Generation of Test Cases from Software Requirements Using Natural Language Processing

Ravi Prakash Verma
Department of Computer Science & Engineering
Integral University
Lucknow, India
raviprakashverma@gmail.com

Dr. (Prof.) Md. Rizwan Beg
Department of Computer Science & Engineering
Integral University
Lucknow, India
rizwanbeg@gmail.com

***Abstract*—Software testing plays an important role in early verification of software systems and it enforces quality in the system under development. One of the challenging tasks in the software testing is generation of software test cases. There are many existing approaches to generate test cases like using uses case, activity diagrams and sequence diagrams; they have their own limitations such as inability to capture test cases for non functional requirements and etc. Thus these techniques have restricted use in acceptance testing and are not effective for verification & acceptance of large software system. If software requirements are stated using semi-formal or formal methods then it is difficult for the testers and other third party domain experts to test the system. It also requires much expertise in interpreting requirements and only limited number of persons can understand them. This paper proposes an approach to generate test case from software requirements expressed in natural language using natural language processing technique.**

Keywords— Knowledge acquisition Knowledge representation, Natural language processing, Software engineering, Software testing

I. INTRODUCTION

OUT of six stage of software test life cycle, test case generation (TCG) is the one of most of difficult stages, consuming 40% to 70 % of the total effort and if compromised then the cost of corrections goes higher [1]. TCG is usually done manually; this is error prone, time consuming and challenging activity [2], [4]. Testing the system early or planning test cases in requirement phase is beneficial [5] in following terms 1) it helps designers & analyst to better understand & express requirements, 2) testability of requirements is ensured, 3) reduces verification and validation cost and 4) avoids misinterpretation of requirements.

If the requirements are expressed using semi-formal or formal techniques then it is difficult for the testers and other third party domain experts to test the system as it requires much expertise in interpreting requirements and only limited number of persons can understand them [16]. Writing requirement in a formal way removes ambiguity and for that there are many techniques such as UML [9], Finite state machine, and other abstract methods etc exist, however they do not promise to solve other problems in requirement engineering. For example, take UML, which fails to capture non-functional requirements. Thus these techniques have restricted use in acceptance testing and are not effective for verification and acceptance of large software system. The most used methods to document requirements is use of natural languages such as English [12] because requirements are easy to specify, verify, use, understand and are universally accepted. This is the reason that majority of requirements specifications are still written in natural language despite the fact that natural language specifications can be ambiguous, incomplete and contradictory [13], [14]. Around 87.7% of the documents are written in natural language and 5.3% of the documents are written in formal languages, while rest is written using other techniques [15].

This paper proposes an approach to generate test case from software requirements expressed in natural language using natural language processing techniques. First, we preprocess the requirements, tag part-of-speech (pos), parse them using parsers that are trained for processing general-purpose natural language text, each parsed tree is converted to knowledge representation graph and these graphs are combined to generate a single graph representing knowledge conveyed by requirement. Finally, we present the method to generate test case.

II. PROPOSED METHOD

Let us consider the requirement “The program takes three integer-values from interval [0,100] and finds whether equation is quadratic and if equation is quadratic then find whether roots are real or imaginary or equal.” [3] The technical or domain specific terms present in requirements are, described in a dictionary or glossary of project and for this particular requirement, we have following description.

1. Program takes integer values from the interval [0, 100].
2. Equation is quadratic if a > 0.
3. Quadratic equation has roots.
4. Discriminant is [b*b] – [4*a*c] substituted as k
5. Quadratic equation have real roots if Discriminant> 0.
6. Quadratic equation have imaginary roots if Discriminant< 0.

7. Quadratic equation have equal roots if Discriminant = 0.

A. Preprocessing

It is normalization of the text [26] includes throwing of unwanted words and doing stemming etc. However, this is domain dependent process, dependent upon the parsing algorithm's ability to recognize the sentence tags in desired format and manual rules are needed to suit ones need. In our case, we concatenate the two words "integer values" to "integer-values", the interval is normally written as [0,100] but we separate "[" and "0" by putting white space. We also split compound sentences to simple sentences. If a statement has the "A" or "a" then this is removed from the sentences and if they appear in a formula then a suitable substitution should be insert at their places or else the POS tagger will treat them as determiner (DT tag). The final requirement statements are as follows

1. Program takes three integer-values from interval [0,100]
2. Program finds whether equation is quadratic.
3. Program finds whether roots of quadratic equation are real or imaginary or equal
4. Quadratic equation has roots.
5. Quadratic equation have real roots if $k > 0$.
6. Quadratic equation have imaginary roots if $k < 0$.
7. Quadratic equation have equal roots if $k = 0$.

B. POS Tagging

It is assignment of part-of-speech to each word in a sentence, useful in information retrieval, word sense disambiguation and it helps in parsing by assigning unique tags to each word thus reducing number of passes [18]. We use a POS tagger that uses a model trained on English data from the Wall Street Journal and the Brown corpus [6], [20] as "The latest model created achieved more that 96% accuracy on unseen data" [20].

C. Parsing

"A natural language parser takes a sentence as input and determines the labeled syntactic tree structure that corresponds to the interpretation of the sentence" [17]. We use shift reduce maximum entropy [21] style parser based on Adwait Ratnaparki's 1998 thesis [17] with the help of tool developed by Richard Northedge [20] based on Penn Treebank [7], [19] to generate the parse trees of each statements and generate parse trees as shown in Fig. 1, 2, 3, 4, 5, 6, 7.

D. Knowledge Representation

To capture very detailed information & knowledge and to represent it there are approaches like OWL, RDF [23], [24] etc. A lightweight graph based approach is sufficient to capture knowledge represented by requirement expressed in natural languages. Where the nouns are represented as nodes and verbs & interjections represent the transactions. The transactions are labeled with conditions from subordinate clauses. The preposition tags and verb will decide the direction of transition for example "take" and "from" is labeled as VBZ so the main noun node will have inward transaction. The rules for constructing graph are as follows.

Start with those statements, which describe the requirements.

1. Start with the deepest S tag.
2. Find -LRB- and corresponding -RRB- tags, concatenate all leaves in between include -LRB- and -RRB- tags leaves.
   a. After concatenation remove all tags, add NP-NN tag, and make concatenated leave as a child of NN tag.
   b. Make NP tag as a child of parent of -LRB- tag.
3. For ADJP - JJ tag we find the previous NN tag and concatenate the leave of JJ with leave of NN tag.
4. Each NN tagged leave forms a node.
5. Each NNS tagged leave forms a node.
6. Each NNP tagged leave forms a node.
7. Nods are connected with a transition; the transitions are labeled with IN tagged leaves if we have NNP-VBP-NS or NNP-VBZ-NS or any model form of verb tags.
8. Repeat the same steps for next higher level of S tag.

For the main requirement

1. Start with the deepest S tag.
2. Find -LRB- and corresponding -RRB- tags, concatenate all leaves in between include -LRB- and -RRB- tags leaves.
   a. After concatenation remove all tags, add NP-NN tag, and make concatenated leave as a child of NN tag.
   b. Make NP tag as a child of parent of -LRB- tag.
3. If we have two consecutive NN tags then tagged leaves are concatenated.
4. For NN-VBZ-JJ tags, concatenate NN tagged leave with JJ tagged leave and rename ADJP-JJ tags as NP-NN tags respectively.
5. Form nodes labeled NN tagged leave and connect next NN tagged leave with a transition labeled with tagged leave of VBZ.
6. If a CD-NNS or CD-NN tagged leaves, the CD goes a label in transaction. If a transaction already has a label then CD, tagged leave is concatenated with the previous label of transaction.
7. If VBZ is tagged with leave "find" or "search" or "get" then a question mark is inserted as a label in transaction.
8. If VBZ-IN is tagged with leave "find" or "search" or "get" then a question mark is inserted as a label in transaction and IN tag is ignored.
9. Repeat the same steps for next higher level of S tag.

The application of aforementioned rules on the statements generates sub graphs see figure 8.

E. Merging graphs

The sub graphs obtained in figure 8 are merged with simple rule as follows if start and end nodes are same and there is intermediate stage then intermediate nodes are included. The transitions labels are concatenate. Where as if a sub graph ends with a node and another sub graph, starts with same node then two sub graphs are connected and redundant nodes are eliminated. These yields result shown in figure 9.

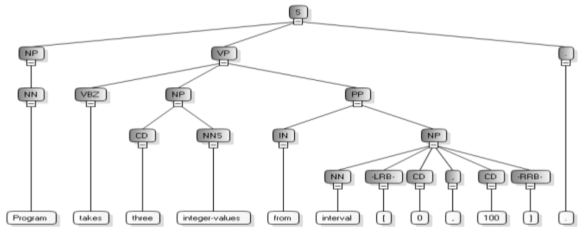

Figure 1. Program takes three integer-values from interval [0, 100].

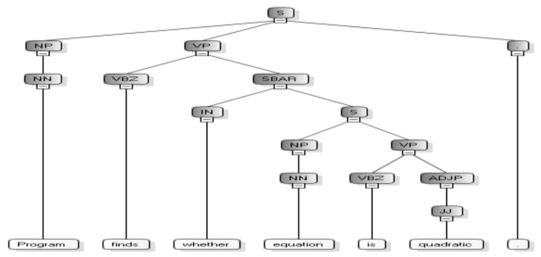

Figure 2. Program finds whether equation is quadratic.

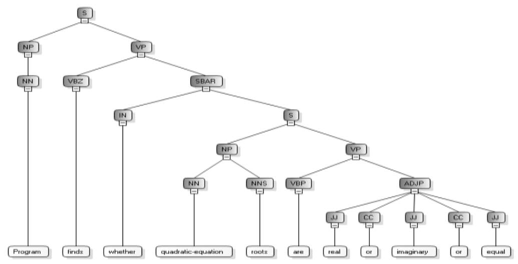

Figure 3.Program finds whether quadratic-equation roots are real or imaginary or equal

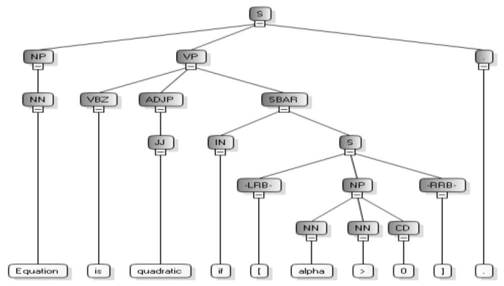

Figure 4. Equation is quadratic if [alpha > 0]

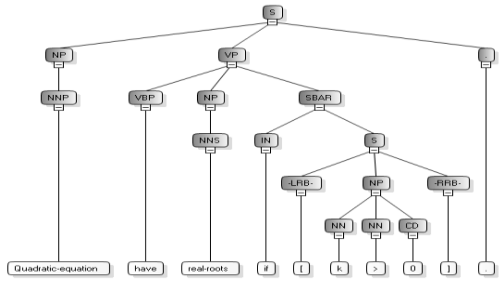

Figure 5. Quadratic-Equation have real roots if [k > 0]

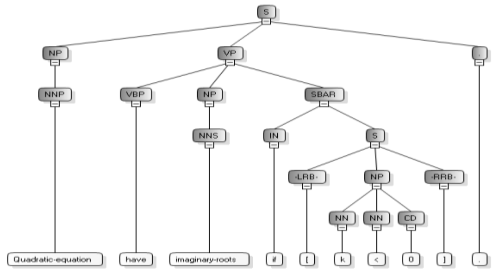

Figure 6. Quadratic-Equation have imaginary-roots if [k < 0].

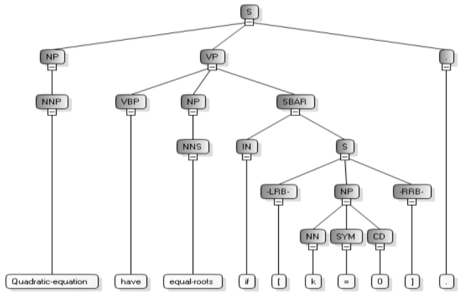

Figure 7. Quadratic-Equation have equal roots if [k = 0].

F. Test case generation

For the automatic generation of test data we use boundary value analysis and combine it with our previous work [31], [32] to generate all input combinations. Let the graph in figure 9 be the set G = {V, E} consisting of set of vertices V as node and arcs as edges E. Start node is the node where there are all out bound edges and end node is the node where we have all inbound edges.

The algorithm for generation of test cases is as follows.

```
FindHeadword ()
{ s = find a node that has all outbound edges only;
  array[i] = all the edges of s; $0 \le i \le n$, where n is number of out bound
  edges of s
  i = 0
```

```
 j = 0
 while (i ≤ n)
 {
   e = array[i]
   while (s ≠ "P")
    { // arrayPreconditions stores the nodes traversed forming preconditions
       arrayPreconditions [j] = s
      // defines a new node s1 after traversing the edge e from node s
      $s_1 \rightarrow_e s$
      j = j + 1
      }
    i = i + 1
  }
}
```

// table is an auxiliary double dimension data structure capable of expanding row and column wise

```
BuildTable()
{ // s is start node and p headword found
   s = p
   // traverse and find arc labeled as "?", symbolizing the action taken by "P"
   array[i] = all the edges of s; 0 ≤ i ≤ n, where n is number of out bound edges of s
while (i ≤ n)
  {
     Add the data value header in the table obtained
     label = Label(array[i])
     if (label == "?")
     { Add column to the table and name the header as value of
       node reachable from node p if not present in the table
      }
  }
 Add additional column of expected result in the table.
}
```

```
TestCaseGeneration()
{ populate the data values row wise
   Select a row data value and traverse the graph starting from s
   For every conditions on labeled transition starting from s
   until we reach end node
    for (i = 0; i ≤ n ; i++)
       { s = p
         for (j = 0; j < m ; j++)
             {
              if (condition == true)
               {
                 table [i][j] = 1
             $s \rightarrow_e s$
                 table [i][m] = table [i][m] + s
             }
           else
               { table [i][j] = 0
             table [i][m] = table [i][m] + "!" + s
            }
          }
        }
 }
```

The test case generation starts with finding "headword" from the main requirement. The headword over here is "P" or "Program". We traverse the graph to find this headword. While traversing the graph all the nodes found before headword serves as the precondition for the tests. After which we traverse the graph and locate "?" symbolizing the action taken by "P" and from a table like table 1. Whenever we find a node or a labeled transition we add column to the table. Finally we add additional column of expected result.

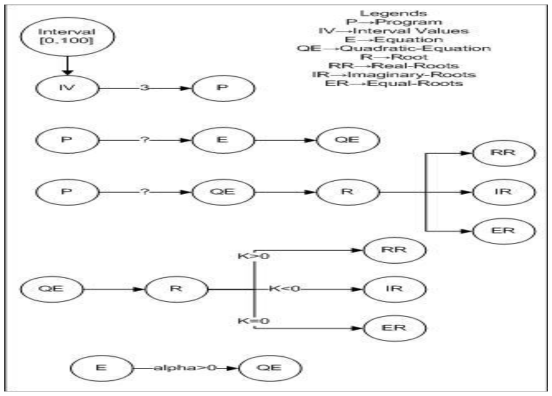

Figure 8. Representing graph for each parse tree.

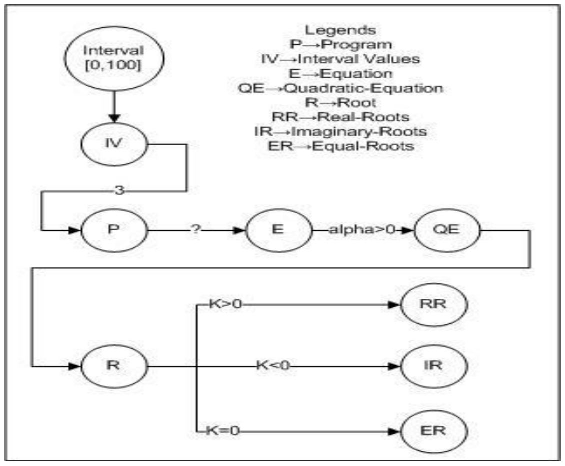

Figure 9. Merged graph

Now we again traverse the graph (see figure 9) starting from the headword. For every condition on labeled transition, we check whether the condition is true or false. If we find a condition false on the particular data (row wise, see table 1) then we fill a cells of the row as “0” or else we fill that condition “1”. If the first node after the headword cannot be reached then fill all cells of the row as “0” and move to next row. If we reach the leaf of graph or end state, then we fill the last column of the table 1 with node encountered whereas, if we are in state from where no transitions are possible then we fill the last state traversed and if it’s cell value is “0” then the state node labeled is negated. All cells, which are left, are filled with zeros.

III. Results and analysis

We have applied our method on the requirement, which includes a mathematical formula. The objective of choosing this particular type of requirement was to demonstrate that currently developed POS tagging models successfully handles simple mathematical formulas, our methods demonstrates a more complex situation in natural language processing and a successful test case generation. The statements were classified into type S → NP | VP with main noun and verb phrase.

Our method is substantial improvement over other methods as it removes the restriction of writing requirements using simple sentences only. We have also tried the same statement with different ways and found that if several graphs cannot be merge to form single graph pertaining to a requirement then this means that either we have missed some key terms or we have not understood them clearly or possibly have not written them well. The test case generation methods proposed by us generates 27 test cases when we use uses the boundary value analysis and when we use full combination technique [27], [28] we generate a total of 125 test cases. This equates the worst case testing technique that is a sub technique of black box testing. Earlier we have generated 124 test case with our previous work of [27] proving the soundness of our technique.

TABLE I Generated test cases (totaling 125)

A	B	C	E	Alpha > 0	QE	R	K > 0	K = 0	K < 0	RR	ER	IR	Expected Result
0	0	0	0	0	0	0	0	0	0	0	0	0	!E
0	0	1	0	0	0	0	0	0	0	0	0	0	!E
0	0	50	0	0	0	0	0	0	0	0	0	0	!E
0	0	99	0	0	0	0	0	0	0	0	0	0	!E
0	0	100	0	0	0	0	0	0	0	0	0	0	!E
0	1	0	1	0	0	0	0	0	0	0	0	0	E, !QE
0	1	1	1	0	0	0	0	0	0	0	0	0	E, !QE
0	1	50	1	0	0	0	0	0	0	0	0	0	E, !QE
0	1	99	1	0	0	0	0	0	0	0	0	0	E, !QE
0	1	100	1	0	0	0	0	0	0	0	0	0	E, !QE
0	50	0	1	0	0	0	0	0	0	0	0	0	E, !QE
0	50	1	1	0	0	0	0	0	0	0	0	0	E, !QE
0	50	50	1	0	0	0	0	0	0	0	0	0	E, !QE
0	50	99	1	0	0	0	0	0	0	0	0	0	E, !QE
0	50	100	1	0	0	0	0	0	0	0	0	0	E, !QE
0	99	0	1	0	0	0	0	0	0	0	0	0	E, !QE
0	99	1	1	0	0	0	0	0	0	0	0	0	E, !QE
0	99	50	1	0	0	0	0	0	0	0	0	0	E, !QE
0	99	99	1	0	0	0	0	0	0	0	0	0	E, !QE
0	99	100	1	0	0	0	0	0	0	0	0	0	E, !QE
0	100	0	1	0	0	0	0	0	0	0	0	0	E, !QE
0	100	1	1	0	0	0	0	0	0	0	0	0	E, !QE
0	100	50	1	0	0	0	0	0	0	0	0	0	E, !QE
0	100	99	1	0	0	0	0	0	0	0	0	0	E, !QE
0	100	100	1	0	0	0	0	0	0	0	0	0	E, !QE
1	0	0	1	1	1	1	0	1	0	0	1	0	E, QE, R, ER
1	0	1	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
1	0	50	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
1	0	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
1	0	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM

1	1	0	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
1	1	1	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
1	1	50	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
1	1	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
1	1	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
1	50	0	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
1	50	1	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
1	50	50	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
1	50	99	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
1	50	100	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
1	99	0	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
1	99	1	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
1	99	50	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
1	99	99	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
1	99	100	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
1	100	0	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
1	100	1	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
1	100	50	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
1	100	99	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
1	100	100	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
50	0	0	1	1	1	1	0	1	0	0	1	0	E, QE, R, ER
50	0	1	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
50	0	50	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
50	0	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
50	0	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
50	1	0	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
50	1	1	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
50	1	50	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
50	1	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
50	1	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
50	50	0	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
50	50	1	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
50	50	50	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
50	50	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
50	50	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
50	99	0	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
50	99	1	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
50	99	50	1	1	1	1	0	0	1		0	1	E, QE, R, IM
50	99	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
50	99	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
50	100	0	1	1	1	1	1	0	0	0	0	0	E, QE, R, RR
50	100	1	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
50	100	50	1	1	1	1	0	1	0	1	1	0	E, QE, R, ER
50	100	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
50	100	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
99	0	0	1	1	1	1	0	1	0	0	1	0	E, QE, R, ER
99	0	1	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
99	0	50	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
99	0	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
99	0	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM

99	1	0	1	1	1	1	1	0	0	0	0	0	E, QE, R, RR
99	1	1	1	1	1	1	0	0	1	1	0	1	E, QE, R, IM
99	1	50	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
99	1	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
99	1	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
99	50	0	1	1	1	1	1	0	0	0	0	0	E, QE, R, RR
99	50	1	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
99	50	50	1	1	1	1	0	0	1	1	0	1	E, QE, R, IM
99	50	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
99	50	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
99	99	0	1	1	1	1	1	0	0	0	0	0	E, QE, R, RR
99	99	1	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
99	99	50	1	1	1	1	0	0	1	1	0	1	E, QE, R, IM
99	99	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
99	99	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
99	100	0	1	1	1	1	1	0	0	0	0	0	E, QE, R, RR
99	100	1	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
99	100	50	1	1	1	1	0	0	1	1	0	1	E, QE, R, IM
99	100	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
99	100	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
100	0	0	1	1	1	1	0	1	0	0	1	0	E, QE, R, ER
100	0	1	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
100	0	50	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
100	0	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
100	0	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
100	1	0	1	1	1	1	1	0	0	0	0	0	E, QE, R, RR
100	1	1	1	1	1	1	0	0	1	1	0	1	E, QE, R, IM
100	1	50	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
100	1	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
100	1	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
100	50	0	1	1	1	1	1	0	0	0	0	0	E, QE, R, RR
100	50	1	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
100	50	50	1	1	1	1	0	0	1	1	0	1	E, QE, R, IM
100	50	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
100	50	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
100	99	0	1	1	1	1	1	0	0	0	0	0	E, QE, R, RR
100	99	1	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
100	99	50	1	1	1	1	0	0	1	1	0	1	E, QE, R, IM
100	99	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
100	99	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
100	100	0	1	1	1	1	1	0	0	0	0	0	E, QE, R, RR
100	100	1	1	1	1	1	1	0	0	1	0	0	E, QE, R, RR
100	100	50	1	1	1	1	0	0	1	1	0	1	E, QE, R, IM
100	100	99	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM
100	100	100	1	1	1	1	0	0	1	0	0	1	E, QE, R, IM

IV. CONCLUSION

We have shown that use of previously trained models can generate POS tags on unseen data having a mathematical formula to generate test cases. For the purpose of POS tagging and parsing if we simultaneously use other corpus such as BNC [8] & corpuses available in public domain then results would be better, but for the sake of simplicity we have used only one. We use a very simple ontology for knowledge representation. Our method is also helpful in representing knowledge & modeling software requirements. This can be used in interpreting the statements of stakeholders and can be used as an elicitation technique, which can be used in interpreting the statements of stakeholders.

"Natural Language (NL) deliverables suffer from ambiguity [10], [11] poor understandability, incompleteness, and inconsistency. However, NL is straightforward and stakeholders are familiar with it to produce their software requirements documents" [22]. Our approach could be successfully be used to study the requirements in requirement elicitation phase. One can analyze whether the requirements are complete and properly understood. If requirements are incompletely expressed then there would be many graphs that cannot be connected. When the graphs are presented to stakeholders then any misinterpretation in requirements could be easily filtered out. Early interpenetration of requirements in this form will be helpful in generating test case, for verification & validation and in acceptance testing of software as whole or it components. The graph generated shown in figure 9 can also serve as test oracle. It can be argued that demonstration is about simple statement but results are positive for even more complex requirement statements. However, the natural language domain (NL) is large and complete coverage of complex requirement statements is not possible and would not appear in one day. We also feel the need for tool to automate this method and use data base such as Hadoop [25] for storing the graphs generated.

REFERENCES

[1] B. Beizer, "Software Testing Techniques", Van Nostrand Reinhold, Inc, New York NY, 2nd edition. ISBN 0-442-20672-0, 1990.

[2] A. Bertolino, "Software Testing Research and Practice", 10th International Workshop on Abstract State Machines (ASM'2003), Taormina, Italy, 2003W.-K. Chen, *Linear Networks and Systems* (Book style). Belmont, CA: Wadsworth, 1993, pp. 123–135.

[3] http://mait4us.weebly.com/uploads/9/3/5/9/9359206/chapter_8_software_testing.pdf

[4] Rosenberg, L., Hyatt,L., Hammer, T., Huffman, L. and Wilson, W. (1998) Testing Metrics for Requirement Quality, presented at the Eleventh International Software Quality Week, San Francisco, CA.

[5] Ramachandran, M. (1996) Requirements-Driven Software Test: A Process-Oriented Approach, ACM SIGSOFT Software Engineering Notes, Vol. 21, Issue 4, pp. 66 – 70

[6] Francis, W. Nelson, Kucera, Henry. THE BROWN CORPUS: A STANDARD CORPUS OF PRESENT-DAY EDITED AMERICAN ENGLISH (computer file), Providence, RI: Department of Linguistics, Brown University [producer and distributor], 1979. See more at: http://dataarchives.ss.ucla.edu/da_catalog/da_catalog_titleRecord.php?studynumber=M091V1#sthash.mVU4zpEh.dpuf

[7] Mitchell P. Marcus and Beatrice Santorini and Mary Ann Marcinkiewicz, Building a Large Annotated Corpus of English: The Penn Treebank, Journal of COMPUTATIONAL LINGUISTICS, year 1993, volume = 19, number 2, pages 313-330

[8] Geoffrey Leech, Roger Garside, Michael Bryant, Claws4: The Tagging Of The British National Corpus In COLING'94 (1994), pp. 622-628

[9] Jon Holt Institution of Electrical Engineers (2004). *UML for Systems Engineering: Watching the Wheels* IET, 2004, ISBN 0-86341-354-4. p.58

[10] K. S Wasson, K. N Schmid, R.R. Lutz, and J. C. Knight, "Using occurrence properties of defect report data to improve requirements," in Proc 13th IEEE International Conference on Requirements Engineering '05, on page(s): 253- 262, 2005.

[11] Joachim Karlssona, Claes Wohlinb, Bjo¨rn Regnellc, "An evaluation of methods for prioritizing software requirements," Information and Software Technology, Elsevier Science B.V 39, pp. 939–947, 1998.

[12] R. Schwitter, A. Ljungberg, "How to Write a Document in Controlled Natural Language", in Proceedings of the Seventh Australasian Document Computing Symposium, Sydney, Australia, December 16, pp. 133-136, 2002.

[13] B. Meyer , "On Formalism in Specification", IEEE Software , 2(1):5–13, January 1985

[14] B. Le Charlier & P. Flener (1998), "Specifications are Necessarily Informal: More myths of formal methods", Journal of Systems and Software, 40(3):275–296, March 1998.

[15] Mich, L., Franch, M. and Novi Inverardi, P., "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, Itlay, 2003.

[16] Cristiaximiliano and Plss, Brian (2010). Generating natural language descriptions of Z test cases. In: Proceedings of the 6th International Natural Language Generation Conference, 7-9 July 2010, Dublin, Ireland.

[17] Ratnaparkhi, A. (1998). Maximum Entropy Models for Natural Language Ambiguity Resolution. Ph.D. thesis, University of Pennsylvania.

[18] http://www.cs.umd.edu/~nau/cmsc421/part-of-speech-tagging.pdf

[19] Mitchell P. Marcus and Beatrice Santorini and Mary Ann Marcinkiewicz, "Building a Large Annotated Corpus of English: The Penn Treebank", Journal of computational linguistics, 1993, volume 19, number 2, pages 313-330

[20] http://www.codeproject.com/Articles/12109/Statistical-parsing-of-English-sentences

[21] Eugene Charniak, "A maximum-entropy-inspired parser", Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference, NAACL 2000, pp. 132-139

[22] Valdivino Alexandre de Santiago Ju´nior, Nandamudi Lankalapalli Vijaykumar, "Generating model-based test cases from natural language requirements for space application software", software quality journal, DOI 10.1007/s11219-011-9155-6.

[23] Lacy, Lee W. (2005). Chapter 9 - RDFS. "OWL: Representing Information Using the Web Ontology Language". Victoria, BC: Trafford Publishing. ISBN 1-4120-3448-5.

[24] Patel-Schneider, Peter F.; Horrocks, Ian (2006-12-19). "OWL 1.1 Web Ontology Language". World Wide Web Consortium. Retrieved 26 April 2010.

[25] Lam, Chuck (July 28, 2010). Hadoop in Action (1st ed.). Manning Publications. p. 325. ISBN 1-935182-19-6.

[26] http://pages.cs.wisc.edu/~jerryzhu/cs769/text_preprocessing.pdf

[27] Ravi Prakash Verma, Bal Gopal and Md Rizwan Beg, "Generation of test cases from software requirements using combination trees", IJCSI, International Journal of Computer Science Issues, Vol. 8, Issue 3, No. 1, May 2011, Page 334-340, ISSN (Online): 1694-0814

[28] Ravi Prakash Verma, Bal Gopal and Md Rizwan Beg, "Data Structure & Algorithm for Combination Tree To Generate Test Case", IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 3, No. 1, May 2011, Page 330-333, ISSN (Online): 1694-0814