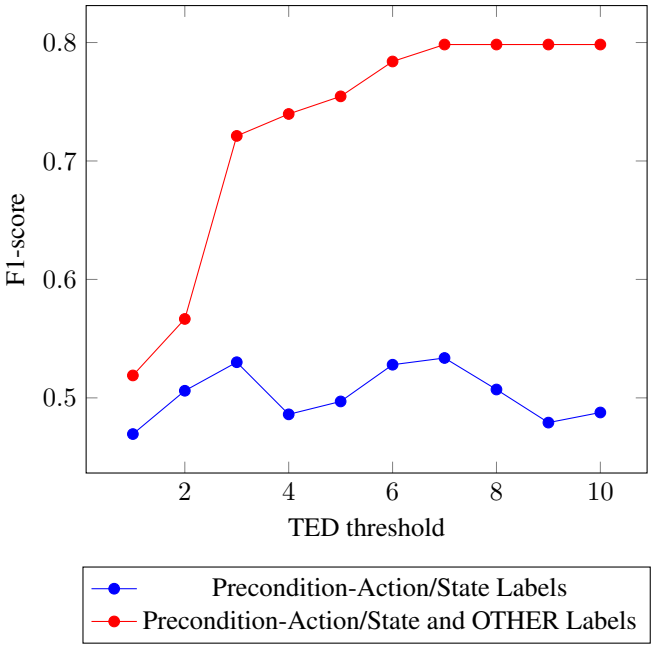

Figure 6: Changes in F1-score with respect to the threshold.

> While locked, if [this button]$_{\text{source}}$ is [pressed]$_{\text{source}}$ for more than 100 msec, the lock will be released and [turn off the lock indicator light]$_{\text{target}}$

The Tree_{s} type tree structure generated from this sentence is shown in Fig. 7. The red nodes in the figure are located on the path from the source node with the Ev label to the target event. Representing the path with a tregex pattern yields:

> (Ev $>_3$ (Adv$_{\text{np}}$ $>_3$ (Adv$_{\text{np}}$ $>_1$ (Hyp $>_1$ (Adv$_{\text{s}}$ $<_3$ Ev))))

Although this exact pattern was not included among the created extraction rules, a tregex pattern with the minimum TED:

> (Ev $>_3$ (Adv$_{\text{np}}$ $>_3$ (Adv$_{\text{np}}$ $>_1$ (Hyp $>_1$ (Adn $>_1$ (Adv$_{\text{np}}$ $<_3$ Ev))))))

was included. This path differs from the path in Fig. 7 only by the presence of Adn, yet the two paths exhibit very similar structures when expressed as tregex patterns. This pattern was created from Fig. 8. Furthermore, this Tree_{s} type tree structure was generated from the sentence:

> 60min0secのときに、[更にタイマボタンを]$_{\text{source}}$1回[押される]$_{\text{source}}$と、1min0secをセットしたことに[なる]$_{\text{target}}$

> At 60 minutes and 0 seconds, if [the timer button]$_{\text{source}}$ is [pressed]$_{\text{source}}$ once more, it [sets]$_{\text{target}}$ the timer to 1 minute and 0 seconds.

This binary relation represents a *Preconditions—Actions/Resulting State* and has a syntactic structure similar to the binary relation in Fig. 7. By considering TED, the rule generated from Fig. 8 was successfully applied for classification.

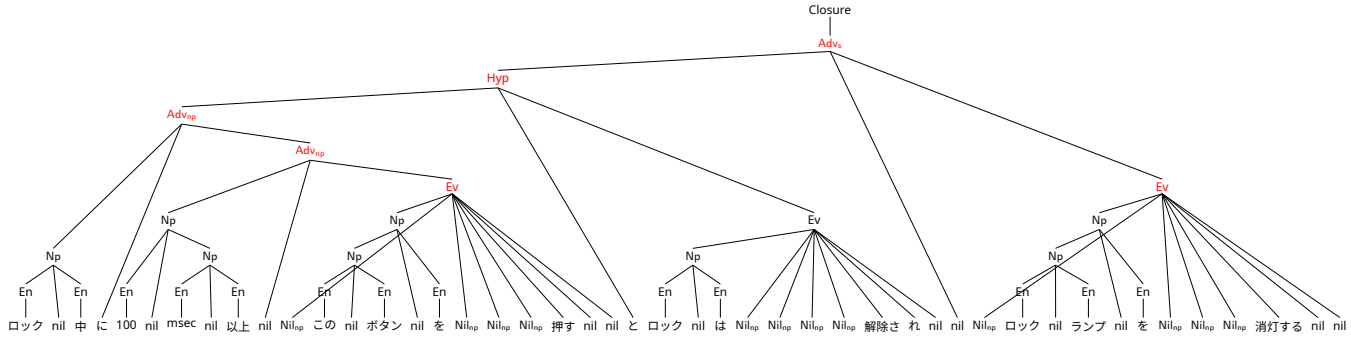

Figure 7: The tree structure generated from one of the test data and the path between the two events that have a *Preconditions*—*Actions* relationship

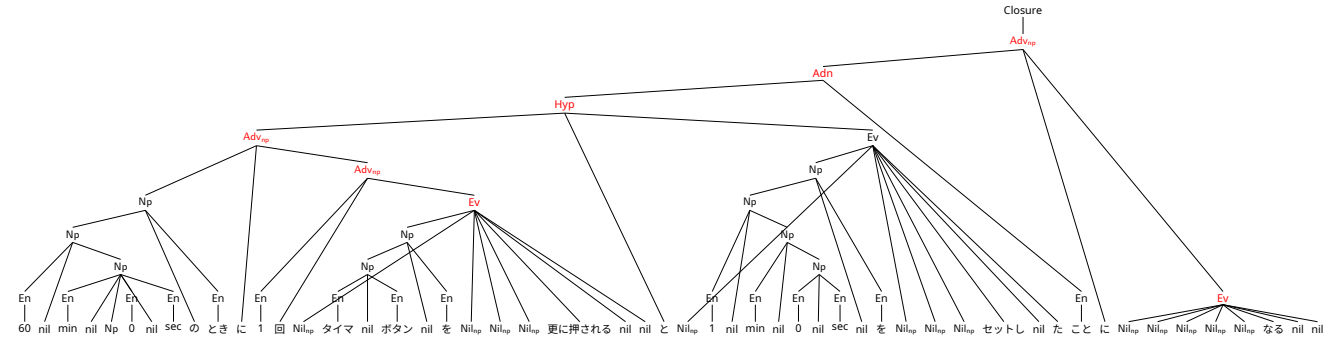

Figure 8: The tree structure generated from one of the training data and the path between the two events that have a condition-action relationship

7 CONCLUSIONS

This study demonstrated the effectiveness of a rule-based approach using syntactic and semantic analysis to extract functional requirements from Japanese specification documents. Notably, the method of minimum TED achieved the highest accuracy, confirming the importance of structural similarity.

Experiments using the functional requirements of *Wadai-Futto pot Ver. 7* showed that the generated rules could accurately identify functional requirements with about 80% precision. This result indicates that even complex rules can be effectively applied, supporting the scalability of our approach.

Future work includes extracting functional requirements that span multiple sentences, improving the accuracy of events/states extraction, and ensuring that no information is lost from the sentences.

ACKNOWLEDGMENT

This research is being partially conducted as Grant-in-Aid for Scientific Research C (21K11826).

REFERENCES

[1] B. Meyer, “On Formalism in Specifications”, In: *Program Verification: Fundamental Issues in Computer Science*, pp. 155–189, (1993).

[2] K. Okano et al., “Analysis of Specification in Japanese Using Natural Language Processing and Review Supporting with Speech Synthesis,” In: *IEICE Technical Report; IEICE Tech. Rep.,* Vol. 117, No. 465, pp. 79–84, (2018).

[3] K. Okano et al., “Analysis of Specification in Japanese Using Natural Language Processing,” In: *Joint Conference on Knowledge-Based Software Engineering*, pp. 12–21, (2018).

[4] H. Ii, K. Okano, and S. Ogata, “Improving Accuracy of Automatic Derivation of State Variables and Transitions from a Japanese Requirements Specification,” In: *Joint Conference on Knowledge-Based Software Engineering*, pp. 20–34, (2020).

[5] J. Fischbach et al., “Towards Causality Extraction from Requirements,” In: *2020 IEEE 28th International Requirements Engineering Conference*, pp. 388–393, (2020).

[6] J. Fischbach et al., “Automatic Detection of Causality in Requirement Artifacts: The CiRA Approach,” In: *Proceedings of Requirements Engineering: Foundation for Software Quality: 27th International Working Conference*, pp. 19–36, (2021).

[7] SESSAME(`http://www.sessame.jp`), *Wadai-Futto Pot (GOMA-1015 Type) 7th Edition*. `https://www.sessame.jp/workinggroup/WorkingGroup2/`

POT_Specification_v7.PDF. (referred February 24, 2025).

[8] J. Sun et al., "PAT: Towards Flexible Verification un-der Fairness," In: *Lecture Notes in Computer Science*, Vol. 5643, pp. 709–714, (2009).

[9] C. A. R. Hoare, *Communicating Sequential Processes*, (1985).

[10] C. A. R. Aoare, *Aommunicating Sequential Arocesses*, https // ntares sip ucm s / luis / doctorado06-07/cspbook.pdf, (referred February 24, 2025).

[11] G. Carvalho et al., "Model-Based Testing from Con-trolled Natural Language Requirements," In: *Formal Techniques for Safety-Critical Systems*, pp. 19–35, (2014).

[12] G. Carvalho et al., "NAT2TESTSCR: Test Case Gen-eration from Natural Language Requirements Based on SCR Specifications," In: *Science of Computer Pro-gramming*, Vol. 95, pp. 275–297, (2014).

[13] S. Barza et al., "Model Checking Requirements," In: *Brazilian Symposium on Formal Methods*, pp. 217–234, (2016).

[14] M. Ohto et al., "Proposal of Extracting State Variables and Values from Requirement Specifications in Japanese by Using Dependency Analysis," In: *Procedia Computer Science*, Vol. 192, Knowledge-Based and Intelli-gent Information & Engineering Systems: Proceedings of the 25th International Conference KES2021, pp. 1649–1658,(2021).

[15] J. Frattini et al., "Causality in Requirements Artifacts: Prevalence, Detection, and Impact," In: *Requirements Engineering*, Vol. 28, pp. 49–74, (2022).

[16] M. Yoshikawa, H. Noji, and Y. Matsumoto, "A* CCG Parsing with a Supertag and Dependency Factored Model," In: *Journal of Natural Language Processing*, Vol. 26, pp. 83–119, (2019).

[17] S. Uematsu et al., "Integrating Multiple Dependency Corpora for Inducing Wide-Coverage Japanese CCG Resources," In: *ACM Transactions on Asian and Low-Resource Language Information Processing*, Vol. 14, No. 1, pp. 1–24, (2015).

[18] P. Aartínez-Gómez et al., "ccg2lambda: A Composi-tional Semantics System," An: *Aroceedings of ACL-2016 System Demonstrations*, pp. 85–90, (2016).

[19] K. Mineshima et al., "Building Compositional Seman-tics and Higher-Order Inference System for a Wide-Coverage Japanese CCG Parser," In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 2236–2242, (2016).

[20] M. Onishi et al., "Reducing Syntactic Complexity for Information Extraction from Japanese Requirement Spec-ifications," In: *2022 29th Asia-Pacific Software Engi-neering Conference*, pp. 387–396, (2022).

[21] The Coq Development Team, *The Coq Proof Assistant: Reference Manual: Version 8.9.0*. (2019).

[22] R. Levy and G. Andrew, "Tregex and Tsurgeon: Tools for Querying and Manipulating Tree Data Structures." In: *Proceedings of the 5th International Conference on Language Resources and Evaluation*, pp. 2231–2234, (2006).

[23] M. Pawlik and N. Augsten, "Tree Edit Distance: Ro-bust and Memory-Efficient," In: *Information Systems*, Vol. 56, pp. 157–173, (2016).

(Received: November 15, 2024)
(Accepted: March 5, 2025)

Maiko Onishi received her M.S. degree in Computer Science from Ochanomizu University, Tokyo, Japan, in 2021. She completed the coursework for a Ph.D. at the same university in 2024. She joined Aisin Corporation. Her research interests include applications of natural language processing.

Kozo Okano received his BE, ME, and PhD degrees in Information and Computer Sciences from Osaka University in 1990, 1992, and 1995, respectively. He was an Assistant Professor and an Associate Professor of Osaka University. In 2002 and 2003, he was a visiting researcher at the Department of Computer Science of the University of Kent in Canterbury, and a visiting lecturer at the School of Computer Science of the University of Birmingham, respectively. Since 2020, he has been a Professor at the Department of Electrical and Computer Engineering, Shinshu University. From 2023 to 2025, he was the Director of Center for Data Science and Artificial Intelligence. Since 2024, he has been the Vice-Director of Data Science Education Headquarters. His current research interests include formal methods for software and information system design, and applying deep learning to Software Engineering. He is a member of IEEE, IEICE, and IPSJ.