Testability Transformation for Efficient Automated Test Data Search in the Presence of Nesting

Phil McMinn
University of Sheffield,
Regent Court,
211 Portobello Street,
Sheffield, S1 4DP, UK
p.mcminn@dcs.shef.ac.uk

David Binkley
Loyola College
4501 North Charles Street
Baltimore,
MD 21210-2699, USA
binkley@cs.loyola.edu

Mark Harman
King's College
Strand, London
WC2R 2LS, UK
mark@dcs.kcl.ac.uk

Abstract

The application of metaheuristic search techniques to the automatic generation of software test data has been shown to be an effective approach for a variety of testing criteria. However, for structural testing, the dependence of a target structure on nested decision statements can cause efficiency problems for the search, and failure in severe cases. This is because all information useful for guiding the search - in the form of the values of variables at branching predicates - is only gradually made available as each nested conditional is satisfied, one after the other. The provision of guidance is further restricted by the fact that the path up to that conditional must be maintained by obeying the constraints imposed by 'earlier' conditionals.

An empirical study presented in this paper shows the prevalence of types of `if` statement pairs in real-world code, where the second `if` statement in the pair is nested within the first. A testability transformation is proposed in order to circumvent the problem. The transformation allows all branch predicate information to be evaluated at the same time, regardless of whether 'earlier' predicates in the sequence of nested conditionals have been satisfied or not. An experimental study is then presented, which shows the power of the approach, comparing evolutionary search with transformed and untransformed versions of two programs with nested target structures. In the first case, the evolutionary search finds test data in half the time for the transformed program compared to the original version. In the second case, the evolutionary search can only find test data with the transformed version of the program.

1 Introduction

The application of metaheuristic search techniques to the automatic generation of software test data has been shown to be an effective approach for functional [11, 21, 20], non-functional [26, 19, 27], structural [12, 13, 4, 29, 10, 17, 25, 16, 15], and grey-box [14, 24] testing criteria. The search space is the input domain of the test object. An objective function provides feedback as to how 'close' input data are to satisfying the test criteria. This information is used to provide guidance to the search.

For structural testing, each individual program structure of the coverage criteria (for example each individual program statement or branch) is taken as the individual search 'target'. The effects

```
Node
        void example(int a, int b, int c, int d)
        {
(1)       if (a > b)
          {
(2)         if (b > c)
            {
(3)           if (c > d)
              {
(4)             // target
        ...
```

Figure 1: Nested targets require the succession of branching statements to be evaluated by the objective function one after the other

of input data are monitored through instrumentation of the branching conditions of the program. An objective function is computed, which decides how 'close' an input datum was to executing the target, based on the values of variables appearing in the branching conditionals which lead to its execution. For example, if a branching statement '`if (a == b)`' needs to be true for a target statement to be covered, the objective function feeds back a 'branch distance' value of $abs(b - a)$ to the search. The objective values fed back are critical in directing the search to potential new test data candidates which might execute the desired program structure.

However, the search can encounter problems when structural targets are nested within more than one conditional statement. In this case, there are a succession of branching statements which must be evaluated with a specific outcome in order for the target to be reached. For example, in Figure 1, the target is nested within three conditional statements. Each individual conditional must be true in order for execution to proceed onto the next one. Therefore, for the purposes of computing the objective function, it is not known that `b > c` must be true until `a > b` is true. Similarly, until `b > c` is satisfied, it is not known that `c > d` must also be satisfied. This gradual release of information causes efficiency problems for the search, which is forced to concentrate on satisfying each predicate individually. For example, inputs where `b` is close to being greater than `c` are of no consequence to the objective function until `a > b`.

Furthermore, the search is restricted when seeking inputs to satisfy 'later' conditionals, because satisfaction of the earlier conditionals must be maintained. If when searching for input values for `b > c`, the search chooses input values so that `a` is not greater than `b`, the path taken through the program never reaches the latter conditional, and thus the search never finds out if `b > c` or not. Instead it is held up again at the first conditional, which must be made true in order to reach the second conditional again. This inhibits the test data search, and the possible input values it can consider in order to satisfy predicates appearing 'later' in the sequence of nested conditionals. In severe cases the search may fail to find test data.

Ideally, all branch predicates need to be evaluated by the objective function at the same time. This paper presents a testability transformation approach in order to achieve this. A testability transformation [7] is a source-to-source program transformation that seeks to improve the performance of a test data generation technique. The transformed program produced is merely a 'means to an end', rather than an 'end' in itself, and can be discarded once it has served its intermediatory purpose as a vehicle for an improved test data search.

The ability to be able to evaluate all branch predicates at the same time results in a significant positive impact in the level of guidance that can be provided to the search. This can be seen by examining the objective function landscapes of the original and transformed versions of programs. Experiments carried out using evolutionary algorithms on two case studies confirm this. In the first study, test data was found in half the number of input data evaluations for the transformed version. In the second study, the test data search was unsuccessful unless the transformed version of the program was used.

An empirical study is presented which examines `if` statement pairs occurring in forty real-world programs. In this study, the latter `if` statement of the pair is nested in the first. The results further serve to show the benefit of the proposed transformation approach. In previous work [3], a method is presented to simultaneously evaluate all nested branch conditions, but only if no further statements occur between each pair of `if` statements. The empirical study shows that this only occurs for 18% of `if` pairs, whereas the transformation approach is also potentially applicable to the additional 82% of cases.

2 Search-Based Structural Test Data Generation

Several search methods have been proposed for structural test data generation, including the alternating variable method [12, 13, 4], simulated annealing [23, 22] and evolutionary algorithms [29, 10, 17, 25, 16, 15]. This paper is interested in the application of the alternating variable method and evolutionary algorithms to structural test data generation.

2.1 The Alternating Variable Method

The alternating variable method [12], is employed in the goal-oriented [13] and chaining [4] test data generation approaches, and is based on the idea of 'local' search. An arbitrary input vector is chosen at random, and each individual input variable is probed by changing its value by a small amount, and then monitoring the effects of this on the branch predicates of the program.

The first stage of manipulating an input variable is called the *exploratory* phase. This probes the neighborhood of the variable by increasing and decreasing its original value. If either move leads to an improved objective value, a *pattern* phase is entered. In the pattern phase, a larger move is made in the direction of the improvement. A series of similar moves is made until a minimum for the objective function is found for the variable. If the target structure is not executed, the next input variable is selected for an exploratory phase.

In the example of Figure 1, the search target is the execution of node 4. Say the program is executed with the arbitrary input `(a=10, b=20, c=30, d=10)`. Control flow diverges away from the target down the false branch from node 1. The search attempts to minimize the objective value, which is formed from the true branch distance from node 1, i.e $a - b$. Exploratory moves are made around the value of `a`. A decreased value leads to a worse objective value. An increased value leads to an improved smaller objective function value. Larger moves are made to increase `a` until `a` is greater than `b`. Suppose the input to the program is now `(a=21, b=20, c=30, d=10)`. Execution now proceeds down the true branch from node 1, but diverges away down the false branch at node 2. The search now attempts to minimize the objective function `b - c` in order to execute node 2 as true. Exploratory moves around `a` have no effect on the objective function. Therefore exploratory moves are made around the values of `b`. A decreased value of `b` leads to a worse objective function value, whilst an increased value leads to execution taking the false branch at node 1 again. Therefore the search explores values around the current value of `c`. Increased values have a negative impact on the objective function, whilst decreased values lead to an improvement. Further moves are made to decrease the value of `c` until input is found which executes node 2 as true. Suppose this is `(a=21, b=20, c=19, d=10)`. Execution now proceeds directly through all branching statements to target node 4.

2.2 Evolutionary Testing

Evolutionary testing [29, 10, 17, 25, 16, 15] employs evolutionary algorithms for the test data search. Evolutionary algorithms [28] combine characteristics of genetic algorithms and evolution strategies, using simulated evolution as a search strategy, employing operations inspired by genetics and natural selection.

Evolutionary algorithms maintain a population of candidate solutions rather than just one current solution, as with local search methods. The members of the population are iteratively

recombined and mutated to in order to evolve successive generations of potential solutions. The aim is to generate 'fitter' candidate solutions within subsequent generations, which represent better candidate solutions. Recombination forms offspring from the components of two parents selected from the current population. The new offspring form part of the new generation of candidate solutions. Mutation performs low probability random changes to solutions, introducing new genetic information into the search. At the end of each generation, each solution is evaluated for its fitness, using a 'fitness' function. The fitness function can be the direct output of an objective function, or this value ranked or scaled in some way. Using fitness values, the evolutionary search decides whether individuals should survive into the next generation or be discarded.

In applying evolutionary algorithms to structural test data generation [29, 10, 17, 25, 16, 15], 'candidate solutions' are possible test data inputs. The objective function evaluates each test data input with regards to the current structural target in question. This is performed in a slightly different way to the alternating variable method. The notion of branch distance is key, but as the search does not work to iteratively improve one solution, the objective function incorporates another metric known as the *approach level* (also known as the *approximation level*) [25] to record how many nested conditionals are left unencountered by an input en route to the target.

Take the example of Figure 1 again. If some test data input reaches node 1 but diverges away down the false branch, its objective value is formed from the true branch distance at node 1, and an approach level value of '2' to indicate there are still two further branching nodes to be encountered (nodes 2 and 3). If the test data input evaluates node 1 in the desired way, its objective value is formed from the true branch distance at node 2, with the approach level value now being one. At node 3, the approach level is zero and the branch distance is derived from the true branch predicate.

Formally the objective function for a test data input is computed as follows:

$$obj\_val = approach\_level + normalize(branch\_dist) \tag{1}$$

where the branch distance *branch_dist* is normalized into the range 0-1 by the function *normalize* using the following formula [1]:

$$normalize(branch\_dist) = 1 - 1.001^{-branch\_dist} \tag{2}$$

thus ensuring the value added to the approach level is close to 1 when the branch distance is very large, and zero when the branch distance is zero.

The approach level, therefore, adds a value for each branch distance which remains unevaluated. Since these values are not known, as the path of execution through the program has meant they have not been calculated, the maximum value is added, i.e. 1 (this 'approximation' to real branch distances is why the approach level is sometimes referred to as the 'approximation level'). As will be seen in the next section, the addition of this value rather than actual branch distance can inhibit search progress.

3 Nested Search Targets

The dependence of structural targets on one or more nested decision statements can cause problems for search-based generation methods, and even failure in severe cases.

The problem stems from the fact that information valuable for guiding the search is only revealed gradually as each individual branching conditional is encountered. The search is forced to concentrate on each branch predicate one at a time, one after the other. In doing this, the outcome at previous branching conditionals must be maintained, in order to preserve the execution path up to the current branching statement. If this is not done, the current branching statement will never be reached. This restricts the search in its choice of possible inputs, narrowing the potential search space.

In case study 1 (Figure 2a), where the target of the search is node 4, the fact that `c` needs to be zero at node 3 is not known until `a == b` is true at node 1. However, in order to evaluate node

```
Node
 (s)    void case_study_1_original{double a, double b)
        {
 (1)        if (a == b)
            {
 (2)            double c = b + 1;
 (3)            if (c == 0)
                {
 (4)                // target
                }
            }
 (e)    }
```

(a) Original program

```
        void case_study_1_transformed(double a, double b)
        {
            double _dist = 0;

            _dist += branch_distance(a == b);

            double c = b + 1;

            _dist += branch_distance(c == 0);

            if (_dist == 0.0)
                // target
        }
```

(b) Transformed version of program

Figure 2: Case study 1

3 in the desired way, the constraint `a == b` needs to be maintained. If the values of `a` and `b` are not -1, the search has no chance of making node 3 true, unless it backtracks to reselect values of `a` and `b` again. However, if it were to do this, the fact that `c` needs to be zero at node 3 will be 'forgotten', as node 3 is no longer reached, and its true branch distance is not computed.

This phenomenon is captured in a plot of the objective function landscape (Figure 3a), which uses the output of Equation 1. The shift from satisfying the initial true branch predicate of node 1 to the secondary satisfaction of the true branch predicate of node 2 is characterized by a sudden drop in the landscape down to spikes of local minima. Any move to input values where `a` is not equal to `b` jerks the search up out of the minima and back to the area where node 1 is evaluated as false again. When stuck in the local minima, the alternating variable method can not alter both input variables at once. As the method will not accept an inferior solution which would place it back at node 1, it declares failure. The evolutionary algorithm, meanwhile, has to change both values of `a` and `b` in order to traverse the local minima down to the global minimum of `(a=-1, b=-1)`.

Case study 2 (Figure 4a) further demonstrates the problems of nested targets, this time with a target within three levels of nesting. This can be seen in a plot of the objective function landscape,

(a) Original program

(b) Transformed version

Figure 3: Objective function landscape for case study 1

```
Node
 (s)    void case_study2(double a, double b, double c)
        {
 (1)        double d, e;
 (2)        if (a == 0)
            {
 (3)            if (b > 1)
 (4)                d = b + b/2;
                else
 (5)                d = b - b/2;

 (6)            if (d == 1)
                {
 (7)                e = c + 2;
 (8)                if (e == 2)
                    {
 (9)                    // target
                    }
                }
            }
 (e)    }
```

(a) Original program

```
void case_study2_transformed(double a, double b, double c)
{
    double _dist = 0;
    double d, e;
    _dist += branch_distance(a == 0);

    if (b > 1)
        d = b + b/2;
    else
        d = b - b/2;

    _dist += branch_distance(d == 1);

    e = c + 2;

    _dist += branch_distance(e == 2);

    if (_dist == 0.0)
      // target
}
```

(b) Transformed version of program

Figure 4: Case study 2

(a) Original program

(b) Transformed version

Figure 5: Objective Function for case study 2, plotted where `c = 0`

seen in Figure 5a. The switch from minimizing the branch distance at node 2 to that of node 6 is again characterized by a sudden drop. Any move from a value of $a = 0$ has a significant negative impact on the objective value, as the focus of the search is pushed back to satisfying this initial predicate. In this area of the search space, the objective function has no regard for the values of `b`, which is the only variable which can affect the outcome at node 6. To select inputs in order to take the true branch from node 6, the search is constrained in the `a = 0` plane of the search space.

3.1 Related Work

Baresel *et al.* [3] consider the nested search target problem where no further statements exist between each subsequent `if` decision statement, as in the example of Figure 1. It is observed that the branch distances of each branching node can simply be measured at the 'top level', i.e. before node 1 is encountered, and simply added together for computing the objective function. However, if statements do exist between pairs of `if` statements, this solution is no longer plausible. In case study 1 (Figure 2), for example, the value of `c` at node 3 is fixed at node 2, which occurs after node 1 is executed. In case study 2 (Figure 4), the value of `d` at node 6 could be fixed at nodes 4 or 5, depending on the input value of `b`. Furthermore, the value of `e` is decided at node 7, which is nested in the true branches of nodes 6 and 2. A burning question therefore, is how often such intermediatory statements occur between `if` pairs in real-world code. Is an extended solution to the nested target problem justified?

3.2 Nesting in real world programs - an empirical study

An empirical study investigated nested `if` statement pairs for forty real-world programs. A description of each program, and its size, measured in lines of code by the tools `wc` and `sloc` can be found in Table 1.

The `if` statement pairs analyzed, for `if` statements P and Q - where Q is nested in P - followed the system dependence graph [9] pattern of the following form:

1. Q is control dependent on P
2. P is not transitively control dependent on Q

Control dependency [5] is informally defined as "for a program node I with two exits (e.g. an if statement), program node J is control dependent on I if one exit from I always results in J being executed, while the other exit may not result in J being executed". Rules 1 and 2, therefore, ensure that Q is nested in P and that P is differentiated from Q.

As outlined in the previous section, the issue of a possible statement sequence A existing between P and Q is an important feature which distinguishes this work from the earlier work of Baresel *et al.* [3]. Such occurrences were checked by the following rules:

3. A (if it exists) depends on some X which is control dependent on P (i.e. A depends on something nested in P)
4. A (if it exists) is not transitively control dependent on Q

A further fifth rule checked if A has a role in determining the outcome at Q, i.e. there is some variable assigned to in A that is used in the predicate at Q:

5. Q is transitively data dependent on A, and this dependency is not loop-carried

The condition that the dependency is not loop-carried ensures that if Q is data dependent on A, A does indeed occur in between P and Q, and does not merely appear after both P and Q within the body of a loop.

Table 1: Details of the real world programs

Program	LOC		Description
	`wc`	`sloc`	
a2ps	63,600	40,222	Postscript formatter
acct	10,182	6,764	Accounting package
barcode	5,926	3,975	Barcode generator
bc	16,763	11,173	Calculator
byacc	6,626	5,501	Berkeley YACC
cadp	12,930	10,620	Protocol engineering tool-box
compress	1,937	1,431	Data compression utility
copia	1,170	1,112	ESA signal processing code
csurf-pkgs	66,109	38,507	Code surfer slicing tool
ctags	18,663	14,298	Produces tags for ex, more, and vi
diffutils	19,811	12,705	File comparing routines
ed	13,579	9,046	Unix editor
empire	58,539	48,800	War game
EPWIC-1	9,597	5,719	Image compression tool
espresso	22,050	21,780	Logic simplification for CAD (from SPECmark)
findutils	18,558	11,843	File finding utilities
flex2-4-7	15,813	10,654	BSD scanner (version 2.4.7)
flex2-5-4	21,543	15,283	BSD scanner (version 2.5.7)
ftpd	19,470	15,361	File Transfer Protocol daemon
gcc.cpp	6,399	5,731	Gnu C Preprocessor
gnubg-0.0	10,316	6,988	Gnu Backgammon
gnuchess	17,775	14,584	Gnu chess game player
gnugo	81,652	68,301	Gnu go game player
go	29,246	25,665	The game go
ijpeg	30,505	18,585	JPEG compressor (from SPECmark)
indent	6,724	4,834	C formatter
li	7,597	4,888	Xlisp interpreter
ntpd	47,936	30,773	Daemon for the network time protocol
oracolo2	14,864	8,333	Array processor
prepro	14,814	8,334	ESA array pre-processing code
replace	563	512	Regular expression string replacement
space	9,564	6,200	ESA ADL interpreter
spice	179,623	136,182	Digital circuit simulator
termutils	7,006	4,908	Unix terminal emulation utilities
tile-forth-2.1	4,510	2,986	Forth Environment
time-1.7	6,965	4,185	CPU resource measure
userv-0.95.0	8,009	6,132	Trust management service
wdiff.0.5	6,256	4,112	Diff front end
which	5,407	3,618	Unix utility
wpst	20,499	13,438	CodeSurfer Pointer Analysis
Sum	919,096	664,083	
Average	22,977	16,602	

Table 2: Nesting in real-world programs

Program	All	Nothing in between	Unrelated in between	Related in between
a2ps	528	98	147	283
acct	105	36	30	39
barcode	116	8	40	68
bc	114	28	29	57
byacc	154	25	51	78
cadp	136	65	26	45
compress	22	6	4	12
copia	1	1	0	0
csurf-pkgs	757	97	267	393
ctags	257	85	31	141
diffutils	263	51	83	129
ed	162	38	45	79
empire	2,915	283	1,132	1,500
EPWIC-1	160	35	38	87
espresso	380	74	103	203
findutils	187	38	42	107
flex2-4-7	203	69	75	59
flex2-5-4	261	80	110	71
ftpd	900	174	203	523
gcc.cpp	187	40	35	112
gnubg-0.0	224	42	87	95
gnuchess	498	134	159	205
gnugo	1,578	384	531	663
go	1,568	375	609	584
ijpeg	277	104	64	109
indent	224	56	46	122
li	121	52	30	39
ntpd	973	197	310	466
oracolo2	282	22	65	195
prepro	263	16	65	182
replace	7	3	0	4
space	283	22	65	196
spice	3,010	428	717	1,865
termutils	77	13	16	48
tile-forth-2.1	44	20	6	18
time-1.7	17	6	8	3
userv-0.95.0	243	44	39	160
wdiff.0.5	49	18	12	19
which	33	4	12	17
wpst	360	41	128	191
average	448.5	82.8	136.5	229.2
%		18.5%	30.4%	51.1%

The results can be seen in Table 2. 'All' is a figure of all `if` statement pairs analyzed. 'Nothing in between' records all `if` P and Q pairs with no A. 'Unrelated in between' records all P and Q pairs with an A, but A does not have an effect on Q. 'Related in between', on the other hand, counts all A's that have an effect on the predicate at Q.

The results show that the 'unrelated in between' case, that is the form of `if` pairs that can be handled by the technique of Baresel *et al.* account for less than a fifth of all `if` pairs studied. A further 30% of the 'unrelated in between' could also be handled, since the extra statements do not affect Q. Therefore, the branch distance calculation could still legitimately take place before P, however data dependency analysis would be required to establish this situation. The remaining 50% of the 'related in between' cases would not be plausibly handled by the approach of Baresel *et al.* This is overcome by the application of a testability transformation approach described in the next section.

4 Applying a Testability Transformation

A testability transformation [7] is a source-to-source program transformation that seeks to improve the performance of a test data generation technique. The transformed program produced is merely a 'means to an end', rather than an 'end' in itself, and can be discarded once it has served its purpose as an intermediary for generating the required test data. The transformation process need not preserve the traditional meaning of a program. For example, in order to cover a chosen branch, it is only required that the transformation preserve the set of test-adequate inputs. That is, the transformed program must be guaranteed to execute the desired branch under the same initial conditions. Testability transformations have also been applied to the problem of flags for evolutionary test data generation [2, 6], and the transformation of unstructured programs for branch coverage [8].

The philosophy behind the testability transformation proposed in this paper is to remove the constraint that the branch distances of nested decision nodes must be minimized to zero one at a time, and one after the other. The transformation takes the original program and removes decision statements on which the target is control dependent. In this way, when the program is executed, it is free to proceed into the originally nested areas of the program, regardless of whether the original branching predicate would have allowed that to happen. In place of the decision is an assignment to a variable `_dist`, which computes the branch distance based on the original predicate. At the end of the program, the value of `_dist` reflects the summation of each of the individual branch distances. This value may then be used as the objective value for the test data input.

The original version of case study 1 (Figure 2a) can therefore be transformed into the program seen in Figure 2b. The benefit of the transformation can be immediately seen in a plot of the objective function landscape (Figure 3b). The sharp drop into local minima of the original landscape (Figure 3a) is replaced with smooth planes sloping down to the global minimum.

Case study 2 (Figure 4) is of a slightly more complicated nature, with the target positioned within three levels of nesting. A further `if-else` decision exists at level one, before the second conditional en route to the target. Within both branches of this decision, a value is assigned to the variable `d`, on which the `if` statement at node 6 is dependent upon. The transformed version of the program can be seen in Figure 4b. Again, the benefits of the transformation can be instantly seen in a plot of the objective landscape (Figure 5b). The sharp drop in the original landscape (Figure 5a) corresponding to branching node 1 being evaluated as true and branching node 2 being encountered, is replaced by a smooth landscape sloping down from all areas of the search space down into the global minimum.

5 Experimental Study

The two case studies introduced were put to the test with an evolutionary approach.

The Genetic and Evolutionary Algorithm Toolbox (GEATbx) [18] was used to perform the

Table 3: Test data evaluations for case study 1

Run	Untransformed Version	Transformed Version
1	35,130	11,910
2	31,350	18,390
3	17,580	13,260
4	24,060	10,560
5	27,300	14,070
6	38,100	13,260
7	39,180	9,750
8	27,300	13,800
9	30,540	12,720
10	32,700	16,500
Average	30,324	13,422

Table 4: Test data evaluations for case study 2

Run	Untransformed Version	Transformed Version
1	54,030	19,470
2	54,030	20,550
3	54,030	16,770
4	54,030	17,850
5	54,030	18,390
6	54,030	19,200
7	54,030	19,740
8	54,030	19,470
9	54,030	15,150
10	54,030	16,500
Average	54,030	18,309

evolutionary searches, which were conducted as follows. 300 individuals were used per generation, split into 6 subpopulations starting with 50 individuals each. Linear ranking is utilized, with a selection pressure of 1.7. The input vectors are operated on by the evolutionary algorithm 'as is', i.e. as a vector of double values. Individuals are recombined using discrete recombination, and mutated using real-valued mutation. Real-valued mutation is performed using "number creep" - the alteration of variable values through the addition of small amounts. Competition and migration is employed across subpopulations. Each evolutionary search was terminated after 200 generations if test data was not found. Each experiment with each program version was repeated ten times.

The domains of each double variable were -1000 to 1000 with a precision of 0.001, giving search space size of 10^{11} for case study 1 and 10^{17} for case study 2.

For case study 1, the evolutionary algorithm generally performed less than half the number of objective function evaluations (i.e. test data evaluations) for the transformed version of the program, compared to the untransformed version (Table 3). The average best objective value plot, in Figure 6, shows search progress for the untransformed version of case study, with sudden improvements in objective as the search navigates from local minimum to local minimum. Search progress for the transformed version, as expected, is more consistent and gradual.

The evolutionary algorithm encountered severe difficulties with the untransformed program for case study 2. Due to the existence of three levels of nesting, the search fails on each occasion. Exactly the same number of test data evaluations are performed on each of the ten repetitions of the experiment (Table 4), terminating in the 200th generation. The search has much more success

Figure 6: Average best objective value plot for case study 1

with the transformed version of the program, finding test data as early as the 55th generation in one of the ten repetitions.

6 Future Work

The transformation algorithm proposed in this paper does not accommodate for decision statements that are looping constructs, such as 'while' or 'for', or for `if` decision statements that are themselves nested within an outer loop. This is because the branch distance value for a conditional could potentially be added more than once. An advanced version of the algorithm might allow for loops by simply recording the minimum value of the branch distance encountered for the conditional, and adding this to the end value of the `_dist` variable. It makes no difference to the transformation algorithm, of course, if intermediate blocks of statements occurring between nested `if` pairs feature self-contained loops.

The transformation algorithm also has issues with certain type of predicates, which need to be detected unless run-time errors are allowed to occur. One example of this is a predicate which tests the possibility of a dynamic memory reference. The following example may lead to a program error if it is transformed, due to the possibility of the array index of
texttti being less than zero or greater than the length of the array, and thus causing an array out of bounds error:

```
if (i >= 0 && i < length_of_a)
{
    printf("%f\n", a[i]);
}
```

Another issue is the possibility of introducing division by zero errors, for example in the following segment of code if the conditional were to be removed:

```
if (d != 0)
{
    r = n / d;
}
```

Figure 7: Average best objective value plot for case study 2

Currently, the transformation algorithm works on a per-target basis - a separate transformation needs to be performed for each search target. An advanced version of the algorithm could modify the predicates of the program to contain function calls. The function call would record the branch distance, and then decide on the basis of the nesting of the current target as to a boolean value to return, and ultimately, whether execution should be allowed to proceed down a specific branch. For example, in the following, `tt_nesting_check` records the branch distance of `a == b` at node A and lets execution flow down through its true branch regardless of whether `a` actually does equal `b` or not. However, `tt_nesting_check` remains true to the original predicate `b == c` at node B, since the current target is not nested within it, but allows execution through the true branch at node C regardless of whether `c == d`.

```
Node
 (A)   if (tt_nesting_check(a == b))
       {
 (B)     if (tt_nesting_check(b == c))
         {
           ...
         }

 (C)     if (tt_nesting_check(c == d))
         {
           // current target nested in here
         }
       }
```

7 Conclusions

This paper has described how targets nested within more than one conditional statement can cause problems for search-based approaches to structural test data generation. In the presence of nesting, the search is forced to concentrate on satisfying one branch predicate at a time, one after the other. This slows search progress and restricts the potential search space available for the satisfaction of branching predicates 'later' in the sequence of nested conditionals.

A testability transformation approach was presented to the problem. A testability transformation is a source-to-source program transformation that seeks to improve the performance of a test data generation technique. The transformed program produced is merely a 'means to an end', rather than an 'end' in itself, and can be discarded once it has served its purpose as an intermediary for generating the required test data.

The main idea behind the testability transformation proposed in this paper is to remove the constraint that the branch distances of nested decision nodes must be evaluated one after the other. The transformation takes the original program and removes decision statements on which the target is control dependent. In this way, when the program is executed, it is free to proceed into the original nested areas of the program, calculating all branch distance values for the purpose in order to compute objective values which are in full possession of the facts about the input data.

The approach was put to the test with two case studies. The case studies are small examples, and by no means represent a worse-case scenario, yet serve to demonstrate the power of the approach. The transformed version of case study 1 allowed the evolutionary search to find test data in half the number of test data evaluations over the original version of the program. Whilst test data could not be found for the search target for the original version of case study 2, the evolutionary algorithm succeeded every time with the transformed version.

The transformation approach deals with assignments to variables in between nested conditionals which may affect the outcome at 'later' conditionals en route to the current structural target. The empirical study of `if` pairs in forty real-world programs, where one of the `if` statements of the pair is nested within the other, showed that this situation occurs just over 50% of the time. These cases can not be dealt with earlier work of Baresel *et al.* [3] which investigated the nesting problem.

References

[1] A. Baresel. Automatisierung von strukturtests mit evolutionren algorithmen. Diploma Thesis, Humboldt University, Berlin, Germany, July 2000.

[2] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 43–52, Boston, Massachusetts, USA, 2004. ACM.

[3] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1329–1336, New York, USA, 2002. Morgan Kaufmann.

[4] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.

[5] J. Ferrante, K. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[6] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1359–1366, New York, USA, 2002. Morgan Kaufmann.

[7] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[8] R. Hierons, M. Harman, and C. Fox. Branch-coverage testability transformation for unstructured programs. *The Computer Journal*, To appear, 2005.

[9] T. Horwitz S., Reps and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12:26–60, 1990.

[10] B. Jones, H. Sthamer, and D. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[11] B. Jones, H. Sthamer, X. Yang, and D. Eyres. The automatic generation of software test data sets using adaptive search techniques. In *Proceedings of the 3rd International Conference on Software Quality Management*, pages 435–444, Seville, Spain, 1995.

[12] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[13] B. Korel. Dynamic method for software test data generation. *Software Testing, Verification and Reliability*, 2(4):203–213, 1992.

[14] B. Korel and A. M. Al-Yami. Assertion-oriented automated test data generation. In *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, pages 71–80, 1996.

[15] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[16] P. McMinn and M. Holcombe. Hybridizing evolutionary testing with the chaining approach. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004), Lecture Notes in Computer Science vol. 3103*, pages 1363–1374, Seattle, USA, 2004. Springer-Verlag.

[17] R. Pargas, M. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[18] H. Pohlheim. GEATbx - Genetic and Evolutionary Algorithm Toolbox, http://www.geatbx.com.

[19] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 134–143, Madrid, Spain, 1998. IEEE Computer Society Press.

[20] N. Tracey. *A Search-Based Automated Test-Data Generation Framework for Safety Critical Software*. PhD thesis, University of York, 2000.

[21] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *Software Engineering Notes, Issue 23, No. 2, Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 1998)*, pages 73–81, 1998.

[22] N. Tracey, J. Clark, and K. Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications*, pages 169–180. Dept of Computer Science, University of Witwatersrand, Johannesburg, South Africa, 1998.

[23] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the International Conference on Automated Software Engineering*, pages 285–288, Hawaii, USA, 1998. IEEE Computer Society Press.

[24] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test data generation for exception conditions. *Software - Practice and Experience*, 30(1):61–79, 2000.

[25] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[26] J. Wegener, K. Grimm, M. Grochtmann, H. Sthamer, and B. Jones. Systematic testing of real-time systems. In *Proceedings of the 4th European Conference on Software Testing, Analysis and Review (EuroSTAR 1996)*, Amsterdam, Netherlands, 1996.

[27] J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275–298, 1998.

[28] D. Whitley. An overview of evolutionary algorithms: Practical issues and common pitfalls. *Information and Software Technology*, 43(14):817–831, 2001.

[29] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing (Application des algorithmes génétiques au test des logiciels). In *5th International Conference on Software Engineering and its Applications*, pages 625–636, Toulouse, France, 1992.