International Journal of Current Engineering and Technology
E-ISSN 2277 – 4106, P-ISSN 2347 - 5161
©2014 INPRESSCO®, All Rights Reserved
Available at http://inpressco.com/category/ijcet

Research Article

Generating and Analyzing Test cases from Software Requirements using NLP and Hadoop

Priyanka Kulkarni[Ȧ*] and Yashada Joglekar[Ȧ]

[Ȧ]Department of Information Technology, Dwarkadas J Sanghvi College of Engineering, Vile Parle(W), Mumbai, India

Accepted 10 Nov 2014, Available online 01 Dec 2014, **Vol.4, No.6 (Dec 2014)**

Abstract

Software testing is the most critical step of software development since it ensures that the system under developments free of errors and unprecedented faults and matches the expectation and requirements elicited from users and stakeholders. However, the process of testing is currently a manual process and is thus prone to mistakes by human testers and time-consuming and arduous. This paper proposes the automation of the task of generating test cases from software requirements written in natural language. This solves the problems of human errors and requirement of manual effort in ensuring coverage of requirements specified during requirements elicitation. It also enables test cases to be generated early on in the software development lifecycle based on requirements documents. The method we propose involves taking software requirements expressed in natural language as input and processing them using natural language processing techniques such as POS tagging and parsing. These NLP constructs are used to represent the requirements in the form of tree structures, which are used to generate knowledge graphs that depict the essential flow of the system. These paths can be traversed using methods such as boundary value analysis, etc. to obtain a suite of test cases.

Keywords: *Natural Language Processing, Knowledge representation, Hadoop, Software testing, Software engineering, Test cases.*

1. Introduction

Testing is a salient step of software development, which is crucial to ensuring the programmed unit works under normal circumstances and to weed out faults prior to deployment. It is the cornerstone of verifying and validating that the expectations and requirements of the users and stakeholders have been met in the system under development. The costs associated with testing involve cost of creating, designing, maintaining and executing and documenting test cases. Minimizing these costs is an important goal for developers. 40% - 70% of the total effort expended is consumed by test case generation. Furthermore, if corrections are needed, the cost incurred increases further (B. Beizer, 1990). Traditional methods of test case generation have moved from manual to automated versions utilizing several models such as sequence diagrams, use case diagrams, activity diagrams, etc. However, these diagrams do not comprehensively and explicitly express all the requirements and also require manual effort themselves. Moreover, they may not best express the vital non-functional requirements of stakeholders. The software requirements specification is the best source for understanding stakeholders' expectations and generating the tests corresponding to the same. However, 79% of level of terminology used in requirements documents is common natural language (L. Mich, M. Franch and P. Novi Inverardi, 2003). So there is a need to transform these requirements into computer-readable format in order to automate the process generate test cases. Natural language processing techniques enable us to morph sentences expressed in natural language into statements that can be understood syntactically and semantically and processed accordingly by a machine.

The system proposed for automating the generation of test cases from software requirements specification using natural language processing outlines the basic process to obtain the test cases, but it is apt for simple or basic requirements (Ravi Prakash Verma, Dr. (Prof.) Md. Rizwan Beg, 2013). The system falters when the requirements become larger and more complex since the knowledge graphs grow into a convoluted network and require too much space and is not efficient enough for testing large systems. It also leads to more computation time and wastage of resources as the complexity grows. We propose an extended approach in this appear which expands on a system proposed to generate test cases from software requirements by augmenting that method with the utilization of Hadoop for storing the knowledge graphs generated and querying the same for subsequent analysis (Lam, Chuck, 2010).

2. Existing Systems

Currently varied approaches are used to generate test cases. Model-based testing (MBT) produces models of a system under test in order to derive test cases. This test

*Corresponding author: **Priyanka Kulkarni**

model depicts the expected behavior or functional aspect of the system. The model is usually manually designed from formal specifications or semiformal design descriptions. The generation of test cases can be from an environmental or behavioral model. This method involves generating abstract tests from the model, concretizing them for execution, executing the tests on the system under test and analyzing the results. However, Model-based testing is insufficient as a solution if used in isolation (Annamariale Chandran, 2011). UML diagrams are commonly used in Model-based testing. A popular approach involves a UML sequence diagram describing the interaction of the components of the system. The sequence diagram forms the basis of the construction of a sequence dependency table, which is further transformed into a graph called the sequence diagram graph. This graph is then traversed using Depth-First Search to derive the test paths (S. Shanmuga Priya, P. D. Sheba Kezia Malarchelvi, 2013).

Other approaches utilize use case diagram, class diagram and sequence diagram in conjunction with constrained language expressions that describe use case templates to generate a sequence diagram graph.

The input and expected output of various scenarios are determined by using the information stored in the nodes of the graph and the OCL expressions to generate the test cases (Vinaya Sawant, Ketan Shah, 2011). Activity diagrams can also similarly be used for test case generation. They can also be used for gray-box testing to generate test cases from high-level design models and considering path coverage criteria such as basic path (W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang, 2004), and simple path (C. Mingsong, Q. Xiaokang, and L. Xuandong, 2006). Other proposed solutions are based on a criterion called activity path coverage criterion, which also ensures minimal loop testing and uncovers synchronization faults (Debasish Kundu and Debasis Samanta, 2009).

Limitations in the existing systems

- Need of skilled model designers for the creation of the abstract and design models
- The gray-box testing method does not handle fork-join efficiently and this limits the scope of the technique.
- Dependency on manual creation of UML diagrams such as sequence diagrams.
- Incorrect test cases may be generated if outdated requirements lead to building of the wrong model.
- Test case generation from UML diagrams is time consuming and fails to capture non-functional requirements.
- The process of generating the sequence diagrams, sequence tables and sequence graphs is lengthy and slow.
- There is no mechanism to map the test cases to the requirements and verify their correctness which may lead to unprecedented faults and increased costs incurred for correction and rework.

3. Proposed Method

The sequence of steps in the proposed solution is depicted in Figure 1.

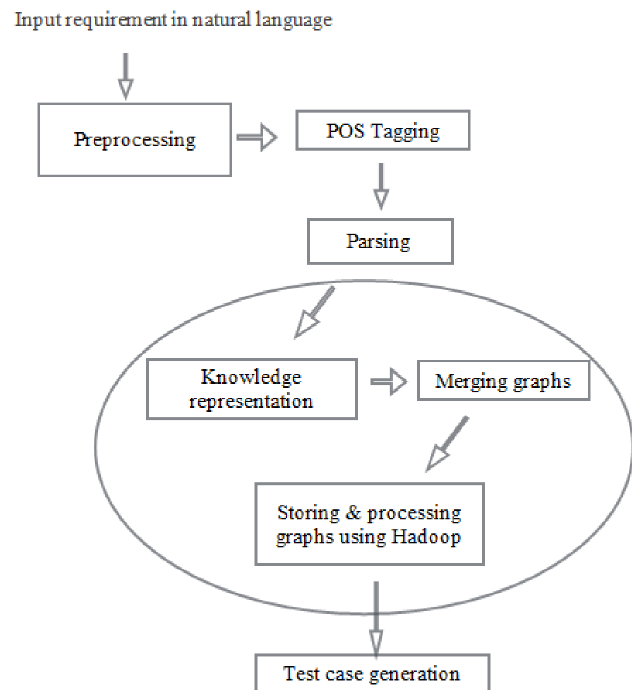

Fig. 1 Proposed Architecture

To illustrate the process, the following statement is selected as a requirement for an ATM system and taken under consideration:

The balance in the account is insufficient if the balance is less than the amount to withdraw.

3.1. Preprocessing

In order to process the requirements, we need to convert the natural language sentences into appropriate forms.
This includes normalization of the text by removing unwanted words, performing stemming etc. If a statement has unnecessary or irrelevant words such as articles, then they are removed, otherwise the POS tagger will treat them as words in the corpus that need to be determined. Considering the example given, the requirement can be rephrased and preprocessed to obtain the following:

Balance is insufficient if balance < withdrawal-amount.

3.2. POS Tagging

In this step, each word in a sentence is assigned a part-of-speech tag, indicating whether it is a noun, verb or adjective etc. By assigning unique tags to each word such as 'VBZ' for verbs, 'NN' for nouns and 'CD' for cardinal numbers, we can categorize the words in the sentence into meaningful groups. These tags are useful in subsequent word sense disambiguation and parsing. POS tagging helps to resolve lexical ambiguity so that each word is correctly assigned its correct semantic connotation.

POS-tagging the requirement stated above results in:

Balance -> NN, is -> VBZ, insufficient -> JJ, if -> IN, etc.

3.3. Parsing

In this step, a sentence is given as input to a natural language parser, which determines the labeled syntactic tree structure that corresponds to the interpretation of the sentence. This step results in a tree structure representing the hierarchy and order of the parsed words. The parse tree obtained for the given example is shown in Figure 2. The steps of pre-processing, POS tagging and parsing can be performed using tools such as Apache OpenNLP based on the maximum entropy framework (A. Ratna Parkhi, 1998) or Stanford's CoreNLP.

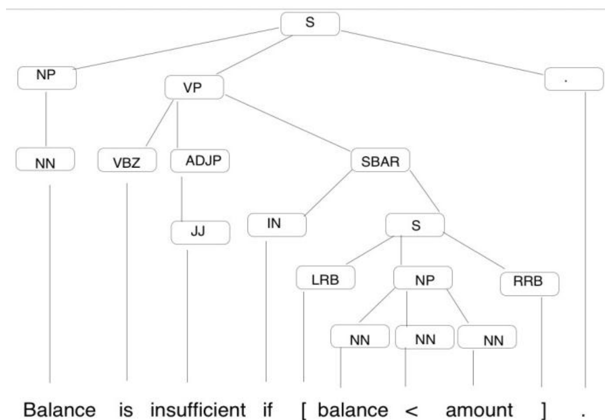

Fig. 2 Balance is insufficient if [balance < amount]

3.4. *Knowledge Representation*

Knowledge is represented from requirements expressed in natural languages using graph-based approach in which the nodes represent nouns and verbs and interjections represent the transactions. The transactions are labeled with conditions from subordinate clauses. The preposition tags and verb will decide the direction of transition. The graph is constructed according to a set of stipulated rules to govern how the nodes are interconnected and the type of transitions between them.

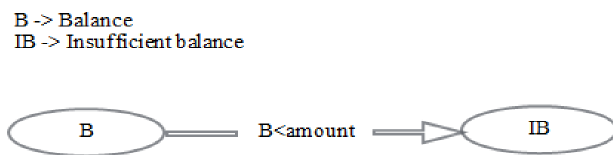

Fig. 3 Knowledge graph generated

3.5. Merging graphs

The sub graphs obtained in the previous step are merged to obtain a single master knowledge graph encompassing each node. The final graph is constructed by merging graphs whose start and end nodes are same and including the intermediate nodes. The transitions labels are concatenated and remaining redundant nodes are eliminated. The final merged graph is stored and subsequently used for traversal to generate the test cases. This merged graph is useful in checking for incorrect or inconsistent requirements. If the requirements are missing or inconsistent, the graph generated will also be incomplete or incorrect, providing a mechanism to detect such flaws in the early stages of the software development lifecycle.

3.6. Storing and Processing Knowledge Graphs using Hadoop

As the software requirements specified become increasingly more complex and interconnected in large systems, the knowledge graphs developed will be convoluted networks of linked data (Jiewen Huang, Daniel J. Abadi, Kun Ren, 2011). As graph data is proliferating, it has tremendous value if analyzed and processed properly leading to tools such as Hadoop becoming increasingly popular.

Hadoop has emerged as the de facto standard for data processing on a large scale. We can improve the efficacy of this automation process by using Hadoop to store the knowledge graphs generated. Since the graphs for large, complex systems can be very convoluted with millions of nodes and edges, traversal of all edges in the graphs and subsequent generation of the test cases can be cumbersome and time-consuming. The power of Hadoop is its rapid processing power at low cost.

By using Hadoop to store the graphs, we can speed up the test case generation process by exploiting its ability to efficiently process large data sets in a distributed environment. This will significantly reduce the time required to generate the test cases, thereby improving the efficiency of this method. Hadoop can further be used at a later stage to compute graph derivations, statistics and data mappings when we want to filter out subsets of specific-scenario test cases for analysis and processing. In order to extract specific test case information from the graphs, we can utilize query languages such as SPARQL to query the linked data. This is akin to sub graph pattern matching which allows you to retrieve specific data from nodes and relationships among them using queries. Another advantage is working with MapReduce jobs for sub graph pattern matching, which performs better than a naive Hadoop solution and achieves higher scalability than other parallelization models. MapReduce query languages like Hive and Pig can be used to query the graphs and help to optimize the computation time for analysis (R.J. Stewart, P.W. Trinder, and H-W. Loidl, 2011). Furthermore, graph query languages like Gremlin empower analysis of graphs, which help in better understanding the test cases and correlation between the nodes of the graphs.

3.7. Test Case Generation

We use boundary value analysis for the automatic generation of test data. The headword is identified, the path the headword is traversed and visited nodes are saved. A table is formulated based on nodes and whether conditions are satisfied or not. Expected result based on conditions is then added to the table. The final table obtained represents the expected result for each path traversal depending on the input values and conditions.

Following table depicts the test case generated for the above said example.

Table 1 Sample Test Case Generated

B	B<amount	IB	Expected Output
0	1	1	IB
1000	0	0	!(IB)

4. Advantages of Proposed System

Implementation of this system to automate generation of test cases is beneficial in the following ways:
1. Automation leads to fast, precise and comprehensive generation of test cases.
2. Time expended in testing is reduced as compared to manual testing.
3. Incorporating Hadoop enables you to perform computations efficiently on not just elementary requirements, but larger and more complex requirements.
4. It ensures that the program paths are exercised adequately without manual checking of path selection criteria.
5. It provides a mechanism to detect incomplete or inconsistent requirements early on in the software development lifecycle.
6. It minimizes cost of correction of the requirements that is incurred since the discrepancies are identified and resolved in the initial stages of software development.

Conclusion

The proposed system will be effective for automating the generation of test cases as well as for studying and analyzing the correctness and completeness of requirements. Inconsistent, incomplete or ambiguous requirements can be detected through this method since the knowledge graphs will reflect the same incongruity.
Using Hadoop to store the knowledge graphs will enable this method to be effective even as scalability of the system under test increases, since it can handle and process complex requirements efficiently and minimize computation time. Further analysis of the knowledge graphs and test cases is possible using graph query languages.

References

Ravi Prakash Verma, Dr. (Prof.) Md. Rizwan Beg (2013), Generation of Test Cases from Software Requirements Using Natural Language Processing, *6th International Conference on Emerging Trends in Engineering and Technology*.

Lam, Chuck (July 28, 2010). Hadoop in Action (1st ed.). *Manning Publications*.p. 325.ISBN1-935182-19-6.

Ratna Parkhi, A. (1998), Maximum Entropy Models for Natural Language Ambiguity Resolution.Ph.D. thesis, University of Pennsylvania.

Jiewen Huang, Daniel J. Abadi, Kun Ren (2011), Scalable SPARQL Querying of Large RDF Graphs.

R.J. Stewart, P.W. Trinder, and H-W. Loidl (2011), Comparing High Level MapReduce Query Languages, Mathematical And Computer Sciences Heriot Watt University.

Debasish Kundu and Debasis Samanta (2009), Novel Approach to Generate Test Cases from UML Activity Diagrams, ETH Zurich, Chair of Software Engineering, Indian Institute of Technology, Kharagpur.

B. Beizer (1990), Software Testing Techniques, Van Nostrand Reinhold, Inc, New York NY, 2nd edition. ISBN 0-442-20672-0.

Mich, L., Franch, M. and Novi Inverardi, P. (2003), Requirements Analysis using Linguistic Tools: Results of an On-line Survey, *Requirements Engineering Journal*, Technical Report 66, Department of Computer and Management Sciences, University of Trento, Italy.

Annamariale Chandran (2011), Model Based Testing - Executable State Diagrams, AVACorp Technology, Step-Auto.

S. Shanmuga Priya, P. D. Sheba Kezia Malarchelvi (2013), Test Path Generation Using Uml Sequence Diagram, *International Journal of Advanced Research in Computer Science and Software Engineering*.

Vinaya Sawant, Ketan Shah (2011), Automatic Generation of Test Cases from UML Models, *International Conference on Technology Systems and Management (ICTSM).*

W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang (2004), Generating test cases from UML activity diagram based on gray-box method, *11th Asia-Pacific Software Engineering Conference (APSEC04), pp. 284-29.*

C. Mingsong, Q. Xiaokang, and L. Xuandong (2006), Automatic test case generation for UML activity diagrams, *International workshop on Automation of software test, pp. 2-8.*