

An Efficient Slicing approach for Test Case Generation

DVLN Somayajulu*, Ajay Kumar Bothra, Prashant Kumar, Pratyush
(E-mail : somadvlns@yahoo.com , ajaybothra@yahoo.com ,
prashant_nitw@yahoo.co.in , pratyush.kanth@gmail.com)

Abstract:

Automated test data generation is a challenging task in software engineering research. Despite all the advancement, software testing continues to be the most time and resource consuming aspect of software development. One aspect of Automated Testing is generation of test data for each predicate involved in the Unified Modeling Language (UML) diagram of the modeled software. The test data generation also needs to consider the dependencies of each predicate on other constraints in the model. In this paper, a new algorithm called Slicing_for_Testcase_Generation (STCG), based on slicing, is proposed for automatic test data generation. Our approach is based on construction of sets by considering the program dependencies which is more efficient compared to the traditional path approach. Depending upon the program dependencies for a given predicate, we obtain a slice for it i.e., for a given model and predicate our approach efficiently generates test cases by considering only the relevant constraints present in the model. The implementation of our approach is presented and experiments were carried out for various UML diagrams. Our experimental results show that the proposed approach is an efficient one than the traditional path approach.

1. Introduction:

Specification, design, implementation, testing, and maintenance are the main activities in any software development life cycle. Model checking of software programs has two goals: one is the verification of correctness of the software. The other is the discovery of errors in faulty software. However, despite the importance and the many resources and man-hours invested by industry (about 30% to 50% of development effort), testing remains quite ad hoc and error prone [5]. Testing activities are often considered to be

the most expensive, and are crucial for establishing confidence in the software [1]. Testing can be done at an early stage in the development process so the developer will often find inconsistency and ambiguity in the specification.

Testing activities consist of designing test cases that are a sequence of inputs, executing the program with test cases, and examining the results produced by this execution [11]. Test case generation is vital part of testing activity. Test case generation is based on evaluation of a predicate i.e. the constraint under consideration, to generate test data satisfying the applicable constraints [10]. Test data are developed for the model using randomly generated values of input variables. A test or test case is a general software artifact that includes test case input values, expected outputs for the test case, and any other inputs that are necessary to put the software system into the state that is appropriate for the test input values. Test cases are usually derived from software artifacts such as specifications, design or the implementations.

The Unified Modeling Language (UML) [4] is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems. It is also used for business modeling and other non-software systems [3]. The UML represents a collection of engineering practices that have been used to model large and complex systems [4]. As a semi-formal modeling language, UML is widely used to describe analysis and design specifications by both academia and industry, thus UML models become the sources of test generation naturally [2].

Path-oriented approach [6] for testing is based on the combination of symbolic execution and constraint solving. Methods for representing expressions and path conditions are used in this approach. The Depth first search (DFS) or Breadth first search (BFS) is

*Address for correspondence: Department of Computer Science & Engineering,
National Institute of Technology – Warangal, Warangal, India 506004

the basis of test case generation in this approach. However this approach is not very efficient. The goal is to find values for input variables such that a terminal state can be reached. If successful, input test data are found (which might reveal a bug in the model of the software) [6]. The other approach for the test case generation involves concept of Slicing [7]. The concept of program slices was first introduced by Weiser [7]. A static slice consists of those parts of program that directly or indirectly affects the value of variable selected at some point of interest. The variable along with point of interest is known as slicing criteria [8]. Korel and Laski [9] introduced the concept of Dynamic Slicing.

A dynamic program slice contains only those statements that actually affect the value of a variable at a program point for a given execution. The approach proposed in this paper extends the concept of Slicing for test data generation. Slicing in Test Case Generation involves construction of Dependency Graph [12], which considers the dependency of each predicate in the modeling diagram of the Software to be tested. The proposed algorithm, *STCG* determines these dependencies. This algorithm finds out all the dependencies in the diagram in one pass by grouping all the depending conditions in similar set while parsing the UML model of the software to be tested. The main advantage of our algorithm is that it generates test data much faster as compared to path approach [6] of generating test cases.

The rest of the paper is organized as follows. In section 2, the basic terminology and definitions are presented. The proposed algorithm is discussed in detail in section 3. In section 4, we compare the Path approach using Depth First Search (DFS) technique with our *STCG Algorithm*. We have shown that the algorithm finds precise dynamic slices, and that it is more time efficient than the existing algorithm involving path approach. The implementation details of the algorithm are discussed in section 5. Section 6 presents experimental results of our analysis which is applied on different UML diagrams. A detail time analysis is also provided to support our work. Conclusions and future works are discussed in section 7.

2. Basic Definitions:

This section discusses the basic definitions that are used in this paper.

Boundary values: Boundary values refers to the set of values for the variables in a predicate such

that any change in the value of any variable in one direction makes the predicate false while value in other direction satisfies the predicate. For example if $x < 5$ is the predicate, which is satisfied when value of x is less than 5. $x = 4$ is the boundary value since increasing the value of x violates the condition while decreasing the value satisfies it. Boundary values are generated randomly to automate the test case generation. To obtain the boundary value from randomly generated initial value, minimization technique is used.

Test case generation: It refers to finding two set of values for the variables in a Predicate P , such that one set satisfies the predicate and another does not. The values set satisfying the predicate should be boundary values and the set not satisfying the predicate should differ from the set satisfying the predicate only in the value of one variable by one.

Constraints: These are the mathematical condition which determines the transition from one node to other in the UML [4] diagram of Software. Conditions may be combined using AND, OR to furnish complex constraints. Nodes in the diagram may be Object, State, Function, Class depending upon the UML [4] diagram. To move from one node to other constraint must be fulfilled.

Predicate: Predicate is a mathematical condition or expression present in the transition from one state to another or itself. For each predicate test case is to be generated satisfying predicate itself, and other depending constraints on which it depends.

Notes: These are the mathematical definitions of variables. Notes are used to define unknown variables or redefine known variables. For example in $x = y + z + 5$, x is being defined in terms of y and z .

Independent variable: A variable whose value does not depend on the value of other variables. More the number of independent variables in the predicate, faster is the generation of test cases as number of depending constraint on predicate is less.

Slice: Specific part of the path which contains the constraints on which, the predicate under consideration depends directly or indirectly.

3.1 Functionality of Slicing for Test Case Generation (STCG) Algorithm:

In this section, we discuss a new slicing algorithm called *STCG algorithm* for finding dependencies. The *STCG algorithm* is given in

Table 1 and the functionality of the algorithm is discussed below.

Initially the condition set or constraint set C and notes (expressions) set N are null. The graph or modeled diagram is traversed to reach the predicate and the constraints encountered on the way are stored in the constraint set C and notes in the notes set N . Let for a particular predicate P (for which test data is to be generated) the constraint set $C = \{C1, C2, C3 \dots\dots\dots, Cn\}$ and the notes set $N = \{N1, N2, N3 \dots\dots Nn\}$. The notes in the set N are replaced in the conditions and the predicate so that the predicate P' and the condition set $C' = \{C1', C2', \dots\dots\dots, Cn'\}$ are obtained. Now we apply the *STCG algorithm* to form a set of only those constraints on which the predicate depends. This set forms the group of the constraints that must be satisfied in order to generate the test cases for the predicate P .

For making groups or sets, we start with the first condition $C1'$ in C' and determine the variables in the condition. The variables along with the condition constitute the first group $G1$. Now the second condition $C2'$ in the set C' is taken and variables in the condition are determined. If any of the variables contained in $C2'$ is present in the set of variables of group $G1$, the variables of $C2'$

are added to the set of variables of $G1$ and the condition $C2'$ is added to the set of conditions of the group $G1$. If none of the variables of the condition $C2'$ is same as the set of variables of the group $G1$, a new group $G2$ is formed. The set of variables of $G2$ contains the variables of the condition $C2'$ and the set of conditions contain the condition $C2'$. Likewise for each condition Cm in the set of conditions C' , if any of the variable present in the condition Cm is also present in the set of variables of any of the groups, the variables are added to the set of variables of the group and the condition is added to the set of conditions of the group. If the variables determined from the condition Cm are present in more than one group, the groups are merged to form a new group. The process is repeated for each condition Cm in the condition set C' .

The variables contained in the predicate P' are determined and the set of conditions of the groups whose set of variables contain at least one variable of P' form the set of relevant conditions R for the predicate P' . This set of relevant conditions R is the set of conditions which are to be satisfied in order to generate the test case for the predicate P' .

```

Begin :
  for each predicate, P in the UML diagram
    Traverse from starting state, S to P.
    Add constraints involved on the way, to C and notes to N
    for every constraint, Cm in C
      for every variable v in Cm
        if (there is any 'note' for v)
          replace v using the 'note' in Cm
    for every variable v in P
      if (there is any 'note' for v)
        replace v using the 'note' in Cm
    for every constraint, Cm in C
      extract all variables in Cm
      if variables of Cm are present in more than 2 groups or sets
        merge the two sets
      else if any variable of Cm is present in some set S
        add variables of Cm to variable set of S and Cm to G
      else if none of variables of Cm are in any of the sets
        form a new group with variable set containing the
        variables of Cm and condition set containing Cm
    extract all variables in P
    merge all groups which contain any of the variables of P to form a
    new set R
End

```

Table 1: Slicing for Test Case Generation (STCG) Algorithm

The condition set of R contains the constraints on which P is dependent. Now random values are generated for the variables contained in P' and the conditions in the condition set of R. These values are manipulated to obtain the boundary values satisfying the conditions in the

condition set of R and predicate P'. These values give the required test case.

3.2 Illustration of the STCG algorithm:

We illustrate working process of the *STCG algorithm* with an example. Consider the state diagram of the Figure 1.

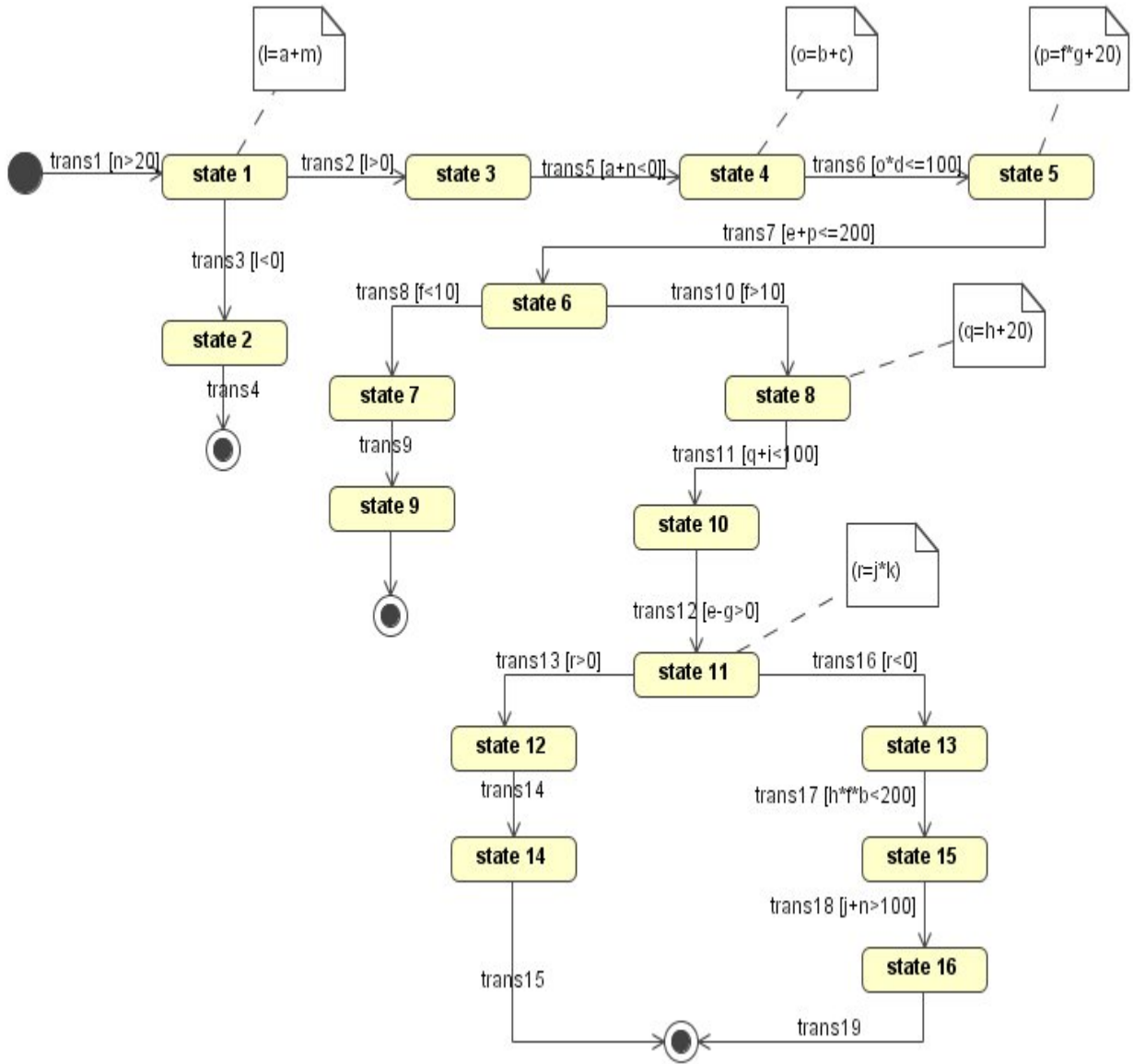


Figure 1- Sample State Diagram

Initially the condition set C and the notes set N are null. As we traverse the state diagram to reach a predicate, we add the conditions to the set C and notes to the set N. For trans18 in Figure. 1, the predicate is $j+n>100$. For this predicate the condition set C is $\{ n>20, l>0,$

$a+n<0, o*d<=100, e+p<=200, f>10, q+i<100, e-g>0, r<0, h+f+b<200\}$ and the notes set N is $\{ l=a+m, o=b+c, p=f*g+20, q=h+20, r=j*k\}$. The notes are replaced in the conditions of set C and predicate to obtain the condition set C' as $\{ n>20, a+m>0, a+n<0, b+c+d<=100, e+f*g<=180, f>10,$

$h+i<80$, $e-g>0$, $j*k<0$, $h+f+b<200$ and the predicate P' is $j+n>100$. Now using the *STCG algorithm*, the groups are made. The first condition in the set C' is $n>20$. The first group is formed with variable set containing the variable n and the condition set containing the condition $n>20$. Now the second condition in the set C' i.e. $a+m>0$ is taken up. As none of the variables in this condition is common to the existing variable

set, a new group is formed containing the variables a and m . The corresponding condition set contains the condition $a+m>0$. The third condition is the set C' is $a+n<0$. The two groups are merged to form a new group because the variables in the condition $a+n<0$ are contained in two groups. The next two conditions are $b+c+d<=100$ and $e+f*g<=180$.

Condition	Variable Set	Condition Set
$n>20$	n	$n>20$
$a+m>0$	n	$n>20$
	a, m	$a+m>0$
$a+n<0$	n, a, m	$n>20, a+m>0, a+n<0$
$b+c+d<=100$	n, a, m	$n>20, a+m>0, a+n<0$
	b, c, d	$b+c+d<=100$
$e+f*g<=180$	n, a, m	$n>20, a+m>0, a+n<0$
	b, c, d	$b+c+d<=100$
	e, f, g	$e+f*g<=180$
$f>10$	n, a, m	$n>20, a+m>0, a+n<0$
	b, c, d	$b+c+d<=100$
	e, f, g	$e+f*g<=180, f>10$
$h+i<80$	n, a, m	$n>20, a+m>0, a+n<0$
	b, c, d	$b+c+d<=100$
	e, f, g	$e+f*g<=180, f>10$
	h, i	$h+i<80$
$e-g>0$	n, a, m	$n>20, a+m>0, a+n<0$
	b, c, d	$b+c+d<=100$
	e, f, g	$e+f*g<=180, f>10, e-g>0$
	h, i	$h+i<80$
$j*k<0$	n, a, m	$n>20, a+m>0, a+n<0$
	b, c, d	$b+c+d<=100$
	e, f, g	$e+f*g<=180, f>10, e-g>0$
	h, i	$h+i<80$
	j, k	$j*k<0$
$h+f+b<200$	n, a, m	$n>20, a+m>0, a+n<0$
	b, c, d, e, f, g, h, i	$b+c+d<=100, e+f*g<=180, f>10, e-g>0, h+i<80$
	j, k	$j*k<0$

Table 2: Illustration of the STCG Algorithm

Two new groups are formed as none of the variables are common with the existing groups. The next condition in the set C' is $f>10$. The variable f is already present in a group so it is added to the set of variables of that group and the condition is added to the set of conditions. The

next condition is $h+i<80$. As neither of h and i is present in any of the groups, a new group is formed. The condition $e-g>0$ is merged with the group with variable set $\{e, f, g\}$ as the variables are already present in the group. The next condition is $j*k<0$ and one more new group is

formed. The last condition is $h+f+b < 200$. As the variables contained in the condition $h+f+b < 200$ are contained in three different groups, these three groups are merged to form a new group as shown in the Table 2 above. Now we have formed groups using the *STCG algorithm*, we take up the predicate P' . The predicate P' is $j+n > 100$. The variables contained in the predicate P' are j and n . The variable n is present in the first group and the variable j is present in the third group of the Table 2. Hence the relevant conditions for the predicate P' will be the condition sets of both the groups. The relevant conditions for the predicate P' are $n > 20$, $a+m > 0$, $a+n < 0$, $j*k < 0$.

Once the relevant conditions for a predicate are found, random values are generated for the variables $\{n, a, m, j, k\}$. These values are bought nearer to the boundary value by reduction of its range. The boundary values thus found is the test case generated.

4. Comparison with DFS approach

STCG algorithm for making groups for finding the test cases is an efficient approach than the DFS or path approach. In Figure 1, the set of constraint C' for the predicate P' ($j+n > 100$) is $\{n > 20, a+m > 0, a+n < 0, b+c+d \leq 100, e+f*g \leq 180, f > 10, h+i < 80, e-g > 0, j*k < 0, h+f+b < 200\}$ when DFS approach is used, whereas in case of slicing it is $\{n > 20, a+m > 0, a+n < 0, j*k < 0\}$. The random values will be generated satisfying the conditions in the constraint set of given predicate P' for the variables $\{a, b, c, d, e, f, g, h, i, j, k\}$ in case of DFS approach in contrast to $\{n, a, m, j, k\}$ in case of slicing approach. As we can see that the number of variables in case of DFS approach is more than the number of variables in the slicing algorithm, more random values generation is amounting to more time. As the number of conditions in DFS approach is greater than the slicing algorithm, more constraint checking and mathematical evaluation is involved. Hence the slicing algorithm is more time efficient compared to the DFS approach. This is further

enforced by the experimental results obtained after implementing both the algorithms for the various UML diagrams in the same environment in the section to follow.

5. Implementation Details:

We have implemented our *STCG algorithm* as well as the traditional path approach in JAVA. The implementation involves parsing of the Extensible Markup Language (XML) file of the corresponding UML diagram. To achieve this, the Document Object Model (DOM) Application Programming Interface (API) of the JAVA is used. For modeling in UML we have used the Magic Draw [13] tool. The corresponding XML codes for the diagrams were also generated by the tool itself. By parsing the XML file we can retrieve all predicates, conditions (constraints), notes and other related information. The time requirements of general path approach and *STCG algorithm* were calculated and compared for various UML Statechart, Activity, Collaboration and Sequence diagrams [15]. The *STCG Algorithm* was found to be much more time efficient as compared to the path approach where the relevant conditions are not extracted from the set of conditions before the generation of final test data.

6. Experimental Results:

A prototype is developed for generating test cases for various UML models. Experiments results show that *STCG* is more time efficient, practical and useful than Path model. All UML diagrams are constructed using MagicDraw [13]. These experiments were executed on a Pentium-based machine with 512MB RAM running Windows-XP, professional edition (Microsoft Corporation) Operating System. We also built a test execution module for automatically generating test cases without human intervention. A result summary for few experiments is presented in Table 3 and Table 4.

Test No.	Time in ms using Path Approach.	Time in ms using STCG Algorithm
1	215.83	156.50
2	228	190
3	242	175.30
4	245	193.6
5	232.3	187.2

Table 3: Time analysis for sequence diagram

Test No.	Time in ms using Path Approach.	Time in ms using STCG Algorithm
1	670.5	629.5
2	720.3	625.2
3	726.5	630.5
4	690.7	625.6
5	708.1	623

Table 4: Time analysis for State diagram

7. Conclusion and Future work:

In this paper we proposed a new algorithm called Slicing_for_Testcase_Generation (STCG) for Test Case Generation from an UML diagram. We have shown that this algorithm is more time efficient as compared to the general DFS Approach. The implementation of our algorithm for various UML diagrams proves that it more efficient compared to the Path approach using DFS technique, irrespective of the type of UML diagram. Also the *STCG algorithm's* efficiency increases with the complexity of the UML diagrams since the number of constraints required to be satisfied is enormously large. In such situations, our proposed *STCG algorithm* works in an efficient manner due to the consideration of relevant conditions only. Such a kind of functionality is not taken care in path approach, which results in time penalty.

As part of future work we would like to build a complete UML testing tool with the help of our approach for the test case generation part.

References:

1. B. Beizer, Van Nostrand Reinhold. Software Testing Techniques. New York NY, 2nd edition, ISBN 0-442-20672-0, 1990.
2. Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong, Zheng Guoliang. Generating test cases from UML activity diagram based on Gray-box method. Software Engineering Conference, 2004. 11th Asia-Pacific Page(s):284 – 291, 30 Nov.-3 Dec. 2004
3. Aynur Abdurazik and Jeff Offutt. A Controlled Experimental Evaluation of Test Cases Generated from UML Diagrams. Dept of Info and Software Engr. George Mason University, Fairfax, USA. ISE-TR-04-03, May 2004.
4. Object Management Group. OMG UML Specification Version 1.3. Available at <http://www.omg.org/uml/>. June 1999

5. Anders Hesselz, Kim G. Larseny, Brian Nielsen, Paul Petterssonz, and Arne Skouyy. Time-optimal Real-Time Test Case Generation using UPPAAL in Proc. of the 3rd Intl. Workshop on Formal Approaches to Testing of Software (FATES'03), 2003
6. Jian Zhang, Chen Xu, Xiaoliang Wang (Chinese Academy of Sciences, China) Path-Oriented Test Data Generation Using Symbolic Execution and Constraint Solving Techniques. 242-250, 2nd International Conference on Software Engineering and Formal Methods (SEFM), 2004.
7. M.Weiser. Programmers use slicing when debugging. Communication of ACM25 (7) 446-452, 1982.
8. G. B. Mund, R. Mall, S. Sarkar. An efficient dynamic program slicing technique. Information & Software Technology 44(2): 123-132, 2002.
9. B. Korel, J. Laski. Dynamic program slicing, Information Processing Letters, 29(3), 155-163, 1988.
10. B. Korel. Automated Software Test Data Generation. IEEE Transactions on Software Engineering Vol. 16(8), 870-879, 1990.
11. Supaporn Kansomkeat, Wanchai Rivepiboon. Automated-Generating Test Case Using UML Statechart Diagrams. Chulalongkorn Univesity, Thailand in Proc. of SAICSIT, 296-300, 2003.
12. K. Ottensein, L. Ottensein. The program dependence graph in software development environment. Proceeding of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices, Vol. 19(5), 177-184, 1984.
13. <http://www.magicdraw.com>: Magic Draw is a visual UML modeling and CASE tool
14. H. Conrad Cunningham. A Programmer's Introduction to Predicate Logic. Technical Report UMCIS 1994-02. Software Methods Research Group, Department of Computer and Information Science, University of Mississippi, 1994.
15. www.agilemodeling.com: Introduction to Diagrams of UML.