1. Introduction

Software testing is an activity of verifying whether the software satisfies its requirements. Pair-wise testing, one of combinatorial testing approaches [1], [2], provides a small set of test cases that cover every pair of parameter values. The test space of combinatorial testing is modeled by a set of parameters, their values, and constraints on the value combinations. The constraints [1] define the combinations that are prohibited and should be excluded from test cases; therefore, constraint handling is crucial in test design. The constraints elicitation process, however, has not been well studied, and it is still known as a cumbersome task [3].

We try to tackle the difficulties of defining constraints in the test design. We, in particular, propose a constraint elicitation process that helps us define constraints. This process exhaustively combines parameters and their values and suggests pairs of parameter values that probably form constraints. The process consists of two steps for dealing with two difficulties in the constraint definition, respectively. These steps use a linguistic approach and collective intelligence to find out a suitable pair of parameters and their values to define constraints.

We conduct experiments on three test models, a cross-browser test model, an ATM system test model, and a real world web application testing, to evaluate the validity of our process. This empirical evaluation revealed that our approach could support the constraint elicitation process by listing up valid rules for constructing constraints.

This paper is a unified and refined version of the paper [4], titled “Towards Automatic Constraints Elicitation in Pair-wise Testing Based on a Linguistic Approach: Elicitation Support Using Coupling Strength” that appears in the Proceedings of the IEEE/AMC 2nd International Workshop on Requirements Engineering and Testing (RET 2015), and the paper [5], titled “Towards Automatic Constraint Elicitation in Test Design: Preliminary Evaluation Based on Collective Intelligence” that appears in the Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW 2015). This paper enhances these previous studies by redefining formulae, providing algorithms for automation, conducting additional experiments, and adding discussion about the applicability of the process based on the experiments.

The rest of the paper is organized as follows: Section 2 gives the background of this study by providing the explanation of pair-wise testing and constraints; Section 3 gives the overview of our approach; Sections 4 and 5 describe how our process finds constraints; Section 6 presents the results of two experimental constraint elicitation, and Sect. 7 explains how we evaluate our approach with the experimental results; Section 8 discusses related work, and Sect. 9 concludes the paper.

2. Pair-Wise Testing and Constraints

Pair-wise testing [1], [2] requires that every pair of parameter values be tested at least once. Consider a web application which may be influenced by various factors including operating systems, browsers and plugins as an example of a System Under Test (SUT). Table 1 lists the possible values of parameters. This test model has three parameters, i.e., OS, Browser, and Plugin, and their parameter values. A test case is a vector of parameter values, e.g., (Mac, Safari, QuickTime), which corresponds to the test case that we check whether QuickTime plugin works on the browser Safari when we use Mac as an OS. From this test model, we
Table 2  Test cases for pair-wise testing constructed without considering constraints.

<table>
<thead>
<tr>
<th>No.</th>
<th>OS</th>
<th>Browser</th>
<th>Plugin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Windows</td>
<td>Safari</td>
<td>Media Player</td>
</tr>
<tr>
<td>2</td>
<td>Windows</td>
<td>Chrome</td>
<td>QuickTime</td>
</tr>
<tr>
<td>3</td>
<td>Windows</td>
<td>IE</td>
<td>Media Player</td>
</tr>
<tr>
<td>4</td>
<td>Mac</td>
<td>Chrome</td>
<td>Media Player</td>
</tr>
<tr>
<td>5</td>
<td>Mac</td>
<td>IE</td>
<td>QuickTime</td>
</tr>
<tr>
<td>6</td>
<td>Mac</td>
<td>Safari</td>
<td>QuickTime</td>
</tr>
<tr>
<td>7</td>
<td>Linux</td>
<td>IE</td>
<td>Media Player</td>
</tr>
<tr>
<td>8</td>
<td>Linux</td>
<td>Safari</td>
<td>QuickTime</td>
</tr>
<tr>
<td>9</td>
<td>Linux</td>
<td>Chrome</td>
<td>QuickTime</td>
</tr>
</tbody>
</table>

Table 3  Test cases for pair-wise testing constructed with considering constraints.

<table>
<thead>
<tr>
<th>No.</th>
<th>OS</th>
<th>Browser</th>
<th>Plugin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Windows</td>
<td>IE</td>
<td>Media Player</td>
</tr>
<tr>
<td>2</td>
<td>Mac</td>
<td>Safari</td>
<td>Media Player</td>
</tr>
<tr>
<td>3</td>
<td>Windows</td>
<td>Chrome</td>
<td>QuickTime</td>
</tr>
<tr>
<td>4</td>
<td>Linux</td>
<td>Chrome</td>
<td>QuickTime</td>
</tr>
<tr>
<td>5</td>
<td>Mac</td>
<td>Safari</td>
<td>QuickTime</td>
</tr>
<tr>
<td>6</td>
<td>Windows</td>
<td>IE</td>
<td>QuickTime</td>
</tr>
<tr>
<td>7</td>
<td>Mac</td>
<td>Chrome</td>
<td>QuickTime</td>
</tr>
<tr>
<td>8</td>
<td>Linux</td>
<td>Chrome</td>
<td>QuickTime</td>
</tr>
</tbody>
</table>

can construct a set of test cases for pair-wise testing as illustrated in Table 2. This set of test cases cover every pair of parameter values.

The test space of pair-wise testing is modeled by a set of parameters, their values, and constraints on the value combinations as well as other testing techniques. In particular, constraints define the combinations that never happen, and therefore, they must be excluded from test cases. For example, when we choose Mac for the parameter OS, we should choose as the browser Safari or Chrome, because Mac OS does not support IE (Internet Explorer). This constraint is defined as follows:

Constraint: OS = “Mac” ⇒ Browser = “Safari” || “Chrome”

In the presence of constraints it is necessary to design a set of test cases such that all of the test cases satisfy the constraints. Otherwise, some test cases would not be executable because of constraint violation, resulting in redoing the test design process. Once we define constraints, we can use test generation tools, such as PICT [6], ACTS [7], and CASA [8], to acquire a set of test cases. These tools accept a test model and constraints as input data and generate the set of test cases. In addition, in [9], we showed how constraints can be handled when constraint programming is used to construct test cases [10], [11]. Table 3 represents a set of test cases for the cross-browser testing constructed with considering constraints. Comparing with the test cases illustrated in Table 2, we can find the difference of the number of test cases and the difference of test cases themselves, i.e., the difference of test space.

3. Our Approach

The goal of this paper is to support identifying which combinations of values define constraints. There are two difficulties in the constraint elicitation: one exists in parameter combination identification and the other exists in value pair determination. In our example explained in Sect. 2, we first have to identify which parameter combinations, such as parameters “OS” and “Browser”, or parameters “OS” and “Plugin”, form constraints. Next, after identifying parameter combinations that form constraints, we also have to determine which value combinations cause constraints. It corresponds to the determination of the value pairs, such as “Mac” and “Safari” or “Mac” and “Chrome” on the parameter combination “OS” and “Browser” in our example.

Therefore, we address the following research questions in this paper:

- **RQ1**: Can we identify the parameter combinations that form constraints without particular domain knowledge?
- **RQ2**: Can we identify the value pairs on the given parameters without particular domain knowledge?

In order to answer the research questions, we propose a constraint elicitation process illustrated in Fig. 1. We assume that the parameters and parameter values are determined before applying our constraint elicitation process. These are determined by manually analyzing specification documents and other artifacts, such as code and GUIs. This process is called test parameter analysis and has been widely studied [12], [13].

Our elicitation process consists of the following two steps:

- **Step 1: Parameter combination identification.** We identify which parameter combinations form constraints. We find such combinations from the relevant document of the test model using a metric that we propose in this paper.
- **Step 2: Value pair determination.** We determine which value combinations probably cause constraints. We use collective intelligence for this determination.

Each step deals with the corresponding difficulty that we described. Subsequent sections explain these steps.

4. Step 1: Parameter Combination Identification

The objective of Step 1 is to identify which parameter combinations form constraints. Czerwonka explains constraints as dependencies among test factors (parameters) [6], meaning that constraints are caused by the dependency relationship between parameters. We expect that most constraints are caused by strong relationships between parameters. In order to find such strong relationships that probably cause constraints, we define a metric called *coupling strength* $\sigma$. The metric $\sigma(f, g)$ between parameters $f$ and $g$ is expressed as follows:

$$
\sigma(f, g) = \frac{\alpha}{d(f, g)}
$$

(1)
where \( d(f, g) \) represents the distance between two parameters, and \( \alpha \) is a normalization factor. This formula represents that the coupling strength is proportional to the reciprocal of the distance between two parameters.

Step 1 identifies this distance by using relevant documents of the test model. This step uses the following formula to calculate the \( d(f, g) \) value:

\[
\sigma(f, g) = \frac{\sum_{p \in P^f} \min_{q \in P^g} |p - q| + \sum_{q \in P^g} \min_{p \in P^f} |q - p|}{|P^f| + |P^g|}
\]

(2)

where \( f \) and \( g \) are parameters and \( F \) and \( G \) are two word groups corresponding to the parameters. The word group is defined as a set whose members are the parameter and the values of the parameters. In our example, \( f \) and \( g \) correspond to “OS”, “Browser”, and “Plugin”, and \( F \) and \( G \) correspond to the word groups, such as \{OS, Windows, Mac, Linux\} or \{Browser, IE, Safari, Chrome\}. \( P^W = \langle p_1^W, \ldots, p_n^W \rangle \) represents the positions of words contained in the word group \( W \) in a relevant document of the test model. We use a document that provides the knowledge about the test model, such as the requirements document, design document, and user manual, as the relevant document. To calculate the \( d(f, g) \) value, we first extract all occurrences of words in the word group \( F \) and \( G \) from the relevant document, that is, construct \( P^W \) for all parameters. After constructing \( P^W \), we find the nearest other group’s word (occurrence). We sum up the distances between these pairs to calculate the \( d(f, g) \) value. This value becomes low when a number of word pairs are located near in the document. A low \( d(f, g) \) value indicates stronger coupling and such coupling, that is the parameter combination, may contain constraints. Note that \( \sigma(f, g) \) should be normalized for each parameter.

**Algorithm 1** Identify parameter combinations that form constraints.

**Input** \( \text{Params} \): parameter list of the test model;

\( \theta \): threshold for extracting parameter combinations;

**Output** \( \text{Combs} \): parameter combinations that form constraints;

1: \( \sigma[i][j] \leftarrow \text{initialized by zero}; \)

2: \( \text{//} \sigma[i][j] \text{: Array of coupling strength} \)

3: for \( i = 0; i < \text{Params.size}(); i++ \) do

4: for \( j = 0; j < \text{Params.size}(); j++ \) do

5: if \( i \neq j \) then

6: \( \sigma[i][j] \leftarrow 1/|d(\text{Params}[i], \text{Params}[j])| \text{ // inverse of distance} \)

7: end if

8: end for

9: end for

10:

11: \( \text{// normalization} \)

12: for \( i = 0; i < \text{Params.size}(); i++ \) do

13: \( \text{sum} \leftarrow 0; \)

14: for \( j = 0; j < \text{Params.size}(); j++ \) do

15: \( \text{sum} \leftarrow \text{sum} + \sigma[i][j]; \)

16: end for

17: for \( j = 0; j < \text{Params.size}(); j++ \) do

18: \( \sigma[i][j] \leftarrow \sigma[i][j]/\text{sum}; \)

19: end for

20: end for

21:

22: \( \text{// extract parameter combinations whose} \sigma \text{ are high} \)

23: \( \text{Combs} \leftarrow \{(\text{Params}[i], \text{Params}[j]) | \sigma[i][j] \geq \theta \}; \)

24: return \( \text{Combs}; \)
Algorithm 2 Calculate distance $d(f, g)$.

**[Input]** $f[1]$, $g[1]$: parameters and their values;
// $f[0]$, $g[0]$: parameters

**[Output]** $d$: distance between parameters $f$ and $g$;

1: // find words in document $doc$
2: List $p^f ←<>$;
3: for ($i = 0; i < f.size(); i++$) do
4: $p^f.add(getPositions(f[i], doc))$;
5: end for
6: List $p^g ←<>$;
7: for ($j = 0; j < g.size(); j++$) do
8: $p^g.add(getPositions(g[j], doc))$;
9: end for
10: // calculate distance
11: $sum ← 0$;
12: for all $p_i$ in $p^f$ do
13: $min ← ∞$;
14: for all $p_j$ in $p^g$ do
15: if $min > |p_i − p_j|$ then
16: $min ← |p_i − p_j|$;
17: end if
18: end for
19: $sum ← sum + min$;
20: end for
21: // calculate distance
22: $sum ← sum + min$;
23: end for
24: $d ← sum / (p^g.size() + p^f.size())$;
25: return $d$;

Algorithm 3 Determine value pairs that form constraints.

**[Input]** Combs: parameter combinations extracted by Algorithm 2;

**[Output]** HighPairs, LowPairs: Value pairs that form constraints;

1: for all $(p_1, p_2)$ in Combs do
2: $hit[i][j] ←$ initialized by zero;
3: $f[1] = p_1.setArray();$ // $f[0], g[0]$ parameters
4: $g[1] = p_2.setArray();$ // $f[1], f[2], ...$, $g[1], g[2], ...$ : values
5: for ($i = 1; i < f.size(); i++$) do
6: for ($j = 1; j < g.size(); j++$) do
7: $hit[i][j]$ ← hit count when the list $< f[0], f[i], p_2(= g[0]), g[j] >$ is given as search words;
8: end for
9: end for
10: // Calculate rate values. We calculate $rate(f, g) (= rate1)$ and $rate(g, f) (= rate2)$, respectively.
11: $rate1[i][j] ←$ initialized by zero;
12: for ($i = 1; i < f.size(); i++$) do
13: $sum ← 0$;
14: for ($j = 1; j < g.size(); j++$) do
15: $sum ← sum + hit[i][j]$;
16: end for
17: $rate1[i][j] ← hit[i][j] / sum$;
18: end for
19: rate2$[i][j] ←$ initialized by zero;
20: for ($j = 1; j < g.size(); j++$) do
21: $sum ← 0$;
22: for ($i = 1; i < f.size(); i++$) do
23: $sum ← sum + hit[i][j]$;
24: end for
25: $rate2[i][j] ← hit[i][j] / sum$;
26: end for
27: $HighPairs ← \{(i, j) | rate1[i][j] ≥ \theta_1 \land rate2[i][j] ≥ \theta_1\}$;
28: $LowPairs ← \{(i, j) | rate1[i][j] ≤ \theta_1 \land rate2[i][j] ≤ \theta_1\}$;
29: return HighPairs, LowPairs;
30: end for

5. Step 2: Value Pair Determination

The objective of this step is to determine which combinations of values define constraints. Constraints represent the value combinations that never happen. In particular, we try to derive this knowledge from collective intelligence. We evaluate the strength of value pairs based on the co-occurrence of the value pairs in the web. If a pair is relatively uncommon in the web, the pair may be the one to be excluded by a constraint; if a pair frequently appears in the web, it probably indicates that one value may restrict the other value, which in turn indicates a constraint. We determine which value combinations probably cause constraints by using a web search engine. We use search hits as a metric for this purpose.

The right part of Fig. 1 illustrates an overview of our value pair determination step (Step2). Step 2 counts hits for each value pair on extracted parameter combinations in Step 1, using a web search engine. The step extracts value pairs whose hit counts are considerably high or low. We assume that excessively high (or low) hit counts indicate that the corresponding value pairs are closely related (or uncommon). A test designer defines constraints using these extracted pairs. Step 2 uses the web search engine with value parameter since $d(f, g)$ is comparative value. We use Algorithm 1 to identify the parameter combinations that probably form constraints. Algorithm 1 uses Algorithm 2 for the calculation of distance $d(f, g)$ in the process. After calculating $\sigma$ values, we extract parameter combinations whose $\sigma$ values are relatively higher than the other parameter combinations one of whose parameters has the same value and the other has a different value.
pairs as search words and compares search hit counts. Considerably high/low hit counts indicate that these pairs could be concerned with constraints.

Algorithm 3 describes how to determine the value pairs that probably form constraints. Step 2 uses the following metric to calculate the rate of the value combination:

$$rate(f_i, g_j) = \frac{\text{hits}(< f, g_j, f_i >)}{\sum_{g \in G} \text{hits}(< g, f, g_j, f_i >)}$$

(3)

where \( f \) and \( g \) are the given parameters, \( f_i \) and \( g_j \) are values of \( f \) and \( g \), respectively, \( G \) is the set of values for parameter \( g \), and \( \text{hits}(L) \) represents the number of hits when list \( L \) is given as a search keyword. A search keyword is a list of words. Since most of the web search engines consider the order of words in a search keyword, we need to fix the order of words in the keyword, unlike in our previous work described in [4]. We order words in a search keyword as follows: \( \langle \text{parameter } g \text{'s name}, \text{parameter } f \text{'s name}, g \text{'s value}, \text{and } f \text{'s value} \rangle \). The first and second words, i.e., parameter names, dominantly exclude the web contents that have less relationship with these parameters. Since the \( rate \) value is a relative value among parameter \( g \text{'s value} \) and it is desirable to emphasize the relative value, we place \( g \text{'s value} \) in the front of \( f \text{'s value} \) in the search keyword (see \( g_j \) and \( f_i \) in Expression (3)). Step 2 sums up the hit counts among all possible pairs \( f_i \) and \( g_j \) \((g \in G)\) and calculate the rate for given \((f_i, g_j)\). The \( rate(f_i, g_j) \) value becomes high (or low) when the appearance of the words \( f_i \) and \( g_j \) in the web contents is more (or less) frequent than the appearance of \( f_i \) and \( g_k \) \((j \neq k)\). An extremely low rate value probably indicates an uncommon value pair, which may mean the pair to be excluded. An extremely high value, on the other hand, probably indicates that one value may restrict the other value, which in turn indicates a constraint.

6. Experiments

In order to evaluate the validity of our elicitation process, we conducted experiments on some test models. First, we evaluated the validity of each step on simple SUT (System Under Test) models (Exp 1 and Exp 2, respectively). Then, we applied our process to a SUT model for the real world web application for evaluating the scalability of our elicitation process (Exp 3).

6.1 Exp 1: Parameter Combination Identification

First, in order to evaluate the validity of our parameter combination identification (Step 1), we conducted an experiment on ATM (Automated Teller Machine) system example described in [14]. This example provides sample artifacts for software development from the requirements document to the implementation code. Table 4 illustrates a SUT model for the ATM software, which we constructed by analyzing the system’s GUI. This SUT model contains six parameters, i.e., Transaction, Accounts (A) and (B), Amount, Card, and PIN. These parameters have parameter values as illustrated in Table 4.

<table>
<thead>
<tr>
<th>Table 4</th>
<th>ATM SUT model.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>Values</td>
</tr>
<tr>
<td>Transaction</td>
<td>withdrawal, deposit, transfer, balance inquiry</td>
</tr>
<tr>
<td>Account (A)</td>
<td>checking, savings, money market</td>
</tr>
<tr>
<td>Account (B)</td>
<td>checking, savings, money market, not selected</td>
</tr>
<tr>
<td>Amount</td>
<td>$20, $40, $60, $100, $200, None</td>
</tr>
<tr>
<td>Card</td>
<td>valid, invalid, unreadable</td>
</tr>
<tr>
<td>PIN</td>
<td>correct, incorrect, non-enterable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5</th>
<th>Coupling strength ( \sigma(f, g) ) values. These values are normalized for each parameter ( f ). “Account” represents the parameters Account (A) and (B).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>( p_1 )</td>
</tr>
<tr>
<td>Transaction</td>
<td>0.264</td>
</tr>
<tr>
<td>Account</td>
<td>0.127</td>
</tr>
<tr>
<td>Amount</td>
<td>0.146</td>
</tr>
<tr>
<td>Card</td>
<td>0.178</td>
</tr>
<tr>
<td>PIN</td>
<td>0.164</td>
</tr>
</tbody>
</table>

Part of possible constraints of this test model are as follows:

- Constraint 1-a: Transaction = “Balance inquiry” ⇒ Account (B) = “Not selected”.
- Constraint 1-b: Transaction = “Balance inquiry” ⇒ Account = “Not selected”.
- Constraint 1-c: Card = “Invalid” || “Unreadable” ⇒ PIN = “Non-enterable”.

Table 5 illustrates the results of \( \sigma(p_1, p_2) \) calculation. From the results, we extracted the following parameter combinations:

- Combination 1-1: (Transaction, Amount), from the row \( p_1 = “Transaction” \).
- Combination 1-2: (Transaction, Account), from the row \( p_1 = “Transaction” \).
- Combination 1-3: (Transaction, Card), from the row \( p_1 = “Transaction” \).
- Combination 1-4: (Account, Amount), from the rows \( p_1 = “Account” \) and \( p_1 = “Amount” \).
- Combination 5: (Card, PIN), from the rows \( p_1 = “Card” \) and \( p_1 = “PIN” \).

Some of the extracted combinations correspond to the constraints previously described — Combination 1-1 matches Constraint 1-b; Combination 1-2 matches Constraint 1-a; and Combination 1-5 matches Constraint 1-c. All extracted combinations also seem to be valid. For example, account type may pose a limitation of withdrawal amount (Combination 1-4).

6.2 Exp 2: Value Pair Determination

Next, we conducted two experiments on simple cross-browser testing (Exp 2-1) and ATM system testing (Exp 2-2) for evaluating the validity of our value pair determination (Step 2). We use the SUT model illustrated in Table 1 for
the cross-browser testing and the SUT model illustrated in 4 for the ATM testing (same as Exp 1), respectively. We use the Google search engine as a web search engine and calculate the rate value for some combinations of parameters to determine value pairs that probably form constraints. Since \( rate(f, g) \) and \( rate(g, f) \) are different by definition due to a difference in denominators, we calculated both rates. We evaluated the validity of suggested value pairs by comparing with the real constraints to be extracted.

### 6.2.1 Exp 2-1: Cross-Browser Testing

As for the cross-browser testing, we focused on the combination of parameters OS and browser. In reality, the following constraints exist on the combination:

- **Constraint 2-a**: OS = “Windows” \( \Rightarrow \) Browser = “IE” || “Chrome”.
- **Constraint 2-b**: OS = “Mac” \( \Rightarrow \) Browser = “Safari” || “Chrome”.
- **Constraint 2-c**: OS = “Linux” \( \Rightarrow \) Browser = “Chrome”.

Tables 6 and 7 show the calculation results of \( rate(f, g) \). We chose search keywords for counting search hits according to the order of words as described in Sect. 5. For example, we used a search keyword “Browser OS IE Windows” when we counted search hits for OS (parameter \( f \)) = “Windows” and Browser (parameter \( g \)) = “IE” in Table 6.

We regarded the rate as considerably high if the rate is higher than 0.5 and more than double of any other parameter values \( g_k \) (\( k \neq j \)). We also regarded the rate as considerably low when the rate is lower than 0.05. Based on the pairs with considerably high rates, we derived the following rules:

- **Rule 2-1**: Browser = “IE” \( \Rightarrow \) OS = “Windows” (from Table 7)
- **Rule 2-2**: Browser=“Chrome” \( \Rightarrow \) OS=“Windows” (from Table 7)

Rule 2-1 is correct under Constraints 2-a to 2-c; Rule 2-2 satisfies Constraints 2-a to 2-c but is too restrictive.

According to the pairs with considerably low rates, on the other hand, we derived the rules which exclude Linux from every browser’s OS choice.

- **Rule 2-3**: OS=“Linux” \( \Rightarrow \) Browser=“IE”||“Chrome” (from Table 6)
- **Rule 2-4**: Browser=“IE” \( \Rightarrow \) OS=“Windows”||“Mac” (from Table 7)
- **Rule 2-5**: Browser=“Safari” \( \Rightarrow \) OS=“Windows”||“Mac” (from Table 7)

These rules satisfy Constraints 2-a to 2-c but are too loose in the sense that they allow generating test cases that should be excluded; however, Rules 2-1 has already implied Rule 2-4. To sum up, our process suggested a correct rule, three loose rules, and a restrictive rule. All real constraints could be found by loosening or restricting the extracted rules.

### 6.2.2 Exp 2-2: ATM Example

As shown in Sect. 6.1, the SUT model of ATM example contains six parameters, Transaction, Accounts (A) and (B), Amount, Card, and PIN. We, in particular, focused on the combination of parameters Transaction and Account (B). A possible constraint for this combination is as follows:

- **Constraint 2-d**: Transaction = “withdrawal” \( || \) “deposit” || “balance inquiry” \( \Rightarrow \) Account (B) = “not selected”.

Tables 8 and 9 show the \( rate(f, g) \) values in this case study. We applied the same rules for determining considerably high/low rates as the ones in Exp 2-1. According to considerably high rate values, we derived the following rules from both the tables:

- **Rule 2-6**: Transaction = “deposit” \( \Rightarrow \) Account (B) = “not selected” (from Table 8)
- **Rule 2-7**: Transaction = “balance inquiry” \( \Rightarrow \) Account (B) = “not selected” (from Table 8)
Rules 2-6 and 2-7 form part of Constraint 2-d. We also derived the following rule from Table 9.

- Rule 2-8: Account (B) = "checking" \(\Rightarrow\) Transaction = "withdrawal" (from Table 9)
- Rule 2-9: Account (B) = "savings" \(\Rightarrow\) Transaction = "withdrawal" (from Table 9)
- Rule 2-10: Account (B) = "not selected" \(\Rightarrow\) Transaction = "deposit" (from Table 9)

Rule 2-10 implies Constraint 2-d, but Rules 2-8 and 2-9 have no relationship with Constraint 2-d. We also extracted rules according to the \((f_i, g_j)\) pairs with considerably low rates, such as the following one.

- Rule 2-11: Transaction = "deposit" \(\Rightarrow\) Account (B) = "checking" \(\land\) "money market" \(\lor\) "not selected" (from Table 8)

Similar to Rules 2-3 to 2-5 in Exp 2-1, most of these rules satisfy the constraint but are too loose. To sum up, the process suggested nine rules including two correct rules, three loose rules, and a restrictive rule. All real constraints could be found by loosen or restricting the extracted rules.

6.3 Exp 3: Real World Web Application

Finally, in order to evaluate the scalability of our approach, we applied our approach to the real world web application. The target application is Confluence\,[15], which is a web application designed for team collaboration.

We prepared an input document by flattening the web pages of the Confluence document site\[16]. We constructed a SUT model for this case study as illustrated in Table 10. This test model was constructed by applying the classifica-

---

From the result listed in Table 11, we identified that the parameters Database and Server could form constraints and applied value pair determination process to these parameters. Same as in Exp 2, we regarded the rate as considerably high if the rate is higher than 0.5 and more than double of any other parameter values $g_k$ ($k \neq j$). We also regarded the rate as considerably low when the rate is lower than 0.05. In reality, the following constraints exist on the combination:

- **Constraint 3-a:** Database = “Microsoft SQL Server” \(\Rightarrow\) Server = “Windows Server”.
- **Constraint 3-b:** Database = “Oracle” \(\Rightarrow\) Server = “Windows Server” || “Unix” || “Linux”.

We list up some derived rules.

- **Rule 3-1:** Server = “Unix” \(\Rightarrow\) Database = “PostgreSQL” || “MySQL” || “Oracle”.
- **Rule 3-2:** Server = “Linux” \(\Rightarrow\) Database = “PostgreSQL” || “MySQL” || “Oracle”.
- **Rule 3-3:** Server = “OS X” \(\Rightarrow\) Database = “PostgreSQL” || “MySQL” || “Oracle”|| “H2”.
- **Rule 3-4:** Database = “Oracle” \(\Rightarrow\) Server = “Windows Server” || “Unix” || “Linux”.
- **Rule 3-5:** Database = “Microsoft SQL Server” \(\Rightarrow\) Server = “Windows Server” || “Unix” || “Linux”.

Comparing the extracted rules with the real constraints, Rule 3-4 matches Constraint 3-b, Rules 3-1 and 3-2 are too restrictive, and Rules 3-3 and 3-5 are too loose. To sum up, the process suggested a correct rule, two loose rules, and five restrictive rules. All real constraints could be found by loosening or restricting the extracted rules.

### 7. Discussion

We now answer the research question in the light of our experiments and discuss the limitations of our current elicitation process.

**RQ1** (*Can we identify the parameter combinations that form constraints without particular domain knowledge?*): The experimental results demonstrate that our parameter combination identification method could extract combinations that form constraints. The extracted parameter combinations are valid in terms of representing the strength of relationship; however, it is not guaranteed that the extracted combinations form constraints. Since the coupling strength is relative value, we need to enhance our identification process by introducing guidelines for using the metric, such as how to determine the threshold for extracting combinations. Which document should be used as the target document that we find words from is also an important choice. From our experiments, we believe that the important aspect of the document selection is the number of words that the document contains rather than the document type, such as a requirements document, design document, and manual. Plenty of words and occurrence of the key words, i.e., parameter and their values, ensure the distance between parameters.

**RQ2** (*Can we identify the value pairs on the given parameters without particular domain knowledge?*): The experimental results demonstrate that we can extract a certain amount of constraints by using the rate of hit counts. Some of suggested pairs corresponded to constraints to be extracted, and others helped define constraints. The rate value, which is calculated based on the co-occurrence of a pair in the web, in some cases, correctly suggested that the pair should be excluded or should be chosen as a constraint, resulting in eliminating the other value combinations. However, we also observed that the most rules were too restrictive or too loose. This may be caused by the two characteristics of the value words: majority/minority and specificity. In Exp2-1, search words containing “IE” hit more web contents than the ones for the other browsers. The search word containing “Withdrawal” in Exp2-2 also hit more contents than the others; the search word containing “H2” in Exp3 hit less contents than the ones for the other DBs. Such a majority/minority issue leads to extract too restrictive or too loose rules. Our current metric rate provides a certain normalized mechanism; however, we should add more refined process to eliminate this issue. One possibility is to add search keywords for restricting the target web contents. The specificity of the value word, on the other hand, makes difficult to capture precise appearance in the web resources. Searches of common phrases, such as “not selected” in Exp2-2, did not provide useful information. As for this issue, we should replace more suitable words to count hits.

We discuss the applicability of our constraint elicitation process. The first step of our process requires a relevant document of the test model. This document should contain words corresponding to parameters or their values. Plenty of these words help acquire more precise results. At the second step, since the process chooses the value pairs that probably form constraints using hit counts of web search, values constituting these pairs must appear in the web contents. Therefore, there must be some web contents that explain the target test domain. It should be noted, even though some web contents contain wrong knowledge, we can still expect valid results owing to the massiveness of web contents. Numerous contents containing precise description help find correct value pairs. If massive contents contain counter examples, our process may suggest contrary results. In this case, a test designer must define right constraints by switching the suggested results, but we can still be aware these suggested value pairs involved in the constraints. Even if there is no web contents that explain the test domain, we can apply the rate metric to existing test cases, if they exist, instead of the web contents. In other words, test cases can be used as the source contents for finding constraints. For example, Blue et al. [3] reduce the number of test cases by eliminating value pairs that do not appear in the test cases.

Next, we discuss the validity of our elicitation process. As Czerwonka explains in [6], constraints are caused by the dependencies between parameters. Accordingly the flow of
our process, that is, choosing parameter combination first and subsequently selecting value pairs, is natural. As for the first step, i.e., the parameter combination identification, the experimental results demonstrate that the coupling strength metric can be used to extract parameter combinations that probably cause constraints. This can be accounted for by the fact that related descriptions tend to be placed at a short distance in documents. For the second step, on the other hand, the experimental results demonstrate that the rate metric could extract a certain amount of correct rules that correspond to constraints but most rules were too restrictive or too loose. Although we constructed correct constraints from the suggested rules, the second step should be improved.

Search-based approaches [16] usually deal with the problems that require us to find potential solutions from search space with no precise rules for computing the best solution. Most of these approaches find solutions using metaheuristic search techniques, such as generic algorithms (GA) [17] and simulated annealing [18]. Our approach is based on an exhaustive matching of parameter values and uses two metrics to find potential value combinations for defining constraints. Compared with ordinary search-based approaches, since the search space of the problem that we deal with is sufficiently narrowed down by sequentially executing the two steps, i.e., parameter combination identification and value pair determination, our approach does not use the metaheuristic search techniques.

8. Related Work

Currently there are few studies on the constraint elicitation. Blue et al. [3] presents test case reduction method, which excludes test cases that contain prohibited value pairs. Their method uses existing test cases and minimizes the set of test cases with covering all pairs that are included in the existing test cases. Their method is useful to define a small set of test cases without violating constraints; however, since their method does not define constraints, it is not guaranteed to provide test cases that covers all possible pairs.

Our current approach uses the distance between words to identify the strength of the relationship between parameters. Co-occurrence-based approach, such as TF-IDF [19], is also known as a method of reasoning the relationship between words. Gabrilovich et al. [20] proposed a method called Explicit Semantic Analysis (ESA), in which a word is represented as a vector whose attributes represent the relatedness to concepts calculated using TF-IDF by using Wikipedia as an input document. In our elicitation process, we choose distance-based approach instead of co-occurrence-based approach. While co-occurrence-based approach requires a large amount of document, the documents related to the target systems are usually limited. Another reason of choosing distance-based approach is the domain dependency. The coupling strength between parameters often depends on target systems. It is sometimes different from the common coupling strength that are extracted from common corpus such as Wikipedia.

Literature in the requirements engineering field has dealt with elaborated linguistic techniques among software engineering fields. Falessi et al. [21] evaluate the performance of a large number of natural language processing techniques for identifying equivalent requirements. They also proposed seven principles for evaluating the performance of these techniques. Some of them could be useful to improve our metrics, i.e., ones for identifying parameter combinations and for determining value pairs. Rahimi et al. [22] proposed an automated goal model generation method from the requirements document. In this method, they use a query augmentation method proposed in [23]. Such a query augmentation technique may improve the correctness of extracted parameter combinations and value pairs of our process.

9. Conclusions

We defined a constraint elicitation process that supports constraint definition for combinatorial test design. Our elicitation process determines parameter combinations that probably form constraints by calculating the coupling strength between parameters. The process also extracts value pairs that probably form constraints by comparing the frequency in the web contents using web search engine. The results of experimental constraint elicitation reveal that the automatically precise constraint acquisition is still difficult but suggest that our approach can support the constraint elicitation by listing up valid candidates for constructing constraints.

The results presented in this paper are promising for further work and improvements. We identified the following two major directions for improving the elicitation mechanism to identify more precise value pairs. First one involves the introduction of consistency checking mechanism. The ambiguity of current extracted rules should be eliminated by checking with other rules or facts in the documents. We also plan to define guidelines for the constraint definition, which include the criteria for determining target documents, thresholds, and search keywords that carefully restrict web contents to ones related to the test model.

Acknowledgments

This work was supported by JSPS Grants-in-Aid for Scientific Research (Grant Numbers 15K00097, 15K00098).

References


Hiroyuki Nakagawa received his B.S. degree in computer science from Osaka University in 1997, his M.S. degree in computer science from the University of Tokyo in 2007, and his Ph.D. degree in computer science from Waseda University in 2013. He worked in Kajima Corporation from 1997 to 2008 and at the University of Electro-Communications as an assistant professor from 2008 to 2013. He joined Osaka University as an associate professor in 2014. His research interests include requirements engineering, self-adaptive systems, software verification techniques, and software evolution.

Tatsuhiro Tsuchiya received his M.E. and Ph.D. degrees from Osaka University in 1995 and 1998, respectively. He is a professor of Dept. of Information Systems Engineering, Graduate School of Information Science and Technology, Osaka University.