Test Case Generation Algorithms and Tools for Specifications in Natural Language

Yusuke Aoyama
Faculty of Comp. Science and Systems Eng.
Kyushu Institute of Technology
Iizuka, Japan

Takeru Kuroiwa
Kyushu Institute of Technology
Iizuka, Japan

Noriyuki Kushiro
Kyushu Institute of Technology
Iizuka, Japan

Abstract—Nowadays, most consumer products are equipped with methods of network communications, and nondeterministic tests, which are originated from random message exchanges via the network, should be carried out. Therefore, the tests of the consumer products with network have obliged us to consume much time to design and conduct. For reducing the labor of designing test cases, algorithms and tools, which help test engineers to convert specifications written in a natural language into semiformal descriptions, and to generate test cases including deterministic and nondeterministic test cases as decision tables, are proposed in the paper. The algorithms and tools were applied to a tiny example for evaluation and confirmed that they have succeeded in generating test cases from documents in a natural language.

Index Terms—Consumer products with software product line engineering, Automatic test case generation for specification in natural language

I. INTRODUCTION

There is a growing interest to introduce Software Product Line Engineering (SPLE) into developing consumer products to respond timely to various requests from worldwide markets. The product in the SPLE utilizes common software frameworks as a core asset, and requires a huge number of regression tests to validate changes of software concerning the frameworks at every product’s development. Moreover, most latest consumer products provide network communications as IoT devices. Both deterministic and nondeterministic tests should be carried out in the regression tests. Therefore, the tests of the consumer products have obliged us to consume much time to design and carry out such tests.

Prior to this study, a hybrid testing environment for deterministic and nondeterministic tests was developed for reducing heavy load of testing consumer products with network in SPLE [1], [2]. In the testing environment, both deterministic and nondeterministic tests were conducted automatically with actual software for the consumer products. As a result, the load to conduct the tests are reduced, however, labor to design test cases still remains serious.

According to our investigation in major consumer manufacturers in Japan, about 700 hours were required for designing test cases for a new product in SPLE (around 600 pages of specifications document), and about 250 hours were consumed for redesigning the test cases at every product to which the core asset in the SPLE was applied. For reducing the labor of designing the test cases, algorithms and tools, which help test engineers to convert specifications written in a natural language into semiformal descriptions, and to generate test cases including both deterministic and nondeterministic test cases as decision tables, are proposed in the paper.

Currently, many researches are focusing on improving efficiency of test case design from specifications in a natural language. One approach is to help engineers detect errors and flaws in specification documents [3]–[6]. The other approach is to generate test cases from specifications in a natural language [7], [8]. In Nat2testscr [7], engineers write specifications with CNL (controlled natural language), and the tool converts the specifications described with the CNL into software test codes. Nat2testscr has succeeded in relaxing engineers’ load to write specifications with formal languages, like VDM++ [9]. However, it has difficulty to apply to existing documents written in a natural language.

The algorithms and tools proposed in the study tried to generate test cases from a huge number of existing specifications written in unconstrained natural language. The remaining sections are organized as follows: In Section II, issues and their solutions for generating test cases for consumer products in SPLE are described. In Section III, the proposed test case generation algorithms and tools for specifications in a natural language are elaborated. In Section IV, the experimental setting and results of the experiments are illustrated. Finally, the conclusion is shown in Section V.

II. ISSUES AND SOLUTIONS IN TEST CASE GENERATION

In Section II-A, legacy process for designing test cases in the manufacturers are modeled with data flow diagram and issues in the legacy process are discussed. Then, the solutions for the above issues are proposed in Section II-B.

A. Legacy Test Case Generation Process

Actual processes for designing test cases were modeled as Fig. 1 through investigation for two major consumer
product manufacturers in Japan. In legacy process, a test engineer reads a document of specifications intensively at the first step. In the second step, the engineer extracts candidates of test items from the document. Then, the engineer designs test cases as a couple of test conditions and expected actions for each test item roughly at the third step. The engineer elaborates test cases by providing concrete values both for test conditions and the expected actions at the fourth step. The outputs from each step are reviewed by a team composed of products engineers and test engineers step by step. Errors in test case design are eliminated heuristically through repetition of reviews, and flaws originated in the specification document are also compensated through the repetitious review processes.

![Legacy Test Cases Design Process for Consumer Products](image1)

Each step is carried out exclusively by a test engineer without any guidelines or standards for designing test cases. Therefore, qualities of the outputs for each step heavily depends on each test engineer’ skill and experience, and the time required for fulfilling each step also depends on them. A lot of man-made errors are included into the outputs due to excessive handiwork of the test engineer. Detecting errors and flaws in the test cases become difficult through the reviews, because all of the design steps are conducted implicitly by one test engineer.

**B. Proposed Test Case Generation Process**

Test case generation process show in Fig. 2 is design based on the issues detected in the legacy handiwork steps, described in Section II-A.

![Proposed Test Cases Design Process for Consumer Products](image2)

All the steps in the design process are entirely supported by algorithms and tools in the proposed process. Each sentence in a document is converted into semiformal descriptions by the tool called “semiformalizer” instead of test engineer’s intensive reading. Logical relations between semiformal descriptions are visualized by the tool “proponet” to help the engineers to detect errors and to compensate flaws including documents through the reviews. The test specifications corrected via the reviews are converted automatically into decision tables by the tool “splosat”. Though the proposed process were elaborated on the survey for actual test design processes in the consumer products manufacturers, it does not seem to be exclusive for testing consumer products and the process can apply widely to products in other domain.

**III. Test Case Generation Tools**

All the steps for generating test cases are defined explicitly as algorithms of the tools, and the reviews for detecting errors and compensating flaws are also supported by the tools in the proposed process. The details of algorithms and tools for each step are explained in the following sections.

**A. Converting Sentences to Semiformal Description**

The “semiformalizer” converts sentences in a document into semiformal descriptions [6] to reduce ambiguity such as orthographical variants and to clarify logical relations among words in a natural language. The algorithm of converting a sentence into semiformal descriptions is shown in Algorithm. 1. The conversion algorithm utilizes “cases” as a grammatical information [10], which provide informations about the role of each word in a sentence, e.g. subject, object, relation, and constraints (candidates for variables and/or values). Fig. 3 shows an example of semiformal descriptions generated with the “semiformalizer”. Words in a sentence are assigned as elements in an atomic proposition according to the “cases”. The “cases” are obtained from results of syntactic parsers, for example, CoreNLP [11] for English, and KNP [12] for Japanese.

![Example of Semiformal Description](image3)

Logical relations among the atomic propositions are described with the fundamental logical symbols: “Not(!)”, “And(&)”, “Or(|)” and “Imply(→)”. Furthermore, the following four symbols: “One”, “Exclusive”, “Inclusive” and “Require”, are supplemented to express complex conditions among the atomic proposition. These relations among the atomic propositions are expressed by using “p_constraint” on the notation syntax shown in Fig. 4. The order of precedence among logical symbols is based on general mathematical order.
B. Logical Relations among Semiformal Descriptions

Semiformal descriptions depict logical relations explicitly; however, it remains still difficult to grasp complex logical relations because of their textual notation. The “proponet” visualizes logical relations among semiformal descriptions as a propositional network (Fig. 5). In the propositional network, each atomic primitive is illustrated as an element of the network (the part circled with broken line in Fig. 5), and logical relation among the atomic primitives, e.g. “&”, “|” and “⇒”, are illustrated with structural expression defined in Fig. 5.

Fig. 5. Propositional Network

B. Logical Relations among Semiformal Descriptions

Sentences included in a system specification document are regarded as a set of statements referring to preconditions and consequences of system requirements. Therefore, the test case design is interpreted as the task extracting statements both for test conditions and for expected actions from each sentence. The “semiformalizer” helps engineers to extract statements of test conditions and expected actions from the sentences and the “proponet” expresses these statements as pairs of semiformal descriptions connected with the implication “⇒”. Thus, the test conditions and the expected actions are identified with the propositional network by rote. The “proponet” emphasizes a boarder between antecedents and consequences as bold bars (see in Fig. 5) so as to make test engineers easy to recognize test conditions and expected actions. From the viewpoint of conducting test cases, similar test conditions should be combined as one test case. The “proponet” also provides a method to find out similar atomic propositions, and to structure them as one propositional network. Fig. 6 is illustrates with Algorithm 2.

Fig. 6. Example for Merging on Similarities of Atomic Propositions

Algorithm 1 Convert Sentences to Semiformal Descriptions

Definition:
children(b) is a list of clauses that modify a clause b.
pop(l) removes the last element e from a list l and return e.
last(l) is the last element of a list l.
reversed(l) is a list in the reverse order of a list l.
apply(e, c) applies an operator c to a semiformal description d and returns a tuple of a semiformal description that is the result of application of e and a list of clauses that is associated to d.
NewSEMILEMENT(b) converts a clause b to an element of the semiformal description.
NewOPERATOR(s1, s2) returns a new operator from two elements of the semiformal description. s1 = (d1, B1) and s2 = (d2, B2), d1 and d2 are a semiformal description d1 modifies d2, B1 and B2 are lists of clauses that are associated to d1 and d2, respectively.

Input:
b: a terminal clause

Output:
s: converted semiformal description

1: procedure Convert(b)
2: stack ← ()
3: C ← children(b)
4: s1 ← (NewSEMILEMENT(b), (b)) ▶ Rule 1
5: s2 ← nil
6: while true do
7: if s1 = nil then
8: if |C| = 0 then
9: break
10: end if
11: c ← pop(C)
12: s2 ← (Convert(c), (c))
13: end if
14: combb1 ← NewOPERATOR(s1, s2) ▶ Rule 2
15: if |stack| = 0 v combb1 is prior to last(stack) then
16: push(stack, combb1)
17: s2 ← nil
18: continue
19: end if
20: combb2 ← last(stack)
21: if combb2 can apply to s2 then
22: s2 ← apply(comb2, s2) ▶ Conversion 1
23: continue
24: end if
25: s1 ← apply(comb1, s1) ▶ Conversion 2
26: s2 ← nil
27: end while
28: for all comb in reversed(stack) do
29: s1 ← apply(comb, s1) ▶ Conversion 3
30: end for
31: return s
32: end procedure

Fig. 4. Syntax for Notation of Semiformal Description
In Fig. 6, two different (unitary) test cases for testing the bold and italic features. A conjunction between both antecedents of the implication: if the user clicks on the B icon and a node, the word processor makes the text bold “&” italic. The algorithm for combining the similar atomic primitives is shown in Algorithm 2.

Algorithm 2 Combine Similar Atomic Propositions

Definition:
- T: a set of node types;
  \[ T = \{ \text{Atomic, And, Or, Imply} \} \]
- n: a node; n = (t, A), t \in T
- A: the arguments of a node;
  \[ A = \begin{cases} (s, o, r, C) & (t = \text{Atomic}) \\ \{ \text{a set of nodes} & (t \neq \text{Atomic}) \end{cases} \]
- s: subject, o: object, r: relation, C: a set of constraints

Input:
- n1; a node: n1 = (t1, A1)
- n2; a node: n2 = (t2, A2)

Output:
- NMr: a set of merged nodes

1: \text{procedure \emph{GROUP}(n1, n2)}
2: \text{A}_{\text{new}} \leftarrow \{ \}
3: \text{merged} \leftarrow \text{false}
4: \text{if } t1 = t2 = \text{Atomic} \text{ then}
5: \text{define } s_{1, 0}, o_{1, 1}, r_{1, 1}, \text{ and } C_{1} \text{ where } A_{1} = (s_{1}, o_{1}, r_{1}, C_{1})
6: \text{define } s_{2, 0}, o_{2, 1}, r_{2, 1}, \text{ and } C_{2} \text{ where } A_{2} = (s_{2}, o_{2}, r_{2}, C_{2})
7: \text{if } s_{1} = s_{2} \land o_{1} = o_{2} \land r_{1} = r_{2} \text{ then}
8: \text{return } \{ (t1, (s1, o1, r1, C1 \cup C2)) \}
9: \text{end if}
10: \text{return } \{ n1, n2 \} \text{ \( \triangleright \) not merged}
11: \text{else if } t1 = \text{Atomic} \lor t2 = \text{Atomic} \text{ then}
12: \text{define } t_k \text{ and } A_k \text{ where } t_k \in \{ t1, t2 \}, A_k \in \{ A1, A2 \}, \text{ and } t_k \neq \text{Atomic}
13: \text{define } n_k \text{ where } n_k = (t_k, A_k)
14: \text{for all } n \in A_k \text{ do}
15: \text{A}_{\text{merged}} \leftarrow \text{\emph{GROUP}(n, n_k)}
16: \text{A}_{\text{new}} \leftarrow \text{A}_{\text{new}} \cup \text{A}_{\text{merged}}
17: \text{merged} \leftarrow \text{merged} \lor |A_{\text{merged}}| = 1
18: \text{end for}
19: \text{if merged then}
20: \text{return } \{ t_k, A_{\text{new}} \} \}
21: \text{end if}
22: \text{return } \{ n1, n2 \} \text{ \( \triangleright \) not merged}
23: \text{end if}
24: \text{if } t1 = t2 \text{ then}
25: \text{if } 3n \in A_k \text{ such that } n \in A_k \text{ then}
26: \text{return } \{ t1, A1 \cup A2 \} \}
27: \text{end if}
28: \text{return } \{ n1, n2 \} \text{ \( \triangleright \) not merged}
29: \text{end if}
30: \text{for all } n \in A_k \text{ do}
31: \text{A}_{\text{merged}} \leftarrow \text{\emph{GROUP}(n, n_k)}
32: \text{A}_{\text{new}} \leftarrow \text{A}_{\text{new}} \cup \text{A}_{\text{merged}}
33: \text{merged} \leftarrow \text{merged} \lor |A_{\text{merged}}| = 1
34: \text{end for}
35: \text{if merged then}
36: \text{return } \{ t_k, A_{\text{new}} \} \}
37: \text{end if}
38: \text{return } \{ n1, n2 \} \text{ \( \triangleright \) not merged}
39: \text{end procedure}

C. Converting Semiformal Descriptions to Decision Table

The “splosat” extracts combinations of execution conditions and expected actions required for each test case from semiformal descriptions, and generates automatically a decision table [13] through the following four steps:
1) Put antecedents for each implication as test conditions in a decision table
2) Put consequences for each implication as expected actions
3) Calculate values of each implication with PyEDA [14]
4) Fill the truth values at cells in the decision table.

Table I is an example of decision table generated by the “splosat” for semiformal descriptions shown in Fig. 6. All possible combinations of test conditions and expected actions required for the test case are included exhaustively in the decision table. Step for combining test conditions and expected actions for each test case consumes much time to conduct and exhausts test engineers. The output of the step may be contaminated with a lot of man-made errors. The “splosat” avoids such kind of contaminations of errors by reducing excessive handiwork of the test engineers.

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Function #</th>
<th>Row Type</th>
<th>Atomic Proposition</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>condition</td>
<td>①</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td></td>
<td>action</td>
<td>②</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Second</td>
<td>condition</td>
<td>③</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td></td>
<td>action</td>
<td>④</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>①: click(user,B_icon)</td>
<td>②: make(word_processor,text,bold)</td>
<td>③: click(user,L_icon)</td>
<td>④: make(word_processor,italic)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“."; don’t care (Both T and F are acceptable.)

IV. Evaluation of Algorithms and Tools

The tools were implemented based on the algorithms explained in the former sections. The tools were applied to a tiny example for evaluation and confirmed that the algorithms and the tools have succeeded in generating test cases for documents in a natural language on the proposed process.

A. Specifications Document for Evaluation

The following specifications for insulin pump [15] are utilized as for evaluation:
1) No insulin shall be delivered, when the readings are below the safe minimum
2) Insulin shall be only delivered if the level of blood sugar is rising and the rate of increase of blood level is increasing, when the readings are within the safe zone
3) Insulin shall be delivered unless the level of blood sugar is falling and the rate of decrease of sugar level is increasing, when the readings are above the recommended level

B. Converting Sentences to Semiformal Descriptions

The above specifications written in a natural language were converted into the following semiformal descriptions with the “semiformalizer”:
1) is(the_readings, below_the_safe_minimum) \( \rightarrow \) !deliver(??, insulin).
2) is(the_readings, within_the_safe_zone) & is(the_level_of_sugar, rising) & is(the_rate_of_increase_of_sugar_level, increasing) \( \rightarrow \) deliver(??, insulin).
3) \( \text{is} (\text{the_readings, above_the_recommended_level}) \&
\left( \neg \text{is} (\text{the_level_of_sugar, falling}) \&
\text{is} (\text{the_rate_of_decrease_of_sugar_level, increasing}) \right) \rightarrow \text{deliver}(??, \text{insulin}). \)

“??” in the semiformal descriptions depicts flaws of the element, which are omitted in sentences of the specifications. Since the flaws are depicted as “??”, test engineers can easily detect flaws of the sentences in a document and inform “the_system” instead of “??” in the above example.

C. Visualizing logical relations

Then, the “proponet” visualizes the above semiformal descriptions as three independent propositional networks shown in Fig 7.

![Fig. 7. Propositional Networks for Insulin Pump](image)

The “proponet” also structures the propositional networks on the similarities of test conditions and/or expected actions. As a result, three independent propositional networks are integrated into two propositional networks (Fig. 8).

![Fig. 8. Propositional Network for Insulin Pump Merged on Similarities](image)

Since the test items are structured on the similarities of test conditions and expected actions by the “proponet”, test engineers can easily construct test cases even if sentences including same test conditions or expected actions are described in different paragraphs in disperse.

D. Creating Decision Tables on Semiformal Descriptions

The “splosat” generates decision tables (Table II, Table III, Table IV) on the propositional networks shown in Fig 8. At the second propositional network depicted in Fig 8, two test conditions are connected with the “Or(())” relationship. The “splosat” generates two test cases independently. If the conditions are connected with the “And(&)”, the “splosat” combines two test conditions and generates one test case.

![Decision Table for the First Test Item](image)

![Decision Table for the Second Test Item](image)

![Decision Table for the Third Test Item](image)

E. Results of Evaluation

Through the evaluation, the followings are confirmed:

1) Test cases were generated through the proposed process with the tools automatically

2) Generated test cases satisfied test items required for validating specifications described in Section IV-A

3) Dependencies of test case design to test engineers’ skills and experiences were minimized

4) Handiwork for generating test cases from the specifications written in a natural language was reduced

In this paper, we explained a tiny example composed of 3 sentences for the evaluation due to the limitation of pages. Besides the above evaluation, each tool was evaluated individually prior to this study. The “semiformalizer” was applied to three different specification documents (composed of 80 sentences, 41 sentences and 90 sentences) for evaluation. The accuracy are evaluated quantitatively
by comparing the results converted by the “semiformalizer” with the results of handiwork by test engineers. Thus, “semiformalizer” has succeeded in converting sentences into semiformal descriptions in 74% accuracy and also has achieved 82% accuracy in identifying logical relations among atomic primitives. The errors of conversion and identification are originated from lack of keywords, which are specifically appeared in the specification documents. Because, the “semiformalizer” assigned words in each sentence with aid of the dictionary, defining keywords. The accuracy will be improved by supplementing keywords in the dictionary.

The “proponet” was also applied to actual reviews by two engineer groups (total 11 engineers including novices and experts) belonging to two major consumer products manufacturers. Therefore, the results visualized by the “proponet” promoted detections of the following errors and flaws both for novices and experts test engineers:

- “??” promotes engineers in detecting flaws of the specifications
- Structuring atomic propositions promotes engineers in noticing omissions of logical relations among atomic primitives in specifications

The semiformal descriptions contributed to detect rhetoric defects in sentences and the propositional networks contributed to detect flaws in tacit, but, significant restrictions for the specifications.

V. Conclusion and Future Plan

Tests of the consumer products with network have obliged us to consume much time for designing test cases and conducting them. For reducing labor of designing test cases, the following algorithms and tools, which help test engineers to convert specifications written in a natural language into semiformal descriptions, and generates test cases as decision tables automatically, are proposed in the paper:

1) Each sentence in a document is converted into semiformal descriptions by the tool called “semiformalizer” instead of test engineer’s intensive reading of the document
2) Logical relations between semiformal descriptions are visualized by the tool “proponet” to promote detection of errors and to compensate flaws through the reviews
3) Test specifications written in semiformal descriptions are converted automatically into decision tables by the tool “splosat”.

All the tools were implemented based on the algorithms and applied to a tiny example for evaluation. As a result, the algorithms and tools have succeeded in generating test cases from documents in a natural language. The following enhancement of the tools are under development:

- To handle specifications including description in tabular form
- To supplement keywords and their usage, which are specifically appeared in specifications documents, into the dictionary

We are going to apply the test case generation tools and the testing environment to actual products in the consumer product manufacturers. Through the applications to real products, we would like to sophisticate the tools and environment and establish test suite for consumer products in the SPLE.

References