Widening the Goal Posts: Program Stretching to Aid Search Based Software Testing.

Kamran Ghani and John A. Clark
Department of Computer Science
University of York
Heslington, York,
YO10 5DD, UK
{*kamran, jac*}*@cs.york.ac.uk*

Abstract

Search based software testing has emerged in recent years as an important research area within automated software test data generation. The general approach of couching the satisfaction of test goals as numerical optimisation problems has been applied to a variety of problems such as satisfying structural coverage criteria, specification falsification, exception generation, breaking unit pre-conditions and software hazard discovery. However, some test goals may be hard to satisfy. For example, a program branch may be difficult to reach via a search based technique, because the domain of the data that causes it to be taken is exceedingly small or the non-linearity of the "fitness landscape" precludes the provision of effective guidance to the search for test data. In this paper we propose to "stretch" relevant conditions within a program to make them easier to satisfy. We find test data that satisfies the corresponding test goal of the stretched program. We then seek to transform the stretched program by stages back to the original, simultaneously migrating the obtained test data to produce test data that satisfies the goal for the original program. The "stretching" device is remarkably simple and shows significant promise for obtaining hard-to-find test data and also gives efficiency improvements over standard search based testing approaches.

1. Introduction

1.1. Dynamic Testing

Dynamic testing — "the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behaviour" [1] — is used to gain confidence in almost all developed software. Various static approaches can be used to gain further confidence but it is generally felt that only dynamic testing can provide confidence in the correct functioning of the software in its intended environment.

We cannot perform exhaustive testing because the domain of program inputs is usually too large and also there are too many possible execution paths. Therefore, the software is tested using a suitably selected set of test cases. A variety of coverage criteria have been proposed to assess how effective test sets are likely to be. Historically, criteria exercising aspects of control flow, such as statement and branch coverage [2], have been the most common. Further criteria, such as data flow [3], or else sophisticated condition-oriented criteria such as MC/DC coverage [4] have been adopted for specific application domains. Many of these criteria are motivated by general principles (e.g. you cannot have much confidence in the correctness of a statement without exercising it); others target specific commonly occurring fault types (e.g. boundary value coverage).

Finding a set of test data to achieve identified coverage criteria is typically a labour-intensive activity consuming a good part of the resources of the software development process. Automation of this process can greatly reduce the cost of testing and hence the overall cost of the system. Many automated test data generation techniques have been proposed by researchers. We can broadly classify these techniques into three categories: random, static and dynamic [5] [6].

Random approaches generate test input vectors with elements randomly chosen from appropriate domains. Input vectors are generated until some identified criterion has been satisfied. Random testing may be an effective means of gaining an adequate test set for simple programs but may simply fail to generate appropriate data in any reasonable timeframe for more complex software (or more sophisticated criteria).

With static techniques an enabling condition is typically generated that is satisfied by test data achieving the identified goal. For example, symbolic execution can be used to extract an appropriate path traversal condition for an identified path. Such enabling conditions are solved by constraint solving techniques. However, current application of static code analysis techniques to generate data is not widespread. Despite much research, these approaches do not scale well, and are problematic for some important code elements, such as loops, arrays and pointers [7].

Recently Search Based Software Engineering (SBSE) [8], [9] has evolved as a major research field in the software engineering community. Major work in the area has con-

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.26

centrated on software testing. In fact it is fair to say that search based software testing has, in many respects, lead the way in SBSE. Work can be dated back to 1976 [10]. The major work in search based software testing itself is search based test data generation. McMinn [7] published a survey paper back in 2004. There has been extensive activity in the field since then. A recent survey on the use of search based techniques to test non-functional properties has recently been published. [11].

2. Search Based Test Data Generation

2.1. General Approach

In search based test data generation (SBTDG) achieving a test requirement is modeled as a numerical function optimisation problem and some heuristic is used to solve it. The techniques typically rely on the provision of "guidance" to the search process via feedback from program executions. For example, suppose we seek test data to satisfy the condition $X <= 20$. We can associate with this predicate a cost that measures how close we are to satisfying it, e.g. $cost(X <= 20) = max(X - 20, 0)$. The value $X == 25$ clearly comes closer to satisfying the condition than does $X == 50$, and this is reflected in the lower cost value associated with the former. The problem can be seen as minimising the cost function $cost(X <= 20)$ over the range of possible values of X.

SBTDG for functional testing generally employs a search/optimisation technique with the aim of causing assertions at one or more points in the program to be satisfied. We may require each of a succession of branch predicates to be satisfied (or not) to achieve an identified execution path; we may require the program preconditions to be satisfied but the postconditions to be falsified (i.e. falsification testing — finding test data that breaks the specification) [12]; or else we may simply require a proposed invariant to be falsified (e.g. breaking some safety condition, or causing a function to be exercised outside its precondition) [13].

There has been a growing interest in such techniques and we see more and more applications of these techniques to software testing. Some of the techniques that have been successfully applied to test data generation are Hill Climbing (HC) [10], [14], Simulated Annealing (SA) [15], Genetic Algorithms (GAs) [16], [17], [18], [13], Tabu Search (TS) [6], Ant Colony Optimisation (ACO) [19], Artificial Immune Systems (AISs) [20], Estimation of Distribution Algorithms (EDAs) [21], Scatter Search (SS) [22] and Evolutionary strategies (ESs) [23].

2.2. Limitations of Search Based Approaches

SBTDG revolves around finding data to cause one or a sequence of assertions (predicates) within a program to be satisfied. Current approaches to search based test data generation have explored the use of various optimisation techniques and various fitness functions to find test data satisfying some particular goal. The approaches tried have seen considerable success, but there are also clear limitations.

Even with the predicate structures involving Boolean relational operators, some targets will be difficult to achieve. It may be because of the very small input domain of data satisfying the goal or because the program structures are not very amenable to the search process. For example, handling Boolean flag variables is hard. In such cases one approach has been to transform the program to an equivalent version more suited to SB approaches [24], [25], [26]. Program transformation has been used to make generation of test data easier. Once the data is found the transformed program may be thrown away. Below we take the idea of program transformation further, viewing transformation not as a one-off activity, but as a fundamental part of the search process.

3. Program Stretching

Currently almost all traditional program transformations are semantics preserving. This is not essential [27]; we can obtain the required test data from a transformed program semantically different from the actual program. In this paper we propose a radical generalisation of current transformation approaches, illustrated with respect to branch and path coverage goals.

We will 'stretch' difficult branches thus making them easier to cover. This can be achieved by generating mutant programs by adding auxiliary variables to the predicates in the difficult branches. To clarify the idea consider the following example.

$if\ (expr1 < expr2)$ .....(I)
{
statements
}
$else$
{
statements
}

$if((expr3 > expr4) \wedge (expr5 < expr6))$.....(II)
{
statements
}
$else$
{
statements
}

Lets suppose that (II) is the branch that SBTDG seems

unable to cover. We now add additional input variables, i.e, var1 and var2 to it as shown below.

$if\ (expr1 < expr2)$ .....(I)
{
statements
}
$else$
{
statements
}

$if\ ((expr3 + var1) > expr4 \wedge (expr5 < (expr6 + var2))$.....(II)
{
statements
}
$else$
{
statements
}

Initially we set the values of $var1$ and $var2$ reasonably large so that it is easy (even trivial) to find test data such that $(expr3 + var1) > (expr4)$ and $expr5 < (expr6 + var2)$. The ranges of additional variable values define a set of "stretched" programs. Our search now proceeds over the set of such stretched programs and test inputs for them. The search trajectory comprises a sequence of pairs $\langle (prog_1, td_1), (prog_2, td_2), \ldots, (prog_{final}, td_{final}) \rangle$. The aim is to end up with end up with an "unstretched" program $prog_{final}$ (with all auxiliary variables set to 0) and test data td_{final} that satisfies the required constraints. Essentially, the search space comprises the original test data input space combined with the set of all variable assignments to auxiliary variables. Although we know the desired eventual value of each auxiliary variable (i.e. 0) allowing it to take positive intermediate values can facilitate the solution of the overall problem. Essentially we find test data satisfying our goal for a highly stretched program, and evolve the test input data and auxiliary variables together to achieve the original aim. This requires a very simple modification to existing SBTDG mechanisms.

3.1. Fitness Function

We use the basic fitness function proposed by [15] for our work as shown in Table 1. This is a modified form of the work proposed by [14]. The fitness function is based on evaluating branch predicates. It gives a value of 0 if the branch predicate evaluates to the desired value and a positive value otherwise. The lower the value, the better is the solution. The table indicates the cost for specific assertions. Where more than one assertion is of interest (e.g. when a sequence of branch predicates must be satisfied to follow an identified path) then the basic costs per predicate are combined in some way. For a sequence of assertions to be satisfied we combine them using the following equation;

$$f(x)_{path} = f(x)_{branch_c} + KN$$

Where K is a constant, $branch_c$ is the current branch and N is the number of uncovered branches of the path to be satisfied.

We need also to incorporate the effect of auxiliary variables. For this purpose we add a new term to the fitness function, which is the summation of all the new variables. i.e,

$$f(x)_{total} = f(x)_{path} + \sum_{i=0}^{n} abs(v_i)$$

where v_i is the ith auxiliary variable

We then bring down $f(x)_{total}$ to zero by decreasing the value of $\sum_{i=0}^{n} abs(v_i)$ in a 'controlled' way.

Table 1. Fitness-Functions

Element	Value#1
$a == b$	if $abs(a-b) = 0$ then 0 else $abs(a-b)$+K
$a \neq b$	if $abs(a-b) \neq 0$ then 0 else K
$a < b$	if $(a-b) < 0$ then 0 else $(a-b)$+K
$a \leq b$	if $(a-b) \leq 0$ then 0 else $(a-b)$+K
$a > b$	if $(b-a) < 0$ then 0 else $(b-a)$+K
$a \geq b$	if $(b-a) \geq 0$ then 0 else $(b-a)$+K
$a \vee b$	$min(cost(a), cost(b))$
$a \wedge b$	$cost(a) + cost(b)$

Table 2 shows how assertions in a program may be stretched.

In most cases this is straightforward. For the case of equality $(a == b)$ we first transform to the equivalent
$(a \geq b \wedge b \geq a)$
before mutating.

A zero cost solution to an identified goal provides test data that achieves the goal for the original program. The extension of the cost function to include punishment of non-zero values of additional variables does not assume any particular mechanism for the traditional cost function component. We have chosen the traditional one with which we are most familiar, but others' approaches can be simply extended in the way we suggest.

4. Experiments and Evaluation

To evaluate the validity of the above concept we performed two sets of experiments. In the first set, we consid-

Table 2. Branch Transformation

Predicates	Transformed Form	Remarks
$a == b$	$(a + var1) \geq b \wedge$ $(b + var2) \geq a$	
$a \neq b$	$a \neq b$	No need for transformation
$a < b$	$a < (b + var)$	
$a > b$	$(a + var) > b$	
$a \leq b$	$a \leq (b + var)$	
$a \geq b$	$(a + var) \geq b$	
$a \vee b$	Apply transformation to expr a and\or b using the above rules	
$a \wedge b$	Apply transformation to expr a and\or b using the above rules	

ered three small case studies of program code of varied McCabe structural complexity. These programs are given in Appendix B. In the second set of experiments, we applied the concept at the architectural level to MATLAB® Simulink® models. We compared performance of the programs on the basis of (i) coverage, i.e. input test data found for branches against total branches to satisfy a coverage criterion, (ii) success rate, i.e. how many times test data was found when a program was run multiple times to satisfy a coverage criterion and, (iii) the average number of executions taken to find the test data.

4.1. Experiment set 1: Code Examples

We used Simulated Annealing (SA) with the following parameters for our first set of experiments . Further experiments can be done to optimise the parameters. However, since we kept the same parameters for both approaches, we believe the results will be similar for the optimise set as well.

Move strategy: fixed

Geometric Temperature decrease rate: 0.8

Number of iteration in inner loop: 250

Maximum number of iteratoin in outer loop: 1000

Stopping criterion: Either a solution is found or maximum number of iterations are reached.

Case study 1 is relatively easy having a McCabe structural complexity [28] of 10 and is experimented with to find the applicability of the stretching principle to different relational operators. Case Study 2 though has a McCabe structural complexity of 8 but is relatively difficult from a search point of view because of the branching structure. Case study 3 has a McCabe structural complexity of 14 and the branching structure is more difficult for search. In each of the above programs, our goal is to achieve the coverage of the last branch as indicated in the appendix. To reach that goal we 'stretched' the intermediate difficult branches.

With program 1, we were able to obtain 100% coverage and success rate using the traditional search based approach and also with program stretching. However, as expected, the average number of iterations to cover the required target in stretched program approach were more than for the traditional approach. Although the goal is satisfied more quickly with the stretched program approach, the search process must expend additional effort reducing the program to the original and dragging the test data with it to maintain goal satisfaction. For easy goals, we believe that the traditional approach should be used instead of program stretching.

In program 2, the search was again 100% successful in each case. However, the task here is more difficult and the results are better for the program stretching approach. Program stretching, in this case, would appear to give efficiency advantages.

In program 3, the results were even better. In this case the search process was successful in 92% of runs for the standard approach and 99.8% of runs using program stretching. The average number of function evaluations (program executions) for successful runs were also better for the program streching approach.

Table 3 and 4 summarise the above results.

Table 3. Number of executions to generate test data Via Standard SBTDG (Based on 500 runs)

	Program1	Program2	Program 3
Average	4825	14248	78212
Max	15037	24315	250050
Min	451	2663	26604
SD	2876	5564	58747
Success Rate(%)	100	100	92

Table 4. Number of executions to generate test data via Program Stretching (Based on 500 runs)

	Program1	Program2	Program 3
Average	7222	11976	61960
Max	24900	18405	140606
Min	1341	5514	14411
SD	3671	2195	35853
Success Rate(%)	100	100	99.8

4.2. Experiment set 2: Simulink Models

As stated above, for the second set of experiments we used MATLAB Simulink models. Appendix A gives a short introduction to Simulink. For this set of experiments we used the models from Zhan and Clark [29], which used input variables of type double with a range of -100 to 100. We used the following configuration for SA parameters. The parameters setting is based on the experimental work by Zhan and Clark[30].

Move strategy: fixed.

Geometric Temperature decrease rate: 0.9
No of inner loop iterations: 100
Max. No. of outer loop iterations: 300
Stopping criteria: Either a solution is found or maximum number of iterations are reached.

We considered only those branches for analysis for which the traditional approach was not hundred percent successful. Zhan and Clark [29] used four models. Model *SmplSw* is rather straightforward and hence is not considered for experiments. *Quadratic* model is also relatively simple but the search is made more difficult by introducing local minima. This was achieved by changing the range of input variables.

Quadratic contains three Switch blocks and hence eight 'paths' or, more specifically combinations. *RandMdl* has four Switch blocks and hence has sixteen combinations. *Combine* has seven Switch blocks and hence one hundred and twenty eight combinations. We targeted each of these combinations in the case of *Quadratic* and *RandMdl*. In case of *Combine* model, we targeted only the case where all switch blocks take the value 'true'. Each model was run thirty times for each of the targets to achieve statistically significant results. For example, for *RandMdl*, we did four hundred and eighty runs in total.

Quadratic model gave much better results for the stretching approach. However, for models *RandMdl* and *Combine* the results in both approaches are very similar in terms of success rate. But the 'stretching' is more expensive as it requires more number of execution to find the input data to cover the required combination. The results have been summarised in Tables 5 and 6.

In both set of experiments, the neighbourhood is searched by considering a small change in any of the variables, either actual or auxiliary, and then the modified fitness function is evaluated. The move is accepted, if the fitness function value is improved. However, by restricting the neighbourhood search, we can get significant improvement in the results. In such case when the value of auxiliary variable is changed, the actual variable is also changed by a fraction of that value. The actual fitness function due to all the variables, actual or auxiliary, as well as the modified fitness function are evaluated. The move is accepted if both fitness functions are improved or if there is an improvement in any of the fitness functions but the other remains unchanged. Table 7 summarises the results for Simulink models for this strategy. We can see that there is a clear improvement in the average number of executions as well as the success rate.

Table 5. SBTDG for Simulink Models without Program Stretching (Based on 30 runs)

	Quadratic	RandMdl	Combine
Total No of Branches	8	16	116
No of Branches for Analysis	4	12	2
Mean Success Rate(%)	49.16	51.9	33
Coverage(%)	100	100	100
Mean No of Executions	587	1931	3122

Table 6. SBTDG for Simulink Models with Program Stretching (Based on 30 runs)

	Quadratic	RandMdl	Combine
Success Rate(%)	90.83	50	36.333
Coverage(%)	100	100	100
Mean No of Executions	5769	5230	7350

5. Previous Work

Program Transformation has been applied to programs most commonly to the removal of problematic program elements such as GOTO statement [31], [32]. In the realm of software testing the work by Harman et al [27], based on previous work [25] is noteworthy. They proposed a simple theory of testability transformation and applied the technique to the removal of flag problem for Evolutionary test data generation. The flag-free programs provided more determinism to the search process improving both the performance of evolutionary test data generation and the adequacy level of the test data so-generated. Hierons et al. [24] applied the technique to transformation of a program with one or more `exit` statement to a branch-coverage equivalent program thus preserving the branch-coverage adequate sets of test inputs. McMinn et al [26] applied the approach to the nested search targets. The approach was applied to the nested-if statement by converting the nested structure to a straight line one. Case studies were performed on two small programs which showed an improvement actor of more than two times in terms of efficiency in the search process. However, the technique still faces challenges posed by some common program constructs: loops, arrays and pointers.

6. Conclusions and Future Work

The program stretching principle is in its initial stages. Initial results are promising and we believe that the technique can be extended to more complex systems. The technique was motivated by the need to satisfy "difficult" goals. The studies indeed show that program stretching has advantages in these cases, either showing greater success or greater efficiency (or both) than standard search based test data generation.

The approach is basically a form of continuous program transformation. A key difference here is that the transformations do not preserve program semantics. The idea of stretching was originally motivated as a form of "topological" deformation. Ideally the input domains corresponding to

Table 7. SBTDG for Simulink Models with Program Stretching and 'Controlled' Neighbourhood (Based on 30 runs)

	Quadratic	RandMdl	Combine
Success Rate(%)	90.83	92.33	77
Coverage(%)	100	100	100
Mean No of Executions	3885	6018	6534

satisfying test goals in the stretched program would map in a straightforward way to those of the original program's test goals. This is not essential. One could easily imagine some input partitions vanishing as the program is stretched — some paths through the program may no longer be possible for example.

This paper has shown how program stretching can be used to find hard-to-find branches, but really this is about generating test data to satisfy hard-to-find conditions. This can easily be extended to the following applications:

Invariant Falsification: Suppose we have a proposed invariant inv for some identified point in the program. If we insert a program statement "if(!inv);" at that point we can view the falsification of the invariant as a branch reachability problem. As a consequence can easily apply the program stretching technique.

Exception Generation: Consider assignment statements of the form "var=expr;". When expr is evaluated the result must be a value inside the type bounds,if an exception is not to be raised. In program reasoning it is common to refer to refer to such constraints as healthiness pre-conditions. We can insert a statement of the form "if(!healthiness); and proceed much as before.

Higher Level Models: In this paper we attempted application of program stretching to the 'path coverage' or more specifically to the 'Combination' coverage [29] of Simulink models. Zhan and Clark [30] also proposed techniques for a more practical branch coverage criterion. We also aim at extending the approach to such criteria. We also believe that our stretching technique may find applications to other higher level models such as statecharts. Similarly, automated SBTDG from specification might be facilitated.

In future we aim to further explore the idea and identify the program structures where we can apply this technique with a greater degree of confidence. The paper has provided an indication that the stretching concept may be used to enable difficult test data to be found more easily. Our evaluation has been limited to a few programs. We believe that more extensive testing is clearly necessary to come to definitive conclusions about the utility of the approach. We have used a rather blunt notion of stretching (adding an auxiliary variable to one side of a relational expression). It remains an open question whether more sophisticated approaches to stretching can be found.

7. Application Outside SBSE

We believe that program stretching is a novel yet appealing concept. Although developed to solve a specific problem in SBTDG, it seems plausible that the approach could find application elsewhere. In abstract terms the fundamental idea is that the "problem" and "solution" are developed together. The problem is essentially relaxed (made easier) until a satisficing solution is found. An attempt is then made to migrate the problem to the original problem of interest with the solution being "dragged" along to maintain satisfaction of the changing constraints. We see no reason in principle why this form of relaxing and dragging should not find wider application within the search based engineering disciplines. We recommend this area to the research community.

Appendix

1. Simulink

Simulink is a software package for modelling, simulating, and analysing system-level designs of dynamic systems. Simulink models/systems are made up of blocks connected by lines. Each block implements some function on its inputs and outputs the results. Outputs of blocks form inputs to other blocks (represented by lines joining the relevant input/output ports). Models can be hierarchical. Each block can be a subsystem comprising other blocks and lines.

Simulink models have their special way of forming branches compared to programs. Blocks such as 'if-else', and Switch are used to form branches. We have used only Switch blocks in our work for branching structure. However, the work can be easily extended to include other blocks as well. A Switch block has three 'in' ports, one 'out' port and a control parameter 'Threshold'. When the value of the second 'in' port is greater than or equal to the threshold parameter, the output will equal to the value carried on the first 'in' port, otherwise the value carried on the third 'in' port will be channelled through to the output. Therefore a Switch block can map to an 'if-then-else' branching structure in code.

Two other important blocks that need to be introduced here are LogicalOperator and RelationalOperator blocks.

A LogicalOperator block has two parameters: operator parameter (which can be 'AND', 'OR', 'NAND', 'NOR', 'XOR', or 'NOT') and input-number parameter (which can be any integer number except that when the operator

parameter is 'NOT', in which case, the input-number must be '1').

A RelationalOperator block has two inputs. There is a parameter defining the desired relation between the two inputs. If the relation is TRUE, the output will be '1'; otherwise, the output will be '0'. There are six options for the relational parameter: ==, $\sim=$, $<$, $<=$, $>=$, and $>$. Figure 1. is an example of a Simulink model showing the above mentioned blocks. This model has three input variables; 'IN-A', 'IN-B' and 'IN-C'. The 'Product' block multiply all it's inputs.The model calculates if an equation of the form:

$$ax^2 + bx + c = 0$$

is a quadratic equation and if it has real-valued solution(s). If it is a quadratic equation and it has one or two real-valued roots, '1' is output; otherwise, the output is '-1'

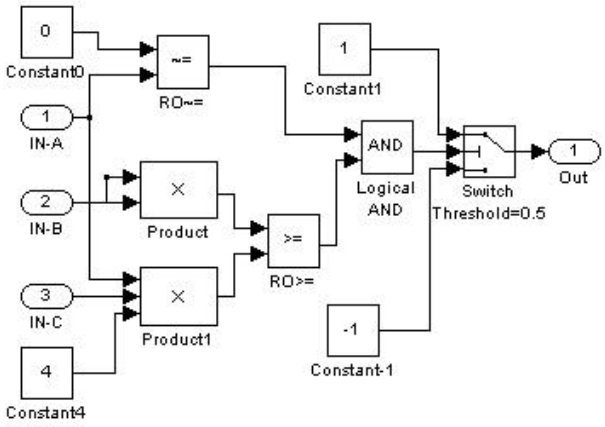

Figure 1. A Simulink Model for the Quadratic Equation.

Simulink models execute (calculate the outputs of) all branches of the models, whether the branches are selected or not, while for programs, only the selected branches are executed. For example, the following code matches the model in Figure 2.

```
program calculation;
 input x,y;
 output z;
    begin
       if x>=y
         z=x-y;
       else
        z=y-x;
     end
```

In the Simulink model, both *'x-y'* and *'y-x'* are calculated although only one of these results is channelled through to the output by the Switch block. However, in the code, only one of them will be executed depending on the evaluation of the predicate *'$x \leq y$'*.

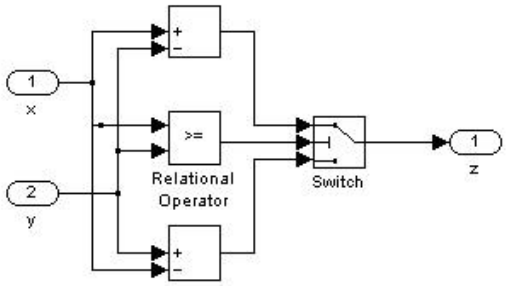

Figure 2. A Simple Simulink Model

2. SBTDG for Simulink Models

Simulink has been popularly used as a higher level system prototyping or design tool in many domains, including aerospace, automobile and electronics systems. This facilitates investigation (e.g. for both verification and validation purposes as well as optimisation) of the system under consideration at an early stage of development. Code can then be generated either manually or automatically. Simulink is playing a more and more important role in system engineering and the verification and validation of Simulink models is becoming vital to users. SBTDG techniques have seen little application to Simulink models, which is surprising since the execution model of Simulink would seem to allow analogous SBTDG techniques to be applied as for code. The only work that is known to us, to the best of our knowledge, is by Zhan and Clark [29], [33], [34]. The work in this paper is based on their earlier work [29].

3. Program 1

```
public class Program1  {


 public void opCheck(int j, int k,
int l, int m){

  if ((j>10) && (j< 15)){

  System.out.println("j>10 && j<15:
executed");


  if (k== j) {

   System.out.println("k=j:executed");

  if ((l!=k) && (m>=j) &&(m==k)) {
```

```
   System.out.println("Target
Reached"); ..........(1)
     }
    }
  }
 }
}
```

3.1. Transformed Program 1.

```
public class Program1Var {

 public void opCheck(int x, int y,
                 int z,int m,
                 int var1,int var2,
                 int var3,int var4){

  if (x+var1>10 && x-var2<15){

    System.out.println("x>10 &&
x<15: executed");

   if (y+var3>=x&&x+var3>=y){

    System.out.println("y==x:executed");

      if (z !=y && m>=x && m==y){

           System.out.println("Target
         reached");
     }
   }
  }
 }
}
```

4. Program 2

```
public class Program2 {

 public void complexCheck(int x, int y, int z) {
   if ((x<10) &&(y>1000) && (z>999)) {

   System.out.println("Branch 1 executed");
  }
  else if ((x > 10) &&(y > 1000) && (z> 999)){

   System.out.println("Branch 1
else-if executed");

   if (((z + y) < 2003) && (x + y< 1400)
&& (x > 390)) {
    System.out.println
   ("Target reached");..........(2)

   }
  }
  else {

System.out.println("the false branch taken");
   }
 }
}
```

4.1. Transformed Program 2.

```
public class Program2Var {

 public void complexCheck(int x, int y,
                    int z, int var1,
                    int var2,int var3){

  if ((x<10) &&(y>1000)
&& (z>999)){

   System.out.println("Branch-1 executed");

   }

  else if ((x+var1 > 10) &&(y+var2 > 1000)
  &&(z+var3> 999)){

   System.out.println("Branch-1
else-if executed");

   if (((z + y) < 2003) && (x + y< 1400)
                              && (x > 390)){

    System.out.println
   ("Target reached");
   }
  }
  else  {

   System.out.println("False branch taken");
  }
}
}
```

5. Program 3

```
public class Program3 {

  public void complexCheck(int a, int b,
int c, int d, int e)  {

  if ((a<10) && (b>1000)&&(c>999  )) {

    System.out.println("Branch-1
 executed");

  }
  else if ((a>10) && (b>1000)&& (c>999)
&& (d>999&& d<1001)) {

   System.out.println("Branch-1 else
 if executed");

   if (c+b<2003&& a+b<1400&&a>390) {

    System.out.println("Inner if
executed");

    if (d+e>2000 && d+e<2003){

     System.out.println
   ("Target reached");..........(3)
   }
  }
}
   else {
```

```
   System.out.println("False
 branch taken");
  }
}
}
```

5.1. Transformed Program 3.

```
public class Program3Var {

 public void complexCheck(int a, int b,
int c,int d,
int e, int var1,
int var2,int var3,
int var4,int var5,
          int var6)  {

  if ((a<10) &&(b> 1000) &&(c> 999)) {

   System.out.println("Branch-1
 executed");

  }
  else if (((a+var1> 10) &&
(b+var2 >1000)) &&
                c+var3> 999)&&
               (d+var4>999)&&
               (d<1001)) {

   System.out.println("Branch-1
else if executed");

   if ((c + b<2003+var5) &&
(a + b<1400)&&(a+var6>390)){

    System.out.println("Inner if
executed");

    if ((d + e> 2001) && (d + e<2003)) {

     System.out.println("Target
reached");
    }
   }
  }
  else {
   System.out.println("False
branch taken");
  }
 }
}
```

References

[1] IEEE, *Guide to software engineering body of knowledge*, 2004, ch. 5, p. 1.

[2] G. Myers, *The Art of Software Testing*. John Wiley & Sons, 1979.

[3] S. Rapps and E. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions of Software Engineering*, vol. 11, no. 4, pp. 367– 375, April 1985.

[4] RTCA, "Software considerations in airborne systems and equipment certification," pp. 2455–2464, 1992.

[5] C. C. Michael, G. McGraw, M. Schatz, and C. C. Walton, "Genetic algorithms for dynamic test data generation," in *Automated Software Engineering*, 1997, pp. 307–308. [Online]. Available: citeseer.ist.psu.edu/michael97genetic.html

[6] E. Diaz, J. Tuya, and R. Blanco, "Automated software testing using a metaheuristic technique based on tabu search," in *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, Oct. 2003, pp. 310 – 313. [Online]. Available: citeseer.ist.psu.edu/diaz03automated.html

[7] P. McMinn, "Search-based software test data generation: a survey: Research articles," *Softw. Test. Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, 2004.

[8] J. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating software engineering as a search problem," *IEE Proceedings — Software*, vol. 150, no. 3, pp. 161–175, 2003.

[9] M. Harman, "The current state and future of search based software engineering," in *Future of Software Engineering 2007*, L. Briand and A. Wolf, Eds., Los Alamitos, California, USA, 2007, pp. 342–357.

[10] W. Miller and D. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 223–226, Sept. 1976.

[11] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, 2009, in Press, Corrected Proof. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0B-4VHXDTD-1/2/9da989f9d874eb88d1f82d9a0878114b

[12] N. Tracey and J. C. K. Mander, "Automated program flaw finding using simulated annealing'," in *proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, Clearwater Beach, Florida, United States, 1998, pp. 73 – 81.

[13] N. J. Tracey, "A search-based automated test-data generation framework for safety-critical softwares," DPhil, University of York, 2000.

[14] B. Korel, "Automated software test data generation." *IEEE Trans. Software Eng.*, vol. 16, no. 8, pp. 870–879, 1990. [Online]. Available: http://www.computer.org/tse/ts1990/e0870abs.htm

[15] N. Tracey, J. Clark, K. Mander, and J. A. McDermid, "An automated framework for structural test-data generation," in *Automated Software Engineering*, 1998, pp. 285–288.

[16] B. F. Jones, H. H. Sthamer, and D. E. Eyres, "Automatic structural testing using genetic algorithms," *Software Engineering Journal*, vol. 11, no. 5, pp. 299–306, Sep 1996.

[17] J. T. Alander, T. Mantere, and P. Turunen, "Genetic algorithm based software testing," in *Artificial Neural Nets and Genetic Algorithms*. Wien, Austria: Springer-Verlag, 1998, pp. 325–328.

[18] R. Pargas, M. J. Harrold, and R. Peck, "Test-data generation using genetic algorithms," *Journal of Software Testing, Verification and Reliability*, vol. 9, no. 3, pp. 263–282, Sep 1999.

[19] H. Li and C. P. Lam, "Software test data generation using ant colony optimization." in *International Conference on Computational Intelligence*, 2004, pp. 1–4.

[20] P. May, K. Mander, and J. Timmis, "Mutation testing: An artificial immune system approach," in *UK-Softest. UK Software Testing Workshop*, University of York. UK., September 2003. [Online]. Available: http://www.cs.kent.ac.uk/pubs/2003/1700

[21] R. Sagarna and J. Lozano, "On the performance of estimation of distribution algorithms applied to software testing." *Applied Artificial Intelligence.*, vol. 19, no. 5, pp. 457–489., 2005.

[22] Sagarna and J. Lozano, "Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms." *European Journal of Operational Research*, vol. 169, no. 2, pp. 392–412., 2006.

[23] E. Alba and J. Chicano, "Software testing with evolutionary strategies." in *The 2nd Workshop on Rapid Integration of Software Engineering Techniques (RISE05)*, vol. 169, no. 2, Heraklion, Greece, September 2005., pp. 50–65.

[24] R. M. Hierons, M. Harman, and C. J. Fox, "Branch-coverage testability transformation for unstructured programs," *Comput. J.*, vol. 48, no. 4, pp. 421–436, 2005.

[25] M. Harman, L. Hierons, A. Baresel, and H. Sthamer, "Improving evolutionary testing by flag removal," in *Genetic and Evolutionary Computation Conference (GECCO 2002).*, 2002. [Online]. Available: citeseer.ist.psu.edu/harman02improving.html

[26] P. McMinn, D. Binkley, and M. Harman, "Testability transformation for efficient automated test data search in the presence of nesting," 2005.

[27] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, 2004.

[28] McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, pp. 308–320, 1976.

[29] Y. Zhan and J. A. Clark, "Search based automatic test-data generation at an architectural level," in *Proceedings of the 2004 Conference on Genetic and Evolutionary Computation (GECCO'04)*. Springer, 2004, pp. 1413–1424.

[30] Y. Zhan, "Search based test data generation for simulink models," Ph.D. dissertation, University of York, 2005.

[31] L. Ramshaw, "Eliminating go to's while preserving program structure," *J. ACM*, vol. 35, no. 4, pp. 893–920, 1988.

[32] A. M. Erosa and L. J. Hendren, "Taming control flow: A structured approach to eliminating goto statements," in *ICCL, IEEE Computer Society 1994 International Conference on Computer Languages*, Toulouse, France, May 1994, pp. 229–240.

[33] Y. Zhan and J. Clark, "Search-based mutation testing for simulink models," in *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO '05)*. ACM, 2005, pp. 1061–1068.

[34] Y. Zhan and J. A. Clark, "The state problem for test generation in simulink," in *Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation (GECCO '06)*. ACM, 2006, pp. 1941–1948.