Evolutionary Testing in the Presence of Loop–Assigned Flags: A Testability Transformation Approach

André Baresel
DaimlerChrysler AG
Alt-Moabit 96a
D-10559 Berlin

David Binkley
Loyola College
Baltimore MD
21210-2699, USA

Mark Harman
Brunel University
Uxbridge, Middlesex
UB8 3PH, UK

Bogdan Korel,
Illinois Institute of Technology,
10 W. 31st Street,
Chicago, IL 60616.

ABSTRACT

Evolutionary testing is an effective technique for automatically generating good quality test data. However, for structural testing, the technique degenerates to random testing in the presence of flag variables, which also present problems for other automated test data generation techniques. Previous work on the flag problem does not address flags assigned in loops.

This paper introduces a testability transformation that transforms programs with loop–assigned flags so that existing genetic approaches can be successfully applied. It then presents empirical data demonstrating the effectiveness of the transformation. Untransformed, the genetic algorithm flounders and is unable to find a solution. Two transformations are considered. The first allows the search to find a solution. The second reduces the time taken by an order of magnitude and, more importantly, reduces the slope of the cost increase; thus, greatly increasing the complexity of the problem to which the genetic algorithm can be applied. The paper also presents a second empirical study showing that loop–assigned flags are prevalent in real world code. They account for just under 11% of all flags.

Categories and Subject Descriptors: **D.2.5 [Testing and Debugging]:** Testing tools **D.2.2 [Design Tools and Techniques]:** Evolutionary prototyping **F.3.3 [Studies of Program Constructs]:** Control primitives

General Terms: Algorithms, Measurement, Performance, Experimentation, Theory.

Keywords

Evolutionary Testing, Testability Transformation, Flags, Empirical Evaluation

1. INTRODUCTION

Evolutionary testing is a search–based software engineering technique [7, 18], based upon evolutionary algorithms [19, 25]. Evolutionary search is typically used where the search-space is large and the worth of a candidate solution can be determined by a 'fitness' function, which gives higher values to better solutions.

In the case of test data generation, the algorithm uses a population of individuals that represent inputs to the program. The population is updated over a sequence of generations. The selection of individuals who survive to the next generation is governed by a fitness function. Between each generation genetic operators are applied to the individuals. These genetic operators mimic the effects of mating and mutation in natural genetics. The overall effect of the evolutionary algorithm is that the population becomes increasingly dominated by fitter individuals over the evolution of generations. This evolution continues until either a point is reached where the population ceases to evolve any further or an individual with a suitably good fitness score has been found.

In the case of test data generation for branch coverage, the fitness function is computed in terms of how close an input comes to executing the target branch. It is a common experience with branch coverage that many branches are easily covered (even with randomly generated test data). However, as the number of remaining uncovered branches decreases, those that remain become ever harder to cover. It is for these few hard to cover branches that evolutionary techniques provide an attractive solution [24, 28, 40].

Evolutionary testing has repeatedly been shown an effective way to automatically generate test data for a variety of test adequacy criteria (not just branch coverage) [20, 21, 24, 26, 28, 31, 37, 41, 43]. A recent survey of work on evolutionary test data generation is provided by McMinn [23]. This paper concentrates on branch coverage, merely for ease of exposition. The approach adopted herein can be applied equally well to any structural (white box) test adequacy criterion.

Although evolutionary testing works well in many situations, it is hampered by the presence of flag variables–variables that hold one of two discrete values, for example, `true` or `false`. Flags are also a problem for other test data generation techniques, such as the chaining approach [15].

The flag problem is best understood in terms of the *fitness landscape*. A fitness landscape is a metaphor for the 'shape' produced by the fitness function. In this landscape, the location of a point is determined by the individual to which the fitness function is applied and the height of a point is determined by the computed fitness value. Using the fitness landscape metaphor, it becomes possible to speak of landscape characteristics such as plateaus and gradients.

As illustrated in the right 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. This landscape 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'.

Embedded systems, such as engine controllers, 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 a serious problem, since generating such test data by hand is prohibitively expensive, yet it is required by many testing standards [6, 33].

This paper presents an algorithm for transforming programs with flag variables into specially tailored versions of the program that compute the fitness for a particular flag–controlled branch. The approach uses testability transformation [17], a form of transformation in which functional equivalence need not be preserved, but which guarantees to preserve test set adequacy. The primary contributions of this paper are

1. A testability transformation algorithm is introduced which can handle flags assigned in loops.
2. Results of an empirical study are reported which show that the approach reduces test effort and increases test effectiveness. The results also indicate that the approach scales well, as the difficulty of the search problem increases.
3. Results from a second empirical study show that the loop–assigned flag problem is prevalent in real programs.

The rest of the paper is organized as follows. Section 2 provides an overview of background information on evolutionary testing, the flag problem and testability transformation. Section 3 introduces the flag replacement transformation and Section 4 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 problem increases. Section 5 presents the empirical study of loop–assigned flags and examples of real world code that contains loop–assigned flags. Section 6 presents related work and Section 7 concludes.

2. BACKGROUND

This section briefly explains the evolutionary testing system used in the empirical study, 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 DaimlerChrysler Evolutionary Testing system [40], built on top of the Genetic and Evolutionary Algorithm Toolbox [30], using a client–server model. The architecture of the system is depicted in Figure 2 where the outer circle provides an overview of a typical procedure for an evolutionary algorithm: First, an initial population is formed usually with random guesses. Each individual within the population is evaluated by calculating its fitness. This results in a spread of solutions ranging in fitness.

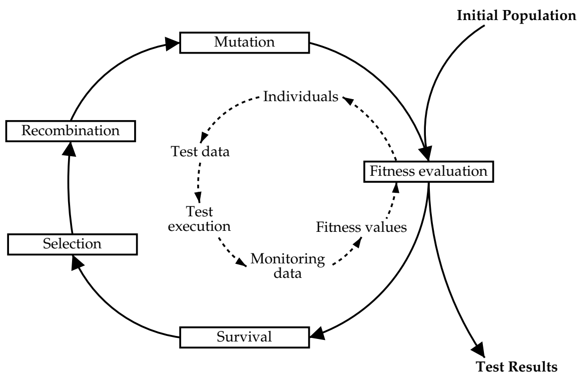


Figure 2: Evolutionary Algorithm for Testing

In the first iteration all individuals survive. Pairs of individuals are selected from the population, according to a pre-defined selection strategy, and combined to produce new solutions. At this point mutation is applied. This models 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 new individuals are evaluated for fitness. Survivors into the next generation are chosen from parents and offspring with regard to their fitness. The algorithm is iterated until the optimum is achieved, or some other stopping condition is satisfied.

In order to automate software tests with the aid of evolutionary algorithms, the test aim must be transformed into an optimization task. This is the role of the inner circle of the architecture depicted in Figure 2. Each generated individual represents a test datum for the system under test. Depending on the test aim is pursued, different fitness functions emerge for test data evaluation.

If, for example, the temporal behavior of an application is being tested, the fitness evaluation of the individuals is based on the execution times measured for the test data [32, 42]. For safety tests, the fitness values are derived from pre- and post-conditions of modules [36], and for robustness tests of fault-tolerance mechanisms, the number of controlled errors forms the starting point for the fitness evaluation [34].

For structural criteria, such as those upon which this paper focuses, a fitness function is typically defined in terms of the program's predicates [1, 4, 20, 24, 28, 40]. 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 the desired (target) path or branch.

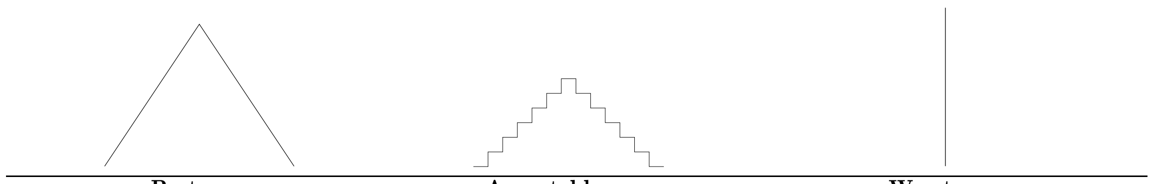

Best case
Smooth landscape with *ubiquitous* guidance toward global optimum.
Acceptable case
Rugged landscape with *some* guidance toward global optimum.
Worst case
Dual plateau landscape with *no* guidance toward global optimum.

Figure 1: The flag landscape: The needle in a haystack problem.

2.2 The Flag Problem

In this paper, a flag variable is any variable that takes on one of two discrete values. Boolean variables are used in the examples. Where the flag only has relatively few input values (from some set S) which make it adopt one of its two possible values, it will be hard to find such a value in S. This problem typically occurs with internal flag variables, where the input state space is reduced, with relatively few 'special values' (those in S) being mapped to one of the two possible outcomes and all others (those not in S) being mapped to the other of the two possible flag values.

A predicate which tests a flag, produces a fitness function that yields either maximal fitness for the 'special values' or minimal fitness for any other value. The landscape induced by the fitness function provides no guidance from lower fitness to higher fitness. This is illustrated in the right of Figure 1.

A similar problem is observed with any n–valued enumeration type, whose fitness landscape is determined by n discrete values. The flag type (where n=2) is the worst case. As n becomes larger the program becomes progressively 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 the flag is assigned a value in a loop, which 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 breakdown in the presence of loop–assigned flags [1, 4, 17].

2.3 Testability Transformation

A testability transformation [17] 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 [10, 29, 39] 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 it has served its role as a vehicle for generating adequate test data. By contrast, in traditional transformation, the original program is replaced by the transformed equivalent.

2. The transformation process need not preserve the traditional meaning of a program. For example, in order to cover a chosen branch, it is only required that the transformation preserve the set of test–adequate inputs. That is, the transformed program must be guaranteed to execute the desired branch under the same initial conditions. By contrast, traditional transformation preserves functional equivalence, a much more demanding requirement.

These two observations have important implications:

1. **There is no psychological barrier to 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 transformation: 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 transformation has taken place.

2. **Considerably more flexibility is available in the choice of transformations 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 transformation 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 flag replacement algorithm is to replace the use of a flag variable with a condition that provides a more gradual landscape. Prior work requires that flag assignments reaching a use not occur within a loop [1, 4, 17]. In contrast, the algorithm presented herein can handle flags assigned inside a loop. It does this by introducing two new variables, `fitness` and `counter`, and by replacing `if(flag)` with `if(fitness==counter)`. The addition of these variables is a kind of instrumentation. The variable `counter` is an induction variable added to count the number of loop iterations that take place (a suitable induction may already exist in the code). The variable `fitness` is a real-valued variable that collects a cumulative fitness score for the assignments that take place during loop execution. Thus, the algorithm essentially transforms the original program into a program tailor to compute a smooth fitness landscape. This landscape has a global optima at the point where the variable `flag` has the desired value.

After transformation, it is possible to simplify the transformed program by taking the slice [3, 35, 45] with respect to the the condition `(fitness==counter)`. Slicing removes unnecessary parts of the program and thus forms a program specialized to the calculation of a smooth fitness function targeting the branch.

The transformation algorithm is presented in Figure 3. It assumes that `flag` is initially assigned `true` and might be subsequently 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`.

Steps 2 and 3 simply insert and initialize the fitness accumulation variable, `fitness`, and the loop iteration counter, `counter`. Step 3 also adds the increment for `counter` at the end of the loop body.

Step 4 uses the bushing and blossom transformation used as part of the testability transformation for non–loop assigned flags [17]. 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. The net effect of this transformation is that the abstract syntax tree becomes a binary tree. Thus, all branches in branching control flow conflate to the same point. Finally, blossoming pushes the assignment statements within a bushed tree to the leaves of the tree. Bushing and blossoming are meaning-preserving.

When combined, bushing and blossoming have the effect of converting the Abstract Syntax Tree (AST) of the body of the loop into a binary tree, in which the internal nodes are predicates and the leaves are a sequence of assignments. Note that blossoming can only be applied when a predicate is side–effect free; thus, for predicates involving side effects, a side–effect removal transformation is first applied [13].

The advantage of bushing and blossoming is that the transformed AST contains one leaf for each path through the body of the loop. Each leaf includes a single assignment to `flag` that reflects the value assigned by a given iteration of the loop body along the associated path. This considerably simplifies the case–based analysis which follows.

Step 5 introduces the update of the fitness accumulation variable, `fitness`. The value added to `fitness` depends upon the value assigned to `flag` along the associated path. If `flag` is assigned `true` then, in essence, assignments in previous loop iterations are irrelevant. To account for this, `fitness` is assigned the current value of `counter`. This assignment overwrites any previously accumulated fitness accumulated in the variable `fitness`.

If `flag` is unassigned along a path, then this current iteration has avoided assigning the value `false` to `flag`. To capture this, the value of the variable `fitness` is incremented. The motivation for this choice is that the loop is one step closer to avoiding an assignment of `false` to `flag`.

Finally, the worst situation is when `flag` is assigned `false`. Here, if no change to `fitness` is made, the resulting fitness landscape is essentially the coarse-grained fitness landscape shown in the middle of Figure 1. Step 5.4 implements a more fine–grained approach, which produces a smoother fitness landscape such as the one seen on the left of Figure 1.

The key observation behind Step 5.4 is that an assignment of `false` to `flag` occurs because a 'wrong decision' was taken earlier in the execution history 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 5.4.2) could avoid the assignment of `false` to `flag`. The value calculated (in Step 5.4.3) for the fitness increment in this case is based upon the standard approach to local fitness calculation in evolutionary testing [40].

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 to 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 it comes to the value of `counter` is determined by how close the loop comes to terminating with `flag` holding the value `true`.

Step 7 is an optional optimization step. It can be ignored, without effecting the functional behavior 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 to evaluate the fitness function. Since evolutionary testing requires repeated execution of the program under test (in order to evaluate fitness), any speed–up will improve the efficiency of the overall approach.

The transformed program is not semantically equivalent to the original. It is a new program constructed simply to mimic the behavior of the original at the target branch. It does so in a way that ensures a more attractive fitness landscape. The standard evolutionary 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`.

Observe that should the initial value of `flag` be *unknown* at the start of the loop, then, because there are only two possible values `flag` can take, a conditional, C can be inserted to test the value of `flag`. The `then` part of C contains a specialized version of the loop which behaves as if `flag` were initially assigned `true`, while the `else` part contains a specialized version of the loop which behaves as if `flag` were initially assigned `false`.

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

```
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
Step 2: Add a variable counter to the loop.
Step 3: Add an assignment fitness=0 as an initialization prior to the loop and an increment of the counter to the end
        of the loop.
Step 4: Bush and Blossom the body of the loop.
Step 5: There are four cases for assignments to flag at the leaves of the AST:
    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 one of the following on a per branch basis.
    Case 5.2: flag is assigned true.
              Add an assignment fitness=counter after the assignment to flag.
    Case 5.3: flag is not assigned a value.
              Add an assignment fitness++ after the assignment to flag.
    Case 5.4: flag is assigned false
        Step 5.4.1: Form the path condition, $\pi$, that leads to the assignment.
                    at which assignments of false to flag can be avoided.
        Step 5.4.2: Backtrack to the first condition, $c$ in $\pi$ at which assignment of false to flag can be avoided
                    (Case 5.1 ensures that such a condition exists).
        Step 5.4.3: Let $f$ = local($c$), where function local is the standard local fitness function.
        Step 5.4.4: Normalize $f$ to a value between 0 and 1, store result in $f'$.
        Step 5.4.5: Add the assignment fitness+=$f'$ after the assignment to flag.
Step 6: Replace the use of flag with fitness==counter.
Step 7: Slice at the replacement predicate fitness==counter, introduced by Step 6.
```

Figure 3: The Transformation Algorithm

4. EMPIRICAL EVALUATION

This section presents an empirical evaluation of the transformation algorithm. Three different transformations are considered. The first leaves the program unchanged. The second applies the transformation algorithm from Figure 3 without **Case 5.4**. The final transformation applies the full transformation algorithm; thus exploiting the 'local fitness calculation' embodied in **Case 5.4**. It is expected that these three transformations will produce landscapes that correspond to those shown in Figure 1.

The program template experimented with is depicted in Figure 4. This program serves as a template because 20 different versions of the program were experimented with for each transformation. In each successive version, the array size is 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; the needle is sought in an increasingly larger haystack. This program is chosen for experimentation because it distills the worst possible case. That is, test data generation needs to find a single value (all array elements set to zero) in order to execute the branch marked `/* target */`. This single value must be found in a search space which is governed by the size, `ELEMCOUNT`, of the array, `a`.

The left hand column of Figure 4 shows the untransformed program. The middle column shows the program produced by the coarse–grained transformation technique (when **Case 5.4** is ignored), and the rightmost column shows the results of the fine–grained transformation. In all three, the final optimization step (**Step 7**) is not applied to facilitate comparisons between the three versions of the program (if applied it would have removed the assignments to `flag`). The function `local` computes the local fitness score for a predicate (the value c in **Step 5.4.2** of Figure 3).

For each transformed program, the evolutionary algorithm was run ten times, to ensure robustness of the results and to allow comparison of the variations between runs for each of the three techniques. An upper limit was set on the number of possible fitness evaluations allowed. This was necessary as some runs failed to find any solution.

The DaimlerChrysler Evolutionary Testing system [2, 40] was used to obtain these results. The 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 DaimlerChrysler developers through an internal company web portal. A full description of the system is beyond the scope of this paper.

The data from selected runs of the two transformed version of the program are presented in Figure 5. The 'no transformation' approach is 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 65536; 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 miniscule chance of finding the 'needle in the haystack'.

The data from all runs of all three techniques are depicted graphically in Figure 6. The top three figures use the same y-axis scale to facilitate comparison. The bottom figures zooms the y-axis by a factor of 10. Notice that there is a spike at array size 10 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 40^{th} generation it has the value 6, but in the 1000^{th} generation this has only reduced to 2. There is a similar spike

```
void f(char a[ELEMCOUNT])
{
  int i;
  int flag = 1;


  for (i=0; i<ELEMCOUNT; i++)
  {
     if (a[i] != 0)
     {
         flag = 0;

     }



  }
  if (flag)
     /* target */
}
```

No transformation

```
void f(char a[ELEMCOUNT])
{
  int i;
  int flag = 1;
  int counter = 0;
  double fitness = 0.0;

  for (i=0; i<ELEMCOUNT; i++)
  {
     if (a[i] != 0)
     {
         flag = 0;

     }
     else
         fitness += 1.0;
     counter++;
  }
  if (counter == fitness)
    /* target */
}
```

Coarse–grained transformation

```
void f(char a[ELEMCOUNT])
{
  int i;
  int flag = 1;
  int counter = 0;
  double fitness = 0.0;

  for (i=0; i<ELEMCOUNT; i++)
  {
     if (a[i] != 0)
     {
         flag = 0;
         fitness = fitness + local(a[i]!= 0);
     }
     else
         fitness += 1.0;
     counter++;
  }
  if (counter == fitness)
    /* target */
}
```

Fine–grained transformation

Figure 4: The program template under test.

Arraysize	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	300	300	300	300	300	300	300	300	300	600
2	1,140	1,140	3,010	1,410	2,244	1,410	1,140	1,677	1,677	3,010
3	1,944	3,810	268,556	5,943	5,943	5,947	3,810	2,478	3,806	2,211
4	155,246	3,276	270,402	269,058	534,644	4,078	534,588	534,516	269,325	534,556
5	271,732	540,492	4,342	4,610	11,548	534,568	534,588	534,576	534,556	269,584
6	2,744	534,584	534,524	801,646	534,532	534,596	1,068,796	271,216	294,996	270,672
7	801,630	534,592	534,540	534,628	801,642	11,828	801,682	5,943	270,939	14,502
8	534,548	7,006	801,646	537,485	273,856	804,133	801,686	534,612	801,706	274,136
9	801,690	534,536	534,528	536,734	269,313	273,344	801,678	801,646	534,576	6,473
10	271,732	1,068,828	801,642	1,335,930	534,580	269,576	540,717	534,572	801,670	542,054
20	1,068,944	801,810	801,814	1,068,944	801,778	534,660	801,838	1,068,992	801,806	1,068,968
30	1,603,328	1,068,932	1,068,948	801,766	1,068,884	1,336,122	2,404,770	801,798	1,068,912	1,068,984
40	1,603,268	1,603,224	1,870,446	1,603,268	2,137,600	1,068,960	2,137,652	801,834	1,068,952	1,336,078

Results for coarse–grained transformation

Arraysize	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	300	300	600	300	600	600	600	600	300	300
2	1,140	1,944	1,410	870	1,410	1,677	1,410	2,211	1,410	2,211
3	4,078	3,276	1,677	2,478	3,276	2,744	1,410	2,211	4,610	2,743
4	5,947	4,881	4,076	6,208	3,010	2,744	3,276	7,814	4,078	5,943
5	5,943	5,678	5,415	5,943	4,877	4,610	5,947	4,343	5,414	4,614
6	7,278	8,350	6,742	5,411	5,411	6,473	6,212	5,415	5,947	11,283
7	7,007	6,742	8,077	15,294	7,278	6,208	7,006	12,354	11,283	8,611
8	8,077	8,344	20,638	6,742	8,081	7,278	6,208	17,697	6,210	44,967
9	8,085	11,018	9,416	8,344	17,960	9,950	16,103	11,282	11,278	26,784
10	6,742	273,619	21,442	16,902	18,498	11,550	16,898	16,634	7,814	11,550
20	20,906	34,000	27,330	26,266	39,081	32,674	27,588	25,456	36,945	25,982
30	225,041	135,253	48,697	72,480	72,474	81,295	121,410	98,920	52,164	77,832
40	51,638	101,578	126,398	389,588	173,960	75,421	75,931	101,870	71,674	68,186

Results for fine–grained transformation

Figure 5: Number of fitness evaluations required for each run of the two versions of the transformation algorithm

at array size 40 in Run 4. Upon investigation, a similar behavior 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 behavior appears to arise from the role chance plays in the underlying evolutionary search algorithm, rather than any properties of the flag problem *per se*.

Figures 7 and 8 shown the averages and standard deviations respectively over all ten runs for each of the three approaches. As can be seen in the graphs, the fine-grained technique outperforms the coarse–grained technique. The coarse–grained technique achieves some success, but its average time is clearly rising and more importantly there is an increase in standard deviation. This increase in variability is a tell-tale sign of increasing randomness in the nature of search with 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 search until a way of moving off a local plateau is found. The average for the 'no transformation' technique is almost uniformly 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 successful search. The high standard deviation for size 2 is evidence that the one solution was a random occurrence.

The qualitative assessment that the fine–grained approach is better than the coarse–grained approach, which in turn, is better than the 'no transformation' approach was confirmed quantitatively using the Mann-Whitney test. This test is a non-parametric test for statistical significance in the differences between two data sets. Because the test is non-parametric, the data is not required to be normally distributed for the test to be applicable. The test reports, among other things, a p-value. Values lower than 0.01 are considered to be highly statistically significant. 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 before return p-values less than 0.0001 indicating that the differences are statistically significant.

5. FLAGS IN REAL CODE

This section first investigates the existence of loop–assigned flag variables "in the large" and then illustrates their existence "in the small." The goal of the "in the large" investigation is to determine if such flags occur in practice. The investigation made use of the dependence graphs output by Codesurfer, a deep structure analysis tool [16]. Traversing dependence edges in a program's dependence graph simplifies the discovery of loop–assigned flag variables. In all, nine programs, with a total of 292,030 lines of code, were analyzed. The programs studied range from the 500 line utility `replace` to the 75,000 line mail handler `sendmail`.

Results for the nine programs are shown in Figure 9. Averaged over all nine programs, 23.6% of the all predicates reference a flag variable (some reference multiple flag variables). Of the 5629 flag variables references, 611 (just under 11%) are loop–assigned. Thus, the problem studied herein is relevant as a significant proportion of the flag used were found to be loop–assigned flags.

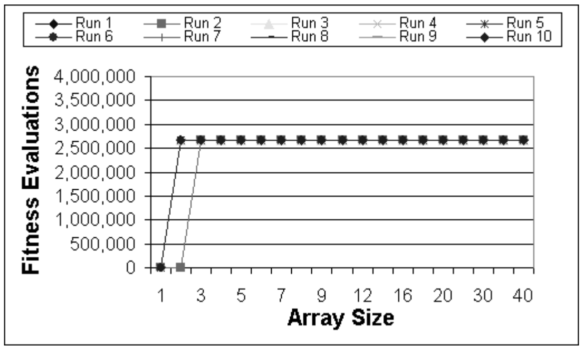
Flag use "in the small" is illustrated in Figure 10, which shows four different source examples that contain loop–assigned flags. These examples illustrate the way in which the loop-flag problem arise naturally in real world source code. The examples come from two systems in current use (a car navigation system and the DaimlerChrysler testing system itself) and from standard algorithms–an ACM network management algorithm and a standard sorting algorithm.

In all four examples, an array is traversed within a loop and the elements are tested against some condition of interest. A flag is set to `true` if this condition arises. Furthermore, the loop is followed by a test of the flag. These are precisely the kind of loop assigned flags considered in this paper. Because the flags denote 'special conditions', it is often hard to find test data to satisfy these conditions.

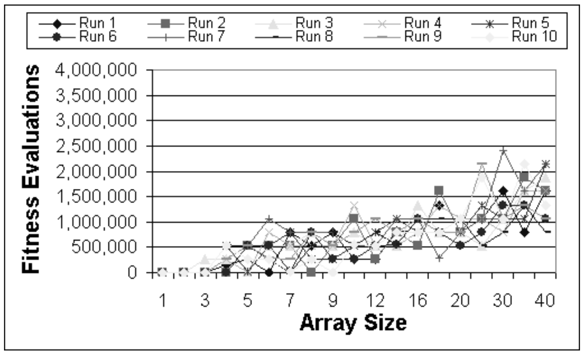
In such situations evolutionary testing is a natural choice, since it is good at optimizing for 'hard to find situations'. However, the presence of the flag variable leads to a degenerate landscapes. Fortunately, these examples can all be transformed using the algorithm from Figure 3. Each example is briefly described below.

Example 1: `update_shps`

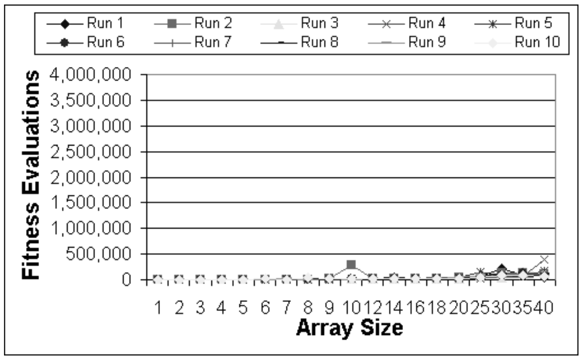
This is a navigation system from a DaimlerChrysler car. The code has been modified to protect non-disclosure of commer-



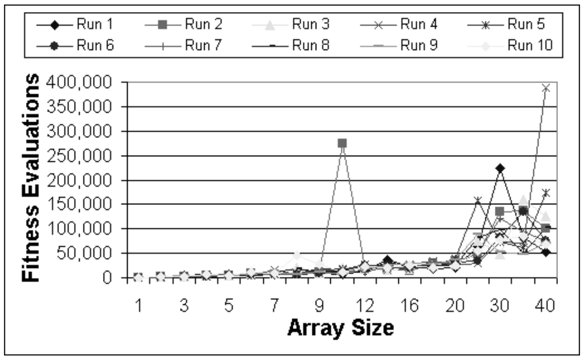
(a) No Transformation



(b) Coarse-Grained Transformation



(c) Fine-Grained Transformation



(d) Fine-Grained Transformation Closeup

Figure 6: Results over ten runs of the evolutionary search for each of the three approaches.

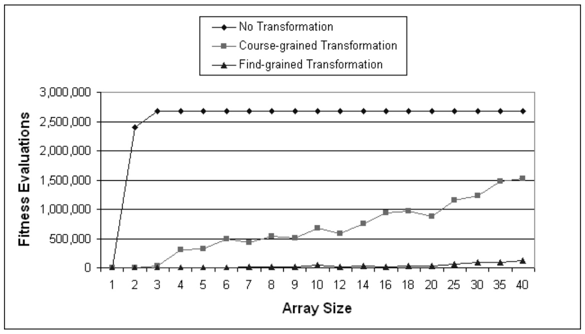


Figure 7: Averages over ten runs of the evolutionary search for each of the three approaches

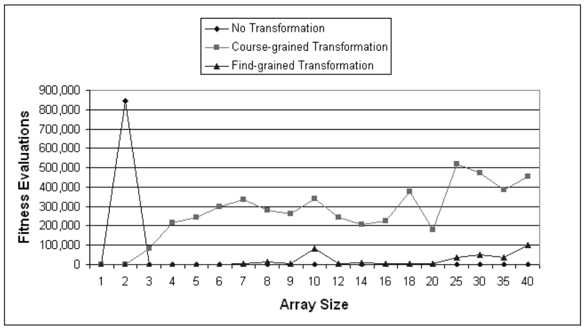


Figure 8: Standard deviation over ten runs of the evolutionary search for each of the three approaches

cially sensitive information. However, these modifications do not effect 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 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 branch marked `/* target */` 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.

Example 2: `handle_new_jobs`

This example is an extract of code from 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 the DaimlerChrysler C++ testing system itself to facilitate 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 shown. In order to execute the `true` branch of this conditional, the assignment `check_work=1;` in the `for` loop must be avoided on every iteration.

	Predicates			Flag Variables		
Program	Total	with Flags		Total	Loop Flags	
`barcode`	235	76	32%	89	13	14%
`empire`	5294	1441	27%	1731	125	7%
`gcc.cpp`	650	163	25%	177	32	18%
`go`	2982	463	15%	538	76	14%
`ijpeg`	1042	153	14%	160	10	6%
`ntpd`	1904	485	25%	560	48	8%
`replace`	54	13	24%	13	7	53%
`sendmail`	3198	719	22%	797	112	14%
`snns`	4858	1256	25%	1564	188	12%
Total	20217	4769	23.6%	5629	611	10.9%

Figure 9: Loop-assigned flags from 9 programs

Example 3: `netflow`

This function is part of an ACM algorithm for 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 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. In this case, it will be set to `true` when `low` is set to a larger value than `high`. In this case the 'special condition' is invalid input.

Example 4: `moveBiggestInFront`

The last example 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.

6. RELATED WORK

Test data must be generated to achieve a variety of coverage criteria to assist with rigorous and systematic testing. Various standards [5, 33] either require or recommend branch adequate testing, and so testing to achieve this is a mission critical activity for applications where these standards apply. Because generating test data by hand is tedious, expensive and error-prone, automated test data generation has, therefore, remained a topic of interest for the past three decades. Several techniques for automated test data generation have been proposed, including symbolic execution [8, 22], constraint solving [11, 27], the chaining method [15] and evolutionary testing [34, 20, 24, 26, 28, 31, 37].

This paper is concerned with evolutionary testing of loop–assigned flags. However, symbolic execution is also known to be hard in the presence of loops [9], because loops force conservative approximations about loop–assigned variables. Back propagation of path information from predicates to form constraints on input variables suffers from a similar problem. Finally, the chaining method also suffers from the flag problem.

The flag problem can be thought of as an example of the high Domain to Range Ratio (DRR) problem which Voas [38] identifies as one source of poor testability. That is, the input space of variables is reduced by assigning to a flag, because the range can take one of only two possible values.

Evolutionary testing in the presence of flags has been studied by three previous authors [1, 4, 17]. Bottaci [4] aims to

```c
#include <math.h>
#define TRUE 1
#define FALSE 0
typedef int int32;
typedef char boolean;
typedef unsigned short shp_index;
struct record {
int32 longitude;
int32 latitude;
};

#define MAX(sizeof(shp_record)/sizeof(struct record))
struct record shp_record[10];
const shp_index next_to_remove = 0;
const shp_index next_to_append = MAX-1;

void update_shps(int32 longitude, int32 latitude) {
  boolean update_points = FALSE;
  shp_index index_zaehler = next_to_remove;

  while ((index_zaehler != next_to_append)) {
  if (labs(shp_record[index_zaehler].longitude-longitude)>50 ||
      labs(shp_record[index_zaehler].latitude-latitude)>50 ) {
    update_points = TRUE;
  }
  else {
        index_zaehler = (index_zaehler + 1) % MAX;
      }
  }

  /* update points and replace by closer ones */
  if (!update_points)
  {
   /* target */
   return;
  }
}
```

Example 1: `update_shps`

```c
typedef enum { NEW,
                INPROGRESS,
                FINISHED }
                Status;


 struct {
   Status  state;
   int     priority;
   char*   messageData;
   } Job;

#define ARRAYSIZE 10
Job joblist[ARRAYSIZE];



/**
 * Only one job per call is handled.
 */

void handle_new_jobs()
{
  int idx=0;
  int check_work=0;
  for (idx=0;idx<ARRAYSIZE;idx++)
  {
    if (joblist[idx].state==NEW)
check_work=1;
  }
  if (!check_work)
  {
    /* target
  ...  code deleted
  */
  }
}
```

Example 2: `handle_new_jobs`

```c
#define ARRAY_SIZE 10
typedef int FlowData[ARRAY_SIZE];
int netflow(
    int low[ARRAY_SIZE],
    int high[ARRAY_SIZE],
    int netconnections[ARRAY_SIZE][ARRAY_SIZE],

    FlowData* in_flows,
    FlowData* out_flows)
{
  /* plausibility check if low and high values */
  int i=0;
  int violation=0;
   while (i<ARRAY_SIZE)
   {
      if (low[i]>high[i]) violation=1;
   }
   if (violation)
     return -1; /* target */
   /*
     code of the netflow integer optimization
  */

  return 0;
}
```

Example 3: `netflow`

```c
#define ARRAY_SIZE 10

int moveBiggestInFront(int data[ARRAY_SIZE])
{
  int i  =1, idx;
  int val =data[0];
  int foundBigger = 0;
  while (i < ARRAY_SIZE)
  {
    if (data[i]>val)
    { foundBigger=1; idx=i; val=data[i]; }
   }
if (!foundBigger)
{
  /* target */
  return;
}
/* do some data exchange,
   not relevant for the test problem
*/
}
```

Example 4: `moveBiggestInFront`

Figure 10: Four Examples of Flags in Loops in Real World Code

correct the instrumentation of the fitness function. by storing the fitness of the initial assignment to a flag variable so that it can be used later on, when the flag variable is used.

Baresel and Sthamer [1] used 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 flow 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. [17] showed how testability transformation could be used to address the flag problem. The approach was to attempt to substitute the definition of the flag for its use in the program under test. Prior transformation ensures that the transformed program is flag–free, allowing standard evolutionary testing techniques to be applied.

All three of these approaches share a similar theme: they seek to connect the last assignment to the flag variable to the use of the flag variable 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 3 could be thought of as a combination of the approaches of Bottaci and Harman et al. It shares the use of auxiliary 'instrumentation variables' with Bottaci's approach, but it uses these in a transformed version of the original program using transformations like the approach of Harman et al.

The most important difference between previous work on the flag problem and that reported here is that none of the previous techniques can be applied when the flag variable is assigned inside a loop, because all the assignments to the flag inside the loop have a bearing upon the final fitness value which must be assigned, rather than just the last assignment.

From a transformation standpoint, the algorithm introduced 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 [44] and the 'evolution transforms' of Dershowitz and Manna [12] and Feather [14]. However, both slices and evolution transforms do preserve some *projection* of traditional meaning. The testability transformation introduce 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.

7. CONCLUSION

This paper has presented a testability transformation for handling flag problems 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 the test data.

The effectiveness of the algorithm is validated with an empirical study that shows how two variations of the algorithm perform for different levels of difficultly of search problem. The results show that the approach scales well to even very difficult search landscapes, for which test data are 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 difficulty of this search problem, the evolutionary testing approach, augmented with the transformation algorithm introduced here finds this value every time.

The paper also presents evidence that the kinds of flag problem considered here arise naturally in a variety of real world systems. Examples from a car navigation system, a test data generator, and network and sorting algorithms are presented. In addition, the paper used results for a separate empirical study to show that the types of flag problem considered here are prevalent among those uses of flags found in a suite of real world programs.

8. ACKNOWLEDGEMENTS

Mark Harman is supported, in part, by EPSRC Grants GR/R98938, GR/M58719, GR/M78083 and GR/R43150 and by two development grants from DaimlerChrysler. Dave Binkley is supported by National Science Foundation grant CCR-0305330.

9. REFERENCES

[1] A. Baresel and H. Sthamer. Evolutionary testing of flag conditions. In *Genetic and Evolutionary Computation (GECCO-2003)*, volume 2724 of *LNCS*, pages 2442–2454, Chicago, 12-16 July 2003. Springer-Verlag.

[2] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1329–1336, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

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

[4] L. Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1337–1342, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[5] British Standards Institute. BS 7925-1 vocabulary of terms in software testing, 1998.

[6] British Standards Institute. BS 7925-2 software component testing, 1998.

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

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

[9] P. D. Coward. Symbolic execution systems - a review. *Software Engineering Journal*, 3(6):229–239, Nov. 1988.

[10] J. Darlington and R. M. Burstall. A tranformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

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

[12] N. Dershowitz and Z. Manna. The evolution of programs: A system for automatic program modification. In *Conference Record of the Fourth*

Annual Symposium on Principles of Programming Languages, pages 144–154. ACM SIGACT and SIGPLAN, ACM Press, 1977.

[13] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Transactions on Software Engineering*, 29(7):665–670, 2003.

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

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

[16] Grammatech Inc. The codesurfer slicing system, 2002.

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

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

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

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

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

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

[23] P. McMinn. A survey of evolutionary testing. *Software Testing, Verification and Reliability*. To appear.

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

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

[26] F. Mueller and J. Wegener. 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)*, pages 144–154, Washington - Brussels - Tokyo, June 1998. IEEE.

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

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

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

[30] H. Pohlheim. Genetic and evolutionary algorithm toolbox for use with Matlab.

[31] H. Pohlheim and J. Wegener. Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1795, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

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

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

[34] A. Schultz, J. Grefenstette, and K. Jong. Test and evaluation by genetic algorithms. *IEEE Expert*, 8(5):9–14, 1993.

[35] F. Tip. A survey of program slicing techniques. Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, Amsterdam, 1994.

[36] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis*, pages 73–81. ACM/SIGSOFT, March 1998.

[37] N. Tracey, J. Clark, and K. Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications (DCIA)*, pages 169–180. IFIP, January 1998.

[38] J. M. Voas and K. W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, May 1995.

[39] M. Ward. Reverse engineering through formal transformation. *The Computer Journal*, 37(5), 1994.

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

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

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

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

[44] M. Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[45] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.