Algorithm for Generating Test Case for Prerequisite of Software Requirement

Ravi Prakash Verma
Department of Computer Science & Engineering
Integral University

Bal Gopal
Dean
Department of Computer Applications
Integral University

Md Rizwan Beg
HOD
Department of Computer Science & Engineering
Integral University

ABSTRACT
Requirements play an important role in conformance of software quality, which is verified and validated through software testing. Requirements may have certain pre requisites which are to be tested first in order to start formal testing process. In this paper we present an approach to generate test case for testing software requirement pre requisites for GUI. Our approach takes pre requisites expressed in natural language and generates test cases from it. It also generates test case to expose the relationship between various components and elements present in prerequisites of GUI based windows forms.

General Terms
Software Requirements, testing

Keywords
Software testing, requirement prerequisite, test case generation

1. INTRODUCTION
The software testing is one the most important activity in the SDLC [8]. It authenticate whether the software being developed solves the intended purpose or not [2]. "Software systems continuously grow in scale and functionality" [1]. Software testing confirms that software being developed as per requirements [12]. At present test cases are written manually by testers [3] [13]. This is most error prone area as important case may be missed out by the tester [3]. Further earlier we find a defect less is the cost of correcting it and we save time, effort and money [4] [14]. But task of generating test case form the software is not an easy task. Many different approaches to express software requirement exist making it more complicated [15]. Out of many approaches the expression of software requirement in English is most popular due to prime reason of understandability by both client and developer [16]. Therefore in this paper we choose to work with the requirements expressed in natural language such as English and we attempt to develop an algorithm to generate test cases to verify the pre perquisite of a GUI requirement (functional). In the GUI based software [9] [10], requirements are implemented having GUI Interface. Where the pre-requisites are whether we have the proper controls for taking user input and associated labels and captions are appropriate, so that even a novice user can operate.

2. PROBLEM DIAGNOSIS
A requirement may have many prerequisites and sub requirements [9] [10] see figure 1. These are implemented using windows form. The windows form interface accepts the user input directly into many controls or objects. For example, consider the requirement "The user should be able to go to the Home page."[7], meaning login page should appear where the user logins in and starts to work on the software system.

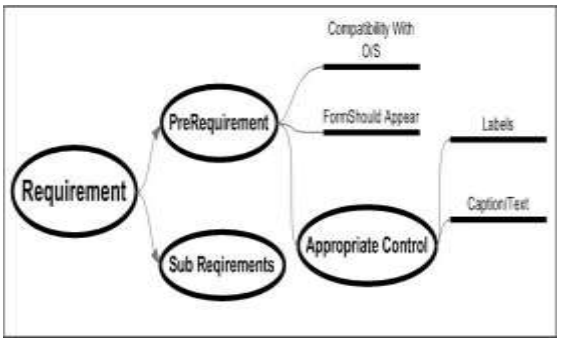

Figure 1. The requirement division into pre-requisite and sub requirement.

Universally the pre-requisite for the requirement implemented though these GUI interfaces are as follows.

1. Software should be compatible with the Operating System.
2. Login page should appear.

Apart from this there are pre-requisites based on the intended purpose of the requirement, which dictates the number of input to be given by the user and depending upon them there could be single or many controls or objects receiving them. For the above we have pre-requisite like as follows.

3. UserId textbox should be available with appropriate label.
4. Password textbox should be available with appropriate label.
5. Submit button with appropriate caption should be available.
6. Cancel button with appropriate caption should be available.

For the pre-requisite involving compatibility, includes "hardware, browser, network, peripherals, compatibility between versions of same software like backward compatibility, software (in connection with other), operating system and database" [5] [6]. Compatibility testing can be automated using automation tools or can be performed manually and is a part of non-functional software testing [6]. There compatibility of software is a complex issues and is beyond the scope of this paper. However we could well start with pre-requisite 2, where the tester can find that is there any page or form that provides login facility to the user or not. So the problem reduces to the simple search and one test case

to search for login form is enough and for the rest of prerequisites we purpose the following approach

3. APPROACH

In order to generate test cases for the pre-requisite involving controls and objects (pre-requisite number 3, 4, 5 & 6) we make few assumptions and use many data structure; these are defined and explained as follows

3.1 Assumptions and primitive structure

We assume that as the requirement exhibits the atomicity therefore pre- requisite for the requirement should also be enforced. The pre-requirements should be expressed in simple sentences. Every sentence is composed of words as follows

Sentence (Pre-requirement)$_i$ = Word$_1$ + Word$_2$ + Word$_3$ + … + Word$_n$

Which can be further simplified in terms of notation as follows

S (Pre-requirement)$_I$ = W_1 + W_2 + W_3 + … + W_n

These are stored in a table (see table 1) in which unique number is given to every sentence. This stored table will be hence forth referred as structure S_1 and individual elements can be accessed as S_1 [Row] [Column]. Every vector S_i [Row] [Column] is = W_1 + W_2 + W_3 + … + W_n

Table 1. Structure S_1 storing pre- requirements

Number	Class of the Object
S1	W_{11} + W_{12} + W_{13} + … + W_{1n}
S2	W_{21} + W_{22} + W_{23} + … + W_{2n}
S3	W_{31} + W_{32} + W_{33}+ … + W_{3n}
…	…

A GUI based interface is usually implemented in a high level language (HLL). Each HLL has its own vocabulary which consists of controls, objects, keywords etc. For example take the case of the C sharp we have textbox, label, buttons etc. commonly referred as object and each has its own set of method, properties and events. We construct the table (see table 2) of available object having unique number and class of object, hence forth will be referred as structure S_2 and individual elements can be accessed as S_2 [Row] [Column].

Table 2. Structure S_2, showing object and its properties

SN	Object	Properties
1	Textbox	…
2	Label	Text
3	Button	Caption/Text
…	…	

A set representing conditions between object and having values is defined as follows

- **Conditions** = {IS, HAS, DOES NOT HAS, HAVE, DOES NOT HAVE, SHOULD, SHOULD NOT, SHOULD BE AVAILABLE, SHOULD NOT BE AVAILABLE, …}

The Condition set is enumerated as follows
Conditions = {C_1, C_2, C_3, …, C_n}
Or
C = {C_1, C_2, C_3, …, C_n}

Adpositions set representing association between two or more that two object and having values like is defined as follows

- **Adpositions** = {WITH, OF, TO, IN, FOR, ON}

The Adpositions set is enumerated as follows
Adpositions = {A_1, A_2, A_3, …, A_n}
Or
A = {A_1, A_2, A_3, …, A_n}

These set have values and these are not limited to the aforementioned values. The testing team can add or remove the words in the set depending upon the dictionary, bibliography of SRS or from the domain knowledge of testing. These sets should be pre constructed before we start generating the test case to test the preconditions.

Additionally we need a tabular structure which will store the objects, conditions and association found in prerequisites statements, this structure, hence forth will be referred as structure S_3 (see table 3) and individual elements can be accessed as S_3 [Row] [Column]. This is defined as follows. The S_3 tabular data structure is initially initialized to null value.

Table 3. Structure S_3 stores objects, conditions and association

SN	Obj_1	Obj_2	Condition	Adpositions
	…	…	…	…

We also create a set in which will store the test case, initially this element is empty. This set could be realized with the help of simple arrays is defined as follows.
S_4 = { Ø }
S_4 = {distinct elements identified in preprocessing of retirement prerequisites}

We also use an array whose data type is string to store prerequisite statement into an array of string. Lastly we use a standard combination function to generate all possible combination in lexicographical order [11], which takes a set of distinct elements and does not generate duplicates.

3.2 Proposed Algorithm

3.2.1. Algorithm for preprocessing

Preprocessing ( )

```
{
 i = 0;
 foreach (sentence s in stucture S1)
  {
   j = 0;
```

```
    foreach (string word in s)
     {
      if (j == 0)
       {
        previous = word;
        j = j + 1;
       }
      else
       {
        if (S3[i].[1] == null)
         { S3[i].[1] = previous + objects(word); }
        else
         {
          if (S3[i].[2] == null)
           {
            if (objects(word) != null)
             { S3[i].[2] = objects(word); }
            else
             { S3[i].[2] = property(word); }
           }

          if (S3[i].[4] == null)
           { S3[i].[4]= adposition(word); }

          temp = temp + word;
          match[k] = condition(temp);
          if (match[k] != null)
           { k = k + 1; }
          temp = temp +  " ";
       }
     }
    S3[i].[3] = finalCondition();
    i = i + 1;
   }
}
```

3.2.1.1. Function for object

```
string object (w)
 { foreach (object o in S2)
    { if (w = = o)
       { return (w) ;}
     }
   return (null);
}
```

3.2.1.2. Function for property

```
string property (w)
 { foreach (property p in S2)
    { if (w = = p)
       { return (w) ;}
     }
   return (null);
}
```

3.2.1.3. Function for adposition

```
string adposition (w)
 { foreach (adposition a in A)
    { if (w = = a)
       { return (w) ;}
     }
   return (null);
}
```

3.2.1.4. Function for condition

```
string condition (w)
 { foreach (condition c in C)
    { if (c = = a)
       { return (w) ;}
     }
   return (null);
}
```

3.2.1.5. Function for final condition

```
string finalCondition ( )
{ for (j = 0; j ≤ count( ); ++ j)
   { temp =j;
     length = match[j].length( );
     for (k = j + 1; k ≤ count( ); ++k)
      { if (length < match[k].length( ))
         { temp = k;
           length = match[k].length( );
         }
       }
    }
  return (match[temp]);
}
```

3.2.1.6. Function for count

```
int count( )
 { i = 0;
   foreach (string s in match)
   { i = i +1; }
   return (i);
 }
```

Now since we have the preprocessed data in the tabular data structure we can precede with proposing algorithm to generate test, which as follows.

3.2.2. Algorithm to generate the test cases

```
GenerateTestCase ( )
{ // add elements
  for ( i = 0; i ≤ numberOfRows(S3); ++ i)
   { S4 = S4 + add element (S3[i][1] · S3[i][3]);
     S4 = S4 + add element (S3[i][2] · S3[i][3]);
   }
 // generate combination of element
  S4 = S4 + Combination (numberOfElementsIn(S4),
numberOfElementsIn(S4));
 // considers the association between the elements
  for ( i = 0; i ≤ numberOfRows(S3); ++ i)
  { for ( j = 0; j ≤ numberOfRows(S3); ++ j)
    { if ((S3[i][1] is object && S3[j][3] is object) || (S3[i][1] is
object && S3[j][3] is associated property ))
       { S4 = S4 + S3[i][1] · S3[j][4] · S3[j][3]); }
  }
}
```

4. ANALYSIS AND RESULT

We start the analysis of the algorithm by considering a pre-requisite "UserId textbox should be available with appropriate label". We store the sentence in the tabular data structure S_1word

by word. After initializing i, j, k to 0 and temp to null we take first word of the sentence since the value of j = 0 we store this in previous as this is the name of the object. After that we take the next word, if it matches "if (S_3[i][1] = = null)" there fore get stored in S_3. The next word "should" get matched in "if (S_3[i][3] = = null)" and gets the storage in match data structure mean while temp variable also holds it value. If we take the next word "be" then we find that it is concatenated in temp variable and same happens for the word "available". Next word in the sentence is "with" which is from the Adpositions set and get a match in the line "if (S_3[i][4] = = null)". The last word "label" gets the match & storage in S_4 at "if (S_3[i][2] = = null)" and "{ S_3[i][2] = = object (w); }" respectively. The supporting functions used in the algorithm are as follows "object (w)", "property (w)", "adposition (w)" and "condition (w)". The "object (w)", "property (w)" functions scan for object & property with the help of S_2 respectively. The "adposition (w)" & "condition (w)" functions scan for adpositions & conditions from the sentence with the help of Adpositions & Conditions sets respectively. The function "finalCondition ( )" is used to differentiate between the two different conditions such as "SHOULD" alone & "SHOULD BE AVAILABLE" and chooses the final condition to used further on the bases of length. The last function is the "count ( )" function which simple find the total number of stored strings in array "match". Finally we get the S_3 as follows for the sentence just processed. When we scan all the sentences we get the following (see Table 4).

Table 4. Structure S_3 after all sentence are scanned.

SN	Object $_1$	Object $_2$	Condition	Adposition
1	User ID Textbox	Label	Should be available	With
2	Password Textbox	Label	Should be available	With
3	Submit Button	Caption	Should be available	With
4	Cancel Button	Caption	Should be available	With

In order to further analyze we would replace text with symbols and when we do this we get table 5.

a = UserID textbox
b = label (for UserID textbox)
c = Password textbox
d = label (for Password textbox)
e = Submit button
f = caption (for Submit button textbox)
g = Cancel button
h = caption (for Cancel button textbox)
i = should be available
j = with

When "GenerateTestCase ( )" gets called it inserts the following elements in the set

S_4 = {a·i, b·j, c·i, d·j, e·i, f·j, g·i, h·j }

After generating combination of the above and adding the test cases for different associations, we get

S_4 = {(a·i, b·j, c·i, d·j, e·i, f·j, g·i, h·j) , (a·j·b), (a·j·d), (a·j·f), (a·j·h), (c·j·b), (c·j·d), (c·j·f), (c·j·h), (e·j·b), (e·j·d), (e·j·f), (e·j·h), (g·j·b), (g·j·d), (g·j·f), (g·j·h)}

Table 5. Structure S_3 when text is replaced by symbols

SN	Object $_1$	Object$_2$	Condition	Adposition
1	a	b	i	j
2	c	d	i	j
3	e	f	i	j
4	g	h	i	j

After eliminating fictitious association we get
S_4 = {(a·i, b·i, c·i, d·i, e·i, f·i, g·i, h·i, a·j, b·j, c·j, d·j, e·j, f·j, g·j, h·j), (a·j·b), (a·j·d), (c·j·b), (c·j·d), (e·j·f), (e·j·h), (g·j·f), (g·j·h)}
So a total of nine test cases were generated

| S_4 | = 9

As we know that using combinations to generate test cases creates enormous amount of test cases. It can be argued that pre-requisite for GUI based user interface or windows form should have all object before further testing can be proceeded. There fore testing team will write test case which includes entire objects in the form but they fail to write test case which explores the association between controls used to design the form. With our algorithm, we explore the possible association between them and test the possibility of cross liking them accidentally, which is very important and which is not possible manually as tester often ignore to write test cases to check them. For example the programmer can code a form, where a label "Password" is placed in font of textbox expecting "User ID" and vice versa. Similarly it can set caption of "Submit" button to "Cancel" and vice versa. In this case there may be nothing wrong with the functionality of the form.

5. CONCLUSION AND FUTURE WORK

We have given the same requirements to various authors and asked them to write test cases for testing prerequisites requirements for GUI, we found that that they always miss the test case to explore the association between the different controls. This is important as we could have cross linking of various controls and resultant code may fail to meet the requirements. Our algorithm, high lights the associations between the various object. The algorithm gives the test case where we have to check whether all necessary controls are present or not. A simple code or script would check that all necessary controls present or not. To check the cross linking or association between controls we have execute the test cases generated by us. The method used gives and impression that there could be enormous number of test cases, which are necessary to execute in order to prove the correctness of GUI. However they are actually less then that because let us suppose that there are n different controls and minimum association degree is 2 there fore we have to execute n/2 test cases to prove the correct linkage between any two controls. If the degree of association is higher the numbers of test cases to prove the correct associations are even less i.e. n/degree of association. However our method is based on assumptions like, it expects requirements to be expressed using simple (atomic) sentences in natural language such as English. The pre-requisites should not be having more than two objects at a time or should only include one

object with one property at a time. In future work we would try to eliminate these assumptions and would try to work on compound & complex sentences.

6. REFERENCES

[1] Kaschner, K., Lohmann, N., "Automatic Test Case Generation for Interacting Services". In Proc. of ICSOC 2008 Workshops. Volume 5472 of Lecture Notes in Computer Science. (2009)

[2] Tony Hoare, "Towards the Verifying Compiler", In The United Nations University / International Institute for Software Technology 10th Anniversary Colloquium: Formal Methods at the Crossroads, from Panacea to Foundational Support, Lisbon, March 18–21, 2002. Springer Verlag, 2002.

[3] Robert V. Binder, "Testing Object-Oriented Systems: Models, Patterns, and Tools", Addison Wesley Longman, Inc., 2000.

[4] M. Jazayeri C. Ghezzi and D. Mandrioli, "Fundamentals of Software Engineering", Prentice Hall, Englewood Cli_s, NJ., 1991

[5] Software Testing Glossary, http://www.aptest.com/glossary .html

[6] Software Testing - Compatibility Testing, http://www. buzzle.com/articles/software-testing-compatibility-testing .html

[7] Software Testing Tutorial, http://www.buzzle.com/articles /software-testing-tutorial.html

[8] S. S. Riaz Ahamed, " Studying the feasibility and importance of software testing: An Analysis", International Journal of Engineering Science and Technology, Vol.1(3), 2009, 119-128.

[9] A. Memon, M. Pollack, and M.L. Soffa, "Plan Generation for GUI Testing", Fifth International Conference on Artificial Intelligence Planning and Scheduling, Brackenridge, Co, April 14-19, 2000

[10] A. Memon, M. Pollack, and M.L. Soffa, "Automated Test Oracles for GUIs", Eighth International Symposium on the Foundations of Software Engineering (FSE2000) , San Diego, CA, Nov. 6-10, 2000.

[11] Dr. James McCaffrey, "Using Combinations to Improve Your Software Test Case Generation", http://msdn.microsoft.com/en-us/magazine/cc163957.aspx

[12] Glenford J. Myers, "The Art of Software Testing", Second Edition, John Wiley & Sons, Inc.

[13] B. Beizer "Software Testing Techniques", Van Nostrand Reinhold , 2nd edition, 1990.

[14] Gilb, Tom, "Principles of Software Engineering Management", Wokingham, England: Addison-Wesley, 1988.

[15] Stephen Withall, "Software Requirement Patterns", Microsoft Press

[16] Rizwan Beg, Qamar Abbas, Alok Joshi, "A Method to Deal with the Type of Lexical Ambiguity in a Software Requirement Specification Document," *icetet*, pp. 1212-1215, 2008 First International Conference on Emerging Trends in Engineering and Technology, 2008. DOI Bookmark: http://doi.ieeecomputersociety.org/10.1109/ ICETET.2008.160