Empirical Evaluation of a Nesting Testability Transformation for Evolutionary Testing

PHIL MCMINN
University of Sheffield
DAVID BINKLEY
Loyola College in Maryland
and
MARK HARMAN
King's College London

Evolutionary testing is an approach to automating test data generation that uses an evolutionary algorithm to search a test object's input domain for test data. Nested predicates can cause problems for evolutionary testing, because information needed for guiding the search only becomes available as each nested conditional is satisfied. This means that the search process can over-fit to early information, making it harder, and sometimes near impossible, to satisfy constraints that only become apparent later in the search. The paper presents a testability transformation that allows the evaluation of all nested conditionals at once. Two empirical studies are presented. The first study shows that the form of nesting handled is prevalent in practice. The second study shows how the approach improves evolutionary test data generation.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic Methods*

General Terms: Verification, Algorithms, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Evolutionary testing, test data generation, testability transformation, search-based software engineering

Authors' addresses: Phil McMinn, University of Sheffield, Department of Computer Science, Regent Court, 211 Portobello Street, Sheffield, S1 4DP, UK. Email: `p.mcminn@dcs.shef.ac.uk` David Binkley, Loyola College, 4501 North Charles Street, Baltimore, MD 21210-2699, USA. Email: `binkley@cs.loyola.edu` Mark Harman, CREST, King's College, Strand, London, WC2R 2LS, UK. Email: `mark.harman@kcl.ac.uk`

This work was supported as follows. Phil McMinn received support from DaimlerChrysler Research & Technology. David Binkley is supported by National Science Foundation grant CCR-0305330. He is also jointly supported with Mark Harman by EPSRC grant EP/F010443. Mark Harman is further supported in part by EPSRC grants EP/F012535, EP/E002919, EP/D050863, GR/T22872 and GR/S93684 and by the EU-funded project EvoTest.

1. INTRODUCTION

The application of metaheuristic search techniques, for example evolutionary algorithms, to the automatic generation of software test data has been shown to be an effective approach for functional [Jones et al. 1995; Tracey et al. 1998a; Tracey 2000], non-functional [Wegener et al. 1996; Puschner and Nossal 1998; Wegener and Grochtmann 1998], structural [Korel 1990; 1992; Ferguson and Korel 1996; Xanthakis et al. 1992; Jones et al. 1996; Pargas et al. 1999; Wegener et al. 2001; McMinn and Holcombe 2006; McMinn 2004], and grey-box [Korel and Al-Yami 1996; Tracey et al. 2000] testing criteria. The search space is the input domain of the test object. A fitness 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 of input data are monitored through instrumentation of the branching conditions of the program. A fitness function is computed, which decides how 'close' an input was to executing the target. This is based on the values of variables appearing in the branching conditionals that 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 fitness function returns a 'branch distance' value of `abs(b - a)` to the search. The fitness values fed back are critical in directing the search to potential new input vector candidates that might execute the desired program structure.

However, it is possible for the search to encounter problems when a target is 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 fitness 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 `a == c` must also be satisfied. This gradual release of information causes difficulty for the search, which is forced to concentrate on satisfying each predicate individually. In this example, all the input variables have to be the same value in order to execute the target, however, this is not reflected in the fitness function until the final branching predicate.

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 or equal to `b`, the path taken through the program never reaches the latter conditional, and thus the search never finds out if `b <= c` is true 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.

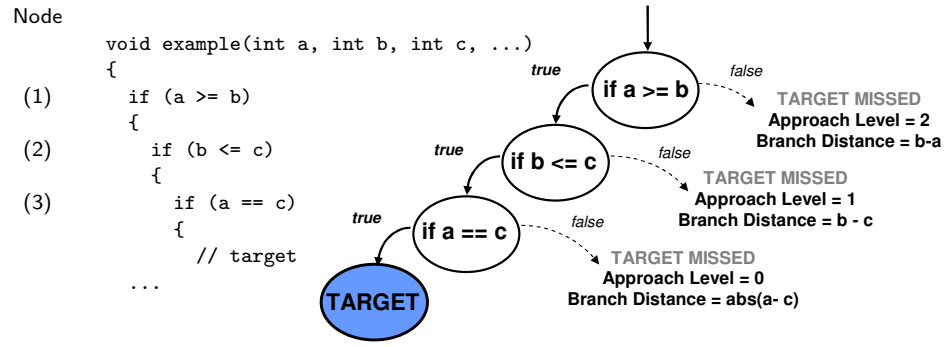


Fig. 1. Nested targets require the succession of branching statements to be evaluated by the fitness function one after the other

Ideally, all branch predicates are evaluated by the fitness function at the same time. This paper presents an approach that achieves this goal by transforming the original program. The type of transformation used, a 'testability transformation' [Harman et al. 2004], is a source-to-source program transformation that seeks to improve the performance of an existing test data generation technique. Here, 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 intermediary purpose as a vehicle for generating test data.

Test data generation is a costly process when performed by hand; thus techniques such as testability transformations that can automate the generation of even a subset of the necessary test data are extremely valuable. Even if only one structure is covered using a transformation that cannot be covered using the original version of the program, human costs associated with the testing process will have been lowered. Therefore, the approach does not have to improve test data generation 100% of the time in order to be useful.

Two empirical studies are used to investigate the impact of the proposed nesting testability transformation. The first examines nesting in 43 real-world programs, establishing that the type of nesting handled by the transformation is prevalent in practice. The second study compares test data generation with and without the nesting transformation, showing that the effectiveness and efficiency of the search can be improved in many cases with significant performance gains, confirmed by applying statistical $t-$tests.

The primary contributions of this paper are therefore as follows:

(1) The paper introduces a testability transformation designed to improve evolutionary structural test data generation for nested program structures.

(2) The paper presents empirical results that show the forms of nesting handled by the approach are highly prevalent in practice.

(3) The paper presents empirical results with 33 different functions taken from 7 production programs that illustrate the way in which the transformation improves the performance of evolutionary testing.

The rest of this paper is organized as follows. Section 2 briefly outlines evolutionary structural test data generation. Section 3 states the research problem to be addressed, while Section 4 describes the testability transformation used to address the problem. Section 6 presents results that indicate that the type of nesting handled is prevalent in practice, whilst Section 7 presents results that show that the approach is useful in improving the performance of evolutionary testing on production code. Section 8 discusses some practical issues with the transformation. Section 9 reviews related work and Section 10 concludes with directions for future work.

2. EVOLUTIONARY STRUCTURAL TEST DATA GENERATION

Several search methods have been proposed for the automation of structural test data generation, including the alternating variable method [Korel 1990; 1992; Ferguson and Korel 1996], simulated annealing [Tracey et al. 1998; Tracey et al. 1998b] and evolutionary algorithms [Xanthakis et al. 1992; Jones et al. 1996; Pargas et al. 1999; Wegener et al. 2001; McMinn and Holcombe 2006; McMinn 2004]. This paper concerns the application of evolutionary algorithms to the problem, an approach known as evolutionary testing [Xanthakis et al. 1992; Jones et al. 1996; Pargas et al. 1999; Wegener et al. 2001; McMinn and Holcombe 2006; McMinn 2004]. Evolutionary algorithms [Whitley 2001] combine characteristics of genetic algorithms and evolution strategies, using simulated evolution as a search strategy, employing operations inspired by genetics and natural selection.

An evolutionary algorithm maintains a population of candidate solutions referred to as *individuals*. Individuals are iteratively recombined and mutated in order to evolve successive generations of potential solutions. The aim is to generate 'fitter' individuals within subsequent generations, which represent better candidate solutions. Recombination forms offspring from the components of two parents selected from the current population. Mutation performs low probability random changes to solutions, introducing next genetic information into the search. The new offspring and mutated individuals form part of the new generation of candidate solutions. At the end of each generation, each individual is evaluated for its fitness with only the fittest individuals surviving into the next generation.

In applying evolutionary algorithms to structural test data generation, the individuals of the search are input vectors. The fitness function to be minimized by the search is derived from the current structure of interest. Thus lower values represent fitter input vectors that are closer to executing the target structure. When a zero fitness value has been found, the required test data has also been found.

Fitness values incorporate two factors. The first, the *branch distance*, is taken from the point at which execution diverged from the target structure for the individual. The branch distance is computed for the alternative branch (*i.e.* the branch having the opposite truth value to the one taken in the course of execution). For example in Figure 1, if execution flows down the false branch from node 1 for an individual, the branch distance is computed using `b - a`. The smaller this value is, the closer the desired true branch is to being taken. The second factor incorporated in the fitness function is a metric known as the *approach level* [Wegener et al. 2001], which records how many conditional statements are left unencountered by

the individual en route to the target. If the execution path resulting from an input vector corresponding to some individual reaches node 1 but diverges away down the false branch, the approach level is 2, since there are two further branching nodes to be encountered (nodes 2 and 3). If the input vector evaluates node 1 in the desired way, its fitness value is formed from the true branch distance at node 2, and the approach level value is 1. At node 3, the approach level is zero and the branch distance is derived from the true branch predicate.

Formally, the fitness function for an input vector is computed as follows:

$$fitness = approach\_level + normalize(dist) \tag{1}$$

where the branch distance *dist* is normalized into the range 0-1 using the following function [Baresel 2000]:

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

This formula ensures 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 can therefore be thought of as adding a value for each branch distance that 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' in the literature [Baresel et al. 2002; Wegener et al. 2001]. As will be seen in the next section, the addition of this rough value rather than actual branch distance can inhibit search progress.

3. THE NESTED PREDICATE PROBLEM

The dependence of structural targets on one or more nested decision statements can cause problems for evolutionary testing, and even failure in severe cases [McMinn et al. 2005]. 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 achieved, the current branching statement will never be reached. In this way, the search is restricted in its choice of possible inputs, *i.e.* the search space is artificially narrowed.

For example, consider the code shown in 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 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 fitness function landscape (Fig-

ure 2c), which uses the output of Equation 1 for *fitness*. 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. The evolutionary algorithm 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)`.

Example 2 (Figure 3a) further demonstrates the problems of nested targets, with the fitness function for the target (node 7) plotted in Figure 3c. 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 fitness value, as the focus of the search is pushed back to satisfying this initial predicate. In this area of the search space, the fitness 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.

4. A TESTABILITY TRANSFORMATION FOR NESTED PREDICATES

A testability transformation [Harman et al. 2004] is a source-to-source program transformation that seeks to improve the performance of an existing test data generation technique where the transformed program is merely a 'means to an end' rather than an 'end' in itself. Thus, it is discarded once it has served its purpose as an intermediary for generating the required test data. 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 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 for that branch. 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 [Baresel et al. 2004; Harman et al. 2002] and the transformation of unstructured programs for branch coverage [Hierons et al. 2005].

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. To capture the removed decisions, assignments are made to a newly introduced variable `_dist`. These assignments compute the branch distance based on each of the original predicates. When the target is reached, the value of `_dist` reflects the summation of each of the individual branch distances and is used as the basis of fitness value computation.

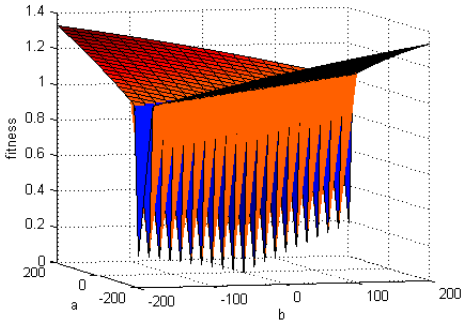
The remainder of this section details the two steps of the testability transformation.

```
Node
     void cs1_original(double a,
                       double b)
     {
(1)    if (a == b)
       {
(2)      double c = b + 1;
(3)      if (c == 0)
         {
(4)        // target
         }
       }
     }
```

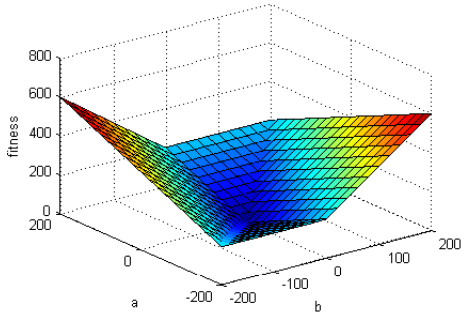
(a) Original program

```
void cs1_transformed(double a,
                     double b)
{
  double _dist = 0;

  _dist += distance(a == b);

  double c = b + 1;

  _dist += distance(c == 0);

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

(b) Transformed version



(c) Landscape for original program



(d) Landscape for transformed program

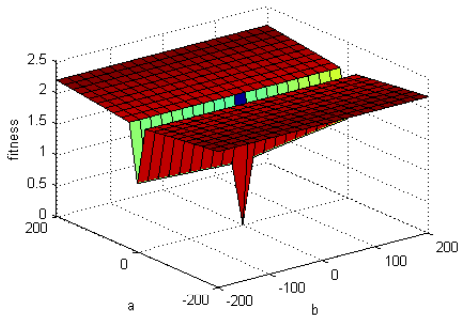
Fig. 2. Example 1, showing (a) the original and (b) the transformed versions of the code. The transformation removes the sharp drop into points of local minima prevalent in the fitness landscape of the original program seen in part (c), with the more directional landscape of the transformed program, seen in part (d)

```
Node
     void cs2_original(double a,
                       double b)
     {
(1)    double c;
(2)    if (a == 0)
       {
(3)      if (b > 1)
(4)        c = b + b/2;
         else
(5)        c = b - b/2;

(6)      if (c == 0)
         {
(7)        // target
         }
       }
     }
```

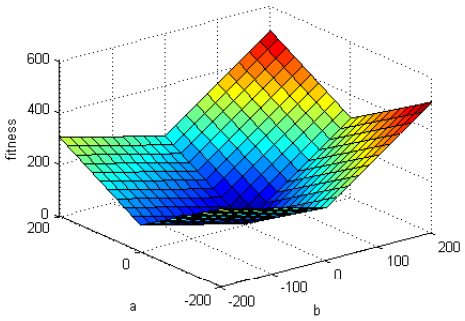
(a) Original program

```
void cs2_transformed(double a,
                     double b)
{
  double _dist = 0;
  double c;
  _dist += distance(a == 0);

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

  _dist += distance(c == 0);

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

(b) Transformed version



(c) Landscape for original program



(d) Landscape for transformed program

Fig. 3. Example 2, showing (a) the original and (b) the transformed versions of the code. The transformation removes the sharp drops in the fitness landscape of the original program as a result of nesting, as seen in part (c), with the more directional landscape of the transformed program, seen in part (d)

The Testability Transformation Algorithm

Step 1 - Check that the transformation can be applied to the target

The first step of the algorithm is to check that two applicability conditions are satisfied. In both, let C denote the set of predicate nodes on which the target is control dependent, either transitively or directly.

(1) Structured code involving the target.
The target and each node in C cannot have more than one direct control dependency. All structured code follows this pattern (a node may have several transitive control dependencies, but only one of these will be a direct dependency). Multiple direct dependencies can be the result of `goto` statements. Thus, the transformation is not applicable to the target in the following piece of code, since it is directly control dependent on both `p` and `q`:

```
if (p) {
    goto x;
}

if (q) {
    x:
    // target
}
```

However, the presence of unstructured code outside of the predicates in C is acceptable. For example, the following is permissible as C does not contain `p` or `q`.

```
if (p) {
    goto x;
}

if (q) {
  x:
  // ...
}

if (r) {
  // target
}
```

(2) Loops.
Neither the target nor any of the nodes in C can be a loop predicate. For example, the transformation is not applicable to the target in the following piece of code, since it is nested in a loop:

```
if (p) {
    while (q) {
        if (r) {
            // target
        }
    }
}
```

However, the following is permissible. The loop does not actually control any of the nodes on which the target is control dependent:

```
if (p) {
    while (q) {
        // ...
    }

    if (r) {
        // target
    }
}
```

Step 2 - Perform the transformation on targets satisfying step 1

(1) Code removal
The nodes in C are connected to the target by a sequence of control dependence edges. Each such edge is labelled either `true` or `false` depending on the branch of the associated `if` statement in which it is found. Statements control dependent on these predicates, but via the opposite edge label, are removed from the program. For example, in the following, the statement `s2` is removed:

```
s1;

if (a == b) {
    s2;
} else {
    s3;
    if (c == d) {
        // target
    }
}
```

(2) Insert uniquely named temporary variable for accumulating branch distances
A new variable, named `_dist`, is introduced. This variable should be of the highest floating point precision and initialised as zero at the start of the program. For example:

```
double _dist = 0;

s1;

if (a == b) {

} else {
    s3;
    if (c == d) {
        // target
    }
}
```

(3) Replace conditionals with branch distance calculations
Each conditional in C is replaced with the relevant distance calculation that would have been inserted automatically by the evolutionary testing instrumentation for the target for that predicate. The result of the calculation should be added to `_dist`. For example, the above program would be modified as follows:

```
double _dist = 0;

s1;

_dist += distance(a != b);

s3;

_dist += distance(c == d);

// target
```

The relevant distance calculations are denoted by `distance(...)` and are not actual C function calls.

(4) Surround target with a new `if` statement
The target is placed within an `if` statement whose predicate checks if `_dist` is equal to zero. For example:

```
double _dist = 0;

s1;

_dist += distance(a != b);

s3;

_dist += distance(c == d);

if (_dist == 0) {
    // target
}
```

For sake of presentation, the transformation assumes all the identifiers in the program are unique (*i.e.*, no two variables at different scopes in the program share the same name), and there is no variable already in the program with the identifier `_dist`. In practice this can easily be ensured through a pre-processing step that simply renames clashing identifiers.

The transformed program is used instead of the original for test data generation. The test data generation process itself is left unchanged; instrumentation is performed as before, and the fitness function calculation is performed in exactly the same way. The difference using the transformed version is that the approach level will always be zero, as there is no nesting. Thus, the branch distance for the target records the difference between the variable `_dist` and zero. The variable `_dist` itself, of course, represents the accumulation of branch distances taken for each predicate that was nested in the original program.

The effect of the transformation on the example of Figure 2 can be seen in the fitness landscape (Figure 2c). The sharp drop into local minima of the original landscape (Figure 2c) is replaced with smooth planes sloping down to the global minimum (Figure 2d).

The example of Figure 3 is of a slightly more complicated nature, with `if-else` code appearing before the nested target. The transformed version of the program can be seen in Figure 3b. Again, the benefits of the transformation can be instantly seen in a plot of the fitness landscape (Figure 3c). The sharp drop in the original

landscape 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 (Figure 3d).

5. RESEARCH QUESTIONS

This section sets out the research questions to be answered through empirical study.

The first research question attempts to justify the approach taken by this paper by establishing the applicability of the transformation in practice. Clearly, if nesting is not prevalent in real-world code, or the rules for applying the transformation as set out in Section 4 are too constrictive, the transformation will not be of much use:

RQ 1: Does the form of nesting handled by the transformation presented occur in practice?

The second research question aims to investigate performance of test data generation using the transformation:

RQ 2: Does the transformation improve test data generation for nested structures in practice?

The third research question investigates two classes of nesting that a structural target can be involved in. The categorisation depends on the nature of the predicates appearing in the `if` statements that the structure is nested in. Two predicates P_i and P_j are defined to be *dependent* when the set of input variables that *influence* their outcome intersect; otherwise, they are *independent*. An *influence* includes the direct use of an input variable, or, an indirect use via intermediate variables that are assigned a value using an input variable. In the following example, P and Q are dependent (both predicates use the input variable `b`), but P and R are independent.

```
        void dependency_example(int a, int b, int c)
        {
(P)         if (a == b) {

(Q)             if (b == c) {
                    // target 1
                }

(R)             if (c == 0)  {
                    // target 2
                }
            }
        }
```

The issue of predicate dependency will affect the search: for target 2, 'moves' made involving the variables `a` and `b` for P will not affect the outcome at R, which is concerned with `c` only. However for target 1, changing the value of `b` for Q could have an adverse effect on P.

It is expected that the removal of nesting through program transformation will have a positive impact on independent predicates, for example target 1, and the predicates involved en route to the target in the example of Figure 3. This is because the search will be free to find their solution 'concurrently' without complicated interactions resulting from changing a variable affecting multiple predicates.

For dependent predicates, the situation is not so clear. The example of Figure 2 depicts a target with dependent nested predicates where the landscapes indicate that the search should benefit from the removal of nesting. However, in general, the effect on the search landscape cannot be predicted. It is possible that the concurrent consideration of dependent predicates could inadvertently introduce further local optima into the search space. Thus, the third research question is as follows:

RQ 3: Does the transformation improve test data generation for nested structures with both 'independent' and 'dependent' predicate types?

The following two sections describe the two empirical investigations that are used to address the three research questions. The first empirical study impacts RQ 1 and RQ 3, while the second impacts RQ 2 and RQ 3.

6. EMPIRICAL STUDY 1: PREVALENCE OF NESTING HANDLED BY THE TRANSFORMATION

The first empirical study is an examination of nested branches in 43 real-world programs, containing a total of just under 800,000 lines of code. The results of the study are summarized in Table I. For each program, the tables includes the size of the program in lines of code and the number of transformable branches (it also includes the percentage of these that are dependent and independent, used to address RQ 3). The table directly answers the first research question:

RQ 1: Does the form of nesting handled by the transformation presented occur in practice?

From the last line of the table, just under 24,000 transformable branches were identified. This is about 3 per 100 lines of code; thus the answer to RQ 1 is clearly 'yes'.

7. EMPIRICAL STUDY 2: TEST DATA GENERATION

The second study provides data related to RQ 2 and RQ 3. It was designed to compare the performance of an evolutionary search algorithm using both transformed and original versions of programs with nested branches. The study selected 33 functions taken from seven production programs (five open-source and two industrial). Table II shows the selected functions as well as details on the predicates uncovered and the domain size used in the search. In all, the code studied contains 670 branches, of which 560 are nested and 394 are transformable. Details of the subjects can be seen in Table II.

Five open-source case studies were selected from the subjects listed in Table I. The program `eurocheck-0.1.0` contains a single function used to validate serial numbers on European bank notes. The program `gimp-2.2.4` is the well-known GNU image manipulation program. Several library functions were investigated that contained branches to which the nesting transformation could be applied, including routines for conversion of different colour representations (for example RGB to HSV) and the manipulation of drawable objects. The program `space` is program from the European Space Agency, and is available from the Software-artifact Infrastructure Repository [Do et al. 2005]. Nine functions were investigated, with three

Table I. Nesting in practice

Program	Lines of Code	Transformable Nested Branches	Dependent (%)	Independent (%)
a2ps	53,900	822	80%	20%
acct-6.3	9,536	160	56%	44%
barcode	5,562	160	78%	23%
bc	14,609	142	85%	15%
byacc	6,337	160	74%	26%
cadp	11,068	290	68%	32%
compress	1,234	40	50%	50%
copia	1,170	2	0%	100%
csurf-packages	36,593	1,326	84%	16%
ctags	16,946	474	86%	14%
diffutils	18,374	316	71%	29%
ed	12,493	184	75%	25%
empire	53,895	2,550	88%	12%
EPWIC-1	8,631	206	67%	33%
espresso	22,050	394	75%	25%
eurocheck-0.1.0	101	6	33%	67%
findutils	16,891	280	81%	19%
flex2-4-7	15,143	338	43%	57%
flex2-5-4	20,252	452	43%	57%
ftpd	15,914	1,050	72%	28%
gcc.cpp	4,079	128	73%	27%
gnubg-0.0	7,229	344	73%	27%
gnuchess	16,659	784	79%	21%
gnugo	15,217	2,086	87%	13%
go	28,547	1,870	88%	12%
ijpeg	24,822	366	90%	10%
indent-1.10.0	6,100	250	78%	22%
li	6,916	182	93%	7%
libgimpcolor	3,230	74	51%	49%
ntpd	45,647	1,176	78%	22%
oracolo2	14,326	498	96%	4%
prepro	14,328	490	96%	4%
replace	563	14	57%	43%
space	9,126	500	96%	4%
spice	149,050	3,360	85%	15%
termutils	6,697	78	62%	38%
tiff-3.8.2	59,649	870	77%	23%
tile-forth-2.1	3,717	52	73%	27%
time-1.7	6,033	28	43%	57%
userv-0.95.0	7,150	326	72%	28%
wdiff.0.5	5,958	56	57%	43%
which	4,880	52	81%	19%
wpst	17,321	622	79%	21%
Total	797,943	23,558	81%	19%

containing branches to which the transformation could be applied. The program `spice` is an open source general purpose analogue circuit simulator. Finally, two functions were investigated, which were clipping routines for the graphical front-end. `tiff-3.8.2` is a library for manipulating images in the Tag Image File Format (TIFF). The functions investigated comprise routines for placing images on pages, and the building of 'overview' compressed sample images.

In addition, two industrial case studies, the programs `dc_f2` (an internal name) and `dc_defroster`, were provided by DaimlerChrysler. An S–Class Mercedes car has over 80 such embedded controllers, which, taken together represent approximately half a gigabyte of object code. The two systems used in this study are production code for engine and rear window defroster control systems. The code is machine generated from a design model of the desired behaviour. As such, it is not

Table II. Production code test objects used in the study

Test Object / Function	Branches			Domain Size (10^x)	
	Total	Nested	Transformable	Setup 1	Setup 2
dc_defroster					
Defroster_main	56	52	52	24	96
dc_f2					
F2	24	8	8	54	81
eurocheck-0.1.0					
main	22	20	6	31	50
gimp-2.2.4					
gimp_hsv_to_rgb	16	14	14	21	37
gimp_hsv_to_rgb4	16	14	14	16	27
gimp_hsv_to_rgb_int	16	14	14	7	12
gimp_hwb_to_rgb	18	16	16	17	27
gimp_rgb_to_hsl	14	12	12	20	37
gimp_rgb_to_hsl_int	14	10	10	7	12
gimp_rgb_to_hsv	10	8	8	20	37
gimp_rgb_to_hsv4	18	12	12	7	12
gimp_rgb_to_hsv_int	14	8	8	7	12
gradient_calc_bilinear_factor	6	4	4	34	51
gradient_calc_conical_asym_factor	6	4	4	31	49
gradient_calc_conical_sym_factor	8	6	6	31	49
gradient_calc_linear_factor	8	6	6	31	49
gradient_calc_radial_factor	6	4	4	21	33
gradient_calc_spiral_factor	8	6	6	37	58
gradient_calc_square_factor	6	4	4	21	33
space					
space_addscan	32	30	0	519	712
space_fixgramp	8	6	6	23	32
space_fixport	6	6	0	125	182
space_fixselem	16	14	0	125	182
space_fixsgrel	72	70	56	524	712
space_fixsgrid	44	42	26	101	120
space_gnodfind	4	2	0	70	89
space_seqrotrg	32	30	0	206	264
space_sgrpha2n	16	14	0	451	614
spice-3f4					
clip_to_circle	42	26	26	23	30
cliparc	64	62	54	44	59
tiff-3.8.2					
PlaceImage	16	6	6	38	59
TIFF_GetSourceSamples	18	18	0	15	15
TIFF_SetSample	14	12	12	10	13
Total	670	560	394		

optimized for human-readability, making manual test data generation non–trivial. The test objects are therefore ideal candidates for search–based testing strategies.

Two further synthetic test objects were designed to specifically investigate the relationship of dependency between the variables of different nested predicates. Figure 4a shows a snippet of code from the 'independent' test object, where for all branches, none of the predicates share any of the input variables. Figure 4b shows a snippet of code from the 'dependent' test object, where consecutive branches share an input variable from the predicate of the last. For both programs, test data generation was attempted with the original version and the transformed versions for each true branch.

7.1 Experimental Setup

The parameters of the evolutionary algorithm are based on those used in the DaimlerChrysler system for evolutionary testing, which has been widely studied in the literature [Baresel and Sthamer 2003; Baresel et al. 2002; Wegener et al. 2001].

```
void independent(double a,
                 double b,
                 double c,
                 double d,
                 double e,
                 double f ...)
{
    if (a == b) {
        if (c == d) {
            if (e == f) {
                ...
            }
        }
    }
}
```

(a) Independent

```
void dependent(double a,
               double b,
               double c,
               double d,
               double e, ...)
{
    if (a == b) {
        if (b == c) {
            if (c == e) {
                ...
            }
        }
    }
}
```

(b) Dependent

Fig. 4. Code snippets of the independent and dependent predicate synthetic test objects. For sake of presentation, only 2 nesting levels are depicted for each test object; in reality nesting is 6 levels deep

The population consists of 300 individuals, split into 6 subpopulations starting with 50 individuals each. Linear ranking is utilized, with a selection pressure of 1.7. Real-valued encodings are used. Individuals are recombined using discrete recombination, and mutated using the mutation operator of the breeder genetic algorithm [Mühlenbein and Schlierkamp-Voosen 1993]. Competition and migration is employed across the subpopulations. Each generation employs a 10% generation gap (*i.e.*, the best 10% of each population are retained from one generation to the next), with the remaining 90% replaced by the best offspring.

The test data generation experiments were performed 60 times using transformed and original versions of the program for each branch. If test data were not found to cover a branch after 100,000 fitness evaluations, the search was terminated. For the evolutionary search, the maximum and minimum values of each ordinal input variable needs to be specified by the tester, and thus different domain sizes are possible. Two domain sizes were used for each function, ranging from approximately 10^7 to 10^{712} across all functions, making for very large search problems. The success or failure of each search was recorded, along with the number of test data evaluations required to find the test data, if the search was successful. From this, the 'success rate' for each branch can be calculated – the percentage of the 60 runs for which test data to execute the branch was found. Success rate is a basis on which the effectiveness of the search can be compared for the branch under original and transformed conditions.

The average number of evaluations required to find test data for each branch was also calculated. The average number of evaluations indicates how much effort was required of the search in finding the test data, and is thus a means of comparing the effort of the search on original and transformed programs.

The sixty runs were performed using the same random seed for each technique, meaning that both searches begin with the same initial population. This allows the

use of paired t-tests in the statistical assessment. A confidence level of 99% was applied. Such tests are necessary to provide robust results in the presence of the inherently stochastic behaviour of the search algorithms.

The study provides answers to RQ 2 and RQ 3:

RQ 2: Does the transformation improve test data generation for nested structures in practice?

Figure 6 gives an overall picture of search effectiveness with the transformation, by plotting the difference in success rate using the transformation for those branches where a change in success rate was experienced when the transformation was applied. Success rate is improved in 59 cases involving independent and dependent nested predicates. 34 of these cases experienced an increase in success rate by more than 10%, with the largest increase being 87% for a branch in the `dc_defroster` program.

Besides effectiveness, search effort is lowered using the transformation in several cases. This can be seen in Table III, which records branches for which the average number of fitness evaluations over the 60 runs is significantly different (when applying paired t-tests) for experiments using the original version of the program for each branch and the transformed version. The average number of evaluations is lower for each of the 58 branches where the transformation is applied, and is cut by over a half for 17 of these.

In 17 cases, however, success rate is worse. Eleven cases suffer a decrease in success rate by more than 10%. Analysis of the code in question revealed the predicates in question were dependent on one another. Consideration of all predicates at once for these branches introduces local optima in to the fitness landscape - optima that do not appear when each predicate is tackled in turn in the original version of the program.

In conclusion then, the results demonstrate that the nesting transformation can indeed improve effectiveness and lower search effort in practice. However this is not always guaranteed. The next research question investigates the reasons behind this more deeply.

RQ 3: Does the transformation improve test data generation for nested structures with both 'independent' and 'dependent' predicate types?

One aim of the first empirical study was to classify nested branches into either the 'dependent' or 'independent' categories. Just under one-fifth of predicates were found to be independent. In theory, removal of nesting in these cases will allow the search to satisfy each predicate *concurrently*, and thus allow for faster test data generation. The distribution of dependent and independent nested predicates across the 43 programs can be seen in Figure 5.

Results obtained with production code show that test data generation can be improved for both independent and dependent nested predicate types, but that this was not always assured for nested predicates of a dependent nature. The synthetic test objects, with either dependent or independent predicate types, were designed to shed more light on this issue.

Figure 7 shows that performance can always be improved for independent predicates, both in terms of effectiveness (*i.e.*, improved success rate) as seen in Figure 7a,

Table III. Search effort using the transformed and original versions of a program for branches where there is a significant difference in the average number of fitness evaluations

Test Object / Branch	Domain Size (10^x)	Average Evaluations		
		Transformed	Original	Significance
dc_defroster				
Defroster_main 12F	96	19,263	29,048	0.000
Defroster_main 14F	96	19,935	39,196	0.000
Defroster_main 15T	96	14,800	21,284	0.000
Defroster_main 16F	96	20,893	48,601	0.000
Defroster_main 16T	96	14,800	21,284	0.000
Defroster_main 18T	96	20,893	48,601	0.000
Defroster_main 20F	96	21,488	49,553	0.000
Defroster_main 20T	96	14,800	21,284	0.000
Defroster_main 22T	96	21,488	49,553	0.000
Defroster_main 30F	96	20,208	38,585	0.000
Defroster_main 31T	96	17,870	31,464	0.000
Defroster_main 36T	96	15,017	21,556	0.000
Defroster_main 39F	96	25,795	66,544	0.000
Defroster_main 39T	96	20,381	40,823	0.000
Defroster_main 40F	96	21,340	48,841	0.000
Defroster_main 40T	96	20,594	40,832	0.000
Defroster_main 44F	96	24,648	63,066	0.000
Defroster_main 44T	96	20,208	38,585	0.000
Defroster_main 45F	96	20,231	38,606	0.000
Defroster_main 45T	96	22,719	46,371	0.000
Defroster_main 48T	96	24,648	63,066	0.000
Defroster_main 49F	96	26,574	70,077	0.000
Defroster_main 49T	96	24,752	63,091	0.000
Defroster_main 55F	96	20,347	40,580	0.000
Defroster_main 58T	96	14,097	21,521	0.000
Defroster_main 60F	96	25,164	65,944	0.000
Defroster_main 60T	96	20,347	40,580	0.000
Defroster_main 61F	96	20,347	40,580	0.000
Defroster_main 61T	96	23,446	55,784	0.000
Defroster_main 63F	96	20,312	41,776	0.000
Defroster_main 63T	96	22,411	46,072	0.000
dc_f2				
F2 20T	54	4,890	5,935	0.001
F2 20T	81	7,017	11,523	0.000
F2 23T	54	4,926	6,746	0.000
F2 23T	81	7,877	11,533	0.000
eurocheck-0.1.0				
main 7T	31	4,524	10,837	0.003
gimp-2.2.4				
gimp_hsv_to_rgb 34T	37	5,780	6,176	0.002
gimp_hsv_to_rgb4 19T	27	5,190	5,492	0.003
gimp_hsv_to_rgb4 29T	27	5,011	5,529	0.001
gimp_hsv_to_rgb4 34T	27	5,105	5,555	0.004
gimp_hsv_to_rgb4 39T	27	5,132	5,821	0.000
gimp_hwb_to_rgb 28T	27	799	1,286	0.002
space				
space_fixgramp 9T	32	7,520	10,980	0.000
space_fixsgrel 102T	524	12,363	18,609	0.006
space_fixsgrel 12T	524	27,813	43,619	0.000
space_fixsgrel 13F	524	29,056	43,309	0.000
space_fixsgrel 26T	524	24,423	40,396	0.007
space_fixsgrel 27T	524	27,389	42,567	0.001
space_fixsgrel 44T	524	29,060	45,287	0.000
space_fixsgrid 18F	101	8,408	12,959	0.000
space_fixsgrid 18F	120	16,128	26,318	0.000
space_fixsgrid 28F	120	15,811	24,909	0.000
space_fixsgrid 40F	101	8,781	11,836	0.002
space_fixsgrid 40F	120	17,748	24,920	0.000
space_fixsgrid 50F	101	3,479	5,627	0.000
space_fixsgrid 50F	120	15,033	24,886	0.000
spice-3f4				
cliparc 15F	59	17,417	21,062	0.002
cliparc 86F	59	6,407	10,395	0.000

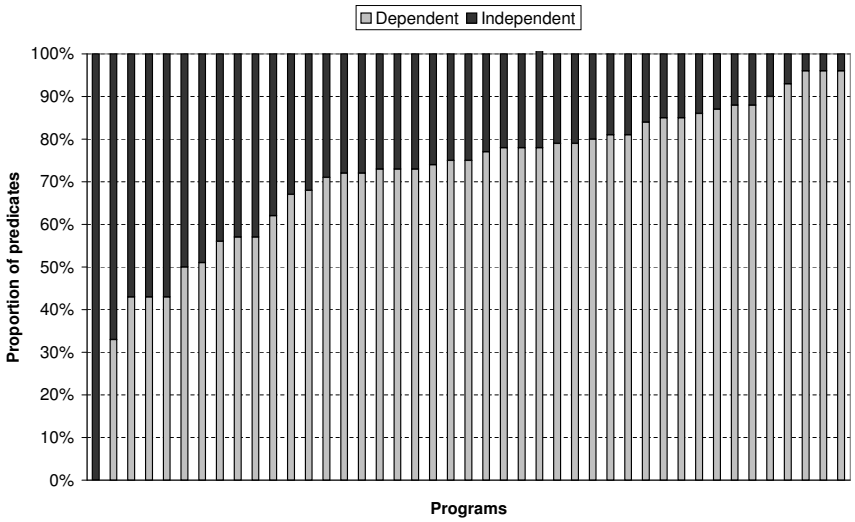


Fig. 5. Distribution of dependent and independent nested predicates for the programs of Table I

and decreased effort (*i.e.*, decreased average number of fitness evaluations) as seen in Figure 7b. The search is 100% successful for all branches with the transformed version of the code, whereas with the original program, the search becomes less effective in more deeply nested predicates and in larger domain sizes. The figure also depicts the relationship for dependent predicates. At shallower levels, performance is improved using the transformation, as the average number of fitness evaluations is lower (Figure 7d). At more deeply nested levels, however, the search with the transformation struggles. Considering all predicates at once, and their inter-dependencies (*i.e.*, having to keep several variables fixed to obey the equality operator in each predicate) makes the search less effective than if the predicates were considered one-at-a-time, as with the original version of the program (Figure 7c). At even deeper levels, however, the search fails using both transformed and original versions of the program.

In conclusion, the results with the synthetic test objects show that the transformation can always improve search performance on nested independent predicates, which from the first empirical study account for about one-fifth of all predicate chains. The relationship for dependent nested predicates is more complicated. At shallow levels, the transformation tends to increase search effectiveness and decrease effort. At deeper levels, however, search performance is less predictable and could be worse with the transformation.

7.2 Threats to Validity

An important part of any empirical study is to consider the threats to the validity of the experiment. This section briefly outlines these potential threats and how they were addressed. The hypotheses studied in this paper concerns the use of a testability transformation to remove nesting in a program for coverage of the branch and its impact on the evolutionary search for test data. One issue to address,

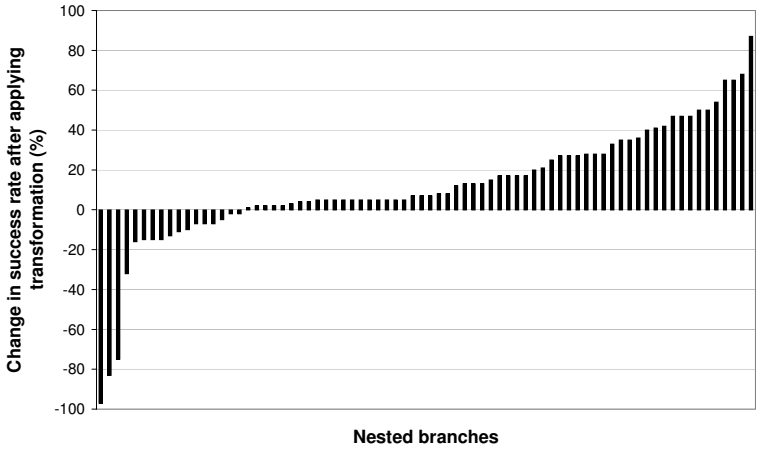


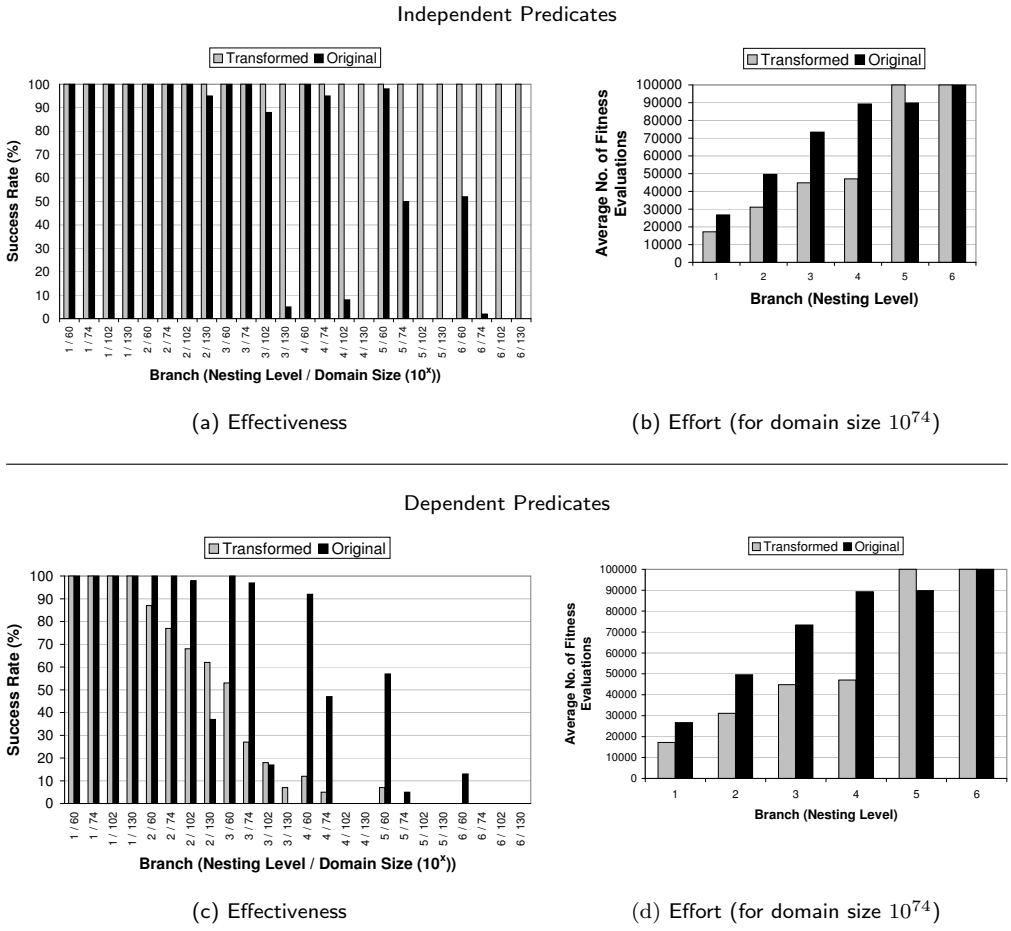
Fig. 6. Nested branches for which there was a change in search success rate after applying the transformation

therefore, is the so-called *internal validity* (*i.e.*, to check whether there has been a bias in the experimental design or an error in its implementation which could affect the causal relationship under study).

One source of error could come from the transformation steps being performed incorrectly. In order to circumvent this, the process was performed automatically with the help of the `javacc` Java Parser Generator and a C grammar. A sample set of transformations were then examined manually to ensure that the process was being performed properly.

A potential source of bias comes from the inherent stochastic behaviour of the evolutionary search algorithm under study. The most reliable (and widely used) technique for overcoming this source of variability is to perform tests for statistical significance on a sufficiently large sample of result data. Such a test is required whenever one wishes to make the claim that one technique produces superior results to another. A set of results are obtained from a set of runs (essentially sampling from the population of random number seeds). In order to compare the performance of the search using transformed and original versions, measured using fitness evaluations, a test was performed to see if there is a statistical significant difference in the means. For the results reported upon here, the t–test was used with the confidence level set at 99%. In order for the t–test to be applicable, it is important to have a sample size of at least 30. To ensure that this constraint was comfortably achieved, each experiment was repeated 60 times.

Another source of bias comes from the selection of the programs to be studied. This impacts upon the *external validity* of the empirical study. That is, the extent to which it is possible to generalise from the results obtained. Naturally, it is impossible to sample a sufficiently large set of programs such that the full diversity of all possible programs could be captured. The rich and diverse nature of programs makes this an unrealistic goal. However, where possible, a variety of programming styles and sources have been used. The study draws upon code from real world programs, both from industrial production code and from open source. Furthermore,



(a) Effectiveness (b) Effort (for domain size 10^{74})

(c) Effectiveness (d) Effort (for domain size 10^{74})

Fig. 7. Search effectiveness and effort for independent and dependent predicates with original and transformed versions of the program for each branch

it should be noted that the number of test problems considered is 394, providing a relatively large pool of results from which to make observations.

Nonetheless, caution is required before making any claims as to whether these results would be observed on other programs, possibly from different sources and in different programming languages. As with all such experimental software engineering, further experiments are required in order to replicate the results contained here.

However, the results show that there do indeed exist cases where the effectiveness of test data generation is improved with the transformation, and effort-wise, there is a statistically significant relationship between the use of transformation for nested branches and decreased 'cost' of search algorithms for test data generation.

8. DISCUSSION

It is possible for the transformation to have issues with the removal of certain types of predicates, although these issues were not encountered in the course of the empirical study. One example 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. The removal of the `if` statement may result in the `printf` statement being executed with a value of `i` that is out of range, thus causing an array out of bounds error:

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

A similar example is the testing of a pointer to see if it points to anything (*i.e.*, whether it is `NULL`). Whilst these types of predicate did appear in code used in the empirical study, they were nested in loops. Thus the transformation could not be applied and as such no abnormal termination errors occurred.

Another issue is the possibility of introducing arithmetic errors. For example in the following segment of code, a division by zero error may result if the conditional were to be removed:

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

The test data generation empirical study revealed cases where the transformation led to poorer search performance (*i.e.*, nested predicates which were dependent). A practical strategy for applying the transformation would therefore be to only apply it to dependent nested predicates if test data cannot be found using the original version of the program. The empirical study which investigated prevalence of nesting found that just under one fifth of nested predicates are independent. For these independent predicates, significant improvements in efficiency are possible.

9. RELATED WORK

This paper has used an approach known as testability transformation, introduced by Harman et al. [2004] as a means of adapting traditional program transformation to allow it to improve the effectiveness of automated test data generation techniques. A testability transformation need not preserve the traditional meaning of the program it transforms. Rather, it need only preserve the sets of adequate test data for the programs studied. This has been found to be applicable to a number of testing problems [Baresel et al. 2004; Hierons et al. 2005; Korel et al. 2005].

Test data generation is a process that is generally performed by hand in industry. This practice is extremely costly, difficult, and laborious. Search–based approaches to testing like evolutionary testing can automate this process and is thus an important research area. Therefore, the solution to the nested predicate problem is also important, because it hinders evolutionary testing - as shown in this paper.

Baresel et al. [2002] study fitness evaluation of composed conditions in C, which is similar to the nesting problem due to the short-circuiting of the `&&` and `||` operators.

In the following piece of code

```
if (a == b && b == c)
{
    // ...
```

the condition `b == c` will not be evaluated until `a == b`, because the `&&` operator breaks off evaluation of the entire condition early. It is noted that if no side-effects exist in the condition, both sub-conditions could be evaluated for the purposes of computing the fitness function. An experimental study shows that the search is more efficient when this is performed. Also discussed is the nested predicate 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 fitness function, in a similar way to the composed condition problem. However, no empirical work was performed. The testability transformation presented in this paper can be applied to these simple cases and more complicated situations where intervening code does exist between nested predicate pairs.

This paper is concerned with the application of evolutionary algorithms to test data generation. Other methods have been proposed, including techniques based on symbolic execution [Boyer et al. 1975; Clarke 1976; King 1976; DeMillo and Offutt 1991], the goal-oriented approach [Korel 1992], and the chaining approach [Ferguson and Korel 1996]. Symbolic execution encounters difficulties with loops and dynamic memory. The goal-oriented approach uses local search to find test data which will execute each nested condition en route to the target node, one after the other. Thus, it too suffers from the nested predicate problem. The use of local search also means that the method cannot escape from local optima in the search space. The chaining approach is concerned with finding data dependencies which may affect the outcome at some problem branching node, at which the flow of execution cannot be changed. The search for data dependencies cannot help the nested predicate problem, as it is rooted in issues of control flow. In fact, the chaining approach is likely to exacerbate the problem, since in the method, the data dependencies also need to be executed, and these may also be nested.

An early method of Miller and Spooner [Miller and Spooner 1976] partially solves the nested target problem. However a straight line version of the program must be produced leading to the structural target of interest. Furthermore, local search is used. Xanthakis et al. [1992] use genetic algorithms, but a full path needs to be specified by the tester. Neither a straight line version of the program, nor the specification of a path up to the nested target are required by the testability transformation approach presented in this paper.

Methods using simulated annealing [Tracey et al. 1998b; Tracey et al. 1998] have also been proposed, but these also follow a strategy of satisfying one nested predicate after another, and therefore also fail to solve the nested target problem. Thus the method proposed in this paper could be further applied here.

In more recent work, Harman and McMinn [Harman and McMinn 2007] applied a theoretical and empirical analysis of the use of genetic algorithms for test data generation, comparing their performance with hill climbing and random search.

Harman et al. [2007] further consider the application search space reduction through the removal of input variables not relevant to a structural target through the use of program analysis.

10. SUMMARY AND FUTURE WORK

This paper has described how targets nested within more than one conditional statement can cause problems for evolutionary structural test data generation. In the presence of such nesting, the search is forced to concentrate on satisfying one branch predicate at a time. This can slow search progress and artificially restricts the potential search space available for the satisfaction of branching predicates 'later' in the sequence of nested conditionals. The paper presented a first empirical study that demonstrated the prevalence of nesting in practice and the possible application sites for the transformation in just under 800,000 lines of production code. A second empirical study showed that evolutionary test data generation can be improved in terms of effectiveness and efficiency for many branches in real-world code. Statistical t-tests showed that performance could be improved significantly in many of these cases. The empirical study showed that improvement gains are always possible where the predicates in nested conditional statements are independent of one another, but improvement is less predictable for dependent predicates.

Future work aims to further tackle the problem of dependent predicates, attempting to remove or reduce local optima in the fitness landscape that might result from an ordering or non-ordering of nested predicate consideration. It also aims to extend the approach to targets nested in loops.

ACKNOWLEDGMENT

The authors would like to thank Joachim Wegener and DaimlerChrylser for providing the two industrial examples used in the empirical study.

REFERENCES

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

Baresel, A., Binkley, D., Harman, M., and Korel, B. 2004. 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)*. ACM, Boston, Massachusetts, USA, 43–52.

Baresel, A. and Sthamer, H. 2003. Evolutionary testing of flag conditions. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724*. Springer-Verlag, Chicago, USA, 2442 – 2454.

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

Boyer, R. S., Elspas, B., and Levitt, K. N. 1975. SELECT - A formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*. ACM Press, 234–244.

Clarke, L. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering 2,* 3, 215–222.

DeMillo, R. A. and Offutt, A. J. 1991. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering 17,* 9, 900–909.

Do, H., Elbaum, S., and Rothermel, G. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering 10,* 4, 405 – 435.

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

Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., and Wegener, J. 2007. The impact of input domain reduction on search-based test data generation. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2007)*. ACM Press, Cavat, near Dubrovnik, Croatia, 155–164.

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

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

Harman, M. and McMinn, P. 2007. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the International Symposim on Software Testing and Analysis (ISSTA 2007)*. ACM Press, London, UK, 73–83.

Hierons, R., Harman, M., and Fox, C. 2005. Branch-coverage testability transformation for unstructured programs. *The Computer Journal 48,* 4, 421–436.

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

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

King, J. 1976. Symbolic execution and program testing. *Communications of the ACM 19,* 7, 385–394.

Korel, B. 1990. Automated software test data generation. *IEEE Transactions on Software Engineering 16,* 8, 870–879.

Korel, B. 1992. Dynamic method for software test data generation. *Software Testing, Verification and Reliability 2,* 4, 203–213.

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

Korel, B., Harman, M., Chung, S., Apirukvorapinit, P., and R., G. 2005. Data dependence based testability transformation in automated test generation. In *16th International Symposium on Software Reliability Engineering (ISSRE 05)*. Chicago, Illinios, USA, 245–254.

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

McMinn, P., Binkley, D., and Harman, M. 2005. Testability transformation for efficient automated test data search in the presence of nesting. In *Proceedings of the UK Software Testing Workshop (UKTest 2005)*. University of Sheffield Computer Science Technical Report CS-05-07, 165–182.

McMinn, P. and Holcombe, M. 2006. Evolutionary testing using an extended chaining approach. *Evolutionary Computation 14*, 41–64.

Miller, W. and Spooner, D. 1976. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering 2,* 3, 223–226.

Mühlenbein, H. and Schlierkamp-Voosen, D. 1993. Predictive models for the breeder genetic algorithm: I. continuous parameter optimization. *Evolutionary Computation 1,* 1, 25–49.

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

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

Tracey, N. 2000. A search-based automated test-data generation framework for safety critical software. Ph.D. thesis, University of York.

Tracey, N., Clark, J., and Mander, K. 1998a. 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)*. 73–81.

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

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

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

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

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

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

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

Xanthakis, S., Ellis, C., Skourlas, C., Le Gall, A., Katsikas, S., and Karapoulios, K. 1992. 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*. Toulouse, France, 625–636.