Generation of test cases from software requirements using combination trees

Ravi Prakash Verma[1], Bal Gopal[2] and Md. Rizwan Beg[3]

[1] Department of Computer Science and Engineering, Integral University
Lucknow, Utter Pradesh, 226026 India

[2] Department of Computer Applications, Integral University
Lucknow, Utter Pradesh, 226026 India

[3] Department of Computer Science and Engineering, Integral University
Lucknow, Utter Pradesh, 226026 India

Abstract

Requirements play an important role in conformance of software quality, which is verified and validated through software testing. Usually the software requirements are expressed natural language such as English. In this paper we present an approach to generate test case from requirements. Our approach takes requirements expressed in natural language and generates test cases using combination trees. However until now we have the tabular representations for combination pairs or simply the charts for them. In this paper we propose the use of combination trees which are far easier to visualize and handle in testing process. This also gives the benefits of remembering the combination of input parameters which we have tested and which are left, giving further confidence on the quality of the product which is to be released.

Keywords: *Software testing, combination trees, Data structures, algorithm, Software Requirements, test cases*

1. Introduction

The software testing is one the most important activity in the SDLC [4]. 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 [5]. At present it is mostly done manually and the test cases are written by the tester, it is the Ad-hoc activity [3] [6]. This is most error prone area as important path or case may be missed out by the tester [3]. The testers develop test cases on the basis of the combinations of value of input parameters taken one at a time, these test cases are represented in the tabular form. It becomes difficult to remember that all the combination have been listed out or not. Further it difficult to visualize that whether we have covered all input parameters decisions that can be taken by the user. The trees can show the decision or action in a sequence which is very important for the software developer and tester to prove the robustness of the software system being developed. Testing done on the bases of combination trees [7] ensures that we are covering every possible action that can be taken by the user or at least can ensure that software system performs correctly if valid condition & action are chosen. In this paper we have proposed the algorithm to generate the test cases from by the use of combination trees and then we combine these trees to generate a single tree. The path traced from root to the node and finally to the leave nodes give the test case.

2. Proposed work

For the sake of understanding we take one example of the requirement and demonstrate the how the test cases are to be generated from software requirements using combination trees. As we know there are lots of software systems being developed which are GUI based. We pick one of common software requirement which is part of in fact every software system which is GUI based, which is "the user should be able to log in to the system". From here onwards we formalize our approach which is as follows.

2.1 Identification of classes of input

As we see that there are six controls on the Login Form namely two Textboxes, two Labels and two buttons. This login form is shown in the figure below (figure 1)

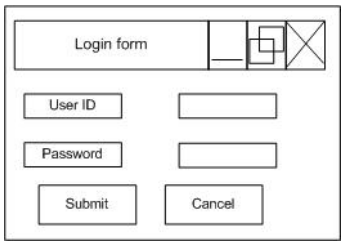

Figure 1. Sample login form

Let us establish which control receives which type of input from the user the "UserID" & "Password" texboxes receive user ID & password respectively, while the labels have fixed caption for the same. The buttons "Submit and Cancel receive the click Events. On the basis of the classes of input controls used in the form we can separate the distinct classes, over here in his case we have "textbox" and "Buttons".

The "Text" input to the control textbox can be any value from the superset as the set
AN = {alpha-numeric characters like a-z, A-Z}
SC = {Special characters like '$','#','!','~','*', ...)}
NC = {(numeric characters like 0-9)}

Text = {AN, SC, NC}

Any input can be classified into valid & invalid class and the in case of text it is constraint by length possibly $c_1 \leq k \leq c_2$, where c_1 and c_2 are finite and $c_1 \neq c_2$. Now we define the input into valid, invalid and show the desired length. Now lets us give each cell a number so that it could be differentiated with each other and handling becomes easy, from now onwards we will use these numbers and to understand what they are indicating to we have to refer the following tables.

Table 1. classification of inputs of Textboxes

SN	*Input*	*Length*	*Valid*	*Invalid*
1	$Text_{UID}$	>6 (1)	alpha-numeric characters {a-z, A-Z} (2)	Special characters like{'$','#','!','~','*',...} numeric characters like{0-9} i.e. Text - AN(3)
2	$Text_P$	> 6 (4)	Text (5)	-

Table 2. classification of inputs of buttons

SN	*Object/ Control*	*Event*	*Embedded procedure/function*	*Action*
1	Submit Button	Event $Click_{SB}$ (6)	Calls Match: which matches user name & password (7)	If Match successful: Go to Home Page (8)
				If Match unsuccessful: Display Message (9)
2	Cancel Button	Event $Click_{CB}$ (10)	Calls Clear All Textboxes (11)	All text boxes are cleared (12)

The condition or statement represented by any number can be complimented as, For example we see that (1) in table 1 represents that the textbox which accepts the user id of the user should allow a user id greater than the length six, so notation (1') means that user id is less than length six. We that the input that is accepted by this form under the above requirement should have (1)·(2) and another statement can be generated by taking the compliment of (1)·(2) which is (1')·(2) which mean the input is any combination from the set AN but length is less than six. "·"implies that both the statements are to be imposed simultaneously. Now we individually take one row from the table and put it into arrays. For table 1, row 1 the arrays elements are 1.2 & 1.3 and it compliment is 1'·2 & 1'·3. For table 1, row 2 the arrays elements are 4.5 and it compliment is 4'·5. Similarly for table 2, row 1 the array elements are 6.7, 8, 9 and for table 2, row 2 the array elements are 10.11 & 10.12. For the array we are generating a combination tree with the following algorithm and creating an orchid with trees representing each array. We will need a following data stricture:

```
struct node {     char [ ] value ;
                  structure node *Parent;
                  structure node *Child [Max];
            }
```

Roots is an array of node which are used to store the different roots of the tree and is defined as follows

```
struct Roots {
            struct  node * N;
            struct node * next;
           } Roots[MaxNumberOfArrays];
```

```
struct Roots  * RootsHead = NULL;
struct Roots  * RootsTail = NULL;

void addRoot(struct * node)
 { if (RootsHead = = NULL && RootsTail = = NULL)
    { RootsHead = (struct *Roots) malloc(sizeof(struct
Roots));
      RootsTail = (struct *Roots) malloc(sizeof(struct
Roots));
      RootsHead→N = node;
      RootsHead→next = NULL;
      RootsTail = RootsHead;
    }
   else
    { struct Roots  * temp = (struct *Roots)
malloc(sizeof(struct Roots));
      temp = RootsTail;
      temp →N = node;
      temp→next = NULL;
      RootsTail→next = temp;
      RootsTail = temp;
      Free(temp);
    }
}

void removeNodeFromHead()
 { if (RootsHead ≠ NULL)
    { struct Roots  * temp = (struct *Roots)
malloc(sizeof(struct Roots));
      temp = RootsHead;
      temp = temp→next;
      RootsHead = temp;
    }
 }

int countRoots(struct Roots  * RootsHead)
 { if (RootsHead ≠ NULL)
    { int i = 1;
     struct Roots  * temp = (struct *Roots)
malloc(sizeof(struct Roots));
      temp = RootsHead;
      while (temp ≠ RootsTail)
       { temp = temp→next;
         i = i +1;
       }
      return (i);
    }
   else
 { return (0); }

}

struct node * makeRootNode(char [] NameOfArray)
 { struct node * temp = (struct * Roots)
malloc(sizeof(struct Roots));
   temp→value = NameOfArrary;
   temp→ Parent = NULL;
   for (int i = 0; i < MAX + 1; ++i)
     { temp→ Child[i] = NULL }
   retrun (temp);
}

bool match(char [ ] NameOfArray)
{ struct node * temp = (struct * Roots)
malloc(sizeof(struct Roots));
  temp = RootsHead;
  while (temp ≠ RootsTail)
    { if (temp→value = NameOfArrary)
       { return (True) ;
         temp = RootsTail;
       }
      temp = temp→next
    }
    return (False);
 }
```

The linked list representation of pointers to nodes is used to store intermediate result. One of the advantages provided by this storage is that it avoids back tacking and traversal. The size of this pointer array first increases then it starts to reduce and finally reduces to zero size in length. This happens because of $\sum_{i=1}^{i=n} {}^{n}c_{i}$, which is $2^{(n-1)}$ -1.

```
struct ParentPointerNode { struct  node * N;
                           struct node * next;
             };

struct ParentPointerNode  * ParentPointerHead = NULL;
struct ParentPointerNode  * ParentPointerTail = NULL;

void addParentPointer(struct * node)
 { if (ParentPointerHead = = NULL &&
ParentPointerTail = = NULL)
    { ParentPointerHead = (struct *ParentPointerNode)
malloc(sizeof(struct ParentPointerNode));
      ParentPointerTail = (struct *ParentPointerNode)
malloc(sizeof(struct ParentPointerNode));

      ParentPointerHead→N = node;
      ParentpointerHead→next = NULL;
      ParentPointerTail = ParentPointerHead;
    }
   else
    { ParentPointerTail →next = node;
      ParentPointerTail = node;
    }
}

void removeNodeFromHead()
```

```
 { if (ParentPointerHead ≠ NULL)
    { struct ParentPointerNode  * temp = (struct
*ParentPointerNode) malloc(sizeof(struct
ParentPointerNode));
       temp = ParentPointerHead;
       temp = temp→next;
       ParentPointerHead = temp;
    }
 }

struct node * makeNode(char [] data )
 { struct node * temp = (struct *ParentPointerNode)
malloc(sizeof(struct ParentPointerNode));
   temp→value = data;
   temp→ Parent = NULL;
   for (int i = 0; i < MAX + 1; ++i)
     { temp→ Child[i] = NULL }
   retrun (temp);
}
```

2.2 Algorithm to create combination tree

```
void createCombinationTree(Element [] Array, int
MaxElementInArray) //, int CountRoot)
 {  if (ParentPointerHead = = NULL)
     { for ( i = 0; i < MaxElementInArray; ++i)
        { struct node * NewNode = makeNode(Array[i]);
          if ( match(Array) = = False)
          { struct node * NewRootNode =
makeRootNode();
            addRoot(NewRootNode);
          }
          NewNode→Parent = NewRootNode;
          NewRootNode→Child[i] = NewNode;
          addParentPointer(NewNode);
        }
     }
    else
     {
       struct node * temp = (struct *ParentPointerNode)
malloc(sizeof(struct ParentPointerNode));
       temp = ParentPointerHead;

       while (temp ≠ ParentPointerTail)
        { struct node * N = (struct *ParentPointerNode)
malloc(sizeof(struct ParentPointerNode));
          Node = ParentPointerHead;
          k= 0;
          char [] tempValue = N→value;
          for (j = 0; j < MaxElementInArray; ++j)
           { if (N→value < Array[i])
             { struct node * NewNode =
makeNode(Array[i]);
               NewNode→Parent = N;
               N→Child[k] = NewNode;
              addParentPointer(NewNode);
               k = k + 1;
             }
           }
         }
        temp→next = node;
        removeNodeFromHead();
      }
   }
}
```

This will create an orchid of as many trees equal to number of arrays, since we have one array for every single row. The orchid is as shown the black dots represent the roots of trees (see figure 2).

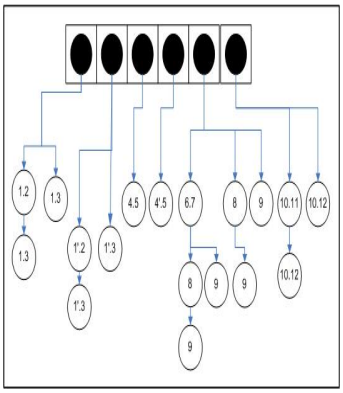

Figure 2. Orchid of combination trees

2.3 Elimination of combination

As in any combination tree there could be absurd combinations which are impossible to reach in practical situations, till now these are not eliminated in our approach. Out of this enormous sample space the reduction is only possible if we fix certain criteria of how the things will happen in the system. This could be easily done by simple control flow graph or establish a simple procedure of how to login into our system (see figure 3). The procedure is as follows

1. enter user ID
2. enter password
3. if you press submit button go to step 5
4. if you press cancel button go to step 6
5. matches user ID and password if matched go to step 7 else go to step 6
6. all textboxes are cleared
7. system takes to the home page

8. system shows home page

With the help of this procedure we can connect the orchid into a single tree

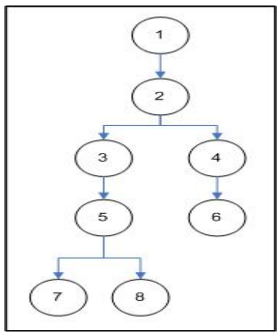

Figure 3. Control flow graph of the procedure to login to the system using login form

2.4 Elimination of child

Combination tree shows all possible combination, it does not considers where they are meaning full or not, certain combinations generated by the above algorithm are impossible to realize for example in the above case we can see that if by pressing "Submit" button the use may go with situation 8 or 9 (see table) but not the both one after the other or if "Click" event of the button is not fired then either 8 nor 9 can be possible. Therefore there could be many such cases present in the combination tree which are infeasible, absurd or not possible altogether. To eliminate such cases we have to parse the entire collection of tree under certain rules which eliminate these combinations. This rule should be developed only for the trusted & standard components, whose behaviors is known and has been thoroughly tested. For example in our case it's the "Button". Following rules can be defined using a rule set.

Definition: Rule set is the set of edges or set of possible productions. Let S be set of rules and L be the set of symbols denoted by $L = \{L_1, L_2, L_3, \ldots, L_n\}$, with which we express the rules or productions. For example in our case the set of symbols is L = {6.7, 8, 9} and the rule set S is defined as follows:

$$S \rightarrow 6.7S \mid S8 \mid S9$$

Now we can produce all applicable rules with the production system these are as follows

Rule 1

$$S \rightarrow 6.7S$$
$$S \rightarrow 6.78$$

Rule 2

$$S \rightarrow 6.7S$$
$$S \rightarrow 6.79$$

We define the production set P = {6.78, 6.79} and apply it over the orchid then we eliminate edges from root to 8, root to 9, and 8 to 9. Similarly for others and the resulting orchid is given in figure below (figure 4).

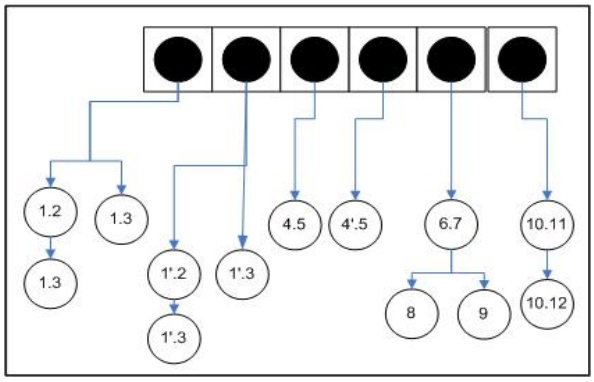

Figure 4. After elimination of children

2.5 Elimination of roots

The elimination of roots is possible by merging the trees which represent the complimentary conditions originating from same steps of control flow graph. As Roots [0] & Roots [1] originate from same step 1 of the flow control and Roots [2] & Roots [3] also originate from same step 1 of the flow control. The new orchid is shown in figure 5.

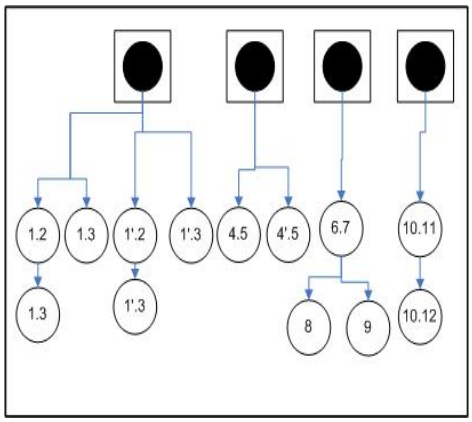

Figure 5. After elimination of roots

2.6 Combining trees

We can see that if we do not reduce the combination tress then we would have huge number of possibility and number of test case generated will be very large. As we have developed a control flow graph for the object under test, if we use that then we could limit the number of possibilities by which user can interact with the form, with the help of this we fix the merger of tree as follows (figure 6)

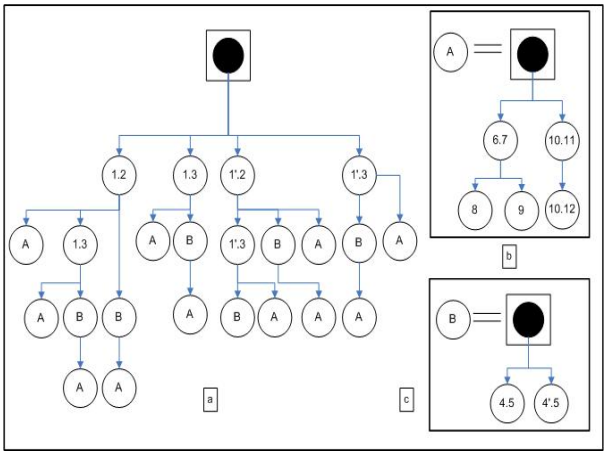

Figure 6. After combining trees

Now we add two additional nodes an extension node, expected result pass node and expected result fail node. The expected result pass node is the node where the software/module/form should comply with the intended purpose of the software requirement further its child fields are set to NULL (see figure 7). The expected result fail node is not actually indicate the failure of the software/module/form instead it indicate that it indicate that software/module/form should raise an error message or it should not allow users to continue. Here also the child fields of expected result fail node are set to NULL. The aforementioned nodes are graphically shown in the figure below. These nodes are attached as leaflet of the tree forming external nodes. We can fix these nodes with help of tables and flow control generated finally we get the following (see figure 8)

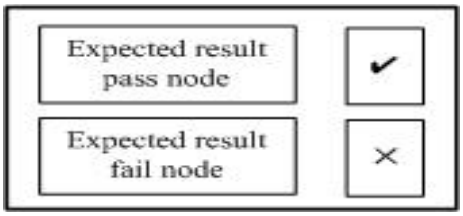

Figure 7. Additional nodes

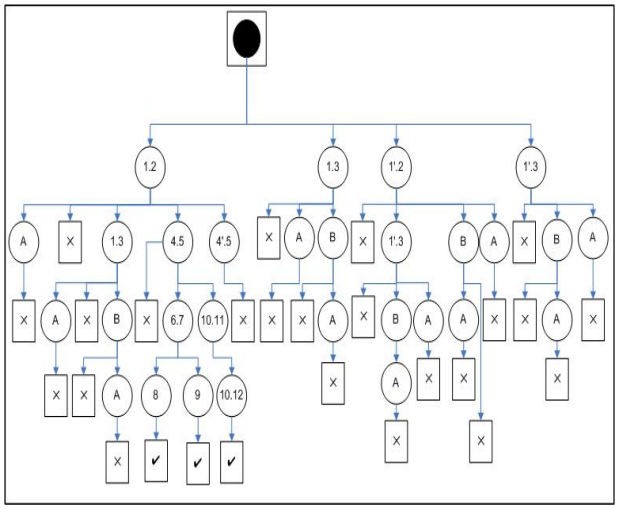

Figure 8. Final combination tree

3. Result and analysis

To get the test case we have to descend from the root to its child and where ever we find a terminating leaves we list the nodes encountered and that becomes the test case with the expected result motioned in the leaves whether it passes or fails. In doing so we get 4 test cases at level 2, 22 test case at level 3, 28 test case at level 4 and finally 9 at level 5. So in total we have 63 test cases. Among all test cases generated so far we have 3 test cases where we have the expected results pass. As we can see that in this simple case can produce enormous amount of test case, however in practice only some are created and only few are executed.

4. Conclusion and future work

It has been impossible to think about such number when we create test cases on ad-hoc bases, however it may not be possible to execute all of them but at least we discover the test cases in which the system should pass successively under given choices of inputs and action by user. We can deliver the system on the bases of selecting the test case in which there is expected result pass while maturing & increasing our confidence on system by performing more test as system is operational. If we find any bugs or fault we can fix them later on. The optimal testing is necessary to establish quality control. Our future work will be to release a tool to support our claim as it is not possible to manually generate such amounts

of test case and we would probabilistically determine the optimality in execution of test cases over such standard software components such as login form.

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] 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.

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

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

[7] Jaroslav Nesetril, "ASPECTS OF STRUCTURAL COMBINATORICS (Graph Homomorphisms and Their Use)", TAIWANESE JOURNAL OF MATHEMATICS Vol. 3, No. 4, pp. 381-423, December 1999