FlagRemover: A Testability Transformation for Transforming Loop Assigned Flags [1]

DAVID W. BINKLEY
Loyola College in Maryland
and
MARK HARMAN and KIRAN LAKHOTIA
King's College London, CREST

Search–Based Testing is a widely studied technique for automatically generating test inputs, with the aim of reducing the cost of software engineering activities that rely upon testing. However, search–based approaches degenerate to random testing in the presence of flag variables, because flags create spikes and plateaux in the fitness landscape. Both these features are known to denote hard optimization problems for all search–based optimization techniques. Several authors have studied flag removal transformations and fitness function refinements to address the issue of flags, but the problem of loop–assigned flags remains unsolved. This paper introduces a testability transformation along with a tool that transforms programs with loop–assigned flags into flag–free equivalents, so that existing search–based test data generation approaches can successfully be applied. The paper presents the results of an empirical study that demonstrates the effectiveness and efficiency of the testability transformation on programs including those made up of open source and industrial production code, as well as test data generation problems specifically created to denote hard optimization problems.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Algorithms

Additional Key Words and Phrases: Evolutionary Testing, Testability Transformation, Flags, Empirical Evaluation

1. INTRODUCTION

Software test input generation has remained a topic of interest for Software Engineering research and practice for three decades. The topic retains its importance because of the enormous cost of inadequate testing [NIST 2002] and the labour–intensive nature of the test data generation process as currently practiced. This

[1] An earlier version of this paper appeared at the International Symposium on Software Testing and Analysis 2004

Author's address: K. Lakhotia, King's College London, CREST, DCS, Strand, London, WC2R 2LS, UK.

reliance on human–centric test input construction makes testing more tiresome, expensive and unreliable than it might be were the generation of test inputs to be automated. Full automation of the test input generation process remains an attractive, though hard open problem.

Several techniques have been proposed for automating test input generation. Of these, Search–Based Testing (SBT) is an approach that has received increasing interest and which has formed the subject of over one hundred and fifty recent papers[1]. Search–based test data generation [Clark et al. 2003; Harman and Jones 2001; Harman 2007] uses metaheuristic algorithms to generate test data. Metaheuristic algorithms combine various heuristic methods in order to find solutions to computationally hard problems where no problem specific heuristic exists.

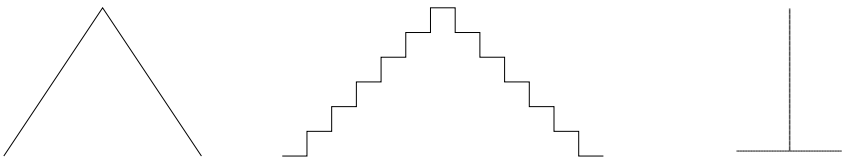
As an optimization technique, SBT incrementally improves either a single, or a pool of candidate solutions. This iterative process continues until either a satisfactory or ideal solution has been found, or another stopping criterion been reached. Due to its nature, SBT works particularly well for problems where the value of a candidate solution can easily be represented numerically in terms of a *fitness function*. A fitness function produces higher values for better solutions and thus allows ranking of solutions based on their fitness value.

SBT has repeatedly shown to be successful [Jones et al. 1996; Jones et al. 1998; Michael et al. 2001; Mueller and Wegener 1998; Pargas et al. 1999; Pohlheim and Wegener 1999; Tracey et al. 1998b; Wegener et al. 1996; Wegener et al. 1997], not only for structural (white box) testing, but also for other forms of testing such as temporal testing [Wegener et al. 1997] or stress testing [Briand et al. 2005]. McMinn [McMinn 2004] provides a comprehensive survey of work on search–based test data generation.

The two most commonly used algorithms in SBT are a hill climb variant known as the Alternating Variable Method (AVM) [Korel 1990] and Evolutionary Algorithms (EAs) [Holland 1975; Mitchell 1996]. EAs are part of the family of metaheuristic algorithms, and the use of EAs for testing is known as Evolutionary Testing (ET). EAs distinguish themselves from other search–based algorithms by applying genetic operations, such as crossover or mutation, to a pool of individuals, known as a population. Each individual in the population represents input parameters to a program or function for structural testing. In a typical EA, the population is updated over a sequence of generations. The selection of individuals who survive into the next generation is governed by a pre–defined selection strategy, based around the fitness values produced by the fitness function. Between each generation, genetic operators are applied to the individuals, loosely representing the effects of mating and mutation in natural genetics. The net effect of these operations is that the population becomes increasingly dominated by better (more fit) individuals. The various steps of an evolutionary cycle are explained in more detail in Section 2.1.

When considering test data generation for achieving branch coverage, as is the case in this paper, the fitness value of an individual is computed in terms of how close it comes to executing a target branch. While some branches are easily covered,

[1]The source of this publication data is the repository of papers on Search–Based Software Engineering at http://www.sebase.org/sbse/publications/, accessed 21st February 2009.



Best case	**Acceptable case**	**Worst case**
Smooth landscape with *ubiquitous* guidance toward global optimum.	Rugged landscape with *some* guidance toward global optimum.	Dual plateau landscape with *no* guidance toward global optimum.

Fig. 1. This figure uses three fitness landscapes to illustrate the effect flag variables have on a fitness landscape, and the resulting 'needle in a haystack' problem.

even by simple methods such as random testing [Harman and McMinn 2007], it is the remaining uncovered branches which challenge test data generation techniques and where SBT provides an attractive solution [Michael et al. 2001; Pargas et al. 1999; Wegener et al. 2001].

Although SBT works well in many situations, it is hampered by the presence of flag variables: variables that hold one of two discrete values: true or false. One place where flag variables are common is in embedded systems, such as engine controllers, which typically make extensive use of flag variables to record state information concerning devices. Such systems can therefore present problems for automated test data generation. This is important, because generating such test data by hand (which is often the case in practice) is prohibitively expensive, yet, having test data is required by many testing standards [British Standards Institute 1998b; Radio Technical Commission for Aeronautics 1992].

The flag problem is best understood in terms of the *fitness landscape*. A fitness landscape is a metaphor for the 'shape' of the hyper–surface produced by the fitness function. In the 2 dimensional case (*i.e.,* one input and one fitness value), the position of a point along the horizontal axis is determined by a candidate solution (*i.e.*, an input to the program) and the height of a point along the vertical axis is determined by the computed fitness value for this input. Using the fitness landscape metaphor, it becomes possible to speak of landscape characteristics such as plateaus and gradients.

As illustrated in the right hand side of Figure 1, the use of flag variables leads to a degenerate fitness landscape with a single, often narrow, super–fit plateau and a single super–unfit plateau. These correspond to the two possible values of the flag variable. While this landscape is not a problem for symbolic execution based techniques, it is well–known to be a problem for many search–based techniques; the search essentially becomes a random search for the 'needle in a haystack' [Baresel and Sthamer 2003; Bottaci 2002a; Ferguson and Korel 1996; Harman et al. 2004].

This paper presents an algorithm for transforming programs containing loop–assigned flag variables, which cannot be handled by previous approaches. The result of the transformation is a tailored version of a program that allows existing approaches to compute representative fitness values for candidate solutions at a particular flag–controlled branch. It uses a testability transformation [Harman

et al. 2004], a form of transformation in which functional equivalence need not be preserved, but in which test set adequacy is preserved. The primary contributions of this paper are as follows:

(1) A testability transformation algorithm is described which can handle flags assigned in loops.
(2) Results of two empirical studies evaluating the algorithm are reported. They show that the approach reduces test effort and increases test effectiveness. The results also indicate that the approach scales well as the size of the search–space increases.
(3) Results from a third empirical study show that the loop–assigned flag problem is prevalent in real programs, thereby validating the need for algorithms to deal with flags in general and loop–assigned flags in particular.

The rest of the paper is organized as follows. Section 2 provides an overview of background information on ET, the flag problem, and testability transformation. Section 3 introduces the flag replacement algorithm and Section 4 outlines how it has been implemented. Section 5 presents an empirical study which demonstrates that the approach improves both test generation effort and coverage achieved and explores the performance of the approach as the size of the search space increases. Section 6 presents the empirical study of loop–assigned flags and examples of real world code that contain loop–assigned flags. Section 7 examines related work and Section 8 concludes.

2. BACKGROUND

This section briefly explains the flag problem and the general characteristics of the testability transformation solution proposed.

2.1 Evolutionary Test Data Generation

The empirical results reported herein were generated using the Daimler Evolutionary Testing system [Wegener et al. 2001], built on top of the Genetic and Evolutionary Algorithm Toolbox [Pohlheim], using a client–server model, and AUSTIN [Lakhotia et al. 2008], a search–based testing tool for programs containing pointer inputs. Figure 2 provides an overview of a typical evolutionary testing process, where the outer circle depicts a typical procedure for an EA: First, an initial population is formed, usually with random guesses. Each individual within the population is evaluated by calculating its fitness value via the fitness function. Starting with randomly generated individuals results in a spread of solutions ranging in fitness because they are scattered around different regions of the search–space.

Subsequently pairs of individuals are selected from the population, according to a pre–defined selection strategy, and combined by the crossover operator to produce new solutions. Once the individuals have been formed, mutation is applied. This mimics the role of mutation in natural genetics, introducing new information into the population. The evolutionary process ensures that productive mutations have a greater chance of survival than less productive ones.

The cycle concludes an iteration by re–evaluating the new individuals with regards to their fitness. Survivors into the next generation are chosen from both

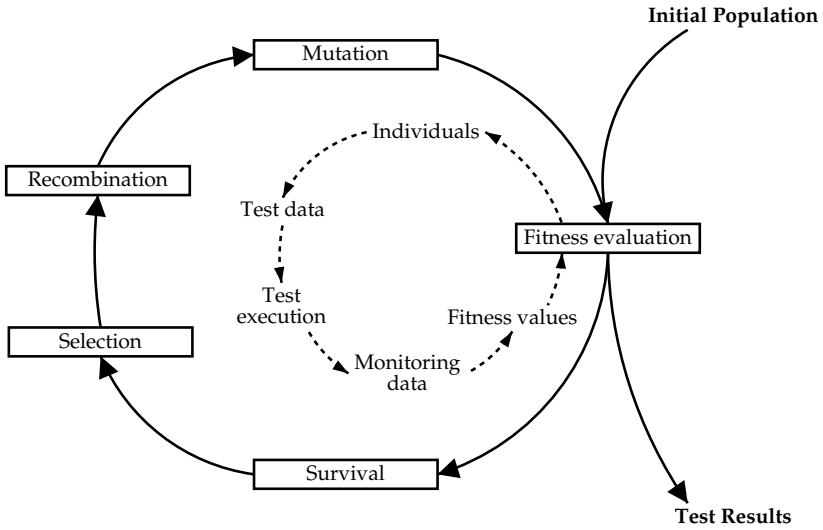


Fig. 2. Evolutionary Algorithm for Testing

parents and offspring, depending on their fitness values and the selection strategy. Typically, 'fitter' individuals survive. However, less fit individuals retain a chance of being carried across into the next generation, thereby maintaining diversity within a population. Diversity is important especially during the early stages of an EA to avoid pre-mature convergence at a local optimum. The algorithm is iterated until the (global) optimum is achieved, or some other stopping condition is satisfied.

At present EA techniques are less efficient than structural analysis techniques for most types of test data generation (*e.g.*, code–based test criteria) [Lakhotia et al. 2009; Harman and McMinn 2007]. In practice, this means they should be applied where other techniques fail to generate test data for a particular subset of structures (*e.g.*, branches). In this way the cost can be amortized.

For software testing to be automated with the aid of EAs, the test aim must be transformed into an optimization task. This is the role of the inner circle depicted in Figure 2. Each generated individual represents a test datum for the system under test. Depending on the test aim pursued, different fitness functions apply for test data evaluation.

If, for example, the temporal behaviour of an application is being tested, the fitness evaluation of the individuals is based on the execution times measured for the test data [Puschner and Nossal 1998; Wegener and Mueller 2001]. For safety tests, the fitness values are derived from pre– and post–conditions of modules [Tracey et al. 1998a], and for robustness tests of fault–tolerance mechanisms, the number of controlled errors forms the starting point for the fitness evaluation [Schultz et al. 1993].

For structural criteria, such as those upon which this paper focuses, a fitness function is typically defined in terms of the program's predicates [Baresel and Sthamer 2003; Bottaci 2002a; Jones et al. 1996; Michael et al. 2001; Pargas et al. 1999; Wegener et al. 2001]. It determines the fitness of candidate test data, which in

turn, determines the direction taken by the search. The fitness function essentially measures how close a candidate test input drives execution to traversing a desired (target) path or branch.

2.2 The Flag Problem

In this paper, a flag variable will be deemed to be any variable that takes on one of two discrete values. Boolean variables are used in the examples. The flag problem deals with the situation where there are relatively few input values (from some set S) that make the flag adopt one of its two possible values. This problem typically occurs with internal flag variables, where the input state space is reduced, with relatively few 'special values' from S being mapped to one of the two possible outcomes and all others being mapped to the other of the two possible flag values. As explained below, the flag problem is the hardest of what is commonly known as the internal variable problem in automated test data generation.

Consider a predicate that tests a single flag variable (*e.g.*, `if ( flag )`). The fitness function for such a predicate yields one of two values: either maximal fitness (for 'special values') or minimal fitness (for any other value). As illustrated in the right of Figure 1, the landscape induced by such a fitness function provides the search with no guidance.

A similar problem is observed with any n–valued enumeration type, whose fitness landscape is determined by n discrete values. The boolean type (where $n = 2$) is the worst case. As n becomes larger the program becomes increasingly more testable: provided there is an ordering on the set of n elements, the landscape becomes progressively smoother as the value of n increases.

The problem of flag variables is particularly acute where a flag is assigned a value inside a loop and is subsequently tested outside the loop. In this situation, the fitness function computed at the test outside the loop may depend upon values of 'partial fitness' computed at each and every iteration of the loop. Previous approaches to handling flags break down in the presence of such loop–assigned flags [Baresel and Sthamer 2003; Bottaci 2002a; Harman et al. 2004].

2.3 Testability Transformation

A testability transformation [Harman et al. 2004] is a source–to–source program transformation that seeks to improve the performance of a previously chosen test data generation technique. Testability transformations differ from traditional transformations [Darlington and Burstall 1977; Partsch 1990; Ward 1994] in two ways:

(1) The transformed program produced is merely a 'means to an end', rather than an 'end' in itself. The transformed program can be discarded once adequate test data has been generated. By contrast, in traditional transformation, the original program is replaced by the transformed equivalent.

(2) The transformation process need not preserve the standard semantics of a program. For example, in order to cover a chosen branch, it is only required that the transformation preserves the set of test–adequate inputs. That is, the transformed program must be guaranteed to execute the desired branch under the same initial conditions as the untransformed program. By contrast, tradi-

tional transformation preserves functional equivalence, a much more demanding requirement.

These two observations have important implications:

(1) **There is no psychological barrier to the transformation**. Traditional transformation requires the developer to replace familiar code with machine–generated, structurally altered equivalents. It is part of the folklore of the program transformation community that developers are highly resistant to the replacement of the familiar by the unfamiliar. There is no such psychological barrier for testability transformations: The developer submits a program to the system and receives test data. There is no replacement requirement; the developer does not even need to be aware that a transformation has taken place.

(2) **Considerably more flexibility is available in the choice of transformation to apply**. Guaranteeing functional equivalence can be demanding, particularly in the presence of side effects, goto statements, pointer aliasing, and other complex semantics. By contrast, merely ensuring that a particular branch is executed for an identical set of inputs is comparatively less demanding.

(3) **Transformation algorithm correctness becomes a less important concern**. Traditional transformation replaces the original program with the transformed version, so correctness is paramount. The cost of 'incorrectness' for testability transformations is much lower; the test data generator may fail to generate adequate test data. This situation can be detected, trivially, using coverage metrics. By contrast, functional equivalence is *undecidable*.

3. THE FLAG REPLACEMENT ALGORITHM

The aim of the replacement algorithm is to substitute the use of a flag variable with a condition that provides a smoother landscape. Prior work with flag variables requires that assignments reaching a use do not occur within a loop [Baresel and Sthamer 2003; Bottaci 2002a; Harman et al. 2004]. By contrast, the algorithm presented in this paper handles flags assigned inside a loop. It does this by introducing two new real valued variables, fitness and counter. These variables replace the predicate use of a flag with an expression that supports a distance based calculation (*e.g.*, if (counter == fitness)) to be used.

The addition of these variables is a form of instrumentation. The variable counter is an induction variable added to count the number of assignments to a flag in all loop iterations. The variable fitness collects a cumulative fitness score from a local fitness function for the flag assignments during loop execution.

Before the formal presentation of the algorithm, the transformation is illustrated to provide some initial intuition. To begin with, Figure 3(a) shows an untransformed program, which contains a single flag variable. In addition to serving as an illustration of the transformation, this program will be used in the empirical study because it denotes the worst possible case for structured code: as the size of the array a increases, the difficultly of the search problem increases. Metaphorically speaking, the needle (all array entries equal to zero) is sought in an increasingly larger haystack.

For illustration, suppose that the goal is to execute the branch at Node 6. To realize this goal requires finding array values that avoid traversing the true branch of Node 3 because if an input causes the program to pass through Node 4, the target branch will be missed. The program in Figure 3(a) produces the landscape shown at the right of Figure 1. Transforming this program to count the number of times the predicate at Node 3 is false, produces the landscape shown at the middle of Figure 1. The transformed program in shown in Figure 3(b). In essence, the counting drives the search away from executing Node 4 because fitness receives a value closer to counter the more times Node 4 is missed.

However, this *coarsely* transformed version does not provide the search with any guidance on finding inputs that make a particular array element zero. It only favours such inputs once found. Thus the stair–step landscape of the middle of Figure 1. The *fine–grained* transformed version, shown in Figure 3(c) calls a *local* fitness function in the true branch of Node 3 that helps guide the search towards individual array values being zero. In this case, the local fitness measures how close the input was at this point to avoiding Node 4.

Local fitness is computed by *negating* the predicate condition at Node 3 and calculating a distance d for the negated predicate, based on a set of rules described by Bottaci [Bottaci 2002a]. In the example, d is equal to the i^{th} value of a, indicating how close $a[i]$ was to being 0 and thus traversing the false (desired) branch of Node 3. Figure 3(d) presents a portion of the local fitness function used in the case of the example function. This portion is for the operator '!='.

After transformation, it is possible to simplify the transformed program by taking the slice [Binkley and Gallagher 1996; Tip 1994; Weiser 1984] with respect to the condition in the transformed predicate. Slicing removes unnecessary parts of the program and thus forms a program specialized to the calculation of a smooth fitness landscape targeting the test goal. This optimization can be used for a variety of test data generation techniques and is independent of the flag replacement algorithm. Any branch in the program may be used as the slicing criterion.

The formal transformation algorithm is presented in Figure 4. It assumes that flag is initially assigned true and might subsequently be assigned false. Clearly there is a complementary version of the algorithm which can be applied when the initial assignment to flag is false.

The rest of this section explains the algorithm's steps in detail. First, Step 1 ensures that all assignments to the variable flag are of the form flag=true or flag=false. This is done by replacing any assignment of the form flag=C for some boolean expression C with if(C) then flag=true else flag=false. Step 2 adds an empty else block to all if statements as a place holder for later code insertions. Steps 3 and 4 simply insert the fitness accumulation variable, fitness, and the assignment counter, counter, both initialized to 0 prior to the start of the loop.

Step 5 introduces the update of the fitness accumulation variable, fitness, and the loop counter, counter. It has three cases. The first, Case 5.1, checks for the special situation when the loop degenerates into a simple assignment. In Cases 5.2 and 5.3 the value added to fitness depends upon the value assigned to flag along the associated path. If flag is assigned true (Case 5.2) then, in essence, assignments in previous loop iterations are irrelevant. To account for this, fitness is assigned the

```
     void f(char a[SIZE]){
        int i;
(1)     int flag = 1;
(2)     for(i=0;i<SIZE;i++){
(3)        if(a[i]!=0)
(4)           flag=0;
        }
(5)     if(flag)
(6)        /*target*/
     }
```

(a) No transformation

```
     void f(char a[SIZE]){
        int i;
        int flag = 1;
        double counter = 0.0;
        double fitness = 0.0;
        for(i=0;i<SIZE;i++){
           if (a[i] != 0){
              counter++;
              flag = 0;
           }else{
              fitness++;
              counter++;
           }
        }
        if(fitness == counter)
           /*target*/
     }
```

(b) Coarse–grained transformation

```
void f(char a[SIZE]){
  int i;
  int flag = 1;
  double counter;
  double fitness;
  char __cil_tmp1;
  char __cil_tmp2;
  double f;
  counter = 0.0;
  fitness = 0.0;
  for(i=0;i<SIZE;i++){
   if (a[i] != 0){
    __cil_tmp1 = a[i];
    __cil_tmp2 = 0;
    f =
    local(__cil_tmp1, "!=",
          __cil_tmp2);
    f = normalize(f)
    flag = 0;
    fitness += f;
    counter++;
   } else {
    counter++;
    fitness++;
   }
  }
  if(fitness == counter)
   /*target*/
}
```

(c) Fine–grained transformation

```
double normalize(double dist){
   return 1 - pow(1.001, -dist);
}

double
local(char arg1, char* op,){
      char arg2){
  double dist;
  if(strcmp(op, "!=") == 0){
     dist = abs(arg1 - arg2);
     if (dist == 0)
        return 0;
     else
        return (dist + 1);
  }
  else if(strcmp(op, "==") == 0){
     ...
  }
}
```

(d) Local fitness function

Fig. 3. An example program before and after applying the coarse and fine–grain transformations. The figures also shows part of the function for computing local fitness.

Suppose that flag is assigned to true outside the loop and that this is to be maintained.

Step 1 Convert all flag assignments to assignments of constants by replacing flag=C with if(C) then flag=true else flag=false for some (side effect free) boolean expression C.

Step 2 Convert any if - then statements that contain a (nested) assignment of flag into if - then - else statements. The added empty branch is filled by Case 5.3 of Step 5 with 'bookkeeping' code.

Step 3 Add variable counter=0 as an initialization prior to the loop.

Step 4 Add an assignment fitness=0 as an initialization prior to the loop.

Step 5 There are three cases for assignments to flag based on the paths through the loop body.

Case 5.1: If all leaves of the AST contain the assignment flag=false (*i.e.*, entering the loop means certain falseness), then the entire loop is treated as flag=!C assuming the original loop is while(C). Otherwise, do the following for each leaf in the loop's AST that assigns to flag.

Case 5.2: flag is assigned true. Increment counter and assign value of counter to fitness immediately after the assignment to flag.

Case 5.3: flag is assigned false.

Step 5.3.1 Create a set, π, containing the critical branching nodes with respect to the flag assignment.

Step 5.3.2 For every critical branching node in π, insert an increment for both counter and fitness as the first instructions in the then or else branches of the node that leads away from the flag assignment (*i.e.*, the target of the branch CFG edge is not post–dominated by the flag assignment), if and only if, the target of the branch CFG edge is not post–dominated by another node in π. Do not add increments for counter and fitness otherwise.

Step 5.3.3 Collect the set of conditions s_c in π at which the assignment of false to flag can be avoided, *i.e.*, the conditions of those nodes in π that contain a branch CFG edge whose target is post–dominated by the flag assignment. Step 5.3.1 ensures that such a condition exists.

Step 5.3.4 For each condition c in s_c do the following.

Step 5.3.4.1 Save the values of the variables used in c in well typed, local, temporary variables for later use (local with respect to the function body, not the enclosing block).

Step 5.3.4.2 Insert the call f = local(...) as the first instruction in the then or else branch of the node containing c that leads towards the flag assignment (*i.e.*, the target of the branch CFG edge is post–dominated by the flag assignment). The function local is the standard local fitness function, and the temporary variables, alongside the binary operator used in c form the arguments of the function call local. As detailed in Section 4, the CIL infrastructure ensures c does not contain any logical operators.

Step 5.3.4.3 Normalize f to a value between 0 and 1.

Step 5.3.4.4 Add f to the existing value of fitness immediately after the flag assignment.

Step 5.3.4.5 Add an increment for counter immediately after the update to fitness (in Step 5.3.4.4).

Step 6 Replace if(flag) with if(fitness==counter).

Step 7 Slice at the replacement predicate if(fitness==counter), introduced by Step 6.

Fig. 4. The Transformation Algorithm

current value of counter (after it has been incremented). This assignment overwrites any previously accumulated fitness.

Case 5.3 addresses an 'undesired' assignment to flag. In this case flag is assigned false. The Control Flow Graph (CFG) is used to identify the set of critical branching nodes for the flag assignment in Step 5.3.1. Critical branching nodes are those decision nodes in a CFG where the flow of control may traverse a branch which is part of a path that can never lead to the flag assignment. In other words, these are the nodes on which the flag assignment is control dependent. Step 5.3.2 iterates over all critical branching nodes and checks if they contain a branch CFG edge which is not post–dominated by either the flag assignment or any other critical branching node for the flag assignment. For each critical branching node which satisfies this requirement, Step 5.3.2 adds an increment of 1 to both counter and fitness as the first instructions to the branch that is not part of the path leading to the flag assignment. This also addresses the case when flag remains unassigned during a path through the loop.

Next, Step 5.3.3 collects the conditions of those branching nodes, which contain a branch CFG edge whose target is post–dominated by the flag assignment. For each of those conditions, Step 5.3.4 implements the more fine–grained approach producing a landscape more like that shown in the left of Figure 1. Smoothing of the fitness landscape improves the search. Here, if no changes to fitness were made, the resulting fitness landscape degenerates to the coarse–grained landscape shown in the middle of Figure 1. Instead Step 5.3.4 implements the transformation shown in Figure 3(c).

Steps 5.3.4.1 and 5.3.4.2 add the necessary instrumentation to compute a fitness increment for the path taken by an input. The result of the fitness computation is saved in a local variable, whose value is normalized in Step 5.3.4.3.

The key observation behind Steps 5.3.4.1 – 5.3.4.3 is that an assignment of false to flag occurs because a 'wrong decision' was taken earlier in the execution of the program. The algorithm therefore backtracks to this earlier point. That is, it finds a point at which a different decision (the decision c of Step 5.3.4) could avoid the assignment of false to flag. The value calculated (in Step 5.3.4.2) for the fitness increment in this case is based upon the standard approach to local fitness calculation in evolutionary testing [Wegener et al. 2001].

Finally, Step 5.3.4.4 adds the fitness increment to fitness immediately after the flag assignment, while Step 5.3.4.5 increments counter.

Step 6 replaces the use of flag with fitness==counter. Observe that the value of fitness can only equal the value of counter in two cases: Either the last assignment to flag in the loop was the value true and there has been no subsequent assignment to flag, or the variable flag has not been assigned in the loop (so its value remains true). In either case, the original program would have executed the true branch of the predicate outside the loop which uses flag. In all other cases, flag would have been false in the original program. For these cases, the value of fitness will be some value less than that of counter. How close together their values are is determined by how close the loop comes to terminating with flag holding the desired value true.

Step 7 is an optional optimization step. It can be ignored, without effecting the functional behaviour of the transformed program or the fitness landscape produced.

The motivation for Step 7 is to reduce the complexity of the program that is executed. Since search–based testing requires repeated execution of the program under test (in order to evaluate fitness of each test case considered), any speed–up will improve the efficiency of the overall approach.

It is important to note that the transformed program need not be semantically equivalent to the original. It is a new program constructed simply to mimic the behaviour of the original at the target branch. It does so in a way that ensures a more attractive fitness landscape. The standard search algorithm (with no modification) can be applied to the transformed program with the goal of finding test data to execute the branch controlled by the newly inserted predicate fitness==counter.

Finally, if flag is assigned in several loops, nested one within the other, then the algorithm is applied to the innermost loop first in order to obtain a fitness value for the innermost loop. This value can then be used as a partial result for the fitness of a single iteration of the enclosing loop. In this manner, the algorithm is applied to each enclosing loop, to accumulate a total fitness value.

4. IMPLEMENTATION

The algorithm has been implemented in a tool which is based on the CIL [Necula et al. 2002] infrastructure for C program analysis and transformation. CIL provides a number of pre–defined program analysis modules, such as control and data flow analysis. It also offers an extensive API (in Ocaml) to traverse the AST of the parsed source code. The tool itself is provided as an Ocaml module and can be run on any platform that has the Ocaml runtime installed.

4.1 Definition of Loop Assigned Flag

For the purpose of the tool, a flag f is considered to be loop–assigned, if and only if it satisfies the following properties:

(1) The definition f_{def} of f is a descendant of the loop statement l_s (*i.e.*, a while or for construct) in the AST
(2) There exists a definition free path for f from f_{def} to f_{use}, where f_{use} is a predicate use of f
(3) f_{use} is not a descendant of l_s in the AST. If it is, it must also be a descendant of another loop statement $l_{s'}$ in the AST.

Flags assigned within loops that arise in a CFG as part of unstructured code (*e.g.*, via the use of goto statements) are not considered to be loop–assigned by the tool, even though the algorithm proposed in Figure 4, in principle, does not necessitate this restriction. As a consequence, the tool might consider more flags to be loop–assigned than strictly necessary, while at the same time leaving loop–assigned flags that arise from unstructured code untransformed.

4.2 Flag Identification

The C language does not contain a dedicated boolean data type, so the question remains how to identify flag variables. Generally speaking, since the aim of the testability transformation is to transform spikes in a fitness landscape, the transformation algorithm does not need to identify flags in a semantically correct way.

The transformation can thus be applied to any variable whose use creates a spike in the fitness landscape. A syntactic check on predicates often suffices to identify such potential 'problem uses'.

Below are two examples of source code constructs that often cause a spike in the fitness landscape:

```
int foo(...){
  ...
  if(C)
    flag = 0;
  ...
  if(flag)//target
  ...
}
```

```
int foo(...){
  ...
  if(C)
    buffer = malloc(...);
  ...
  if(buffer)//target
  ...
}
```

Notice even though buffer is not a flag, the fitness landscape for the //target branch in the right column exhibits the same features as the flag controlled branch in the left column.

For the implementation described in this paper, a variable is considered to be a flag if it is used inside a predicate in one of these ways:

(1) if (variable)

(2) if (!variable)

(3) if (variable == constant)

(4) if (variable != constant)

4.3 Flag Removal

Before applying the transformation algorithm, the parsed source code needs to be simplified. To this end a number of CIL options, as well as some custom pre–processing transformations are used.

By default CIL transforms compound predicates into equivalent if - then - else constructs [2]. For in-line predicates containing logical operators, CIL uses temporary variables for its code transformation. Figure 5 illustrates the CIL code transformations with examples.

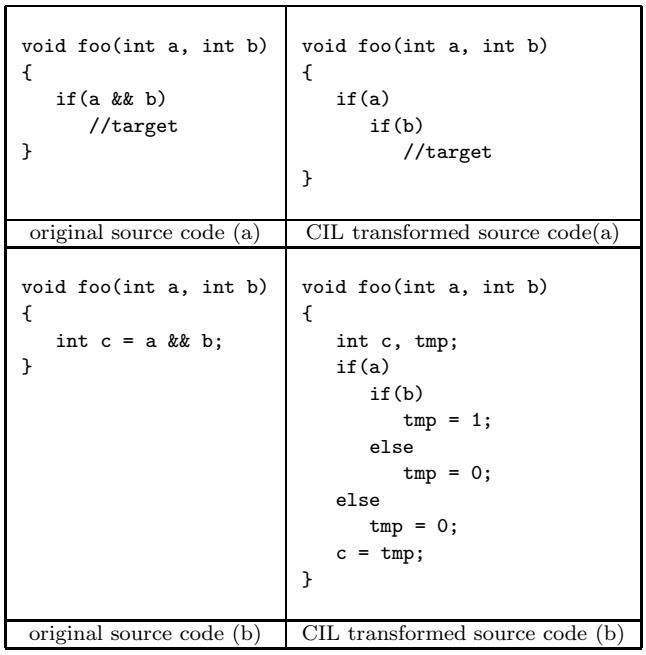
Besides the transformation of compound predicates, the tool requires the following code transformations prior to applying the flag transformation.

Simplify: This CIL option (–dosimplify) transforms the source code into simpler three–address code.

Simple Memory Operations: This option (–dosimpleMem) uses well–typed temporary variables in order to ensure each CIL lvalue involves at most one memory reference.

Prepare Control Flow Graph (CFG): This option (–domakeCFG) converts all

[2]The flag Cil.useLogicalOperators enables the use of logical operators. The tool uses CIL in its default mode where this flag is set to false.



```
void foo(int a, int b)
{
   if(a && b)
      //target
}
```

original source code (a)

```
void foo(int a, int b)
{
   if(a)
      if(b)
         //target
}
```

CIL transformed source code(a)

```
void foo(int a, int b)
{
   int c = a && b;
}
```

original source code (b)

```
void foo(int a, int b)
{
   int c, tmp;
   if(a)
      if(b)
         tmp = 1;
      else
         tmp = 0;
   else
      tmp = 0;
   c = tmp;
}
```

CIL transformed source code (b)

Fig. 5. Two examples illustrating how CIL transforms compound predicates into single predicate equivalents. The transformation works for both, predicates in conditional statements (see (a)), and in-line predicates (see (b)).

break, switch, default and continue statements and labels into equivalent if and goto constructs.

Uninline: This custom step converts all in-line predicates (without logical operators) into equivalent if - then - else statements. Further, this module also implements Step 1 from Figure 4.

The implementation of Case 5.3.1 from Figure 4, requires control dependence information for each statement. CIL provides a module to compute a function's CFG by identifying each statement's control predecessor and successor, as well as a module to compute the immediate dominator information for a statement. The tool combines these two modules by first inverting the edges of the CFG (adding a unique exit node when necessary), and then computing the immediate dominator information for the inverted CFG. This is equivalent to computing the post domination information for each statement. Based on the post dominator tree, the control dependence information is computed for each statement in the AST.

Next, flags are collected by iterating over the CIL AST, performing a syntactic check on if statements. The pre–processing steps ensure that all predicates appear in the form of if statements. When a predicate matches the flag pattern described in Section 2.2, information about the predicate and its parent statement (*i.e.*, the if statement) are stored in a hash table.

For each entry in the hash table, the tool uses the CIL reaching definitions module to collect the set of definition statements for a flag reaching the predicate use of

Test Subject	real	user	sys
synthetic examples	0.1252	0.0440	0.0568
EPWIC	0.3092	0.1888	0.0896
bibclean	0.1752	0.0816	0.0632
ijpeg	1.8326	1.7232	0.0584
time	0.1654	0.0760	0.0648
plot2d	1.7412	1.6128	0.0752
tmnc	0.1738	0.0920	0.0544
handle_new_jobs	0.1548	0.0664	0.0632
netflow	0.1454	0.0568	0.0648
moveBiggestInFront	0.1306	0.0648	0.0520
update_shps	0.1664	0.0816	0.0608

Table I. Runtime of the transformation (in seconds) for the test subjects as reported by the time utility. The measurements are averaged over five runs. The column **real** refers to the wall clock time, **user** refers to the time used by the tool itself and any library subroutines called, while **sys** indicates the time used by system calls invoked by the tool.

the flag. For each of these definitions, the tool checks whether they occur within a loop, and further that the flag use is not contained in the same loop. This is achieved by traversing the CIL AST. Loop assigned flags are labeled as such.

For each loop based flag assignment, the control dependence information of the containing statement is used to derive a local fitness function. An example is given in Figure 3(d). All flag variables of a given type share the same local fitness function. The necessary type information can easily be extracted via a call to CIL's typeof function, which returns the data type of its argument. Finally, the statement containing the predicate use of the flag is transformed as described in Step 6 of Figure 4.

The tool can be run in two modes. By default, the local fitness function is used to 'punish' an assignment to flag, as illustrated in Figure 3(c). Sometimes a flag assignment may be desired however, and thus the transformation can be used to guide the test data generation process towards the statement containing the flag assignment. In this mode, fitness is incremented by the local distance function (not inverting its second parameter) in the branches avoiding the flag assignment.

4.4 Runtime

For the transformation to be applicable in practice, the tool should perform in reasonable speed. The tool was therefore run on each of the test subjects used in Section 5, and timing information was recorded via the GNU time utility. The time measurements were collected on a HP Compaq 6715b laptop, with an AMD Turion 64 processor, running Ubuntu Intrepid. For each test subject, the runtime of the tool was recorded five times to allow for slight variations in the measurements reported by time. The data, averaged over the five runs, is shown in Table I. The tool did not require more than 2 seconds to apply the transformation to any of the test subjects.

4.5 Limitations

Currently the tool only implements an intraprocedural transformation. As a result, global flags are ignored by the tool, as are flags passed by reference. Furthermore, the tool does not include any alias analysis and, as a consequence, does not handle intraprocedural or interprocedural aliasing of flag variables. The tool further distinguishes between function assigned flags and other flags. Function assigned flags are variables whose value depends on the return value of a function call. These types of flags can be handled by a different testability transformation [Wappler et al. 2007]. Other kinds of flags include the loop–assigned flags addressed in this paper, and simply assigned flags addressed by previous work [Alshraideh and Bottaci 2006; Harman et al. 2004; Baresel and Sthamer 2003; Bottaci 2002b].

Both the algorithm presented in Figure 4 and the tool are incomplete in the presence of unstructured flow of control within the body of loops. In the work of Liu et al. [Liu et al. 2005] the authors present a synthetic example which illustrates this incompleteness. In practice we only observed this limitation in 2 out of 17 functions examined during the empirical study in Section 5. Nevertheless we aim to resolve this issue in future versions of the tool.

5. EMPIRICAL ALGORITHM EVALUATION

This section presents two empirical evaluations of the transformation algorithm's impact. It first reports on the application of the transformation to the synthetic 'needle in the haystack' example from Figure 3(a). After this, it considers the application of the transformation to a collection of flags extracted from several production systems.

The synthetic benchmark program was chosen for experimentation because it denotes the worst possible case for the search. Twenty versions of this program were experimented with using no transformation, the coarse–grained transformation, and the fine–grained transformation. In each successive version, the array size was increased, from an initial size of 1, through to a maximum size of 40. As the size of the array increases, the difficultly of the search problem increases; metaphorically speaking, the needle is sought in an increasingly larger haystack. This single value must be found in a search space, the size of which is governed by the size of the array, `a`. That is, the test data generation needs to find a single value (all array elements set to zero) in order to execute the branch marked `/* target */`.

The evaluation of the transformation on the twenty versions of the program was done using both the Daimler Evolutionary Testing system [Baresel et al. 2002; Wegener et al. 2001], and AUSTIN [Lakhotia et al. 2008]. For the test subjects listed in Table II only AUSTIN was used to evaluate the transformation.

The Daimler Evolutionary Testing system is capable of generating test data for C programs with respect to a variety of white box criteria. It is a proprietary system, developed in–house and provided to Daimler developers through an internal company web portal. A full description of the system is beyond the scope of this paper. AUSTIN is a search–based test data generation tool for C programs and uses a combination of the AVM and a set of constraint solving rules for pointers and dynamic data structures.

The AVM was shown to be particularly effective when applied to the branch

coverage adequacy testing criterion [Harman and McMinn 2007]. It is a type of hill climb which systematically explores the neighbourhood of each element in a program's input vector. The size of an element's neighbourhood move changes proportional to the number of successful moves for that element. Initially it consists of the smallest increment (decrement) for an element type (*e.g.*, ± 1 for integers). When a change in value leads to an improved fitness, the search tries to accelerate in that direction by making ever increasing moves. The formula used to calculate the size of a move is: $m_i = s^{it} * dir * acc_i$, where m_i is the move for the i^{th} input variable, s is the *repeat base* (2 by default) and it the repeat iteration of the current move, $dir \in \{-1, 1\}$, and acc_i the accuracy of the i^{th} input variable. The accuracy applies to floating point variables only. For all the experiments reported in this study the precision for these variables was set to 2 digits.

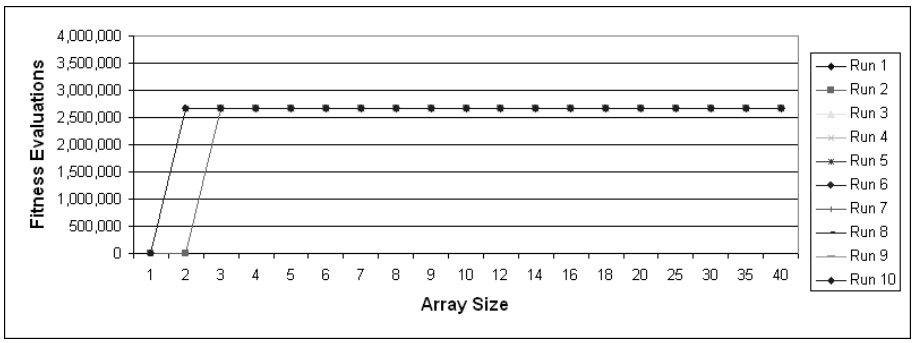
When no further improvements can be found for an element, the search continues by exploring the next element in the vector. Once the entire input vector has been exhausted, the search recommences with the first element if necessary. In case the search stagnates (*i.e.*, no move leads to an improvement) the search restarts at another randomly chosen location in the search space. This overcomes local optima and enables the hill climb to explore a wider region of the search space. A fitness budget, how many potential solutions the search is allowed to evaluate, ensures the search terminates when no solution can be found. For the purpose of this study, the fitness budget was set to $100,000$ evaluations for both AUSTIN and the Daimler Evolutionary Testing System. Due to the stochastic nature of both tools, the experiments were repeated 10 times to ensure robustness of the results and to allow comparison of the variations between runs.

5.1 Synthetic Benchmarks

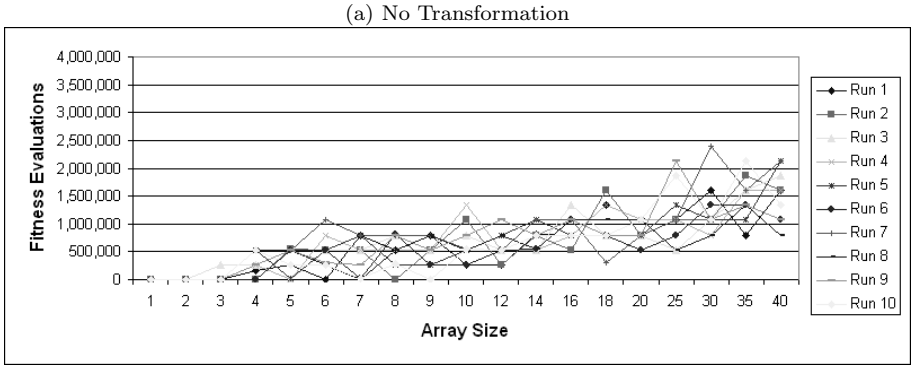
The analysis of the synthetic benchmark begins by applying the Daimler Evolutionary Testing system to the program without any transformation, after the coarse–grained transformation (when Case 5.3 is ignored), and finally after the fine–grained transformation. Figure 6 shows the results with the 'no transformation' case shown at the top. To facilitate comparison, the three graphs use the same y–axis scale. Figure 7 show a zoomed–in version of the graph for the fine–grained transformation.

The 'no transformation' case is largely uninteresting as it fails to find any test data to cover the branch in all but two situations. The first of these is where the array has size one. In this instance there is a 1 in 256 chance of randomly finding the 'special value' in each of the ten runs. At array size two, the chances of hitting the right value at random have diminished dramatically to 1 in $65,536$; only one of the ten runs manages to find this needle. For all other runs, no solution is found. In all cases, without transformation, the evolutionary search degenerates to a random search. Such a random search has a minuscule chance of covering the target branch.

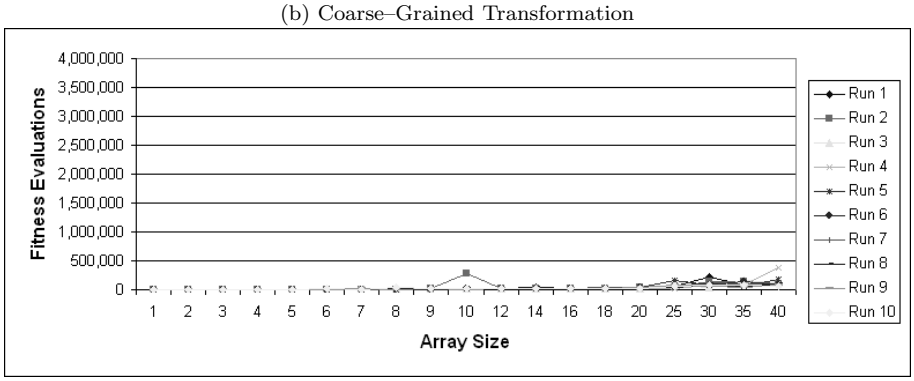
The coarse–grained technique achieves some success, but its trend for larger array sizes is clearly rising and more importantly there is an accompanying increase in the variation of the fitness evaluations. This increase in variability is a tell–tale sign of increasing randomness in the nature of the search using the coarse–grained approach. That is, where the landscape provides guidance, the evolutionary algorithm can exploit it, but when it does not, the search becomes a locally random



(a) No Transformation



(b) Coarse–Grained Transformation



(c) Fine–Grained Transformation

Fig. 6. Results over ten runs of the evolutionary search for each of the three approaches part 1.

search until a way of moving off a local plateau is found. As is visually apparent, the fine–grained technique outperforms the coarse–grained technique.

Also visually apparent in Figure 7 is the spike at array size ten in Run 2. This outlier was investigated and can be explained as follows: The search has almost found a solution with a similar number of fitness evaluations as the other nine runs. That is, it solves the nine–element array size problem, but the tenth array element does not reduce to zero for many additional generations. For instance, in the 40^{th} generation it has the value 6, but in the 1000^{th} generation this has only reduced

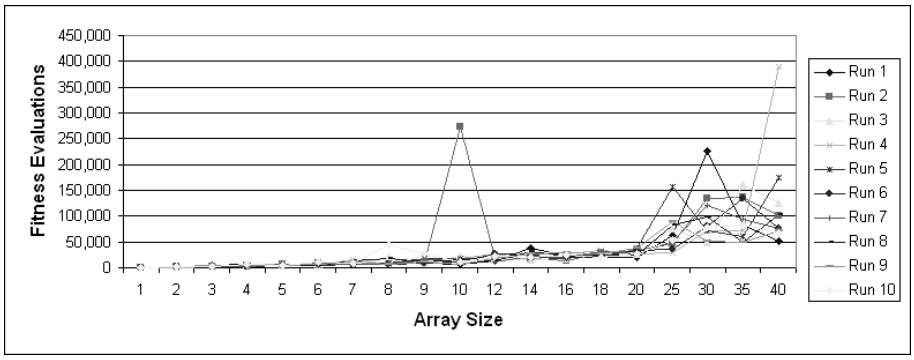
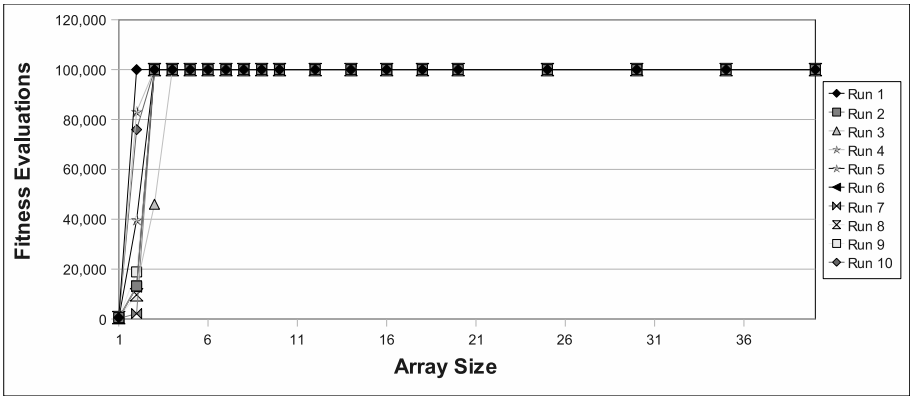
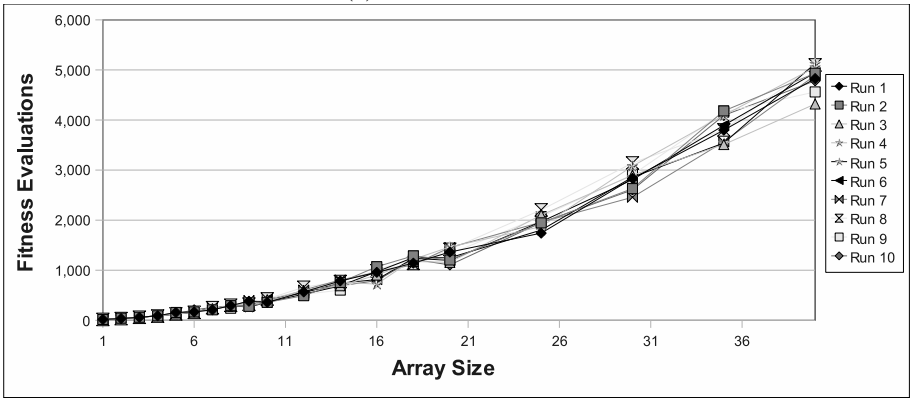


Fig. 7. Results over ten runs of the evolutionary search for the fine–grained transformation approach close–up.



(a) No Transformation



(b) Fine–Grained Transformation

Fig. 8. Results over ten runs of the alternating variable method for the 'no transformation' and fine–grained transformation approach.

to 2. There is a similar spike at array size 40 in Run 4 (again in the top graph). Upon investigation, a similar behaviour was observed. The search finds a solution for the array size 39 problem, but the search progresses very slowly with the final array element. In both cases this behaviour arises from the role chance plays in the underlying evolutionary search algorithm, rather than any properties of the flag problem *per se*.

Next, AUSTIN was applied to the synthetic benchmark yielding similar results. The results using no transformation and the fine–grained transformations are depicted in Figure 8. As with the Daimler Evolutionary Testing system, AUSTIN fails to find any test data to cover the branch for array sizes greater than three when working with the untransformed program. In one run the search manages to cover the branch at an array size of three, in nine runs with an array size of two, and in all runs when the array contains only one element. For AUSTIN to cover the branch using the untransformed program, the search needs to either randomly find the solution, or randomly choose a starting point which is exactly one neighbourhood move away from the solution (*e.g.*, a = {-1,0,0}, {1,0,0}, etc.).

For the fine–grained transformed program, AUSTIN requires far fewer fitness evaluations than the Daimler Evolutionary Testing system to cover the target branch. This is visually evident when comparing the y–axis scales of the chart in Figure 7, which shows the Daimler Evolutionary Testing system results, with that of the chart in Figure 8(b), which shows the AUSTIN results. This difference is consistent with the findings of Harman and McMinn who showed that a relatively simple optimization algorithm, such as the AVM, can often outperform a more complex search strategy [Harman and McMinn 2007].

The final two charts, shown in Figures 9 and 10, present the averages and standard deviations respectively over all ten runs for each of the three approaches using the Daimler Evolutionary Testing system. The pattern for AUSTIN is similar. The average for the 'no transformation' technique is almost uniformly the worst–case, while its standard deviation is zero, in all but the cases for array size 1 and 2 (where some random chances led to a successful search). The high standard deviation for size 2 is evidence that the one solution was a random occurrence.

The qualitative assessment presented in Figure 6 and Figure 8 clearly suggests that the fine–grained approach is better than the coarse–grained approach, which in turn, is better than the 'no transformation' approach. The 'trend' of the fine–grained transformation outperforming the 'no transformation' approach manifests itself as the size of the problem increases. It is also the case that as the difficulty of the problem increases (*i.e.*, as the size of the array increases), the test data generation process will get harder for both, the fine–grained and 'no transformation' approaches. This can be seen in Figure 8(b) and Figure 7. Despite this, the fine–grained approach will always be able to offer the search guidance, whereas the 'no transformation' approach remains a random search for the 'needle' in an increasingly larger 'haystack'.

To complement this qualitative assessment quantitatively, an assessment using the Mann–Whitney test, a non–parametric test for statistical significance in the differences between two data sets, is performed. Because the test is non–parametric, the data is not required to be normally distributed for the test to be applicable. The

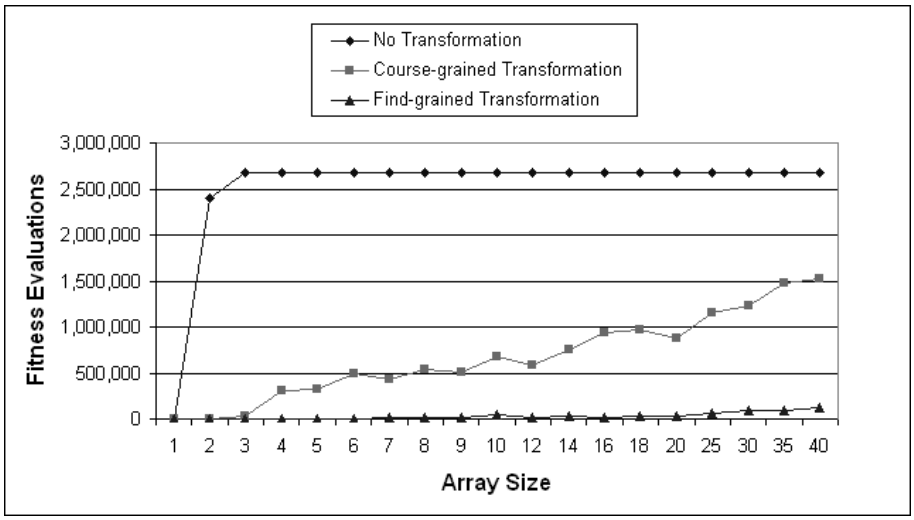


Fig. 9. Averages over ten runs of the evolutionary search for each of the three approaches

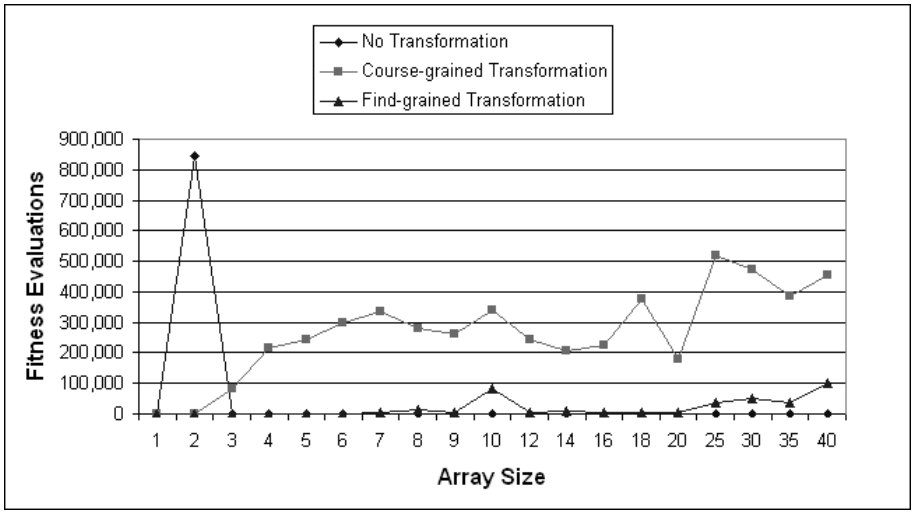


Fig. 10. Standard deviation over ten runs of the evolutionary search for each of the three approaches

test reports, among other things, a p–value. The p–value for the test that compares the 'no transformation' results with the 'coarse–grained transformation' results and that which compares the 'coarse–grained transformation' results with the 'fine–grained transformation' results, all return p–values less than 0.0001 indicating that the differences are 'statistically significant at the 99% level'.

The results obtained from the synthetic benchmarks clearly show that the test data generation process on the transformed (fine–grained) version outperforms the test data generation process on the untransformed version.

Test subject	Function under test
EPWIC	run_length_encode_zeros
bibclean	check_ISBN check_ISSN
ijpeg	emit_dqt format_message next_marker pbm_writepbmrowraw write_frame_header
time	getargs
plot2d	CPLOT_DetermineSeriesStatistics
tmnc	rule_l_intercept
handle_new_jobs	handle_new_jobs
netflow	netflow
moveBiggestInFront	moveBiggestInFront
update_shps	update_shps

Table II. Test subjects

5.2 Open Source and Daimler Programs

In addition to the study of the synthetic program, which represents a problem that is synthetically generated to be hard for search–based approaches to solve, the seventeen C functions shown in Table II were used to evaluate the impact of the testability transformation of Figure 4. These functions were extracted from a combination of ten open–source and Daimler programs. Each of the seventeen is first described in some detail to provide an indication of flag usage in each program. This is followed by a discussion of the empirical results obtained using these functions.

EPWIC is an image compression utility. The selected function run_length_encode_zeros loops through a stream of data (encoded as a C string) and counts the number of consecutive occurrences of 0. If a zero is found, a counter is incremented and the flag found_zero is set. The flag is initialized to 1 and the test problem is to find inputs which avoid setting the flag to 0.

bibclean is a program used to check the syntax of BibTeX files and pretty print them. Two functions were selected. The first, check_ISBN, loops through an input string to check whether it represents a valid ISBN number. It contains the flag new_ISBN, which is initialized to 1 and set to 0 at the start of every loop iteration whenever its value is 1. The flag is reset to 1 in the body of the loop only when a given number of parsed characters in the range $0-9$, including 'x' and 'X', represent an invalid ISBN number, or, the string does not represent a valid ISBN number at all, but contained more than 10 valid ISBN characters. The test data generation challenge is to discover a string with more than 10 valid ISBN characters which do not represent a valid ISBN number. The search has to navigate through the entire input domain of the function, which is approximately 10^{120}. The second function, check_ISSN works exactly as check_ISBN except that ISSN instead of ISBN numbers are checked for.

ijpeg implements an image compression and decompression algorithm, of which five functions were tested. The first, emit_dqt, contains the flag prec. An array of 64

unsigned integers is iterated over. If an element exceeds the value 255, the flag is set to true. The target branch is dependent on the flag being set to true.

The second function, format_message, formats a message string for the most recent JPEG error or message. It contains the flag isstring, which is used to check if the format string contains the '%s' format parameter. The test data generation problem is to find an input string which contains the character sequence '%s'.

The third function, next_marker, contains a loop assigned flag c, which is part of the termination criterion for a do {} while() loop. An input buffer is traversed and the current character assigned to c. The loop terminates if c is not equal to 255, thus the challenge is to discover that an array of inputs is required that contains the value 255 at least once.

The fourth function, pbm_writepbmrowraw, contains the local variable bitshift of type int which is initialized to 7. The inputs to the function are a file pointer, a pointer to the start of a row in a matrix of unsigned character types, and the number of columns in the row. A loop goes through each column and checks its entry for a non–zero value. Whenever a non–zero character is encountered, the value of bitshift is decremented by one. When bitshift takes on the value -1 it is reset to 7. After the body of the loop, the function checks for bitshift not being equal to 7. In this case the hard to cover target branch is the false outcome of this check.

The final function, write_frame_header, contains the loop assigned flag is_baseline, which is initialized to 1 and assigned 0 in the body of the loop. The target branch depends on the flag retaining its initial value. To complicate matters, the flag may be initialized to 0 before the start of the loop if two properties of the function's input domain are true. This assignment is not part of the transformation *per se* (apart from ensuring that the state of the flag is correctly represented by the helper variables regardless of the path taken to reach the start of loop), thus it remains an additional goal of the search algorithm to find inputs which avoid initializing the flag to 0.

time is a GNU command line utility which takes as input another process (a program) with its corresponding arguments and returns information about the resources used by the process (*e.g.*, the wall–clock and CPU time used). The function getargs contains three loop assigned flags, outfile, append and verbose. The function parses the command line arguments and sets the option flags accordingly. The challenge during the test data generation process is to find input parameters encoded as strings, that are valid options, setting these flags.

plot2d is a small program that produces scatter plots directly to a compressed image file. The core of the program is written in ANSI C. However the entire application includes C++ code. Only the C part of the program was considered during testing. The function CPLOT_DetermineSeriesStatistics contains the loop assigned flag computeStats, which is initialized with 1 and only ever assigned 1 in the body of the loop. The branch dependent on the false outcome of the flag is therefore infeasible and the true branch trivially covered.

tmnc is a C implementation of the TMN protocol. The function rule_l_intercept

loops through an array of sessions (containing, *inter alias*, information about the initiator and responder), validating a session object. If the session is valid, a flag is set.

handle_new_jobs is a job scheduler responsible for management of a set of jobs stored in an array. Each job has a status and priority as well as additional data used during job execution. This code is part of the Daimler C++ testing system itself: it facilitates parallel execution of test processes. The input space is the job array (the 'data' entries are unimportant for coverage). The test problem is to find the right input data for the flag, check_work, tested in the last condition of the function. In order to execute the true branch of this conditional, the assignment check_work=1; in the for loop must be avoided in every iteration.

netflow is part of an ACM algorithm for performing net flow optimization. The function has many input parameters configuring the net to be optimized, for example connected nodes and connection capacity. The two parameters of the function are low and high. The netflow function begins with some plausibility checks on the input parameters. The flag variable violation is typical of a test for special conditions which cannot be handled by the regular algorithm. As an invalid input check, violation is set to true when low is set to a larger value than high.

moveBiggestInFront is part of a standard sorting algorithm. A while loop processes the elements of an array, checking whether the first element is the biggest. If no such value exists, this constitutes a special case with the result that the flag assignment is not executed in any iteration.

update_shps is a navigation system used by Daimler in vehicular control systems. The code has been modified to protect commercially sensitive information. However, these modifications do not affect the properties of the code with respect to flag variable use. The navigation system operates on a 'Shape Point Buffer' which stores information from a digital street map. Streets are defined by shape points. The buffer contains map locations (points) near to the current location of the car. For testing, the input space is formed from the set of shape point buffer data stored in a global array and the position of the car supplied as the parameters of the function. The function uses a flag, update_points, to identify a situation where an update is required. The flag is assigned inside a loop traversing the shape point buffer. The flag becomes true if any shape point from the buffer is outside a certain area. The target branch is hard to execute because input situations rarely lead to update_points being assigned false. The search space for the predicate if (!update_points) is precisely the worst case flag landscape described in Figure 1.

These seventeen functions capture the full range of difficulties for the search. At the easy end of the spectrum, test data for the flag use in the predicate from plot2d was always found in a single evaluation, both before and after transformation. Code inspection revealed that every path through the loop (and in fact the function) assigned true to the flag. Prior to the body of the loop, the flag is initialized to true. After the loop, the function contains a check for the value true of flag. Since the false branch of this check is clearly infeasible, it is not clear if this code was written anticipating some structural addition; perhaps it is a bug.

At the other end of the spectrum, the evaluation budget was exhausted in both the transformed and untransformed versions of three test subjects: getargs_append, getargs_verbose, and next_marker. The first of the three, getargs_append, comes from the command line argument processing of the program time. This example uncovered a limitation in the current AUSTIN and Daimler tool implementations. The tools do not properly handle C static variables, which are used to hold values across multiple calls to a function. In this case, time uses the function getopt, which records an index and look–ahead character in the static variables optind and nextchar. These two static variables effectively prevent the search from returning to a previous solution after exploring an inferior neighbour.

The second function getargs_verbose, also from time, and the third next_marker from ijpeg both contain unstructured control flow. In this case an exit within the loop; however, the impact of such control statements (*e.g.*, a return statement) would be similar. In essence, such statements prevent the search from exploiting accumulated fitness information. In both cases, the search does not execute the transformed predicate (created by Step 6 of the algorithm shown in Figure 4).

Observe that none of the aforementioned issues denote flag problems. Rather the application of search–based testing techniques to real world programs has thrown up subsidiary issue and barriers to test data generation that have nothing to do with flags. It should be recognized that no 'perfect' solution to the test data generation problem yet exists; all techniques have language features that present difficulties. The purpose of this paper is to demonstrate that the barrier to search–based testing raised by the presence of flag variables can be lowered by the testability transformation approach advocated in the paper. However, there will remain further work required on these other issues of search–based testing.

The remaining thirteen functions fall in between these two extremes. In each case, the transformation improves the search for test data. The average success over all ten runs of each test subject is reported in the top of Figure 11. This table and the one below it are sorted based on the number of fitness evaluations performed using the untransformed program. Overall the transformation led to a 28% increase in successful test-date generation.

The lower table in Figure 11 shows the number of fitness evaluations used by all ten runs for each program. This data is shown graphically in Figure 12, which uses a log scale on the y–axis. Overall, the transformation leads to a 45% improvement, reducing the total number of evaluations needed from $4,589$ to $2,522$, which represents a statistically significant reduction (students t-test p-value $= 0.031$). It produces an improvement for all but four of the test subjects. In several cases the improvement is dramatic. For example, with the function moveBiggestInFront the total number of evaluations drops from $9,011$ to 10. Even in cases where the untransformed programs never exhausts its evaluation budget, there are large improvements. For example, with the function format_message there is a 91% decrease from $2,811$ to 245.

These results have to be treated with statistical caution. We take the average results of 10 runs for each flag use of each predicate in each function studied. These average values form a sample from two paired populations: the 'with treatment' population and the 'without treatment population' for flag uses in predicates. In

Program	Function	Covered Branches untrans -formed	trans -formed
time	getargs_append	0	0
time	getargs_verbose	0	0
ijpeg	next_marker	0	0
ijpeg	write_frame_header	0	10
netflow	netflow	0	10
moveBiggestInFront	moveBiggestInFront	3	10
tmnc	rule_l_intercept	2	4
EPWIC	run_length_encode_zeros	10	10
ijpeg	format_message	10	10
bibclean	check_ISBN	10	10
bibclean	check_ISSN	10	10
ijpeg	emit_dqt	10	10
time	getargs_outfile	10	10
update_shps	update_shps	10	10
handle_new_jobs	handle_new_jobs	10	10
ijpeg	pbm_writepbmrowraw	10	10
plot2d	CPLOT_DetermineSeriesStatistics	10	10
	average	6.2	7.9
	percent improvement		28%

Program	Function	Fitness Evaluations untrans -formed	trans -formed	savings	percent reduction
time	getargs_append	10,000	10,000	0	0%
time	getargs_verbose	10,000	10,000	0	0%
ijpeg	next_marker	10,000	10,000	0	0%
ijpeg	write_frame_header	10,000	412	9588	96%
netflow	netflow	10,000	61	9939	99%
moveBiggestInFront	moveBiggestInFront	9,011	10	9001	100%
tmnc	rule_l_intercept	8,767	8,451	316	4%
EPWIC	run_length_encode_zeros	3,401	2,066	1335	39%
ijpeg	format_message	2,811	245	2566	91%
bibclean	check_ISBN	1,385	543	842	61%
bibclean	check_ISSN	843	664	179	21%
ijpeg	emit_dqt	835	145	690	83%
time	getargs_outfile	478	218	260	54%
update_shps	update_shps	271	45	226	83%
handle_new_jobs	handle_new_jobs	202	6	196	97%
ijpeg	pbm_writepbmrowraw	7	5	2	29%
plot2d	CPLOT	1	1	0	0%
	average	4,589	2,522		
	percent improvement		45%		

Fig. 11. Results from empirical study of functions extracted from open source software

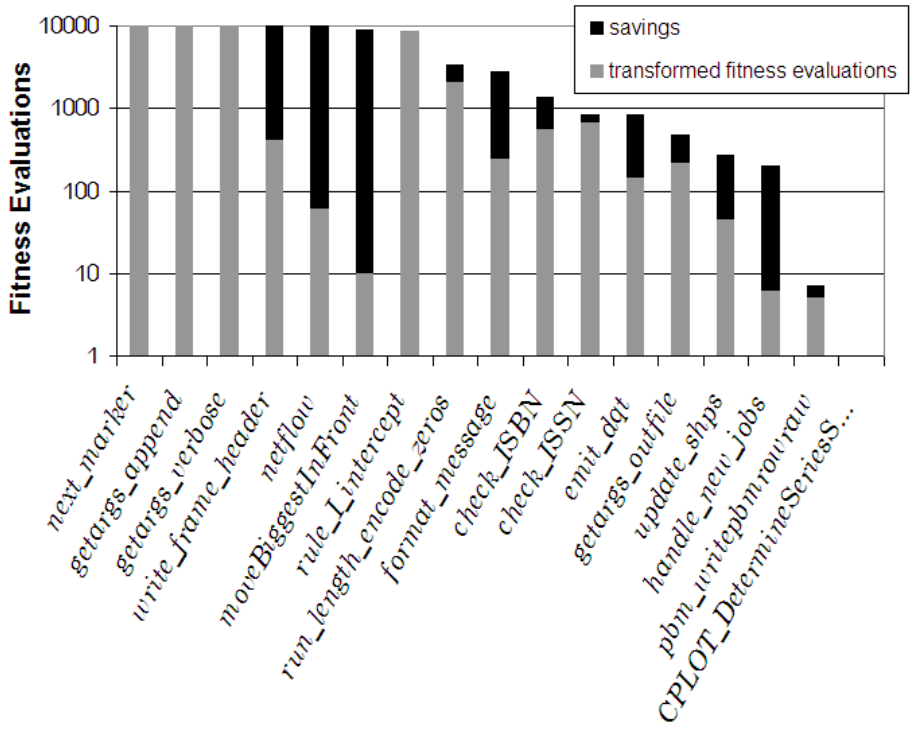


Fig. 12. Chart of data from second empirical study

this case, 'with treatment' means with some form of transformation aimed to improve the test data generation process.

The samples involved in this test are not (and, indeed, cannot be) sampled in an entirely random and unbiased manner. They are samples from the space of all possible loop–assigned flag uses in all predicates in all functions in all C programs. There is even an issue here as to what constitutes the 'population'; should it be, for example, all C programs possible; all those currently written, or all those in use in production systems? These questions bedevil any attempt to make reliable claims for statistical significance of results for studies involving samples of program code. In using the statistical tests, we are merely seeking to give a rough indication of the strength of the results for the programs studied, rather than to make claims about the behaviour of untried predicates in programs as yet unconsidered in the study.

6. EMPIRICAL VALIDATION

This section investigates the existence of loop–assigned variables 'in the large'; that is it considers if flags occur widely in practice. This is important to validate the existence of the loop–assigned flag problem: if flags are not prevalent then there would be little point in trying to remove them.

The investigation makes use of the dependence graphs output by Codesurfer,

Program	Total Predicates	With flags	Total flags	Program	Total Predicates	With flags	Total flags
EPWIC-1	435	104	107	gnugo	2584	687	731
a2ps	1886	677	754	go	2982	489	570
acct	349	168	179	ijpeg	1042	189	199
barcode	235	83	98	indent-1.10.0	537	140	172
bc	298	99	105	li	364	94	99
byacc	620	298	309	named	5794	1820	2080
cadp	480	174	175	ntpd	1904	552	595
compress	59	10	12	oracolo2	541	379	382
cook-c_incl	706	218	228	prepro	530	374	377
cook-cook	2203	732	768	replace	54	13	13
cook-cook_bom	564	194	202	sendmail-8.7.5	3198	1026	1161
cook-cookfp	510	176	183	snns-batchman	4858	1811	2201
cook-cooktime	518	185	200	snns-convert2snns	4394	1496	1829
cook-file_check	462	169	176	snns-ff_bignet	4450	1527	1876
cook-find_libs	458	163	169	snns-isnns	4398	1499	1832
cook-fstrcmp	467	166	173	snns-linknets	4541	1527	1865
cook-make2cook	683	254	271	snns-netlearn	4386	1495	1828
cook-roffpp	505	179	185	snns-netperf	4387	1495	1828
copia	15	7	16	snns-pat_sel	4396	1494	1827
csurf-packages	2043	924	979	snns-snns2c	4600	1541	1874
ctags	1063	336	387	snns-snnsbat	4669	1636	1995
cvs	6053	2617	2817	snns-td_bignet	4451	1531	1879
diffutils	909	365	442	snns-ui_main	6433	2118	2512
ed	619	228	302	space	503	345	348
empire	5294	1240	1406	spice	7254	2368	2478
espresso	1043	290	305	termutils	218	99	107
findutils	686	257	309	tile-forth-2.1	193	50	54
flex2-4-7	584	317	364	time-1.7	40	17	17
flex2-5-4	706	378	432	userv-0.95.0	679	222	241
ftpd	1765	723	823	wdiff.0.5	164	88	125
gcc.cpp	650	187	203	which	77	30	35
gnubg-0.0	497	113	119	wpst	989	482	516
gnuchess	1001	212	277	**total**	119,976	40,877	47,121

Fig. 13. Predicates and flags in the 65 programs

a deep structure analysis tool [Grammatech Inc. 2002]. Traversing dependence edges in a program's dependence graph simplifies the discovery of loop–assigned flag variables. In all, 65 programs (including both Daimler and open source programs), with a total of 119,976 predicates were analyzed. The programs studied range from utilities with as few as fifteen predicates to the electronic circuit simulation program spice containing 7,254 predicates.

Results for the programs are shown in Figures 13, 14 and 15. The data presented in these figures may be useful to other researchers addressing the flag problem. It can be used to assess the kinds of flags used and their prevalence in a reasonably large corpus of code. In this way the data provides base line information about how prevalent the problem is and the various precise forms taken by the flag problem. Other researchers can also use these data to determine whether their code base is similar to that used in this study. This may be valuable to determine the degree to which subsequent work replicates the findings reported here.

In Figure 13, the data reported for each program consists of the total number

Program	Total Flags	Loop Flags	Simple			Single							
			gf	pf	bf	la	ca	oa	cas	ma	pa	cnst	?f
EPWIC-1	107	21	11	18	0	32	1	0	0	20	4	0	0
a2ps	754	69	85	155	4	284	9	5	19	121	0	3	1
acct	179	19	82	13	2	34	2	22	1	4	0	0	0
barcode	98	16	3	7	0	49	0	0	2	10	11	0	0
bc	105	21	10	38	0	23	0	0	0	11	2	0	0
byacc	309	18	96	5	0	158	0	0	1	16	15	0	0
cadp	175	13	14	106	0	31	0	0	0	10	1	0	0
compress	12	0	6	0	0	5	0	0	0	1	0	0	0
cook-c_incl	228	56	36	32	0	65	3	5	4	27	0	0	0
cook-cook	768	81	122	90	0	340	23	5	7	82	18	0	0
cook-cook_bom	202	47	35	25	0	60	4	5	4	22	0	0	0
cook-cookfp	183	45	29	22	0	52	3	5	4	23	0	0	0
cook-cooktime	200	51	28	24	0	52	3	5	4	27	6	0	0
cook-file_check	176	40	28	19	0	51	3	5	4	25	1	0	0
cook-find_libs	169	40	28	19	0	49	3	5	4	21	0	0	0
cook-fstrcmp	173	44	28	19	0	49	3	5	4	21	0	0	0
cook-make2cook	271	51	54	33	0	80	3	5	4	35	6	0	0
cook-roffpp	185	47	30	22	0	52	3	5	4	22	0	0	0
copia	16	0	10	0	0	6	0	0	0	0	0	0	0
csurf-packages	979	36	75	515	2	269	0	0	1	48	22	11	0
ctags	387	76	27	34	2	185	6	1	4	50	0	2	0
cvs	2817	202	531	470	1	834	39	111	16	372	239	2	3
diffutils	442	49	69	89	1	116	3	4	8	70	33	0	1
ed	302	41	72	58	1	68	0	4	3	48	7	0	3
empire	1406	216	273	203	3	400	43	4	2	229	33	0	0
espresso	305	45	116	27	0	69	0	3	13	30	2	0	2
findutils	309	27	21	95	1	97	2	4	7	55	0	0	0
flex2-4-7	364	24	206	28	0	42	0	9	0	45	10	0	0
flex2-5-4	432	7	242	41	0	59	0	14	0	38	31	0	1
ftpd	823	108	356	52	1	132	83	15	0	53	23	0	0
gcc.cpp	203	51	52	29	0	40	0	8	0	17	6	0	0
gnubg-0.0	119	15	51	25	0	16	1	1	1	6	2	1	3
gnuchess	277	31	37	78	0	82	1	2	7	25	6	8	2
gnugo	731	52	158	239	1	167	2	12	2	68	19	11	0
go	570	98	34	66	0	238	0	0	5	118	10	1	0
ijpeg	199	17	16	55	0	86	0	0	1	21	3	0	1
indent-1.10.0	172	40	52	5	0	19	0	6	0	40	10	0	0
li	99	19	4	31	0	22	0	0	1	8	10	4	2
named	2080	271	91	495	8	733	44	15	14	293	116	0	6

Fig. 14. Part 1 of 2: Loop–assigned flags from 65 programs. Column headings are defined in Part 2 of the table.

of predicates in the program, the number of predicates that include one or more flags, and the total number of flags. Comparing the last two columns, most flag–containing predicates include a single flag. Figures 14 and 15 show a break down of the flags into twelve categories. Of particular interest is the third column, which shows loop–assigned flags. In the 65 programs, a total of 47,121 flags were found in all predicates, of these 14% are loop–assigned. Thus, the problem studied herein is relevant, as a significant proportion of the flags used were found to be loop–assigned.

Program	Total Flags	Loop Flags	Simple			Single							
			gf	pf	bf	la	ca	oa	cas	ma	pa	cnst	?f
ntpd	595	70	203	62	1	142	10	13	0	66	28	0	0
oracolo2	382	16	1	46	0	281	0	0	0	38	0	0	0
prepro	377	16	1	46	0	277	0	0	0	37	0	0	0
replace	13	7	0	0	0	6	0	0	0	0	0	0	0
sendmail-8.7.5	1161	161	142	167	8	362	32	19	37	172	59	2	0
snns-batchman	2201	312	534	167	0	649	21	10	20	241	231	16	1
snns-convert2snns	1829	300	442	133	0	605	22	9	0	216	86	16	0
snns-ff_bignet	1876	301	456	137	0	626	21	9	0	224	86	16	0
snns-isnns	1832	302	441	132	0	606	21	9	0	219	86	16	0
snns-linknets	1865	318	442	132	0	621	21	10	0	218	86	17	0
snns-netlearn	1828	300	441	132	0	607	21	9	0	216	86	16	0
snns-netperf	1828	300	441	132	0	605	21	9	0	216	88	16	0
snns-pat_sel	1827	300	441	132	0	605	21	9	0	217	86	16	0
snns-snns2c	1874	308	444	132	0	638	21	12	0	216	86	17	0
snns-snnsbat	1995	302	471	140	0	619	21	47	0	268	110	17	0
snns-td_bignet	1879	302	456	137	0	626	21	9	0	226	86	16	0
snns-ui_main	2512	334	745	213	0	755	26	10	4	299	110	16	0
space	348	16	1	7	0	283	0	0	0	41	0	0	0
spice	2478	513	161	390	0	881	3	17	54	382	76	1	6
termutils	107	23	8	23	0	39	0	0	0	11	3	0	2
tile-forth-2.1	54	6	9	10	0	23	0	2	0	4	0	0	0
time-1.7	17	3	4	5	0	4	0	0	0	1	0	0	0
userv-0.95.0	241	12	30	12	0	147	21	2	0	11	6	0	0
wdiff.0.5	125	8	71	8	0	15	0	0	0	17	6	0	0
which	35	0	11	3	0	8	0	6	1	6	0	0	0
wpst	516	29	36	240	0	152	0	0	1	30	16	12	0
Percent	100%	14%	20%	13%	<1%	33%	1%	1%	<1%	12%	4%	<1%	<1%

Fig. 15. Part 2 of 2: Loop–assigned flags from 65 programs. Column headings are as follows
gf - global flags not assigned to in the module,
pf - parameter flags not assigned to in the module,
bf - combination of unassigned global and parameter used (*e.g.*, if(f==g))
la - single assignment to a local variable
ca - single assignment from the result of a call
oa - other single assignments
cas - parameters assignment from within conditional
ma - multiple reaching assignments to the flag exist
pa - assignments through a pointer
cnst - flags with constant value while(true)
?f - unknown flags

7. RELATED WORK

Many internationally accepted testing standards [British Standards Institute 1998a; Radio Technical Commission for Aeronautics 1992] either recommend or require branch adequate testing for quality assurance and safety. These standards apply in particular to embedded system controllers, where flag use is common. Generating test data by hand in order to meet these standards is tedious and error–prone even though most embedded code is relatively simple in structure; thus, automated test data generation has been a sustained topic of interest for the past three decades.

Past techniques used for generating test data automatically include symbolic

execution [Clarke 1976; King 1976], dynamic symbolic execution [Godefroid et al. 2005; Cadar and Engler 2005; Sen et al. 2005; Tillmann and de Halleux 2008; Burnim and Sen 2008; Xie et al. 2009; Cadar et al. 2008], constraint solving [DeMillo and Offutt 1993; Offutt 1990; Offutt et al. 1999], the chaining method [Ferguson and Korel 1996] and evolutionary testing [Schultz et al. 1993; Jones et al. 1996; Michael et al. 2001; Mueller and Wegener 1998; Pargas et al. 1999; Pohlheim and Wegener 1999; Tracey et al. 1998b; Godefroid and Khurshid 2002].

The loop–assigned flag problem discussed in this paper is relevant for all these methods. Symbolic execution formulates predicates as mathematical expressions with the aid of symbolic variables, which are then passed to an automated reasoner when used as part of a test data generation process. Loops force conservative approximation about loop–assigned variables, because they may be non–deterministic. Constraint solving techniques, which back propagate path information from predicates to form the constraints on the inputs suffer from a similar problem. Dynamic symbolic execution aims to overcome many of the problems typically associated with symbolic execution. To date, most variations of dynamic symbolic execution are based on exploring all feasible execution paths; they do not have the capability of targeting specific branches. Hence, branches controlled by loop–assigned flags may not be covered if the dynamic symbolic execution gets 'stuck' in unbounded loops, or, they may take a very long time to be covered. For example, CUTE [Sen et al. 2005] fails to find test data within 1000 iterations for the synthetic example shown in Figure 3(a) with an array size of 10 or greater [3]. This is because the number of feasible paths increases to more than 1024. By comparison, the AVM takes, on average, 386.6 fitness evaluations (evaluations correspond to iterations in CUTE), to cover the target branch for an array size of 10.

Finally, the chaining approach also suffers from the loop–assigned flag problem. It tries to identify sequences of nodes in a programs control flow graph which need to be executed in order to reach a specified target branch. Loop–assigned variables may lead to 'infinite' chains or result in loss of information, because it is not known a–priori how often a node inside a loop needs to be executed.

For a subset of a C–like language, Offutt et al. introduce a dynamic variant of constraint solving that performs dynamic domain reduction [Offutt et al. 1999]. The algorithm can be applied to the flag problem. First a path to the target is selected for execution. Then the domains of each input variable are refined as execution follows this path. Domain refinement takes place at assignments and decision statements. For example, if the domain of a and b were both $1 \cdots 10$ before if (a != b), then in the true branch of the if statement, a and b would be assigned the domains $1 \cdots 5$ and $6 \cdots 10$ respectively.

Loops are handled in a similar fashion by marking the loop predicates and dynamically reducing the input domain of the variables involved in loop constraints. However, the domain reduction requires knowing a–priori a path to the target. Thus, for this dynamic domain reduction technique to cover the flag controlled branch in the program shown in Figure 3(a) requires that it first select the path through the body of the loop which avoids each assignment to flag. Because this

[3]CUTE was run with the -r option in order to initialize (primitive) inputs with random numbers.

section is done essentially by chance and the set of feasible paths is large, this may take some time.

Flags often present the worst case scenario to search based test data generation techniques. In particular when only very few sub–paths will result in a flag taking on one of its two values. In contrast, the approach presented in this paper is able to offer the search coarse and fine–grained guidance. This makes the approach more applicable to the flag problem in the presence of loop–assigned flags.

SBT in the presence of flags has been studied by four different authors [Bottaci 2002a; Baresel and Sthamer 2003; Harman et al. 2004; Liu et al. 2005]. Bottaci [Bottaci 2002a] aims to establish a link between a flag use and the expression assigning a flag. This is done by storing the fitness whenever a flag assignment occurs so it can be used later on.

Baresel and Sthamer [Baresel and Sthamer 2003] use a similar approach to Bottaci. Whereas Bottaci's approach is to store the values of fitness as the flag is assigned, Baresel and Sthamer use static data analysis to locate the assignments in the code, which have an influence on the flag condition at the point of use. Baresel and Sthamer report that the approach also works for enumeration types and give results from real–world examples, which show that the approach reduces test effort and increases test effectiveness.

Harman et al. [Harman et al. 2004] illustrate how a testability transformation originating from an amorphous slicing technique can be used to transform flag containing programs into flag–free equivalents. They achieve this by substituting a flag use with the condition leading to, as well as the definition of a flag, with the aid of temporary variables.

Liu et al. [Liu et al. 2005] present an approach for unifying fitness function calculations for non–loop assigned flags, and consider the problem of loop–assigned flags in the presence of break and continue statements [Liu et al. 2005].

Three of the approaches share a similar theme: they seek to connect the last assignment to the flag variable to the use of the flag at the point where it controls the branch of interest. In Bottaci's approach the connection is made through auxiliary instrumentation variables, in that of Baresel and Sthamer it is made through data flow analysis and, in the approach of Harman et al. , a literal connection is made by substitution in the source code.

The algorithm presented in Figure 4 and the approach by Liu et al. could be thought of as a combination of the approaches of Bottaci and Harman et al. They share the use of auxiliary 'instrumentation variables' with Bottaci's approach, but use these in a transformed version of the original program using transformations like the approach of Harman et al.

Alshraideh and Bottaci [Alshraideh and Bottaci 2006] proposed an approach to increase diversity of population–based test data generation approaches, for situations where the fitness function results in plateaux. This partly addresses issues relating to plateaux by increasing the chances that random mutations will move the population off the plateau. One of the underlying features of the flag problem is the way in which plateaux are present. In the case of hard–to–test flags, the landscape is formed of one very large plateau (with a tiny spike; the needle in the haystack) and, even with increased diversity, the search–based approach reduces to

random search.

A preliminary version of the fine–grained transformation advocated in the present paper, made use of a bushing and blossoming technique [Baresel et al. 2004]. Bushing takes a program which may contain if - then statements, and replaces these with if - then - else statements. It also copies in the rest of the code sequence from the block into the then and else branches of the conditional. Blossoming pushes the assignment statements within a bushed tree to the leaves of the tree. When combined, bushing and blossoming have the effect of converting the AST of the code sequence into a binary tree, in which the internal nodes are predicates and the leaves are a sequence of assignments. The advantage of bushing and blossoming is that the transformed AST contains one leaf for each path. This considerably simplifies the case–based analysis from Figure 4. However it also introduces new levels of nesting, and, as a study by McMinn et al. [McMinn et al. 2005] shows, this causes problems in an evolutionary search. Thus it was decided not to include bushing and blossoming in the implementation described in Section 4, and only transform if - then into if - then - else statements, minimizing the levels of nesting introduced by the transformations.

All of the evolutionary approaches can benefit from general improvements in genetic algorithms. For example, Godefroid and Khurshid consider a framework for exploring very large state spaces often seen in models for concurrent systems [Godefroid and Khurshid 2002]. They describe an experiment with an implementation based on the VeriSoft tool. They experiment with heuristics that improved the genetic algorithm's mutation operators and also show how Partial–order reduction can allow a greater search space to be considered.

Finally, from a transformation standpoint, the algorithm presented here is interesting as it does not preserve functional equivalence. This is a departure from most prior work on program transformation, but it is not the first instance of non–traditional–meaning preserving transformation in the literature. Previous examples include Weiser's slicing [Weiser 1979] and the 'evolution transforms' of Dershowitz and Manna [Dershowitz and Manna 1977] and Feather [Feather 1982]. However, both slices and evolution transforms do preserve some projection of traditional meaning. The testability transformation introduced here does not; rather, it preserves an entirely new form of meaning, derived from the need to improve test data generation rather than the need to improve the program itself.

8. SUMMARY

This paper presents a testability transformation that handles the flag problem for evolutionary testing. Unlike previous approaches, the transformation introduced here can handle flags assigned in loops. Also, unlike previous transformation approaches (either to the flag problem or to other more traditional applications of transformation) the transformations introduced are not meaning preserving in the traditional sense; rather than preserving functional equivalence, all that is required is to preserve the adequacy of test data.

An implementation of the algorithm is discussed. The implementation is based on CIL, an infrastructure for C program analysis and transformations. The modular nature of both CIL and the tool allows the system to be extended in the future and

incorporate different transformation algorithms, thus forming an effective transformation tool, geared toward improving evolutionary testing.

The behaviour of the algorithm is evaluated with two empirical studies that involve a synthetic program and functions taken from open source programs. The synthetic examples are used to illustrate how two variations of the algorithm perform for different levels of difficulty. The results show that the approach scales well to even very difficult search landscapes, for which test data is notoriously hard to find. The worst case considered involves finding a single adequate test input from a search space of size 2^{320}. Despite the size and difficulty of this search problem, the search–based testing approach, augmented with the transformation algorithm introduced here, consistently finds this value.

The paper also presents evidence that the flag problem considered herein arises naturally in a variety of real world systems. It uses results from a separate empirical study to show that the flag problem considered is prevalent among those uses of flags found in a suite of real world programs.

REFERENCES

Alshraideh, M. and Bottaci, L. 2006. Using program data-state diversity in test data search. In *Proceedings of the 1st Testing: Academic & Industrial Conference - Practice and Research Techniques (TAICPART '06)*. 107–114.

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 ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, G. S. Avrunin and G. Rothermel, Eds. ACM, 108–118.

Baresel, A. and Sthamer, H. 2003. Evolutionary testing of flag conditions. In *Genetic and Evolutionary Computation (GECCO-2003)*. LNCS, vol. 2724. Springer-Verlag, Chicago, 2442–2454.

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

Binkley, D. W. and Gallagher, K. B. 1996. Program slicing. In *Advances in Computing, Volume 43*, M. Zelkowitz, Ed. Academic Press, 1–50.

Bottaci, L. 2002a. Instrumenting programs with flag variables for test data search by genetic algorithms. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers, New York, 1337–1342.

Bottaci, L. 2002b. Instrumenting programs with flag variables for test data search by genetic algorithms. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, Eds. Morgan Kaufmann Publishers, New York, 1337–1342.

Briand, L. C., Labiche, Y., and Shousha, M. 2005. Stress testing real-time systems with genetic algorithms. In *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, H.-G. Beyer and U.-M. O'Reilly, Eds. ACM, 1021–1028.

British Standards Institute. 1998a. BS 7925-1 vocabulary of terms in software testing.

British Standards Institute. 1998b. BS 7925-2 software component testing.

Burnim, J. and Sen, K. 2008. Heuristics for scalable dynamic test generation. Tech. Rep. UCB/EECS-2008-123, EECS Department, University of California, Berkeley. Sep.

Cadar, C., Dunbar, D., and Engler, D. R. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, R. Draves and R. van Renesse, Eds. USENIX Association, 209–224.

Cadar, C. and Engler, D. R. 2005. Execution generated test cases: How to make systems code crash itself. In *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings*. Lecture Notes in Computer Science, vol. 3639. Springer, 2–23.

Clark, J., Dolado, J. J., Harman, M., Hierons, R. M., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., and Shepperd, M. 2003. Reformulating software engineering as a search problem. *IEE Proceedings — Software 150,* 3, 161–175.

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

Darlington, J. and Burstall, R. M. 1977. A tranformation system for developing recursive programs. *Journal of The ACM 24,* 1, 44–67.

DeMillo, R. A. and Offutt, A. J. 1993. Experimental results from an automatic test generator. *ACM Transactions of Software Engineering and Methodology 2,* 2 (Mar.), 109–127.

Dershowitz, N. and Manna, Z. 1977. The evolution of programs: A system for automatic program modification. In *Conference Record of the Fourth Annual Symposium on Principles of Programming Languages*. ACM SIGACT and SIGPLAN, ACM Press, 144–154.

Feather, M. S. 1982. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems 4,* 1 (Jan.), 1–20.

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

Godefroid, P. and Khurshid, S. 2002. Exploring very large state spaces using genetic algorithms. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, Grenoble, France.

Godefroid, P., Klarlund, N., and Sen, K. 2005. DART: directed automated random testing. *ACM SIGPLAN Notices 40,* 6 (June), 213–223.

Grammatech Inc. 2002. The codesurfer slicing system.

Harman, M. 2007. The current state and future of search based software engineering. In *Future of Software Engineering 2007*, L. Briand and A. Wolf, Eds. IEEE Computer Society Press, Los Alamitos, California, USA, 342–357.

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

Harman, M. and Jones, B. F. 2001. Search based software engineering. *Information and Software Technology 43,* 14 (Dec.), 833–839.

Harman, M. and McMinn, P. 2007. A theoretical and empirical analysis of genetic algorithms and hill climbing for search based structural test data generation. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA To Appear*.

Holland, J. H. 1975. *Adaption in Natural and Artificial Systems*. MIT Press, Ann Arbor.

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

Jones, B. F., Eyres, D. E., and Sthamer, H. H. 1998. A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal 41,* 2, 98–107.

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

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

Lakhotia, K., Harman, M., and McMinn, P. 2008. Handling dynamic data structures in search based testing. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*. ACM, Atlanta, GA, USA, 1759–1766.

Lakhotia, K., McMinn, P., and Harman, M. 2009. Automated test data generation for coverage: Haven't we solved this problem yet?. In *Testing: Academic & Industrial Conference, Practice And Research Techniques (TAIC PART09)*. IEEE Computer Society Press, To Appear.

Liu, X., Lei, N., Liu, H., and Wang, B. 2005. Evolutionary testing of unstructured programs in the presence of flag problems. In *APSEC*. IEEE Computer Society, 525–533.

Liu, X., Liu, H., Wang, B., Chen, P., and Cai, X. 2005. A unified fitness function calculation rule for flag conditions to improve evolutionary testing. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, D. F. Redmiles, T. Ellman, and A. Zisman, Eds. ACM, 337–341.

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 *UK Software Testing Workshop (UK Test 2005)*. Sheffield, UK.

Michael, C., McGraw, G., and Schatz, M. 2001. Generating software test data by evolution. *IEEE Transactions on Software Engineering* 12 (Dec.), 1085–1110.

Mitchell, M. 1996. *An Introduction to Genetic Algorithms*. MIT Press.

Mueller, F. and Wegener, J. 1998. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)*. IEEE, Washington - Brussels - Tokyo, 144–154.

Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. *Lecture Notes in Computer Science 2304*, 213–228.

NIST. 2002. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3.

Offutt, A. J. 1990. An integrated system for automatically generating test data. In *Proceedings of the First International Conference on Systems Integration*, R. T. Ng, Peter A.; Ramamoorthy, C.V.; Seifert, Laurence C.; Yeh, Ed. IEEE Computer Society Press, Morristown, NJ, 694–701.

Offutt, A. J., Jin, Z., and Pan, J. 1999. The dynamic domain reduction approach to test data generation. *Software Practice and Experience 29,* 2 (January), 167–193.

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

Partsch, H. A. 1990. *The Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer.

Pohlheim, H. Genetic and evolutionary algorithm toolbox for use with Matlab.

Pohlheim, H. and Wegener, J. 1999. Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Vol. 2. Morgan Kaufmann, San Francisco, CA 94104, USA, 1795.

Puschner, P. and Nossal, R. 1998. Testing the results of static worst–case execution-time analysis. In *19th IEEE Real-Time Systems Symposium (RTSS '98)*. IEEE Computer Society Press, Madrid, Spain, 134–143.

Radio Technical Commission for Aeronautics. 1992. RTCA DO178-B Software considerations in airborne systems and equipment certification.

Schultz, A., Grefenstette, J., and Jong, K. 1993. Test and evaluation by genetic algorithms. *IEEE Expert 8,* 5, 9–14.

Sen, K., Marinov, D., and Agha, G. 2005. CUTE: a concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, M. Wermelinger and H. Gall, Eds. ACM, 263–272.

Tillmann, N. and de Halleux, J. 2008. Pex-white box test generation for.NET. In *TAP*, B. Beckert and R. Hähnle, Eds. Lecture Notes in Computer Science, vol. 4966. Springer, 134–153.

Tip, F. 1994. A survey of program slicing techniques. Tech. Rep. CS-R9438, Centrum voor Wiskunde en Informatica, Amsterdam.

Tracey, N., Clark, J., and Mander, K. 1998a. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis*. ACM/SIGSOFT, 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 (DCIA)*. IFIP, 169–180.

Wappler, S., Baresel, A., and Wegener, J. 2007. Improving evolutionary testing in the presence of function–assigned flags. In *Testing: Academic & Industrial Conference, Practice And Research Techniques (TAIC PART07)*. IEEE Computer Society Press, 23–28.

Ward, M. 1994. Reverse engineering through formal transformation. *The Computer Journal 37,* 5, 795–813.

Wegener, J., Baresel, A., and Sthamer, H. 2001. Evolutionary test environment for automatic structural testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms 43,* 14, 841–854.

Wegener, J., Grimm, K., Grochtmann, M., Sthamer, H., and Jones, B. F. 1996. Systematic testing of real-time systems. In *4th International Conference on Software Testing Analysis and Review (EuroSTAR 96)*.

Wegener, J. and Mueller, F. 2001. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems 21,* 3, 241–268.

Wegener, J., Sthamer, H., Jones, B. F., and Eyres, D. E. 1997. Testing real-time systems using genetic algorithms. *Software Quality 6*, 127–135.

Weiser, M. 1979. Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. Ph.D. thesis, University of Michigan, Ann Arbor, MI.

Weiser, M. 1984. Program slicing. *IEEE Transactions on Software Engineering 10,* 4, 352–357.

Xie, T., Tillmann, N., de Halleux, P., and Schulte, W. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*.

Received Month Year; revised Month Year; accepted Month Year