Data Dependence Based Testability Transformation in Automated Test Generation

Bogdan Korel
Computer Science Department
Illinois Institute of Technology
Chicago, IL USA

Mark Harman
King's College London
Strand, London
WC2R 2LS, UK

S. Chung, P. Apirukvorapinit,
R. Gupta, Q. Zhang
Computer Science Department
Illinois Institute of Technology
Chicago, IL USA

Abstract

Source-code based test data generation is a process of finding program input on which a selected element, e.g., a target statement, is executed. There exist many test generation methods that automatically find a solution to the test generation problem. The existing methods work well for many programs. However, they may fail or are inefficient for programs with complex logic and intricate relationships between program elements. In this paper we present a testability transformation that transforms programs so that the chances of finding a solution are increased when the existing methods fail using only the original program. In our approach data dependence analysis is used to identify statements in the program that affect computation of the fitness function associated with the target statement. The transformed program contains only these statements, and it is used to explore different ways the fitness may be computed. These explorations are inexpensive when using the transformed program as compared to explorations using the original program. As a result, executions in the transformed program that lead to the evaluation of the fitness function to the target value are identified. The identified executions are then used to guide the search in the original program to find an input on which the target statement is executed. In this paper, the approach is evaluated using a case study which demonstrates the potential for this testability transformation to improve the efficacy of the test generation.

1. Introduction

Test data generation in white-box testing (source-code based testing) is a process of finding program input on which a selected element (e.g., a not yet covered statement) is executed. Finding such input test data can be very labor-intensive and expensive. Several automated test generation methods for white-box testing have been proposed in the literature, e.g., [2, 4, 5, 6, 7, 8, 9, 10, 12, 14, 15, 16, 20]. There are different types of automated test data generation methods: random test data generation [2], path-oriented test data generation [4, 5, 10, 15], the chaining approach [7, 11], and test generation methods based on genetic and evolutionary algorithms [1, 9, 12, 13, 14, 16].

Existing test data generation methods use different types of information about a program in order to guide the search process, e.g., a control flow graph, control dependencies, data flows, etc. Most methods use a program control flow graph to guide the search process. However, control information about a program is frequently not sufficient to efficiently guide the search. The search may be guided to parts of the program that have nothing to do with finding the solution. As a result, a lot of search effort may be wasted in these parts of the program. The chaining approach [7] partially alleviates this problem by using data flow information of the program to guide the search when problem statements are encountered during the search. However, this approach still uses a control flow graph as the primary source of information to guide the search. Although existing methods work well for many programs, for programs with complex logic and intricate relationships between program elements they either fail or are highly inefficient. The principal reason for this expense and failure lies with the reliance upon the control flow graph to guide the search. Frequently finding a solution requires executing parts of the program that are seemingly (from the control flow graph perspective) unrelated to the solution. Therefore, the search needs to exploit additional information that indicates ways in which different parts of the program may affect the solution.

In this paper, we propose to use data dependence analysis to form novel transformations of the original program that are used to guide the search process. Data dependences capture "computational" relations between statements and thereby identify statements that affect a solution. In particular, we use a data dependence graph to explore different computations, represented as data dependence paths in the data dependence graph. Using this approach may require the exploration of a large number of data dependence paths until the path is identified that leads to the solution. Exploring some paths in the original program may be expensive. Therefore, in our approach, we use a testability transformation [18] that transforms the original program so that the chances of finding a solution are increased. This approach is applied when the existing methods fail using the untransformed program, and so it augments rather than replaces existing approaches.

The transformed program is constructed in such a way that it makes it much easier to explore different computations that are likely to prove important. As a result, the search can find the solution much faster. The transformed program contains only the statements identified by data dependence analysis. The transformed program is used to explore different ways (paths) the fitness function may be computed. As a result, promising paths in the transformed program are identified, i.e., executions that lead to the evaluation of the fitness function to the target value. These promising paths are then used to guide the search in the untransformed (original) program to find the final solution, i.e., an input on which the target statement is executed in the untransformed program. Our initial experience with the approach shows that it may efficiently find a solution for many programs for which existing methods fail or have difficulties finding the solution.

In the next Section we briefly present the existing methods of test generation. In Section 3, a data dependence based search is presented. In Section 4, data dependence based testability transformation and its application in the search is presented. In Section 5, the approach is extended to multiple-variable data dependences. In Conclusions future research is discussed.

2. Test generation methods

This section briefly overviews the existing methods of automated test data generation for white-box testing. Test data generation in white-box testing is a process of finding program input on which a selected code element, e.g., a statement or a branch, is executed. In this paper, we concentrate on test generation for a statement, referred to as a target statement (or a target node). There are different types of test data generation methods that could be applied for this problem: *random* test data generation [2], path-oriented test data generation [4, 5, 10, 15], the chaining approach [7, 11], and test generation methods based on genetic and evolutionary algorithms [9, 12, 14, 16].

The path-oriented approach reduces the problem of test data generation to a 'path' problem, i.e., a program path is selected automatically which leads to the target node. Then a program input that results in the execution of the selected path is derived. If the program input is not found for the selected path, a different path is selected which also leads to the target node. This process is repeated until a program path is selected for which a program input is found, or it is terminated when the designated resources are exhausted, e.g., search time limit. Two methods have been proposed to find program input to execute the selected path, namely, symbolic execution [4, 5, 8], and execution-oriented test data generation [10]. The symbolic execution generates a path constraints (algebraic expressions over the symbolic input values), which consist of a set of equalities and inequalities on the program's input variables. These constraints must be satisfied for the selected path to be traversed. A number of algorithms have been used for solving these constraints [4, 5, 8]. An alternative approach, referred to as an execution-oriented approach of test data generation, for finding program input for the selected path was proposed in [10]. This approach is based on actual execution of a program under test and function minimization methods. The goal of finding a program input is achieved by solving a sequence of sub-goals, where function minimization methods [10] are used to solve these sub-goals. The major weakness with the existing path-oriented test data generators is in the path selection where significant computational effort can be wasted in exploring non-executable paths that are generated.

The goal-oriented approach of test data generation [11] differs from the path-oriented test data generation in that the path selection stage is eliminated. The approach starts by initially executing a program with arbitrary program input. When the program is executed, the program execution flow is monitored. During the program execution, the search procedure decides whether the execution should continue through the current branch or an alternative branch should be taken because, for example, the currently executed branch does not lead to the execution of the selected node. If an undesirable execution flow at the current branch is observed, then a real-valued function is associated with branch. Function minimization search algorithms are used to find automatically new input that will change the flow of execution at this branch.

The goal-oriented approach was extended to an approach, referred to as a chaining approach, in which data flow analysis is used to guide the search process [7]. The chaining approach starts by executing a program for an arbitrary program input $\boldsymbol{x}$. If an undesirable execution flow at some branch (p,q) is observed (because, for instance, branch (p,q) does not lead to the target node), then a real-valued function is associated with this branch. Function minimization search algorithms are used to find automatically new input that will change the flow of execution at this branch. If, at this point, the search process cannot find program input $\boldsymbol{x}$ to change the flow of execution at branch (p,q), then the chaining approach attempts to alter the flow of execution at node p by identifying nodes that have to be executed prior to reaching node p. As a result, the alternative branch at p may be executed. We refer to node p of the branch (p,q) as a *problem* node.

The chaining approach finds a set of last definitions of all variables used at problem node p, in order to identify program nodes that have to be executed prior to the execution of node p. Such a sequence of nodes to be executed is referred to as an *event sequence*, and it is used to control the execution of the program by the chaining approach.

The problem of finding input data in the chaining approach is reduced to a sequence of sub-goals where each sub-goal is solved using function minimization search techniques that use branch predicates to guide the search process. Each branch (p,q) in the flow graph is labeled by a predicate, referred to as a *branch predicate*, describing the conditions under which the branch will be traversed. There are two types of branch predicates: Boolean and arithmetic predicates. Boolean predicates involve Boolean variables. On the other hand, arithmetic predicates are of the following form: A_1 *op* A_2, where A_1 and A_2 are arithmetic expressions, and *op* is one of $\{<, \leq, >, \geq, =, \neq\}$. Each branch predicate A_1 *op* A_2 can be transformed to the equivalent predicate of the form: *F rel* 0. *F* is a real-valued function, referred to as a *fitness function*, which is (1) positive (or zero if rel is <) when a branch predicate is false or (2) negative (or zero if rel is = or ≤) when the branch predicate is true. Clearly, *F* is a function of program input ***x*** (more detailed discussion about the construction of a fitness function can be found in [7, 10]). For example, a fitness function for the true-branch of node 20 in the program of Figure 1 is: 100-*top* < 0. The fitness function is evaluated for any program input by simply executing the program.

Several automated test generation methods based on genetic and evolutionary algorithms [9, 12, 14, 16] have been proposed. These methods are in the category of heuristic-global optimization techniques and are designed to find good approximations to the optimal solution in large complex search spaces. These methods are quite promising. However, the major problem with these methods is their performance and their sensitivity on a selection of good fitness functions.

3. Data-dependence based search

The existing test data generation methods use different types of information about a program to guide the search process, e.g., a control flow graph, control dependencies, data flows, etc. Most methods use a program control flow graph to guide the search process. However, control information about a program is frequently not sufficient to efficiently guide the search. The search may be guided to parts of the program that have nothing to do with finding the solution. As a result, a lot of search effort may be wasted in these parts of the program. Therefore, the search needs another type of information that can be used to guide the search. In this paper, we propose to use data dependence analysis to guide the search process. In particular, the search uses the data dependence graph to identify promising computations in the program that lead to the solution.

The presented approach is used when the existing test generation methods have difficulty, or fail, to find a solution. The search failure is typically caused by a relatively small number of test nodes (predicates of conditional statements) where the search is not able to find an input to execute their branches that lead to the target node. The idea is to first identify a "problem" node(s) using the existing test generation methods, i.e., a test node(s) where the existing methods fail to find input to traverse a "desired" branch of the test node, before using the presented approach.

For example, consider a C function of Figure 1. Suppose the goal is to find an input to execute statement 21 that is a target node. The existing methods of test generation have difficulties finding the solution to this test generation problem. For example, path-oriented methods require over 10^{101} paths of increased size to be explored before the solution is found whereas the goal-oriented approach and the chaining approach fail to find the solution. By using the existing test generation methods (e.g., [7, 11]), node 20 (a predicate of an if-statement) is identified as a problem node because it is difficult to find an input on which the "desired" true-branch of this node is taken to reach the target node. Notice that in order to execute node 21, node 13 has to be executed 101 times before reaching node 20.

In order to guide the search to find the solution, we propose to use data dependence analysis during the search. In what follows we introduce the notion of data dependence and then introduce the concept of a data dependence graph. Note that in this paper we use a statement and a node interchangeably. Data dependence captures the notion that one statement (node) assigns a value to a variable and another statement may potentially use this value. More formally, there exists *data dependence* between statements n_i and n_k with respect to variable *v* if: (1) *v* is assigned a value at n_i, (2) *v* is used in n_k, and (3) there exists a path in the program from n_i to n_k along which *v* is not modified. For example, in the function of Figure 1 there exists a data dependence between nodes 13 and 20 because node 13 assigns a value to variable *top*, node 20 uses variable *top*, and there exists a control path (13, 14, 15, 19, 23, 27, 6, 7, 8, 9, 15, 19, 20) from 13 to 20 along which variable *top* is not modified. Notice that the shortest path from 13 to 20 (i.e., 13, 14, 15, 19, 20) is not executable.

Data dependencies in the program can be graphically represented by a graph, referred to as a *data dependence graph*, where nodes represent statements and directed arcs represent data dependences. The data dependence graph of the program is used to derive a data dependence sub-graph for the problem node. Such a sub-graph contains nodes and dependencies of the data dependence graph for which there exists a path in the data dependence graph to the problem node. For example, Figure 2 shows a data dependence sub-graph with respect to problem node 20 of the program of Figure 1. This data dependence sub-graph contains nodes that may influence the problem node. This data dependence sub-graph is used during the search to explore different ways in which the problem node may be affected.

```
1       void F(int A[], int C[]) {
          int AR[100];
          int a, i, j, cmd, top, f_exit;
2         i=1;
3         j = 1 ;
4         top = 0 ;
5         f_exit=0;
6         while (f_exit==0) {
7               cmd = C[j] ;
8               j = j + 1 ;
9               if (cmd == 1) {
10                    a = A[i] ;
11                    i = i + 1 ;
12                    if (a > 0) {
13                          top = top + 1 ;
14                          AR[top] = a ;
                      };
                };
15              if (cmd == 2) {
16                    if (top>0) {
17                          write(AR[top]);
18                          top = top - 1 ;
                      };
                };
19              if (cmd==3) {
20,21                 if (top>100) {write(1);}
22                    else write(0);
                };
23              if (cmd==4) {
24,25                 if (top==0) {write(1);}
26                    else {write(0);};
                };
27,28           if (cmd>=5) f_exit=1;
          }; //endwhile
29      }
```

Figure 1. A sample C function

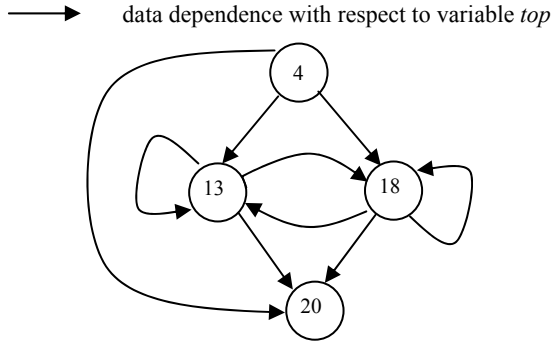


Figure 2. Data-dependence sub-graph

Suppose a data dependence sub-graph contains only data dependences related to only one variable *v*, e.g., a data dependence sub-graph of Figure 2 contains only data dependences related to variable *top*. In such a case, a *data dependence path* is a path in the data dependence graph from a node that has no incoming dependences to it and ending at the problem node. A data dependence path for multi-variable dependences is discussed in Section 5. In this paper we refer to a data dependence path as a *path*. For example, the following is a path in the data dependence sub-graph of Figure 2: 4, 13, 18, 20.

```
1     float TransFunc(int PathSize, int S[], int R[]) {
        int i, j, top;
2       i=1;
3       while (i<=PathSize) {
4         switch (S[i]) {
5           case 4: {top = 0;            // 4
6                   break; }
7           case 13: {top = top + 1;   // 13
8                   for (j=1;j<R[i];j++) top = top + 1;
9                   break; }
10          case 18: {top = top - 1;   // 18
11                  for (j=1;j<R[i];j++) top = top - 1;
12                  break; }
13        }
14        i++;
15      };
16       return 100-top; //computation of the fitness function at node 20
      }
```

Figure 3. A transformed function of the C function of Figure 1 with respect to problem node 20

There are different strategies of generating paths for exploration in the data dependence sub-graph. The simplest strategy is to explore initially all acyclic paths in the dependence sub-graph to the problem node. If the solution is not found, then paths of increased size are generated until the solution is found or designed resources are exhausted. When exploring a selected path, the search tries to execute the program under test in such a way that nodes listed in the path are executed in the same order. In addition, for each node in the path variable *v* is not modified during program execution before reaching the next node in the path. In this paper, we use the chaining approach [7] to explore selected data dependence paths. Each path corresponds to a chain in the chaining approach.

For example, for the problem node 20 of the function of Figure 1, five acyclic paths in the data dependence sub-graph of Figure 2 are generated:

P_1: 4, 20	P_2: 4, 18, 20	P_3: 4, 13, 20
P_4: 4, 13, 18, 20	P_5: 4, 18, 13, 20	

For path P_3 = 4, 13, 20, it is expected that the execution reaches node 4, then execution reaches node 13 without modifying variable *top*, and finally the execution reaches node 20 without modifying variable *top*. For all acyclic paths listed above, the search fails to find the solution. Therefore, paths of increased size are generated and explored. This search continues until a path 4, 13, 13, …, 13, 20 in which node 13 occurs 101 times is generated. For such a path the chaining approach can find the solution to execute node 21.

Data-dependence based search works well for many programs. However, the effectiveness of the search depends on the way paths are generated [17] and selected for exploration. For example, over 101 data dependence paths [17] need to be explored to find a test data to execute node 21 in the function of Figure 1.

4. Testability Transformation

A testability transformation [18] is a source-to-source program transformation that seeks to improve the performance of test data generation technique. Using testability transformation, test data is generated from the transformed program, though it is to be applied to the original (untransformed) program. Testability transformations differ from traditional transformations; they are used solely to assist with testing. Once test data has been generated, the transformed program can be discarded. This has the important consequence that the transformed program needs not preserve the meaning of the original program, nor needs it even preserve some abstract interpretation of the meaning of the original program.

In this paper, transformation produces a version of the program that is clearly non-meaning preserving. However, this is not a problem as it would be with conventional transformation. If the transformation helps us to generate test data then we generate the test data and throw away the transformed program. We only resort to testability transformation when all else has failed, so we have little to lose if it does not help.

Compare this situation with conventional program transformation, where the transformed program must be equivalent to the original from which it is constructed and where the equivalence question is, of course, undecidable. For testability transformation, the decision problem is simply: "do we cover the aspect of interest in the original program, when it is executed using the test data generated from the transformed program?". This is clearly decidable.

In our approach we use a testability transformation that transforms a program under test to a format that is more appropriate for data-dependence based test generation. By using a testability transformation, the chances of finding a solution are increased when the existing methods fail using only the untransformed program, and, in addition, it is much easier to explore different data dependence paths in the transformed program than in the untransformed program. As a result, the search may find a solution more efficiently. The major idea of the testability transformation is to create a transformed program that is used to efficiently evaluate the fitness function associated with the given problem node. The transformed program contains only statements that are responsible for computation of the fitness function. The major advantage of the transformed program is that it is easy to execute any statement that affects the fitness function. As a result, the transformed program allows exploring efficiently different data dependence paths in order to identify paths that lead to the target value of the fitness function.

For a given problem node and the fitness function associated with this node, a data dependence graph is constructed to identify statements that affect computation of the fitness function, and these statements are included in the transformed program, which is used to explore different data dependence paths. If the fitness function evaluates to the target value for a given path in the transformed program, the path is considered a promising path (in many cases, a partial program input may be also generated). If, on the other hand, the fitness function fails to evaluate to the target value in the transformed program, the given path is considered as unpromising and it is rejected. Identified promising paths in the transformed program are then used to guide the search in the untransformed program to find the final solution, i.e., an input on which the target statement is executed in the untransformed program.

The transformed program, e.g., a transformed program of Figure 3, consists of statements that are part of the data dependence sub-graph related to the problem node. The main input to the transformed program is the data dependence path represented as array *S* and integer *PathSize* that indicates the length of the path. Array *S* contains a sequence of integers indicating *ids* of statements that are in the data dependence path. The transformed program executes statements according to array *S*. In particular, the transformed program contains a while-loop and a switch-statement inside of it. The while-loop is used to execute all statements as indicated in array *S*. The switch-statement is used to select a statement to be executed according to *S*. Each statement that belongs to the data dependence sub-graph is included in the transformed program in a case-statement of the switch-statement. A case-statement contains an integer indicating the statement's id. The statement itself is included in the body of the case-statement. The number of iterations of the while-loop is determined by the value of *PathSize*.

Some nodes in the data dependence sub-graph have self-looping data dependencies. In order to explore an influence of self-looping data dependences on the fitness function, the corresponding nodes need to be repeatedly executed. Therefore, we introduce another input (array *R* of integers) to the transformed program that indicates a number of repetitions of executions of statements that have self-looping data dependences. If S[i] indicates *id* of a statement to be executed, R[i] indicates a number of repetitions of this statement. If a statement contains a self-looping data dependence, a for-loop with a counter R[i] is added in the corresponding case-statement of the transformed program. This loop is used to repeat multiple executions of the statement. A transformed program is represented as a function (a transformed function) that has input parameters: *S*, *PathSize*, and *R*. In addition, some input parameters of the program under test may also be included. The transformed function returns the value of the fitness function that is associated with the problem node.

An algorithm to construct a transformed program is presented below:

for each input variable *v* of the program under test **do**
  if *v* is used by a statement of the data dependence sub-graph
  then include *v* as an input variable of the transformed program
endfor
for each node *n* in the data dependence sub-graph **do**
  if *n* does not have a self-looping data dependence
  then include the following statement in the switch-statement:
    case *id_of_n*: { *text_of_n*; break; }
  if node *n* has a self-looping data dependence
  then include the following statement in the switch-statement:
    case *id_of_n*: { *text_of_n*;
      for (j=1;j<R[i];j++) *text_of_n*;
      break; }
endfor
Include an expression computing the fitness function in the return statement of the transformed program

where,
id_of_n is an unique id (integer) associated with statement *n*
text_of_n is the text of statement *n* in the main function

A transformed function for the function of Figure 1 and the problem node 20 is shown in Figure 3. The transformed function contains all statements that are part of the data dependence sub-graph of Figure 2, i.e., statements 4, 13, and 18. These statements affect computation of the fitness function associated with problem node 20. For statements 13 and 18, for-loops are included because these statements have self-looping data dependences in the data dependence sub-graph. The transformed function does not have in its interface any input parameters of the function of Figure 1 because none of these input parameters is used by statements 4, 13 and 18.

After the transformed function is constructed, the search generates different data dependence paths and explores them on the transformed function. If the fitness function evaluates to the target value for a given path in the transformed function, the path is considered a promising path. Otherwise, the path is considered as unpromising and it is rejected. Identified promising paths in the transformed program are then used to guide the search in the untransformed program to find the final solution.

When the search generates a data dependence path for exploration, the path is represented by *S* and *PathSize*. This is the main input to the transformed function. For a given *S* and *PathSize*, the goal is to find values for array *R* and eventually input parameters of the untransformed function when included in the transformed function, such that the fitness function evaluates to the target value. The search uses the existing test generation techniques [7, 12, 16] to find an input on which the fitness function evaluates to the target value in the transformed function.

For example, for the problem node 20 of the function of Figure 1, five acyclic paths in the data dependence sub-graph of Figure 2 may be initially generated:

P_1: 4, 20	S[1]=4; PathSize=1
P_2: 4, 18, 20	S[1]=4; S[2]=18; PathSize=2
P_3: 4, 13, 20	S[1]=4; S[2]=13; PathSize=2
P_4: 4, 13, 18, 20	S[1]=4; S[2]=13; S[3]=18; PathSize=3
P_5: 4, 18, 13, 20	S[1]=4; S[2]=18; S[3]=13; PathSize=3

All these paths do not result in the solution when using data dependence search presented in the previous section. Suppose the transformed function of Figure 3 is used to identify promising paths. For each path, the corresponding inputs *S* and *PathSize* to the transformed function are shown at the right side. For each path, the goal is to find values for elements of input array *R* such that the value of the fitness function returned by the transformed function is negative. The existing test generation methods [7, 12, 16] are used in this search. The search fails for paths P_1 and P_2. These paths are considered as unpromising and are rejected. When path P_3 is explored (S[1]=4; S[2]=13; PathSize=2), the search finds values to elements of array *R* (R[1]=1; R[2]=101) on which the fitness function evaluates to the negative value in the transformed function. Therefore, path P_3 with 101 repetitions of node 13 is considered as a promising path. When this path is used to guide the search in the function of Figure 1, an input is easily identified for which the target node 21 is executed.

Using the transformed function of Figure 3, it is possible to find a solution only with at most five explorations of data dependence paths as opposed to over one hundred of path explorations when the transformed function is not used.

5. A case study for programs with multiple-variable data dependences

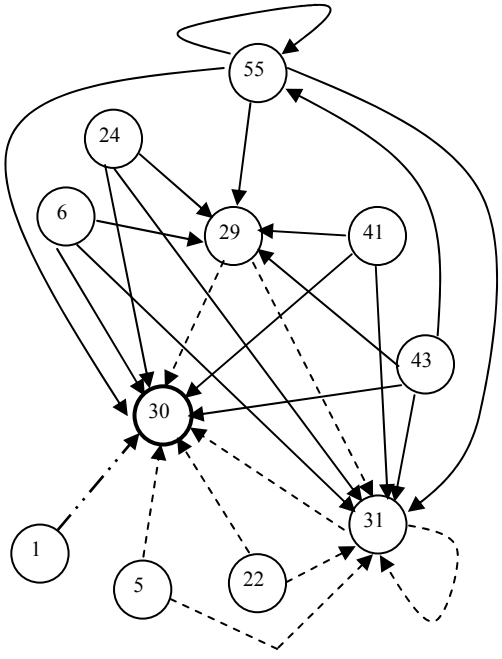
In this section we present an extension of the presented approach and a case study for programs with data dependence sub-graphs with multiple-variable data dependences. Consider a C function of Figure 4. This function accepts as an input three inputs: integers *iTextSize* and *maxpos*, and array of characters *A*. There is a precondition imposed on the input: $maxpos \geq 0$. The goal of the test data generation is to find an input on which target statement 33 is executed. The existing test generation methods have difficulties or fail to find the solution. For example, path-oriented methods require an extremely large number of paths (over 10^{10} paths) to be explored before the solution is found whereas the goal-oriented approach and the chaining approach fail to find the solution. The existing methods identify node 30 as a problem node. The following fitness function is associated with the true-branch of this problem node:

FitnessFunction = maxpos - (linepos + wordlen + 1)

Precondition: maxpos ≥ 10

```
1     void format(int iTextSize, char *A, int maxpos)
      {int i, j, linepos, wordlen, curpos, firstblank, exitflag;
      char ch1, ch;
2     j=0;
3     i = 0;
4     ch = A[i];
5     linepos = 0;
6     wordlen = 0;
7     firstblank = 1;
8     exitflag=0;
9     while((ch != 27) && (i < iTextSize)) {
10        if(ch == 10) {
11            if(linepos > 0) {
12,13             if((linepos+wordlen+1)<=maxpos) printf(" ");
14                else printf("\n");
              }
15            j = 1;
16            while(j <= wordlen) {
17                ch1 = A[curpos];
18                curpos ++;
19                printf("%c", ch1);
20                j++;
              }
21            printf("\n");
22            linepos = 0;
23            firstblank = 1;
24            wordlen = 0;
          }
          else {
25            if(ch == ' ') {
26                if(firstblank == 1) {
27                    firstblank = 0;
28,29               if(linepos == 0) linepos = wordlen;
                    else {
30                      if((linepos + wordlen + 1) <= maxpos) {
31                          linepos += wordlen + 1;
32                          printf(" ");}
                        else {
33                          linepos = wordlen;
34                          printf("\n");
                        }
                    }
                }
35              j = 1;
36              while(j <= wordlen) {
37                  ch1 = A[curpos];
38                  curpos ++;
39                  printf("%c",ch1);
40                  j ++;
                }
41              wordlen = 0;
            }
            else {
42              curpos = i;
43              wordlen = 1;
44              exitflag = 0;
45              firstblank = 1;
46              while(exitflag != 1) {
47                  i ++;
48                  ch = A[i];
49,50               if(ch == 10) exitflag = 1;
                    else {
51,52                   if (ch == 27) exitflag = 1;
                        else {
53,54                       if(ch == ' ') exitflag = 1;
55                          else wordlen ++;
                        }
                    }
                }
56              if(wordlen == 0) {
57                  i ++;
58                  ch = A[i];
                }
            }
59      if(wordlen > 0) {
60,61       if((linepos+wordlen+1)<=maxpos) printf(" ");
62          else printf("\n");
63          j = 1;
64          while(j <= wordlen) {
65              ch1 = A[curpos];
66              curpos ++;
67              printf("%c",ch1);
67              j++;
            }
        }
}
```

Figure 4. A sample C function



Note: node 33 is not included because it is a target node.

Figure 5. Data-dependence sub-graph for node 30

In order to identify nodes that affect the problem node, a data dependence sub-graph is constructed with respect to the problem node 30 and is shown in Figure 5. Notice that this data dependence sub-graph contains data dependences with respect to three variables: *linepos*, *worldlen*, and *maxpos*. Based on this data dependence graph the transformed function is constructed and it is shown in

Figure 6. For nodes 31 and 55 a for-loop is inserted into the corresponding case-statements because both nodes have a self-looping data dependences. In addition, the interface of the transformed function contains input variable *maxpos* because there is a data dependence between node 1 and 30 with respect to *maxpos* in the sub-graph of Figure 5.

In the next step the search should generate data dependence paths from the data dependence sub-graph. These paths are explored on the transformed function. In the case of a single-variable dependences in the data dependence sub-graph, a path is a simple path in the data dependence graph. However, in a case of data dependences with multiple variables, a data dependence path is a graph, referred to as a *data dependence execution graph*. A data dependence execution graph can be viewed as a dynamic data dependence graph [19]. This graph represents possible data dependences between executed statements that may occur during program execution. Each node represents a possible statement execution and data dependences between these nodes represent data dependences that occur between executed statements. For example, Figure 7 shows a sample data dependence execution graph derived from the data dependence sub-graph of Figure 5. This graph shows that statements 43 and 55 are executed twice and statement 29 is executed once and shows data dependences that may occur during program execution. This graph represents a possible execution in the program of Figure 4 that influence problem node 30. Data dependence execution graphs can be derived from the data dependence sub-graph by traversing backwards from the problem node. An algorithm to generate data dependence execution graphs from the data dependence sub-graph is presented in Figure 8.

Data dependence based search generates data dependence execution graphs and then explores them in the program. Each execution graph represents a different way the fitness function associated with a problem node may be computed. Since exploration of execution graphs in the original program may be expensive, we use the transformed program to explore execution graphs in order to identify execution graphs on which the fitness function evaluates to the target value in the transformed program. The identified execution graphs are then used to guide the search in the original program to find the final solution.

For each data dependence execution graph, a sequence of node executions in the execution graph is generated. This execution sequence represents a possible sequence of executions of nodes in the execution graph in such a way that all data dependences in the execution graph are preserved. There may be many execution sequences for a given execution graph. However, it is sufficient to explore only one sequence in the transformed program. A possible execution sequence for the execution graph of Figure 7 is: 1, 43, 55, 29, 43, 55, 30. Notice that the following sequence 1, 43, 55, 43, 55, 29, 30 is not a valid execution sequence because a data dependence between the first execution of node 55 and 30 is lost.

An execution sequence associated with an execution graph under exploration becomes an input, *S* and *PathSize*, to the transformed program. For a given *S* and *PathSize*, the goal is to find values of array *R* and eventually input parameters of the untransformed function (if included in the transformed function), so that the fitness function evaluates to the target value. The search uses the existing test generation techniques [7, 12, 16] to find an input on which the fitness function evaluates to the target value. Notice that array R is used to explore possible repetitions of dependences for nodes with self-looping data dependences.

```
float TransFunc(int maxpos, int PathSize, int S[], int R[]) {
int i;
int linepos, wordlen;
i=1;
while (i<=PathSize) {
      switch (S[i]) {
          case 5:  {linepos = 0;                  //5
                   break; }
          case 6:  {wordlen = 0;                  //6
                   break; }
          case 22: {linepos = 0;                  //22
                   break; }
          case 24: {wordlen = 0;                  //24
                   break; }
          case 29: {linepos = wordlen;            //29
                   break; }
          case 31: {linepos += wordlen + 1;       //31
                   for (j=1;j<R[i];j++) linepos+= wordlen+1;
                   break; }
          case 41: {wordlen = 0;                  //41
                   break; }
          case 43: {wordlen = 1;                  //43
                   break; }
          case 55: {wordlen ++;                   //55
                   for (j=1;j<R[i];j++) wordlen ++;
                   break; }
      }
      i++;
};
return maxpos - (linepos + wordlen + 1);
}
```

Figure 6. A transformed function of the C function of Figure 4

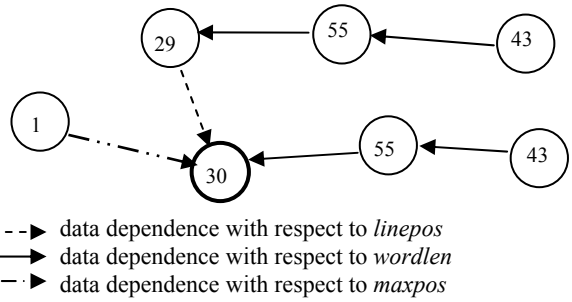


Figure 7. A sample data dependence execution graph

For example, when an execution graph of Figure 7 is explored, the following execution sequence is provided to the transformed function of Figure 6:

PathSize=5; S[1]=43;S[2]=55;S[3]=29;S[4]=43;S[5]=55;

The goal for the search is to find values for *maxpos* and elements of array *R* so that the transformed function returns the target value of the fitness function, i.e., a negative value. The search can easily find the following solution:

maxpos=10; R[2]=7; R[5]=1;

on which the transformed function returns a negative value.

Therefore, the execution graph of Figure 7 and the corresponding execution sequence with 7 repetitions of node 55 is considered as a promising execution graph. When this graph is used to guide the search in the function of Figure 4, an input is easily identified for which the target node 33 is executed.

Input: Problem node *n*
Data dependence sub-graph
Maximum size of execution graphs *MaxSize*
Output: Execution graphs *EG* of size less or equal *MaxSize*

EG, EG_1 execution graphs
$U(x)$ a set of variables used in node x
$LD(n,v)$ a set of nodes that have a data dependence on node n with respect to variable v

```
procedure expand (EG, n)
1    if number of nodes in EG is greater than MaxSize then exit
2    if for all nodes x in EG for which there is no incoming
         dependence (arc) to x in EG and U(x) = ∅ then
3       output EG
4       exit
5    endif
6    for every c ∈ LD(n,v1) × LD(n,v2) × … × LD(n,vm),
         where m=|U(n)| do
7       Let c = <x1, x2, …, xm>
8       EG1 = EG
9       for every node xi in c do
10         Add node xi into EG1
11         Insert data dependence between xi and n w.r.t. vi
12      endfor
13      for every node x in c do
14         if U(x) ≠ ∅ then expand(EG1, x)
15      endfor
16   endfor
end expand

17   Set execution graph EG as empty
18   Add a problem node p to EG
19   expand (EG, p)
```

Figure 8. An algorithm to generate execution graphs

The algorithm to generate execution graphs is presented in Figure 8. The algorithm generates execution graphs of sizes limited by *MaxSize*, where the size is determined by a number of nodes in the execution graph. After inserting a problem node into an empty execution graph in lines 17 and 18, the algorithm invokes *expand* procedure. In this procedure, the size of the current execution graph is checked for its size in line 1. If the size exceeds the limit, the execution graph is rejected. In lines 2-3, the procedure checks whether the current execution graph has some nodes for expansion. Notice that node x in the execution graph can be expanded if x does not have any incoming edges and x uses at least one variable. If there are no more nodes for expansion, the execution graph is outputted in line 3 for exploration. In the for-loop in lines 6-16, node n is expanded by adding all combinations of nodes related to data dependences of different variables used in n. These nodes, each related to a variable used in n, are added to the execution graph and corresponding data dependences are inserted between these nodes and n. In the for-loop in lines 13-15, each newly added node is expanded (if possible).

For each execution graph an execution sequence is generated. The execution sequence must preserve all dependences that do exist in the execution graph. This sequence is then used on the transformed function to determine whether this is a promising sequence (execution graph). If no execution sequence is found, the execution graph is discarded and not explored. For example, consider a sample execution graph of Figure 9 generated from the data dependence sub-graph of Figure 5. For this execution graph it is not possible to create an execution sequence such that all data dependences are preserved. The problem of finding an execution sequence in the execution graph is equivalent to the problem of lining-up nodes of a graph with colored arcs in such a way that for any two consecutive nodes the intersection of arcs between these nodes does not contain two arcs with the same color that originate from different nodes. The algorithm to generate an execution sequence from an execution graph is not presented in this paper because of space limitations.

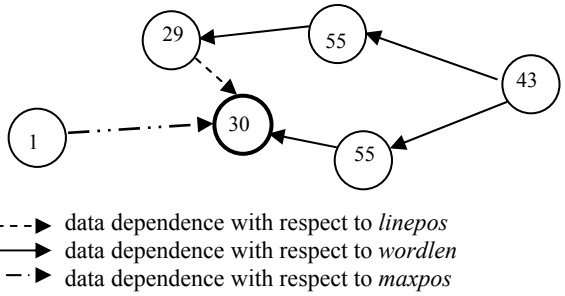


Figure 9. A sample data dependence execution graph

When the approach was used to find a test data to execute node 33 of the function of Figure 4, it efficiently identified the solution by exploring less than 88 execution graphs in the transformed function of Figure 6 before the solution was found. Notice that each exploration in the transformed function is very inexpensive because the search does not have any difficulty executing statements in

orders specified in execution sequences. During the search using the transformed function, all explored execution graphs were identified as unpromising ones except the execution graph of Figure 7 that led to the solution. On the other hand, exploring execution graphs in the untransformed function of Figure 4 is very expensive because the complex logic of this function.

6. Conclusions

In this paper, we presented an approach that uses data dependence analysis to form novel transformations of the original program that can be used to guide the search process in test generation. In particular, we use a data dependence graph to explore different computations, represented as data dependence paths and execution graphs, that affect the solution. The presented testability transformation transforms programs so that the chances of finding a solution are increased when the existing methods fail using only the untransformed program. In our approach data dependence analysis is used to identify statements in the program that affect computation of the fitness function associated with the problem statement. The transformed program contains only these statements. The transformed program is constructed in such a way that it makes it much easier to explore different ways the fitness function may be computed. As a result, the search can find the solution much faster by identifying promising executions that are then used to guide the search in the untransformed program to find the final solution. The primary contributions of this paper are as follows:

1. It is shown how data dependence analysis can be used to guide speculative transformations to improve testability.
2. The approach is evaluated using a case study which demonstrates that the transformations can improve the test data generation.
3. The transformations employed do not preserve the meaning of the program, yet it is shown that this is unimportant in the context of test data generation. This is an interesting and, perhaps, somewhat surprising finding.

The presented approach has been incorporated into our test generation environment TESTGEN [7]. The initial experience with the presented approach shows that it may effectively and efficiently find solutions for many programs for which the existing methods fail or have difficulties in finding the solution. In the future, we plan to perform an experimental study to investigate the effectiveness and efficiency of the presented approach in automated test generation for commercial software.

7. References

[1] A. Baresel, H. Sthamer, "Evolutionary testing of flag conditions," Proc. Genetic and Evolutionary Computation Conference, 2003, pp. 2442-2454.

[2] D. Bird, C. Munoz, "Automatic generation of random self-checking test cases," IBM Systems J., 22(3), 1982, pp. 229-245.

[3] B. Boehm, R. McClean, D. Urfig, "Some experience with automated aids to the design of large-scale reliable software," Proc. Intern. Conference on Reliable Software, 1975, pp. 105-113.

[4] R. Boyer, B. Elspas, K. Levitt, "SELECT - A formal system for testing and debugging programs by symbolic execution," SIGPLAN Notices, 10(6), 1975, pp. 234-245.

[5] L. Clarke, "A system to generate test data and symbolically execute programs," IEEE Transactions on Software Engineering, 2(3), 1976, pp. 215-222.

[6] R. DeMillo, A. Offutt, "Constraint-based automatic test data generation," IEEE Transactions on Software Engineering, 17(9), 1991, pp. 900-910.

[7] R. Ferguson, B. Korel, "Chaining Approach for Automated Test Data Generation," ACM Tran. on Software Eng. and Methodology, 5(1), 1996, pp. 63-86.

[8] W. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," IEEE Transactions on Software Engineering, 4(4), 1977, pp. 266-278.

[9] B. Jones, et al., "Automatic structural testing using genetic algorithms," Software Eng. Journal, 11(5), 1996, pp. 299-306.

[10] B. Korel, "Automated test data generation," IEEE Transactions on Software Engineering, 16(8), 1990, pp. 870-879.

[11] B. Korel, "Dynamic method for software test data generation," Journal of Software Testing, Verification, and Reliability, 2, 1992, pp. 203-213.

[12] C. Michael, et al., "Generating software test data by evolution," IEEE Tran. on Software Engineering, 27(12), 2001, pp. 1085-1110.

[13] P. Mcminn, M. Holcombe, "The state problem for evolutionary testing," Proc. Genetic and Evolutionary Computation Conference, 2003, pp. 2488-2498.

[14] R. Pargas, et al., "Test data generation using genetic algorithms," Journal of Software Testing, Verification, and Reliability, 9, 1999, pp. 263-282.

[15] C. Ramamoorthy, S. Ho, W. Chen, "On the automated generation of program test data," IEEE Transactions on Software Engineering, 2(4), 1976, pp. 293-300.

[16] J. Wegener, et al., "Evolutionary test environment for automatic structural testing," Information and Software Technology, 43, 2001, pp. 841-854.

[17] B. Korel, S. Chung, P. Apirukvorapinit, "Data dependence analysis in automated test generation," Proc. 7th IASTED Intern. Conf. on Software Eng. and Applications, 2003, pp. 476-481.

[18] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, M. Roper, "Testability Transformation," IEEE Transactions on Software Engineering. 30(1), 2004, pp. 3-16.

[19] B. Korel, J. Laski, "Dynamic Program Slicing," Information Processing Letters, vol. 3, No. 3, 1988, pp. 155-163.

[20] P. McMinn, "Search-Based Software Test Data Generation: A Survey," Software Testing, Verification and Reliability, vol. 14, 2004, pp. 105-156.