Generation of Patterns for Test Cases

V. Vani1*, G. S. Mahalakshmi2 and J. Betina Antony2

1Department of Information Technology, Easwari Engineering College, Chennai - 600089, Tamil Nadu, India; v.vani465@gmail.com
2Department of Computer Science and Engineering, College of Engineering Guindy, Anna University, Chennai - 600025, Tamil Nadu, India; gsmaha@annauniv.edu, betinaantony@gmail.com

Abstract

This work aims at generating all possible patterns of test cases for the given use case. Given a use case, generate all possible test cases that are mandatory for the given use case is actually the problem statement. For a particular use case, there can be hundreds and thousands of test cases. It can be even one to ten. This depends upon the programmer or developer who actually analyzes the use case while developing. Thus a programmer may miss some test cases that are mandatory and other could write unnecessary test case that has nothing to do with the use case. Thus this project aims at generating those test cases which are mandatory for the given use case and ensure that there cannot be more test cases than the generated one. The initial phase of the project involves Natural Language Processing and the later phases involve our own algorithm for generating these test case.

Keywords: Part-of-Speech Tagging and use Cases, String Matching, Test Case Generation

1. Introduction

Writing test cases is more complicated than developing. A test case, in software engineering, is a set of conditions to determine whether software, application, a system or one of its features is working as it was intended. A test oracle is the method of determining the success or failure of a test case. The test oracle may be a use case, a requirement or even a heuristic. Many test cases are usually involved to determine that a software program or system is considered sufficiently scrutinized for deployment. Test cases are often referred to as test scripts and are usually collected into test suites. One of the important phases in the software development cycle is developing and testing. Test cases have to develop before developing the software. If test cases are missed, then this will be pointed in the testing phase. The project aims to automate the generation of test cases.

Most of the early works of pattern generation concentrated on get test patterns for logic circuits. This commonly employed technology called Test Pattern Generation (TPG) was an important step in the process of automating test situations2,7 to validate the working condition of an assembled circuit. The concept of generation patterns for software test cases is still a work in progress. However, automated test cases are generated from various sources. Parameter based test case generation8 shows the importance of combinational design based approach over other state-of-art approaches. Another requirements-based approach to functional testing of product lines, based on a formal test generation tool outlines how product-specific test cases can be automatically generated from PF functional requirements expressed in UML4. In our work, we combine the concepts of pattern generation into software test cases by considering the requirement attributes as parameters.

*Author for correspondence
and finding the level of coverage of these cases on the attributes.

2. Module Description

2.1 Minimizing Description

The description of a given use case is in plain English with several statements. These statements have to be minimized because not every part of the statement is necessary for further phases. Only the part that helps in generating the test case is enough. The following steps are involved in minimizing the description. 1. Tokenization, 2. Part-of-speech tagging, 3. String Matching.

There are many intermediate steps. But these plays important role in minimization.

2.1.1 Tokenizer

In lexical analysis the process of breaking an array of text up into meaningful elements such as words, phrases or symbols called tokens is known as Tokenization. These tokens become input for further processing such as parsing or text mining. Tokenization is used in many linguistics applications such as text segmentation in lexical analysis. Tokenization in general is found at the word level. Often a tokenizer relies on simple heuristics such as: i) Punctuations or white spaces may separate characters, ii) Contiguous string of characters or numbers may constitute a token. Languages with inter-word spaces find this method of tokenization quite straight forward. However, they have few special cases such as contractions, hyphenated words, emoticons, and URIs which again may account to single words. A classic example is “Time Square-based”, which a naive tokenizer may break into “Time and Square-based” at the space even though the better break is at the hyphen (Time Square and based). Tokenization is particularly difficult for languages written in agglutinated form which exhibit no word boundaries such as Ancient Greek, Chinese or Thai.

2.1.2 POS Tagging

A Part-of-Speech Tagger (POS Tagger) is software that assigns grammatical part of each word (and other token). Some POS tags are noun, verb, adjective, etc. In general, computational applications use more fine-grained POS tags like ‘noun-plural’, ‘verb-phrase’ etc. In corpus linguistics, the purpose of Part-of-Speech tagging (POS tagging or POST) is to mark a word in a text (corpus) to a particular part of speech, based on its definition and the context in which it is mentioned. The relationship of a word with its adjacent and related words in a phrase, sentence, or paragraph can affect its part of speech. Sometimes POS Tagging is also called grammatical tagging or word-category disambiguation. POS-tagging algorithms fall into two distinctive groups: i) Rule-based approaches, ii) Stochastic methods. E. Brill’s tagger, one of the first and most widely used English POS-tagger, which employs rule-based algorithms. Some of the current state-of-arts algorithms for part-of-speech tagging include the Viterbi algorithm, Brill Tagger, Constraint Grammar, and the Baum-Welch algorithm (also known as the forward-backward algorithm). The Brill tagger however suffers defects as it learns a set of patterns, and then applies these rather than optimizing a statistical quantity. The most commonly used model now is the “Stanford POS Tagger”.

2.1.3 String Matching

String matching plays a great role everywhere. A typical example of string matching is “Find and Replace” in text file editors. Not just in text editors but every application
we come across. POS tagging is done to remove stop words, foreign words and similar words that don't have anything to do with the use case. The input text to string matching contains only the important words that have something to do with the test case. The input file (use case description) specifies the fields, basic flows and alternate flows. In a flow, once an action is done corresponding success flow and failure flow should be mentioned following the action. Thus string matching here helps to identify the field the current action description is about. Once the field is identified corresponding success and failure flows are extracted. The algorithm used for string matching is KMP (Knuth Morris Pratt) Pattern Searching algorithm.

2.2 Pattern Generation
The aim of this module is to generate all possible combinations of field values. By concept of combination, when there are N fields number of combination do not exceed 2 to the power N. Although the number of combinations does not exceed the maximum combination, it does not mean that the maximum number of possible test cases is 2 to the power N. Because there can be hundreds of alternate flow for a single test case.

For example, in an ATM use case for “Wrong Pin” one can handle the following alternate flow. 1. “Be in same page if number of tries is less than three”. 2. “Block the card if the number of tries exceeds three”. Handling such alternate flows is done in the upcoming phases.

2.3 Test Case Elimination
Not all the test cases generated so far is mandatory. Thus unnecessary test cases have to be removed. This removal is done in many ways. The first and simplest way of accomplishing this is based on “ranks”. The fields are specified in the order of rank in the input file. Lower the index higher is the rank of the field. If a field with higher rank has resulted in failure then don’t care about the field which has rank lower than that (i.e.) in a ATM use case if Card insertion action has resulted in failure, then do not care all the fields following the card-insertion (need ask for Pin). So such tests will definitely there be in the pattern. This has to be removed. Once removal is done with this method next phase of removal removes all the unnecessary test cases. The methodology is “include a pattern into solution set if the set has all VALID before the first INVALID”. Once these two removals are done there will be only mandatory test cases that are needed for the use case.

3. Sample use Case Scenario: ATM use Case
ATM plays a great role in today environment. Different people can write different number of test cases for a single use case. An ATM use case can be built just single test case i.e. if all of the fields is success give the money else do not give. But this is not sufficient because it does not handle all the flows. There a question might arise from the bank customer “What if I have entered the pin wrong carelessly?” These kinds of questions can be answered only with test cases i.e., the answer that the customer expects is that the ATM should alert that he had entered wrong Pin. This can be done by handling the case by writing test cases in prior. The entire scenario is discussed with some fields that are the key to ATM. Let us limit the number of fields. The fields are: 1. “Card-Insertion”, 2. “Pin”, 3. “Amount”, 4. “Confirm”. There are four fields. Therefore there can be sixteen combinations. Out of these sixteen combinations five combinations might have more than one alternate flow. So the number of test cases increases. So whatever may my value for the field, the only value it produces is either “VALID” or “INVALID”. These combinations of “VALID” and “INVALID” represents the test cases. Example test case 1: If all fields are “VALID” then dispense the cash. Example test case 2: If Pin is wrong; either blocks the card or asks the Pin again depending upon the number of tries. Example Test case 3: If Amount entered is not multiple of Hundred then alert that message. Example Test case 4: If Amount entered is greater than the maximum withdrawal amount then alert a message saying amount exceeded maximum amount of withdraw. Thus the number of test cases goes on increasing.

4. Result and Evaluation
The project has many different outputs on many phases. But here only the important phase’s outputs are discussed. The phase which gives all possible test cases and the final output that gives the mandatory test cases that are actually important for the use case (Figure 2).

Above image (Figure 3) contains all possible patterns for a given use case. The given use case contains four fields and so there are sixteen combinations.
Figure 2. Input use case.

Figure 3. All possible patterns.

Figure 4. Patterns after elimination by set difference.

Figure 5. Patterns after handling alternate flows.

4.1 Coverage Analysis

Coverage shows the extent to which a test case covers the attributes in the given use case field. The same process is carried on for all the 20 input sequences present in the data set to obtain the mean coverage of all sequences, which is shown on the left. The consolidated list of mean
coverage values shows that the values of coverage lie predominantly between 0.4 and 0.6 (Figure 5).

Test case evaluation is done using pattern elimination comparison which is a comparison with another set of test cases. A self-evaluating test case generating methodology will serve the purpose better and still remains an unexplored research topic for which this work may serve as a basis.

5. Future Work

The sole purpose of this project is to generate all the test cases that are actually mandatory. This suits well if the use case description is simple that is constraints are imposed on the use case description. The project could be further developed to work for descriptions that are complex. This could be done if more of Natural Language Processing is involved.

6. References

1. Brill E. A simple rule-based part of speech tagger. Proceedings of the third conference on Applied natural language processing (ANLC ’92). Stroudsburg, PA, USA: Association for Computational Linguistics; 1992. p. 152–5.
2. Korel B. Automated software test data generation. IEEE Trans Software Eng; 1990. p. 870–9.
3. Memon AM, Pollack ME, Soffa ML. Hierarchical GUI test case generation using automated planning. IEEE Transactions on 27.2 Software Engineering; 2001. p. 144–55.
4. Nebut C, et al. Automated requirements-based generation of test cases for product families. Proceedings 18th IEEE International Conference on Automated Software Engineering; 2003.
5. Part-of-Speech, Stanford Log-linear. Tagger; 2009.
6. Rayadurgam S, Heimdahl MPE. Coverage based test-case generation using model checkers. Proceedings 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems; 2001.
7. Schulz MH, Elisabeth A. Advanced automatic test pattern generation and redundancy identification techniques. IEEE 18th International Symposium on Fault-Tolerant Computing. Digest of Papers; 1988.
8. Tung Y-W, Aldiwan WS. Automating test case generation for the new generation mission software system. IEEE Aerospace Conference Proceedings; 2000. p. 1.

Appendix

Table 1. Sequence 1 - Test cases

| # | Valid card | Pin | Amount | Transaction | Result |
|---|-----------|-----|--------|-------------|--------|
| 1 | Failure   | XXX | XXX    | XXX         | Rollback |
| 2 | Success   | Failure | XXX | XXX | Pin incorrect-Redirect(If tries<3) |
| 3 | Success   | Failure | XXX | XXX | Pin incorrect-Block Card(If tries=3) |
| 4 | Success   | Success | Failure | XXX | Insufficient Amount |
| 5 | Success   | Success | Failure | XXX | Amount<100 Minimum 100 |
| 6 | Success   | Success | Failure | XXX | Amount>25000 Maximum 25000 |
| 7 | Success   | Success | Success | Failure | Transaction Failed |

Table 2. Sequence 2 - Test cases

| Machine status | Valid | Card | Enter amount | Reenter amount | Enter pin | Mode of transaction | Withdrawal | Retry pin 2 | Retry pin 3 | Result |
|----------------|-------|------|--------------|----------------|-----------|---------------------|------------|-------------|-------------|--------|
| 1              | Success | Success | XXX | Success | Success | Success | XXX | XXX | Successful |
| 2              | Success | Success | XXX | Failure | Success | Success | Success | XXX | Successful |
| 3              | Success | Success | XXX | Failure | Success | Success | Failure | Success | Successful |
| 4              | Success | Success | Failure | Success | XXX | XXX | XXX | XXX | Successful |
| 5              | Failure | XXX | XXX | XXX | XXX | XXX | XXX | XXX | Unsuccessful |
Table 3. Sequence 3 - Test cases

| card-insertion | PIN-number1 | PIN-number2 | PIN-number3 | amount-entered-correct | amount-reentered-correct | transaction-confirmation | Result         |
|----------------|-------------|-------------|-------------|------------------------|-------------------------|--------------------------|----------------|
| 1              | Success     | Success     | XXX         | XXX                    | Success                 | XXX                      | Successful     |
| 2              | Success     | Success     | Success     | XXX                    | Success                 | XXX                      | Successful     |
| 3              | Success     | Failure     | Failure     | Success                | Success                 | XXX                      | Successful     |
| 4              | Success     | Failure     | Failure     | XXX                    | XXX                    | XXX                      | Unsuccessful   |
| 5              | Success     | Success     | XXX         | XXX                    | Failure                 | XXX                      | Unsuccessful   |
| 6              | Success     | XXX         | XXX         | Failure                | Success                 | Successful               | Unsuccessful   |

Table 4. Test case coverage for sequence 1

| Test Case | Coverage |
|-----------|----------|
| TC1       | 4/5      |
| TC2       | 1/5      |
| TC3       | 3/5      |
| TC4       | 3/5      |
| TC5       | 4/5      |
| TC6       | 2/5      |
| TC7       | 5/5      |
| Mean coverage | 0.53 |

Table 5. Test case coverage for sequence 2

| Test Case | Coverage |
|-----------|----------|
| TC1       | 6/9      |
| TC2       | 7/9      |
| TC3       | 8/9      |
| TC4       | 4/9      |
| TC5       | 1/9      |
| TC6       | 2/9      |
| TC7       | 5/9      |

Table 6. Test case coverage for sequence 3

| Test Case | Coverage |
|-----------|----------|
| TC1       | 4/7      |
| TC2       | 5/7      |
| TC3       | 6/7      |
| TC4       | 4/7      |
| TC5       | 4/7      |
| TC6       | 5/7      |
| Mean coverage | 0.57 |