formerly Typeset

Open access • Book Chapter • DOI:10.1007/978-3-540-78917-8_11

Testability transformation: program transformation to improve testability

— Source link

Mark Harman, André Baresel, David Binkley, Robert M. Hierons ...+4 more authors

Institutions: King's College London, Daimler AG, Loyola University Maryland, Brunel University London ...+3 more institutions

Published on: 01 Jan 2008 - Formal Methods

Topics: Testability and Program transformation

Related papers:

- Testability transformation
- Empirical evaluation of a nesting testability transformation for evolutionary testing
- Search-based software test data generation: a survey
- Open Problems in Testability Transformation
- Branch-Coverage Testability Transformation for Unstructured Programs

Share this paper:

View more about this paper here: https://typeset.io/papers/testability-transformation-program-transformation-to-improve-23kyt0xick

Testability Transformation — Program Transformation to Improve Testability

Mark Harman[1*], André Baresel[2], David Binkley[3], Robert Hierons[4], Lin Hu[1], Bogdan Korel[5], Phil McMinn[6], Marc Roper[7]

[1] King's College London, Strand, London, WC2R 2LS.
[2] DaimlerChrysler, Alt Moabit 96a, Berlin, Germany.
[3] Loyola College, 4501 North Charles Street, Baltimore, MD 21210-2699, USA.
[4] Brunel University, Uxbridge, Middlesex, UB8 3PH, UK.
[5] Illinois Institute of Technology, 10 W. 31st Street, Chicago, IL 60616.
[6] University of Sheffield, Regent Court, 211 Portobello Street, Sheffield, S1 4DP, UK.
[7] Strathclyde University, 26 Richmond Street, Glasgow G1 1XH, UK.
[*] Corresponding Author.

Abstract. Testability transformation is a new form of program transformation in which the goal is not to preserve the standard semantics of the program, but to preserve test sets that are adequate with respect to some chosen test adequacy criterion. The goal is to improve the testing process by transforming a program to one that is more amenable to testing while remaining within the same equivalence class of programs defined by the adequacy criterion. The approach to testing and the adequacy criterion are parameters to the overall approach. The transformations required are typically neither more abstract nor are they more concrete than standard "meaning preserving transformations". This leads to interesting theoretical questions. but also has interesting practical implications. This chapter provides an introduction to testability transformation and a brief survey of existing results.

1 Introduction

A *testability transformation* (TeTra) is a source-to-source program transformation that seeks to improve the performance of a previously chosen test data generation technique [28]. Testability transformation uses the familiar notion of program transformation in a novel context (testing) that requires the development of novel transformation definitions, novel transformation rules and algorithms, and novel formulations of programming language semantics, in order to reason about testability transformation.

This chapter presents an overview of the definitions that underpin the concept of testability transformation and several areas of recent work in testability transformation, concluding with a set of open problems. The hope is that the chapter will serve to encourage further interest in this new area and to stimulate research into the important formalizations of test-adequacy oriented semantics, required in order to reason about it.

As with traditional program transformation [12, 36, 43], TeTra is an automated technique that alters a program's syntax. However, TeTra differs from traditional transformations in two important ways:

1. The transformed program 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 adequate test data generation. By contrast, in traditional transformation, it is the original program that is discarded and replaced by the transformed version.
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 preserves the set of test-adequate inputs for the branch. 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 three important implications:

1. **There is no psychological barrier to transformation**. Tradition transformation requires the developer to replace familiar code with machine-generated, structurally altered equivalents. It is part of the fokelore 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 need not even be aware that transformation has taken place.
2. **Considerably more flexibility is available in the choice of transformations to apply**. Guaranteeing functional equivalence is demanding, particularly in the presence of side effects, `goto` statements, pointer aliasing and other complex language features. By contrast, merely ensuring that a particular branch is executed for an identical set of inputs is comparatively less demanding.
3. **Transformation algorithm correctness is less important**. 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 is one of degree and can be detected, trivially, using coverage metrics. By contrast, functional equivalence is *undecidable*.

2 Testability Transformation

Testability transformation seeks to transform a program to make it easier to generate test data (*i.e.*, it seeks to improve the original program's 'testability'). There is an apparent paradox at the heart of this notion of testability transformation:

> Structural testing is based upon structurally defined test adequacy criteria. The automated generation of test data to satisfy these criteria can be impeded by properties of the software (for example, flag variables, side effects, and unstructured control flow). Testability transformation seeks to remove the problem by transforming the program so that it becomes easier to generate adequate test data. However, transformation alters the structure of the program. Since the program's structure is altered and the adequacy criterion is structurally defined, it would appear that the original test adequacy criterion may no longer apply.

The solution to this apparent paradox is to allow a testability transformation to co-transform the adequacy criterion. The transformation of the adequacy criterion ensures that adequacy for the transformed program with the transformed criterion implies adequacy of the original program with the original criterion. These remarks are made more precise in the following three definitions.

First, a *test adequacy criterion* is any set of syntactic constructs to be covered during testing. Typical examples include a set of nodes, a set of branches, a set of paths, etc. For example, to achieve "100% branch coverage", this set would be the set of all branches of the program. Observe that the definition also allows more fine grained criteria, such as testing to cover a particular branch or statement.

Second, a *testing-oriented transformation* is a partial function that maps a program and test adequacy criteria to an updated program and updated test adequacy criteria [20]. (In general, a program transformation is a partial function from programs to programs.) Finally, a *testability transformation* is a testing-oriented transformation, τ such that for all programs p and criteria c, $\tau(p, c) = (p', c')$ implies that for all test sets T, T is adequate for p according to c if T is adequate for p' according to c' [20].

A simple example of a testability-transformation is the removal of code that does not impact a target statement or branch. One approach to such a removal is program slicing [46, 6]. Removing code allows existing techniques to better focus on the statement of branch of interest. A more involved example is given in Section 4.3.

3 Test Data Generation

One of the most pressing problems in the field of software testing revolves around the issue of automation. Managers implementing a testing strategy are soon confronted with the observation that large parts of the process need to be automated in order to develop a test process that has a chance to scale to meet the demands of existing testing standards and requirements [8, 39].

Test data must be generated to achieve a variety of coverage criteria to assist with rigorous and systematic testing. Various standards [8, 39] either require or recommend branch coverage 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 remained a topic of interest for the past three decades [9, 16, 24].

Several techniques for automated test data generation have been proposed, including symbolic execution [9, 23], constraint solving [13, 34], the chaining method [16], and evolutionary testing [40, 22, 32, 33, 35, 37, 42]. This section briefly reviews two currently used techniques for automating the process of test data generation, in order to make the work presented on testability transformation for automated test data generation in this chapter self contained.

3.1 Evolutionary Testing

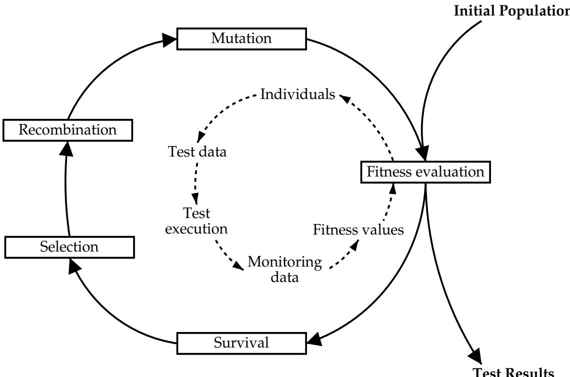


Fig. 1. Evolutionary Algorithm for Testing

The general approach to evolutionary test data generation is depicted in Figure 1[1]. The outer circle in Figure 1 provides an overview of a typical procedure for an evolutionary algorithm. First, an initial population of solution guesses is created, usually at random. Each individual within the population is evaluated by calculating its *fitness*: a measure of how close the individual comes to being a solution (fitness is formalized later in this section). The result is a spread of solutions ranging in fitness.

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 genetics, introducing new information into the population. The evolutionary process ensures that productive mutations have a greater chance of survival than less productive ones.

[1] This style of evolutionary test data generation is based on the DaimlerChrysler Evolutionary Testing System [44].

The new individuals are evaluated for fitness. Survivors into the next generation are chosen from parents and offspring with regard to fitness. The algorithm is iterated until the optimum is achieved or some other stopping condition is satisfied.

In order to automate software test data generation using evolutionary algorithms, the problem must first be transformed into an optimization task. This is the role of the inner circle of the architecture depicted in Figure 1. Each generated individual represents a test datum for the system under test. Depending on which test aim is pursued, different fitness functions emerge 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 [38, 45]. For safety tests, the fitness values are derived from pre- and post-conditions of modules [41], and for robustness tests of fault-tolerance mechanisms, the number of controlled errors forms the starting point for the fitness evaluation [40].

For structural criteria, such as those upon which this chapter focuses, a fitness function is typically defined in terms of the program's predicates [4, 7, 22, 32, 35, 44]. 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.

Typically, the test-data generation tool first instruments each predicate to capture fitness information, which guides the search to the required test data. For example if a branching condition `a == b` needs to be executed as true, the values of `a` and `b` are used to compute a fitness value using `abs(a-b)`. The closer this "branch distance" value is to zero, the closer the condition is to being evaluated as true, and the closer the search is to finding the required test data.

As a simple example, consider trying to test the true branch of the predicate `a > b`. While typical execution of a genetic algorithm might include an initial population of hundreds of test inputs, for the purposes of this example, consider two such individuals, i_1 and i_2. Suppose that, when executed on the input i_1, `a` equals `b`, and when run on i_2, `a` is much less than `b`, then i_1 would have a greater chance of being selected for the next generation. It would also have a better chance of being involved in (perhaps multiple) crossover operations with other potential solutions to create the children that form the next generation.

3.2 The Chaining Method

The chaining approach uses data flow information derived from a program to guide the search when problem statements (conditional statements in which a different result is required) are encountered [16]. The chaining approach is based on the concept of an event sequence (a sequence of program nodes) that needs to be executed prior to the target. The nodes that affect problem statements are added to the event sequence using data flow analysis.

The alternating variable method [25] is employed to execute an event sequence. It is based on the idea of 'local' search. An arbitrary input vector is chosen at random, and each individual input variable is probed by changing its value by a small amount, and then monitoring the effects of this on the branches of the program.

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

For example, consider again, the predicate `a` $>$ `b`. Assuming `a` is initially less than `b`, a few small increases in `a` improves the objective value (the difference between `a` and `b`). Thus, the pattern phase is entered, during which the iteration of ever-larger increases to the value of `a` finally produce a value of `a` that is greater than `b`, satisfying the desired predicate and locating an input that achieves coverage of the desired branch.

4 Three Application Areas for Testability Transformation

The effectiveness of test data generation methods, such as the evolutionary method and the chaining method, can be improved through the use of testability transformation (TeTra). This section presents three case studies that illustrate the wide range of testability transformation's applicability. The first two subsections concern applications to evolutionary testing, while the third concerns the chaining method.

4.1 TeTra to Remove Flags for Evolutionary Testing

Testability Transformation was first applied to the *flag problem* [19]. This section considers the particularly difficult variant of the flag problem where the flag variable is assigned within a loop. Several authors have also considered this problem [7, 4]; however, at present, testability transformation offers the most generally applicable solution. Furthermore, this solution is applicable to other techniques such as the chaining method and symbolic execution [11], which are known to perform poorly in the presence of loop assigned flags.

A *flag* variable is any boolean variable used in a predicate. Where the flag only has relatively few input values (from some set S) that make it adopt one of its two possible values, it will be hard for any testing technique to uncover a value from 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.

The fitness function for a predicate that tests a flag yields either maximal fitness (for the "special values") or minimal fitness (for any other value). In the landscape induced by the fitness function, there is no guidance from lower fitness to higher fitness. This is illustrated by the landscape at the right of Figure 2.

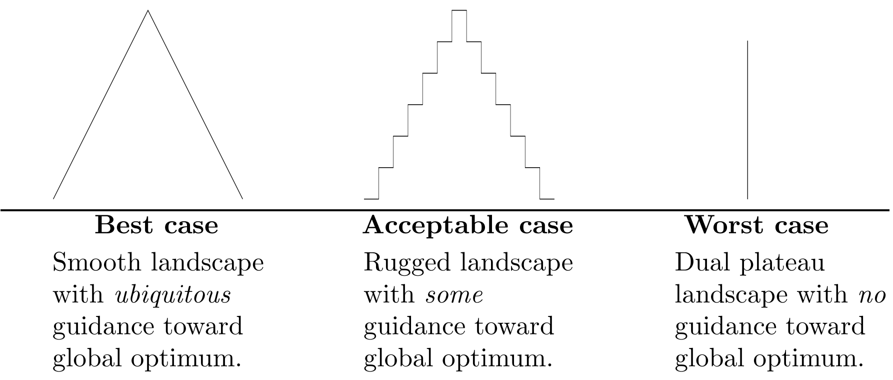


Fig. 2. The flag landscape: the needle in a haystack problem. The y-axis measures fitness while each x-axis represents the input space.

A similar problem is observed with any k-valued enumeration type, whose fitness landscape is determined by k discrete values. As k becomes larger the program becomes progressively more testable; provided there is an ordering on the k elements, the landscape becomes progressively more smooth as k increases. The landscapes in the centre and then left of Figure 2 illustrate the effect of increasing k.

The problem of flag variables is particularly acute where the flag is assigned a value in a loop and then used later outside the loop. For example, consider the variable flag in the upper left of Figure 3. 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. Many previous approaches to the flag problem breakdown in the presence of loop-assigned flags [4, 7, 20]. These simpler techniques are effective with non-loop-assigned flags.

The aim of the loop-assigned flag removal algorithm is to replace the use of a flag variable with an expression that provides better guidance. The algorithm has two steps. The first adds two variables: a new induction variable, `counter`, is added to the loop to count the number of iterations that take place. The second new variable, `fitness`, is a real-valued variable that collects a cumulative fitness score for the assignments that take place during the loop. When applied to code from the upper left of Figure 3, the result of the first step is shown in the upper right of Figure 3. Where "if (flag)" has been replaced with "if (counter == fitness)".

The variable counter measures the number of times the loop passes down the desired path (the one which executes the assignment to flag in a way that

```
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 */
}
```

Original Untransformed Program

```
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

Fig. 3. Illustration of the coarse and fine grain loop-flag removal transformation.

gives the desired final value for flag). This gives rise to the improved but coarse grained landscape as shown in the centre of Figure 2 [2]. The coarseness comes because loop iteration is deemed either to traverse the desired path (with a consequent increase in accumulated fitness) or to miss this path (with no change in accumulated fitness).

A further improvement is possible using an additional transformation that instruments the program to compute, for iterations that fail to traverse the de-

scribed path, how close the iteration comes to traversing the desired path. The transformed code, shown it the lower section of Figure 3, employs the computation of a "local fitness calculation" (the function local), which captures the proximity of each loop iteration to the desired branch. This produces the smoothest fitness landscape (shown at the left of Figure 2).

The function local is a macro expansion that implements a different 'local' or 'branch' fitness [28]. The particular expansion applied depends upon the predicate to be optimized and can, as such, be viewed as a parameter to the overall approach.

Once the transformation has added these variables, the algorithm's second step slices [46, 6] the resulting program with respect to the transformed predicate. Slicing removes parts of the program that do not influence the predicate. The result is a program specialized to the calculation of a smooth fitness function targeting the single branch of interest. In this way, the algorithm has essentially transformed the original program into a fitness function, tailor-made to have a smooth fitness landscape with a global optimum at the point where the variable `flag` has the desired value.

To provide empirical data as to the impact of loop assigned flag removal, the three programs depicted in Figure 3 were studied. (The effect of the slicing step is not shown in the figure to facilitate comparisons between the three versions of the program.) 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 of the array. The program is thus a template and 20 different versions were experimented with for each technique. In each successive version, the array size is increased, from an initial value 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 large haystack.

The DaimlerChrysler Evolutionary Testing system was used to obtain the results [5, 44]. This system generates test data for C programs using a variety of white box criteria. It is a proprietary system, developed in-house and provided to DaimlerChrysler developers through a web portal.

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

The ten-run averages for each of the three approaches are depicted in Figure 4. As can be seen, the fine-grained technique outperforms the coarse-grained technique. The coarse-grained technique achieves some success, but the test effort is noticeably worse than for the fine-grained technique. Both do considerably better than the no transformation approach which fails to find suitable test data on most runs.

The data from all 10 runs of each program are depicted in Figure 5. The no

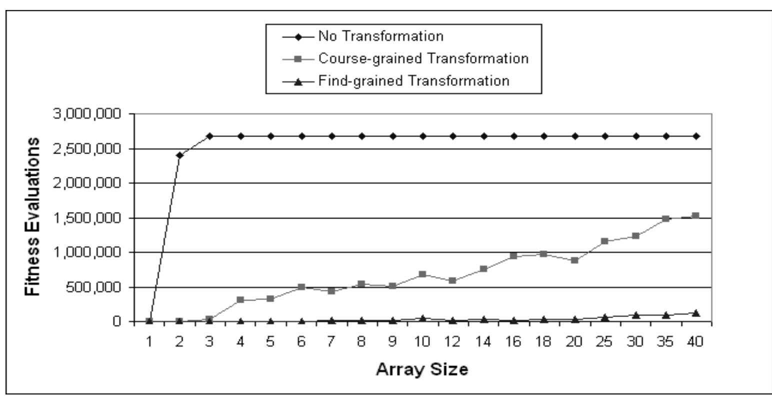


Fig. 4. Ten-run averages of the evolutionary search for each of the three approaches.

transformation approach fails to find any test data to cover the branch in all but two situations. The first is where the array is of size one. In this instance there is a high chance of finding the "special value" by random search, and all ten runs achieve this. At array size two, the chances of hitting the right value at random have diminished dramatically; only one of the ten runs manages to find a solution. 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 finding the "needle in the haystack".

The data for the course grained approach shows success for all runs with a steady increase in effort required. Perhaps more importantly, as seen in the middle graph of Figure 5, there is an increase in variability (the height difference from fewest to most fitness evaluations is growing as the problem becomes more difficult). This is a tell-tale sign of the partially random nature of search. 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 the local plateau is found.

Finally, the only interesting aspect of the data for the fine-grained transformation are two spikes (at array size 10 and 40). These are essentially the mirror image of the "good luck" the untransformed algorithm had finding a solution randomly. Here, the algorithm gets to test data in which all but one array entry is zero, but then through random "bad luck" takes longer to find the solution. In this case it only serves to slow the search. It does not prevent the search from finding the desired test data.

Statistically, the claim that the fine-grained approach is better than the coarse-grained approach, which in turn, is better than the no transformation approach was confirmed using a Mann-Whitney test [1]. 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 nor-

mally distributed for the test to be applicable. Both comparisons report high statistically significant difference ($p < 0.0001$).

4.2 TeTra for Nested Predicates to Assist Evolutionary Testing

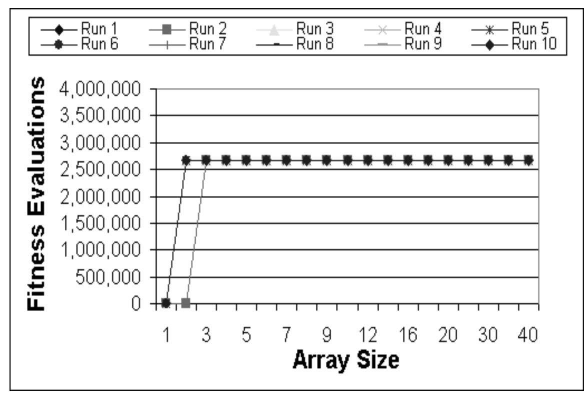
The second example considers the problem that predicate nesting causes evolutionary test data generation. Evolutionary techniques face two key problems when encountering nested predicates: first, constraints on the input are only revealed as each individual predicate is satisfied, and second, the information guiding the search is treated locally at each predicate. For example, consider the code shown in Figure 6a. Here the condition `c == 0` is not encountered until after `a` equals `b`. Thus, the search does not find out that `c` is important until `a == b` is satisfied. Furthermore, while attempting to make `c == 0` true, the search must operate in the smaller search space defined by the predicate `a == b`. Any adjustment in the values of `a` or `b` could potentially put the search back at 'square one'. Thus, once test data has been found to execute a conditional in a certain way, the outcome at that condition must be maintained so that the path to the current condition is also maintained.

The latter problem causes problems for the search when predicates are not mutually exclusive. For example, in Figure 6a, variable `c` must be made zero without changing the values of `a` and `b`. However `c` is actually `b+1` (Statement 2). Therefore `b` needs to be -1 for Statement 3 to be executed as true. If values other than -1 have been selected for `a` and `b`, the search has no chance of making the condition at Statement 3 true. That is unless of course it backtracks to reselect values of `a` and `b`. However, if it were to do this, the fact that `c` needs to be zero at Statement 3 will be 'forgotten,' as Statement 3 is no longer reached, and its fitness is not computed.

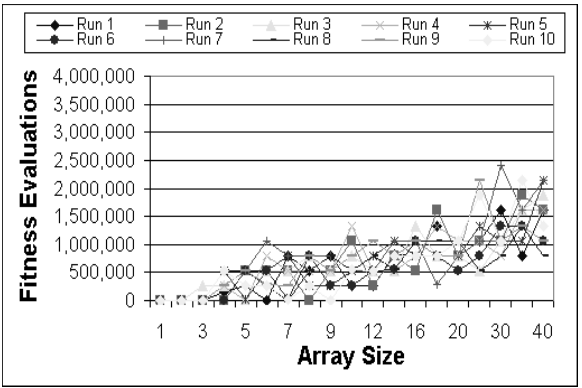
This phenomenon is captured in a plot of the fitness landscape (Figure 6c). The shift from satisfying the predicate of Statement 1 to the secondary satisfaction of the predicate of Statement 2 is characterized by a sudden drop in the landscape down to spikes of local optima. Any move to input values where `a` is not equal to `b` yanks the search up out of the optima and back to the area where Statement 1 is evaluated as false again.

McMinn *et al.* proposed a solution to the nested predicate problem based on testability transformation [31]. In essence, their approach evaluates all the conditions leading to the target at the same time. This is done by flattening the nesting structure in which the target lies and is non-trivial when code intervenes between conditionals (for example, it could contain a loop). The transformation takes the original program and removes decision statements on which the target is control dependent. In this way, when the program is executed, it is free to proceed into originally nested regions, regardless of whether the original branching predicate would have allowed that to happen.

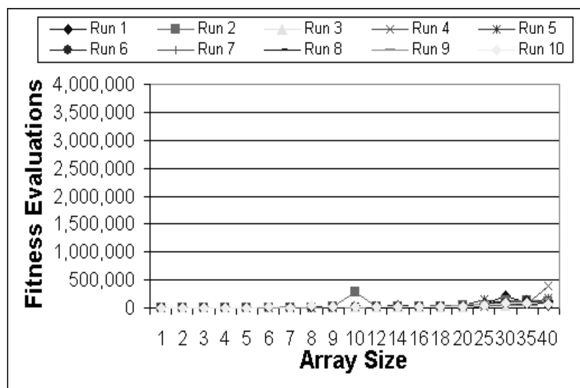
In place of each predicate an assignment to a new variable `_dist` is added. These assignments compute the branch distance based on the original predicate. At the end of the program, the value of `_dist` reflects the summation of each of the individual branch distances. This value may then be used as the fitness



(a) With No Transformation



(b) With Coarse-Grained Transformation



(c) With Fine-Grained Transformation

Fig. 5. The ten runs of the evolutionary search for each of the three approaches. For a given array size, each chart shows the total number of fitness evaluation required to find a solution or reaching the fixed limit.

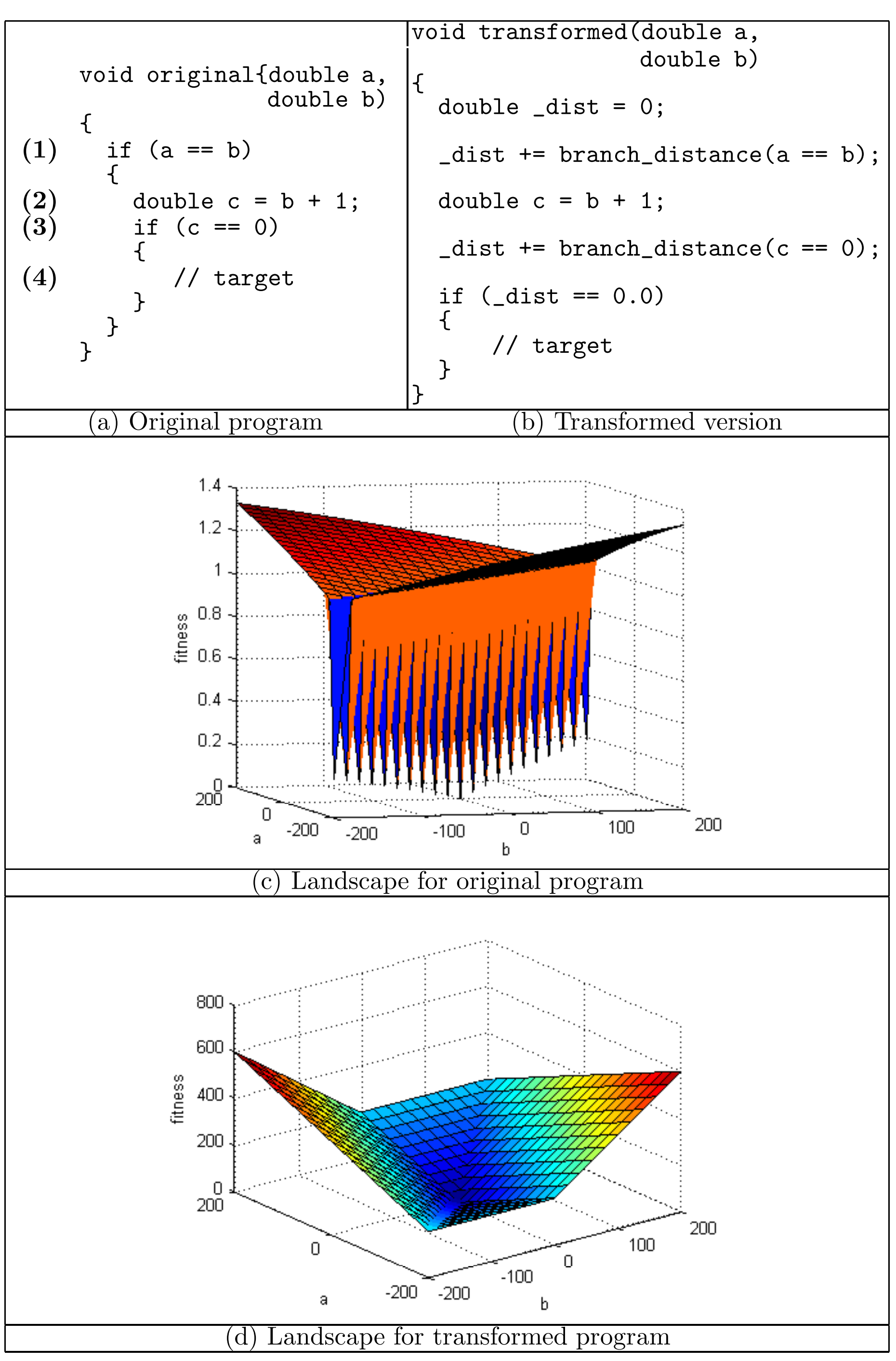


Fig. 6. Case study showing (a) the original and (b) the transformed versions of the code. The transformation removes the sharp drop into points of local minima prevalent in the fitness landscape of the original program seen in part (c), with the more directional landscape of the transformed program, seen in part (d).

value for the test data input. This inline accumulation of fitness information within the program body is not unlike the fine-grained transformation method employed by Baresel *et al.* [2] for collecting information in loop bodies involving assignments to flags.

Figure 6 shows an example of this transformation where the original program, shown in Figure 6a, is transformed into the program seen in Figure 6b. The benefit of the transformation is seen by comparing the fitness landscapes shown in Figures 6c and 6d where the sharp drop into local minima of the original landscape is replaced with smooth planes sloping down to the global minimum. The improvement comes because the search can concentrate on both conditionals at the same time and is in possession of all the facts required to make them both true at the very beginning of the search.

The solution is not free of challenges, as the transformed program can potentially have issues with the removal of certain types of predicates that prevent the occurrence of run-time errors. One example of this is a predicate that tests and then references heap allocated storage. For example, transformation of the conditional `if (p != NULL) { if (p->x > 0) ··· }` would include, in the computation of `_dist`, the dereferencing of the potentially `NULL` pointer `p`.

Therefore the transformed test object must be evaluated in its own 'sandbox' to prevent any potential abnormal termination from affecting the rest of the test data generation system. The fitness function is calculated using all fitness information that was successfully accumulated. In this way, search performance is unlikely to be worse than if the original version of the program were being used. Improving the treatment of such predicates forms one area for future work, while the issue of nesting within a loop body forms another.

In their study of nested `if` statements McMinn *et al.* provide two key empirical results from a collection of forty real-world programs [31]. The first result shows the prevalence of nested predicates in real-world code (on average 3 nested predicate pairs per 100 lines of non-comment non-blank code). Over 80% of these include intervening code two-thirds of which affected the second predicate and thus cannot simply be reordered out of the way. The second result compares finding test data for the original and transformed versions of two programs. The first program is that of Figure 6. TeTra allowed the evolutionary search to find test data using half the effort (test data evaluations) of the untransformed program. The second program studied included three levels of nesting. For the original version of this program, the evolutionary algorithm failed to find test data, while it succeeded every time using the transformed version.

4.3 TeTra for Data Dependence Transformation to Assist the Chaining Method

The third case study, considers the speculative application of testability transformation to the chaining method [16]. Existing test data generation methods use different types of information about a program in order to guide the search process (*e.g.*, a control flow graph, control dependencies, data flows, etc). Although

existing methods work well for many programs, complex logic and intricate dependence relationships between program elements can pose a challenge to test generators. Thus, without transformation, test data is hard for a data-flow technique to generate. Transformation is used in this case is to remove the barrier created by control dependencies in discovering a good 'chain'.

This third case study, exploits the fact that testability transformation need not preserve the standard semantics. In this more radical form of testability transformation, the transformations may yield programs for which it is known that the wrong test data will be produced. However, this technique can be used to speculatively generate test data. As a result, the search can find the solution where other techniques fail. This approach is more expensive and thus typically applied only after existing cheaper methods fail to find the required test data. If the transformation fails, then nothing additional is lost, so the method need only improve test data generation in some cases in order to be valuable [27].

The goal of the transformation is to produce a program that contains only the statements responsible for the computation of the fitness function. The major advantage of the transformed program is that it is easy to execute any statement that affects the fitness function. As a result, the transformed program allows efficient exploration of different paths in order to identify paths that lead to the target value of the fitness function. The technique has five steps. First a data-dependence subgraph is built. This is then used to generate the transformed program. In the next step, paths in the data-dependence subgraph are selected for exploration. For each selected path test data is generated using the transformed program to identify *promising* paths (*i.e.*, paths that lead to the target value of the fitness function). Finally, *promising* paths are used to guide the search for test data using the original program. A good data-flow analysis tool is only well suited to handle the first step [27].

The technique is data-dependence based and thus stands in contrast with existing techniques that are strongly tied to program control flow [26]. To motivate this choice, the authors note that finding test data can frequently require executing parts of the program that are (from the control flow perspective) unrelated. Data dependence analysis, however, ties these regions together as it captures the situation in which one statement assigns a value to a variable that another statement uses. For example, in the function of Figure 7 there exists a data dependence between Statements 13 and 20 because Statement 13 assigns a value to variable `top`, Statement 20 uses variable `top`, and there exists a the control path (13, 14, 15, 19, 23, 6, 7, 8, 9, 15, 19, 20) from 13 to 20 along which variable `top` is not modified.

The technique's first step builds a data dependence graph and then extracts the subgraph for a particular statement. In a data dependence graph nodes represent statements from the program, and directed arcs represent data dependencies [17]. For a chosen node, the extracted data-dependence subgraph includes all the nodes for which there exists a path to the selected node. These represent the statements that may influence the chosen statement. For example, Figure 8 shows the data dependence subgraph for the node corresponding to Statement

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

Fig. 7. A sample C function.

20 from Figure 7.

This statement is referred to as a *problem* statement because Statement 21 is difficult for other test-data generation techniques to generate test data for (it's execution requires Statement 13 to be executed 101 times before reaching Statement 20). The next step uses the subgraph extracted for a problem statement to guide the construction of the transformed program. Each statement that belongs to the subgraph is included in the transformed program as the case of a switch-statement. This program includes the statements whose nodes appear in the extracted subgraph (*e.g.*, see Figure 9).

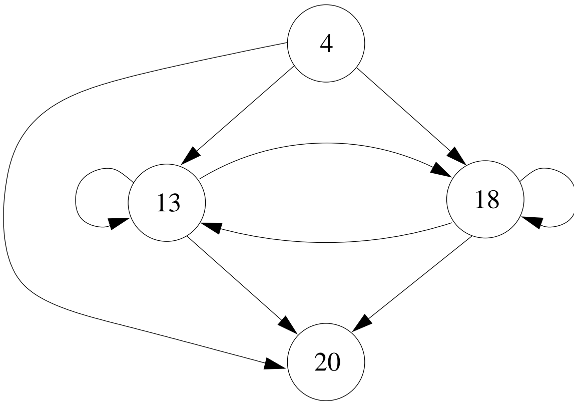


Fig. 8. Data Dependence Subgraph

In addition, to the input parameters from the original program, the transformed program includes two new input parameters, S and R. The array S represents *data dependence paths* from the extracted subgraph. Only paths that begin with a node that has no incoming arcs and end at the problem node are considered. For example, 4, 13, 18, 20 is a data dependence path in the subgraph shown in Figure 8. Array S indicates the sequence of statements from the data dependence path that are to be executed. The transformed program contains a while-loop with a switch-statement inside it. These combine to execute the statements as indicated by S.

Some nodes in the data dependence subgraph have self-loops (statements such as i++ within a loop depened on themselves). In order to explore the influence of such data dependences on the fitness function, the corresponding nodes need to be repeatedly executed. This is the purpose of the input array, R; thus, $S[i]$ indicates the particular statement to be executed and $R[i]$ indicates a number of repetitions of this statement. In essence, the transformation has produced a function from the inputs S and R to the value of the fitness function for the problem node.

For example, a transformed version of the function from Figure 7 for problem Statement 20 is shown in Figure 9. The transformed function contains the statements that are part of the data dependence subgraph from Figure 8 (statements 4, 13, and 18). The end point is omitted because it does not modify the state. These statements affect the computation of the fitness function associated with problem Statement 20. For statements 13 and 18, for- loops are included because these statements have self-looping data dependences in the data dependence subgraph.

After transformation, a search generates different data dependence paths for exploration. A path is represented by S. For each S, the goal is to find values for

```
1   float transformed(int S[], int R[])
    {
      int i, j, top;
2     i=1;
3     while (i <= length(S))
      {
4       switch (S[i])
        {
5         case 4: top = 0;
6                 break;
7         case 13: top++;
8                  for (j=1; j<R[i]; j++)
                      top++;
9                  break;
10        case 18: top--;
11                 for (j=1; j<R[i]; j++)
                      top--;
12                 break;
13      }
14      i++;
15    }
16    return 100-top;
    }
```

Fig. 9. Transformed version of the code from Figure 7 for the data dependence subgraph of Figure 8. The result value is the fitness function for Statement 20 of Figure 7

R such that the fitness function evaluates to the target value. The search uses the existing test generation techniques [16, 32, 44] to find an input on which the fitness function evaluates to the target value in the transformed function. If the target value is achieved, the path is considered a promising path. Otherwise, the path is considered to be unpromising and it is rejected. Finally, promising paths in the transformed program are used to guide the search in the untransformed program.

For example, the transformed program for problem Statement 20 from Figure 7, shown in Figure 9, captures the five acyclic paths in the data dependence subgraph of Figure 8. The following table shows these paths and their corresponding inputs

path	corresponding input
P_1: 4, 20	$S[1] = 4$
P_2: 4, 18, 20	$S[1] = 4, S[2] = 18$
P_3: 4, 13, 20	$S[1] = 4, S[2] = 13$
P_4: 4, 13, 18, 20	$S[1] = 4, S[2] = 13, S[3] = 18$
P_5: 4, 18, 13, 20	$S[1] = 4, S[2] = 18, S[3] = 13$

For each path, the goal is to find values for array R such that the value of the fitness function returned by the transformed function is negative. The search fails for paths P_1 and P_2 and these paths are rejected. When path P_3 is explored,

the search finds the values $R[1] = 1; R[2] = 101$ for which the fitness function evaluates to the negative value in the transformed function. Therefore, path P_3 with 101 repetitions of Statement 13 is considered as a promising path. When this path is used to guide the search of the function of Figure 7, an input is easily identified for which target Statement 21 is executed. Using the transformed function of Figure 9, it is possible to find a solution although only five data dependence paths need be considered as opposed to over one hundred path explorations when the transformed function is not used.

5 A Road Map for Future Work on Testability Transformation

This chapter has surveyed the current state-of-the-art of testability transformation. There remain many open problems. This section sets out a road map for future work on TeTra.

1. **Algorithms**
   Currently there are several algorithms for test-data generation using testability transformation. These tackle a variety of problems such as flag variables [2], nesting [31], and unstructured control flow [21]. The existence of these algorithms demonstrates the potential and wide applicability of testability transformation. However, there remain many open problems in test data generation for which algorithms have yet to be developed. For example, the problems of internal state [29, 30], continuous input [3], and the loop problem for symbolic execution [11].
2. **Semantics**
   As shown in Section 2 the ideas behind testability transformation require a new notion of semantic correctness since the transformations are neither more abstract nor more concrete than the standard semantics. There are a number of open problems in testability transformation work, relating to the semantic foundations of the approach. Initial work has explored the proof obligations for testability transformation [21]. This work considers a transformation that reduces multi-exit loops to single exits loops and illustrates the need for different kinds of proof obligation in reasoning about testability transformation. Specifically, the proof obligations are not that the transformations preserve the behaviour of the program, rather that any test suite that provides 100% branch coverage for the tranformed program is guaranteed to provide 100% coverage for the original program. However, no semantic investigation of correctness has yet been performed for other testability transformation algorithms. Such a proof would be relatively uncomplicated for the work on flag removal reported in Section 4.1 (because this algorithm aims to preserve standard semantics). However, for the other two algorithms, described in Sections 4.2 and 4.3, the proof obligations require the formulation of an alternate semantics. This result is a more challenging and enticing problem as both transformations preserve only *aspects* of the semantics.

3. **Raising the Abstraction Level**
   Existing work in testability transformation has considered the problem of applying standard and non-standard code level transformation rules and tactics to improve testability at the source code level of abstraction. However, there is a general movement in testing and design away from the code level to the model level, and so there may be a case for the application of testability transformation at higher levels of abstraction, such as the design and specification level. There is a particularly strong current interest in model driven development, with a consequent interest in development of approaches for testing at the model level. It is likely that there will be many problems for test data generation at the model level and this suggests the possibility of applying testability transformation at the model level of abstraction.

   There has also been much work on development of techniques for testing from formal specifications. Here too, it may be possible to use testability transformation to transform specifications into a format more amenable to testing.
4. **Other Kinds of Testability Transformation**.
   The focus of this chapter has been upon testability transformations that transform programs (and possibly the adequacy criterion). However, it is not hard to imagine scenarios in which the criterion is transformed while the program remains unchanged. For example, one could imagine a testability transformation, that takes a program, p, and a test adequacy criterion, c, and returns the test adequacy criterion, c' that is the 'lowest' possible criterion below c in the subsumes relationship for which all adequate test sets are identical to those for p and c. This would make it possible to capture the way in which certain programs are constructed in such a way that weaker test data generation methods will suffice for achieving stronger testing results. Such an approach complements cost reduction techniques considered in the existing literature such as test-set minimization [18].

   For example, consider the program fragment

   ```
   if (E) x=1; else x=2;
   ```

   In this simple example, covering all statements will also cover all branches of E. However, in general this is not the case. The problem is to compute the weakest adequacy criterion (in the subsumes lattice [47]) that is sufficient to meet a given adequacy criterion c for a given program p. This general problem is more challenging and can be formulated as a testability transformation. Such a formulation would have useful practical ramifications. Where, for example, it can be determined that a weaker test data generation technique can be used, then it is possible to employ a test data generation tool with performance advantages that accrue from its attempt to satisfy only the weaker criterion. This is particularly advantageous when the original, more demanding adequacy criteria, has no tool capable of generating test data.
5. **Other Testing Oriented Transformation**.
   The definition of Testability transformation (in Section 2) is couched in terms

of a "Testing Oriented Transformation." This is a transformation that takes and returns a pair containing the program under test and the adequacy criterion under consideration. It may be that there are other forms of testing-oriented transformation that may turn out to be useful. There may also be other interesting relations on programs, test data, and test adequacy criteria that remain to be explored.

6 Related Work

Testability transformation is a novel application of program transformation that does not require the preservation of functional equivalence. This is a departure from most work on program transformation, but it is not the first instance of non-traditional meaning-preserving transformation. Previous examples include Weiser's slicing [46] and the "evolution transforms" of Dershowitz and Manna [14] and Feather [15]. However, both slices and evolution transforms do preserve some *projection* of traditional meaning. Testability transformation as 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 improved the program itself.

There has only been non-transformation based previous work on the first of the three application areas considered in Section 4. For the other two applications of testability transformation (to the nested predicate problem and the chaining method), testability transformation is currently the only method to have been applied. The first application, to the problem of evolutionary testing in the presence of flags, has been considered in three previous papers [7, 4, 20]. Bottaci [7] introduces an approach which aims to correct the instrumentation of the fitness function. Baresel and Sthamer [4] used a similar approach to Bottaci [7]. 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 that have an influence on the flag condition at the point of use.

The paper that introduced testability transformation by Harman et al. [20] presented a testability transformation approach to the flag problem, based upon substituting a flag variable with its computation. The approach could not handle loop-assigned flags. Work since then, summarised in this chapter, has improved the technique in terms of its generality, applicability, and effectiveness.

7 Conclusion

Testability transformation is a new application for program transformation. It concerns the application to aid testing rather than the more familiar application areas of optimization, comprehension, re-engineering, or program development through refinement. However, testability transformation is more than merely a novel application area of a long-standing area of research and practice; the fundamental nature of the transformations takes a different form than conventional transformation.

Testability transformations are applied in order to improve testing. The equivalence that needs to be preserved is not functional equivalence (as with almost all prior work on transformation). Rather, it is the set of adequate test sets. This has been shown to be neither more abstract nor more concrete than normal transformation, with the result that testability transformation is not simply an instance of abstract interpretation [10]. Rather, it includes novel transformation rules and algorithms and suggests the need for novel formulations of programming language semantics in order to reason about testability transformations.

Furthermore, testability transformation is a means to an end and not an end result in itself. This has important practical ramifications, such as a reduced importance for correctness of transformations and a lower psychological barrier to acceptance of transformation. if a transformation rule is incorrect, the consequence is not an errant program, it is merely the (possible) failure to find desired test data. This is both a less critical consequence and it is also an easily computable outcome. That is, should a conventional transformation be incorrect, the problem of determining whether there has been an impact upon the transformed program is undecidable, whereas the problem of determining whether test adequacy has been satisfied is usually trivial as it simply requires running the program. With the continued need for extensive unit testing and the growing available of 'spare' processor cycles, techniques such as testabiliy tansformation should continue to see increased interest and application.

8 Acknowledgments

The work summarized in this chapter has been largely conducted as a result of the EPSRC funded project TeTra — Testability Transformation (GR/R98938) and by the project's collaborators. David Binkley is funded, in part by National Science Foundation grant CCR0305330. More details concerning the TeTra project are available on the TeTra website at

http://www.dcs.kcl.ac.uk/staff/linhu/TeTra

References

1. D. G. Altman. *Practical Statistics for Medical Research*. Chapman and Hall, 1997.
2. A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 43–52, Boston, Massachusetts, USA, 2004. ACM.
3. A. Baresel, H. Pohlheim, and S. Sadeghipour. Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724*, pages 2428 – 2441, Chicago, USA, 2003. Springer-Verlag.
4. André Baresel and Harmen 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.

5. André Baresel, Harmen Sthamer, and Michael Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1329–1336, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.
6. David Wendell Binkley and Keith Brian Gallagher. Program slicing. In Marvin Zelkowitz, editor, *Advances in Computing, Volume 43*, pages 1–50. Academic Press, 1996.
7. Leonardo 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.
8. British Standards Institute. BS 7925-1 vocabulary of terms in software testing, 1998.
9. Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.
10. Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.
11. P. David Coward. Symbolic execution systems - a review. *Software Engineering Journal*, 3(6):229–239, November 1988.
12. John Darlington and Rod M. Burstall. A tranformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
13. Richard A DeMillo and A Jefferson Offutt. Experimental results from an automatic test generator. *ACM Transactions of Software Engineering and Methodology*, 2(2):109–127, March 1993.
14. Nachum Dershowitz and Zohar 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.
15. Martin S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, January 1982.
16. Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, January 1996.
17. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
18. Todd Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the 20th International Conference on Software Engineering*, pages 188–197. IEEE Computer Society Press, April 1998.
19. Mark Harman, Lin Hu, Robert Hierons, André Baresel, and Harmen Sthamer. Improving evolutionary testing by flag removal ('best at GECCO' award winner). In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1359–1366, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.
20. Mark Harman, Lin Hu, Robert Mark Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, January 2004.
21. Robert Hierons, Mark Harman, and Chris Fox. Branch-coverage testability transformation for unstructured programs. *The Computer Journal*, 48(4):421–436, 2005.

22. B.F. Jones, H.-H. Sthamer, and D.E. Eyres. Automatic structural testing using genetic algorithms. *The Software Engineering Journal*, 11:299–306, 1996.
23. James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.
24. K. N. King and A. Jefferson Offutt. A FORTRAN language system for mutation-based software testing. *Software Practice and Experience*, 21:686–718, 1991.
25. B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.
26. Bogdan Korel, S. Chung, and P. Apirukvorapinit. Data dependence analysis in automated test generation. In *Proceedings: 7th IASTED International Conference on Software Engineering and Applications*, pages 476–481, 2003.
27. Bogdan Korel, Mark Harman, S. Chung, P. Apirukvorapinit, and R. Gupta. Data dependence based testability transformation in automated test generation. In 16th *International Symposium on Software Reliability Engineering (ISSRE 05)*, pages 245–254, Chicago, Illinios, USA, November 2005.
28. P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.
29. P. McMinn and M. Holcombe. The state problem for evolutionary testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724*, pages 2488–2497, Chicago, USA, 2003. Springer-Verlag.
30. P. McMinn and M. Holcombe. Evolutionary testing of state-based programs. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005)*, pages 1013–1020, Washington DC, USA, 2005. ACM Press, New York.
31. Phil McMinn, David Binkley, and Mark Harman. Testability transformation for efficient automated test data search in the presence of nesting. In *UK Software Testing Workshop (UK Test 2005)*, Sheffield, UK, September 2005.
32. C.C. Michael, G. McGraw, and M.A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, (12):1085–1110, December 2001.
33. 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.
34. A. Jefferson Offutt. An integrated system for automatically generating test data. In Raymond 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, April 1990. IEEE Computer Society Press.
35. 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.
36. Helmut A. Partsch. *The Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer, 1990.
37. Hartmut Pohlheim and Joachim Wegener. Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1795, San Francisco, CA 94104, USA, 13-17 July 1999. Morgan Kaufmann.

38. 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, Los Alamitos, California, USA, 1998. IEEE Computer Society Press.
39. Radio Technical Commission for Aeronautics. RTCA DO178-B Software considerations in airborne systems and equipment certification, 1992.
40. A. Schultz, J. Grefenstette, and K. Jong. Test and evaluation by genetic algorithms. *IEEE Expert*, 8(5):9–14, 1993.
41. 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.
42. Nigel Tracey, John Clark, and Keith 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.
43. Martin Ward. Reverse engineering through formal transformation. *The Computer Journal*, 37(5), 1994.
44. Joachim Wegener, André Baresel, and Harmen 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.
45. Joachim 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.
46. Mark Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.
47. Hong Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering*, 22(4):248–255, 1996.

This article was processed using the LaTeX macro package with LLNCS style