Testability Transformation

Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer,
André Baresel, and Marc Roper, *Member*, *IEEE Computer Society*

Abstract—A testability transformation is a source-to-source transformation that aims to improve the ability of a given test generation method to generate test data for the original program. This paper introduces testability transformation, demonstrating that it differs from traditional transformation, both theoretically and practically, while still allowing many traditional transformation rules to be applied. The paper illustrates the theory of testability transformation with an example application to evolutionary testing. An algorithm for flag removal is defined and results are presented from an empirical study which show how the algorithm improves both the performance of evolutionary test data generation and the adequacy level of the test data so-generated.

Index Terms—Evolutionary testing, search-based software engineering, automated test data generation, transformation.

1 Introduction

A testability transformation is a source-to-source program transformation that seeks to improve the performance of some chosen test data generation technique. This paper introduces a simple theory of testability transformation and illustrates its application, defining a testability transformation algorithm for improving the performance of evolutionary test data generation in the presence of flag variables.

As with traditional program transformation [8], [20], [24], a testability transformation alters the syntax of the original program to which it is applied. However, testability transformations differ from traditional transformations in three important respects:

- The transformation of a program may be accompanied by a corresponding transformation to the test adequacy criterion, so that test data generated for the transformed program (and transformed criterion) will be adequate for the original program and original criterion. By contrast, traditional transformation is applied only to programs.
- 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 adequate test data generation. By contrast, in traditional transformation, it is the original program which is replaced by the transformed equivalent.
- The transformation process need not preserve the traditional meaning of a program. Testability transformation requires new definitions of program equivalence; testability transformations need not preserve functional equivalence. For example, it is shown that branch coverage preservation is not merely an abstract interpretation of functional equivalence, but a distinct meaning in its own right. By contrast, traditional transformation preserves functional equivalence.

The present paper illustrates the application of testability transformation to evolutionary testing for branch coverage in the presence of flags. The choice of evolutionary testing, branch coverage, and flag problems is made purely for illustration. The definitions of testability transformation can be applied to any automated or semi automated test data generation approach (not just evolutionary testing) and transformation can be used to remove any structural impediment to testing (not just flags). The flag removal transformation is also useful for other coverage criteria (not merely branch coverage).

The rest of this paper is organized as follows: Section 2 introduces the theory of testability transformation, while Section 3 describes the novel aspects of this new approach to program transformation. Section 4 explains the "flag problem" for evolutionary test data generation. Section 5 introduces an algorithm which removes flags to address this problem, showing that the transformation is a valid testability transformation according to the theory introduced. Section 6 presents the results of an empirical study[1] concerning the transformation of flag programs. The study indicates that the approach works well in practice and indicates the particular kinds of flag problem which present the greatest (and the least) difficulty for evolutionary test data generation. Section 7 presents related work and Section 8 concludes.

2 Testability Transformation

When presented with problems of programming style, a natural solution is to seek to transform the program to

- *M. Harman, L. Hu, and R. Hierons are with the Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex, UB8 3PH, UK.*
E-mail: {Mark.Harman, Liu.Hu, Rob.Hierons}@brunel.ac.uk.
- *J. Wegener, H. Sthamer, and A. Baresel are with DaimlerChrysler, Alt-Moabit 96a, D-10559 Berlin, Germany.*
- *M. Roper is with Strathclyde University, Livingstone Tower, 26 Richmond St., Glasgow G1 1XH, UK.*

Manuscript received 26 Feb. 2003; revised 7 July 2003; accepted 7 Aug. 2003.
Recommended for acceptance by J. Offutt.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0002-0203.

[1] An earlier version of the empirical results was presented at GECCO 2002 [13].

remove the problem. In the context of improving testability, this will be termed "testability transformation." Testability transformation seeks to transform a program to make it easier to generate test data for it (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, flags, 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 criteria 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 cotransform 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 by the definitions below.

Testing-Oriented Transformation. *Let* $\mathbf{P}$ *be a set of programs and* $\mathbf{C}$ *be a set of test adequacy criteria.*[2] *A program transformation is a partial function in* $\mathbf{P} \rightarrow \mathbf{P}$. *A* Testing-Oriented Transformation *is a partial function in* $(\mathbf{P} \times \mathbf{C}) \rightarrow (\mathbf{P} \times \mathbf{C})$.

The test adequacy criterion, $\mathbf{C}$ is any set of syntactic constructs which are to be covered during testing. For example, a set of nodes, a set of branches, a set of paths, and so on. So, for "100 percent branch coverage," $\mathbf{C}$ would be the set of branches of the program. Observe that the definition also allows more fine grained criteria, such as testing to cover a particular branch or statement.

Testability Transformation. *A Testing-Oriented Transformation,* τ *is a* Testability Transformation *iff, for all programs* p *and criteria* c, *if* $\tau(p, c) = (p', c')$, *then, for all test sets* T, T *is adequate for* p *according to* c *if* T *is adequate for* p' *according to* c'.

The term "test pair" will be used to refer to a program and its associated test adequacy criterion. A testing oriented transformation is any function on test pairs. A testability transformation is a testing oriented transformation for which test data generation from the transformed test pair produces adequate test data for the original test pair. A testability transformation from (p, c) to (p', c') will be referred to as a "c to c' testability transformation."

c-**Preserving Testability Transformation.** *Let* τ *be a testability transformation. If, for some criterion,* c, *for all programs* p, *there exists some program* p' *such that* $\tau(p, c) = (p', c)$, *then* τ *is called a* c*-preserving testability transformation.*

For some criterion, c, a c-preserving testability transformation, if it exists, guarantees that the transformed program is suitable for testing with respect to the original criterion. For many applications, this will be the ideal; it means that the easier-to-test program can simply be used as a replacement for the original in order to generate test data and that the test data so-generated can be used on the original program. The test data generation process need not change, it is simply applied to a more amenable form of the program. Where a c-preserving testability transformation exists, it is therefore likely that this will be preferred, over a non-c-preserving testability transformation.

A testability transformation only guarantees that sufficient test data will be generated to meet the original test adequacy criterion. It does not demand that the test data generated contains no redundant test cases. A test case is redundant if it can be removed without affecting satisfaction of the original test adequacy criterion. A reversible testability transformation produces just enough test cases to meet the original test adequacy criterion and no more.

Reversible Testability Transformation. *A testability transformation,* τ, *is a* Reversible Testability Transformation *iff its inverse is a testability transformation.*

3 Novel Aspects of Testability Transformation

3.1 Cotransformation of the Test Adequacy Criterion

In traditional transformation it is only the program that is transformed. Testability transformation also allows for cotransformation of the test adequacy criterion. Consider the transformation below:

```
while (b1)
{ if (b2) break;    ⇒    while (b1 && !(b2)) s1;
s1; }
```

This transformation preserves the (traditional) meaning of the original, but it causes edges of the Control Flow Graph (CFG) of the original program to disappear. The transformation is therefore not branch coverage preserving. It might therefore be thought inappropriate for branch coverage applications. However, such a transformation might turn out to be desirable because it restructures the program. The transformed program achieves this restructuring at the expense of destroying edges, so branch coverage for the original program will not correspond to branch coverage for the transformed program.

However, there is a relationship between the original and transformed program: Any Modified Decision/Condition Coverage (MC/DC) adequate test set for the transformed program will be guaranteed to be branch adequate for the original. A test set is MC/DC adequate iff it achieves branch coverage and, additionally, for all possible subconditions (b1 and b2 in this case), there is a test case pair which demonstrates that the subcondition independently affects the outcome of the overall predicate evaluation [4]. MC/DC is clearly a "tougher" testing criterion because any test set which is MC/DC adequate clearly is also branch

[2] The precise structure of an adequacy criterion is deliberately left unspecified. Each adequacy criterion might, for example, consist of one or more subcriteria, each of which have to be met.

	b1	b2	b1 && !b2
Test case 1	T	F	T
Test case 2	T	T	F
Test case 3	F	T	F

Fig. 1. Test cases which give branch coverage but not MC/DC.

adequate (covers all branches), but the opposite is not necessarily the case.

In the original program, MC/DC and branch coverage are equivalent. In the transformed program, they are not. In the transformed program, any test set which is MC/DC adequate, will be branch adequate for the original, so the transformation is a Branch-to-MC/DC testability transformation. Notice, however, that the transformation is not a reversible testability transformation. That is, although any MC/DC adequate test set for the transformed program is branch adequate for the original, there are branch adequate test sets of the original which do not give MC/DC coverage of the transformed program. For example, consider a test set which contains three test cases which produce the values at b1 and b2 shown in Fig. 1. These three test cases achieve branch coverage, but not MC/DC coverage.

This illustrates the way in which the flexibility of the definition of testability transformation allows the test data generation "difficulty" to be traded between the program structure and the adequacy criterion. Where the structure of the program prevents test data generation, the problem may be ameliorated by increasing the stringency of the adequacy criterion, while removing the difficulty for the program.

3.2 Disposable Transformed Programs

Traditionally, the goal of transformation has been to produce a program which is improved either for efficiency (for instance, compiler optimization) or is improved for the human reader (for instance restructuring). In all cases, the transformed program is an end in itself. Testability transformation is different. The transformed program is used merely as an input to an automated test data generation process. The transformation guarantees that this process will produce adequate test data according to the original criterion, but, once the test data has been generated, the transformed program can be discarded.

3.3 Nonstandard Meaning Preservation

Program transformation has traditionally been concerned with the preservation of functional equivalence. By contrast, a transformation must provide a wholly different guarantee: Test data generated from the transformed program is adequate for the original program. This means that the transformations applied need not preserve any notion of traditional equivalence, opening up the possibility of defining novel sets of program transformation rules and algorithms. At first sight, this appears to be abstract interpretation; some abstraction of the traditional semantics of the program must surely be preserved during the transformation process? However, testability transformation is not abstract interpretation. A simple example which shows this is the fragment of code below:

```
if (e) skip; else skip;
```

Suppose that e is some side-effect free expression. Using conventional transformation, this would be removed as dead code. However, to do so, would remove two branches and, thus, test data which is adequate for branch coverage of the transformed program would not be adequate for the original program. Therefore, testability transformation is not, in general, more abstract than traditional transformation; not all traditional transformations are testability transformations. This is only the simplest example which illustrates that traditional transformations which preserve functional equivalence need not be valid testability transformations for branch coverage. Other examples of non-branch coverage preserving transformations include the well-known restructuring transformations [21], which reduces an arbitrary unstructured program into a structured program. Such transformations remove goto statements and preserve functional equivalence, but do not preserve branch adequate test sets. Also, not all testability transformations are traditional transformations. As a simple example, consider the transformation:

```
if (e) x=1; else x=2;  ⇒   if (e) x=2; else x=1;
```

Clearly, this is not traditionally meaning preserving; it could hardly be less meaning preserving in the traditional sense, since the original and transformed program differs on the final value of the variable x for every possible initial state. Nonetheless, this transformation would preserve branch adequate test sets. In practice, the authors have found that many traditional transformations are also branch adequacy preserving. For example, the transformation algorithm presented in Section 5 uses *only* traditionally meaning preserving transformation steps. However, this section demonstrates that testability transformation is *not* simply a new application for transformation (to testing) but that the application of transformation to testing establishes a new kind of transformation, with its novel equivalence relations. This novel equivalence relation can be useful in practice. For example, suppose that the goal is to execute some branch and that the program contains a large computation, S, which does not contribute to whether or not the program follows S. In such a situation S is not of interest to the goal in question and can be removed using conditioned slicing [6], [7]. This is important, because automated test data generation techniques such as evolutionary testing, typically require many executions of the test subject.

4 The Flag Problem for Evolutionary Test Data Generation

Evolutionary testing is based upon evolutionary computation techniques [14], [15], [18]. These techniques are search-based approaches to optimisation problems. They are typically used where the search-space is large and where the value of a candidate solution can be determined by a "fitness" function, which gives higher values to better solutions. In test data generation, the fitness function is determined by the coverage achieved and an individual in the search space is a single test input. The search seeks to find a test input to achieve a chosen test goal. A test goal is the execution of an edge of interest. In this way, by repeatedly performing evolutionary searches, test data can be generated to cover all edges of the program. A similar approach can be used to achieve other forms of test goal, but the present paper will focus on branch testing of which the test goals consist of edges to be targeted.

In order to automate software test data generation with the aid of evolutionary algorithms, the test goal must itself be formulated as an optimization task. A numeric representation of the test goal is formulated, from which a suitable fitness function for evaluation of the generated test data can be derived. Depending on which test goal is pursued, different fitness functions emerge for test data evaluation. For structural testing, the fitness functions are typically based on the computation of a distance for each individual test case [16], [17], [19], [22], [25], indicating how far away it is from executing the program predicate in the desired way.

The approach to forming a fitness function typically involves a global measure and a local measure. The global measure assesses how close execution comes to reaching the predicate which controls the branch of interest. The local measure assesses how close the predicate comes to evaluating to either true or to false (depending upon whether the target node is controlled by the true or false edge emerging from the predicate).

To define the local fitness for branch coverage, the predicate controlling a branch of interest is used. For example, if a branch condition `x==y` needs to be evaluated to `true`, then the fitness function[3] may be defined as $\texttt{MAX}-|\texttt{x}-\texttt{y}|$, for some suitable value of `MAX`. Each individual of the population represents a test datum with which the test object is executed. For each test datum the execution is monitored and the fitness value is determined for the corresponding individual. This approach is the standard approach to computation of fitness in structural evolutionary testing work.

Evolutionary testing has been shown to be an effective way of automatically generating test data for white box (or structural) test adequacy criteria for programs which contain no flag variables [19], [22], [25]. However, for programs with flag variables, the performance of evolutionary testing can be severely diminished.

A flag variable is one whose value is either `true` or `false`. Flags typically "flag" the presence of some special condition of interest. Embedded systems, such as engine controllers, often make extensive use of flag variables to record state information concerning the devices controlled. 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 a value in S. This problem typically occurs with internal flag variables, where the input state space is reduced, with relatively few values (those in S) being mapped to one of the two possible outcomes and all the other (those not in S) being mapped to the other of the two possible flag values.

Such systems can be hard to test to high levels of coverage using evolutionary testing approaches to automated test data generation. This is a serious problem since generating such test data by hand is prohibitively expensive. Also, since flags typically test for conditions which represent exceptions, they are likely to exhibit the state-squashing effect, where relatively few input values correspond to one of the two flag outcomes.

The presence of flag variables creates a coarse fitness landscape, within which the search takes place. This greatly reduces the effectiveness of the search. That is, the fitness landscape consists of two plateaus, corresponding to the two possible flag values. One of these plateaus will be superfit and the other superunfit. A search-based approach, such as evolutionary testing, will not be able to locate the superfit plateau any better than a random search because the fitness landscape provides no guide to direct the search from unfit to fit regions of the landscape. Where the fit plateau may be very small relative to the unfit plateau, this makes the program hard to test. Such "small plateaux" are comparatively common since flags often test for some form of "special" or "unusual" condition, for which special behavior is required.

The problem is illustrated in Fig. 2. Suppose that `A` is some variable whose values are in the range 0 to 20 (the problem has been simplified, without loss of generality, for ease of exposition). The left-hand section of the figure shows the fitness landscape for a predicate which tests to see if the value of `A` is 10. The right-hand section shows a program which performs the same test using a flag variable.

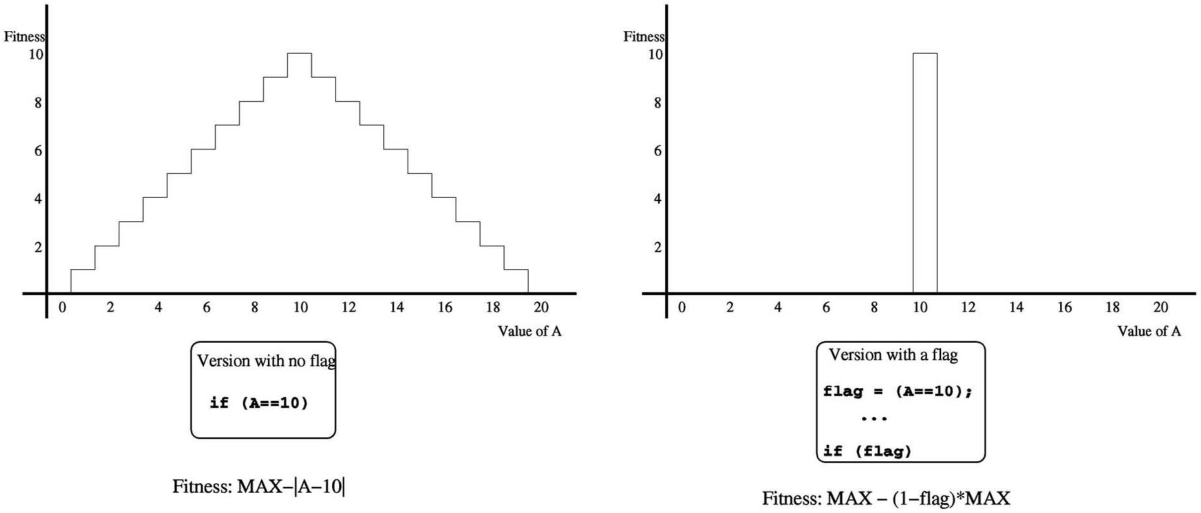
The fitness landscape induced by the flag-free predicate is smooth and there is a clear "guide" from areas of low fitness to those with higher fitness. By contrast, the landscape for the version with a flag, produces a fitness function which yields either maximal fitness for the "special value" 10 for A or minimal fitness for any other value. There is no guide from lower fitness to higher fitness. The flag fitness landscape thus has the characteristic "needle in a haystack" form which is known to present problems for search-based approaches because any search degenerates into a random search. Moreover, random search for such a needle in a haystack is unlikely to succeed.

5 A Testability Transformation Solution to the Flag Problem

Five levels of increasing difficulty of flag problem are summarized in Fig. 3. The term $\mathbf{REF}(e)$ denotes the referenced [1] variables of e, where e is either an expression or a statement. Each level represents an increasing level of effort required to transform the flag-using program into a flag-free program. The effort level is defined with respect to the flag variable concerned (denoted *flag*) and the predicate in which it is used (denoted p). Interleaved assignments and uses of *flag* will be split up into "define" and "use" sections, where all defines precede uses.

This section develops algorithms for transforming flags out of programs for levels of problem difficulty one through to four inclusive. Level five is not handled by the algorithm, neither does it take account of concurrency. The effect of the program transformation on the search landscape is to transform it from the form depicted on the

[3] In this exposition, the goal is to maximize fitness. In much work on Evolutionary Testing, the goal is to minimize fitness, but this is simply an orthogonal view.



```
if (A==10)
```

```
flag = (A==10);
    ...
if (flag)
```

Fig. 2. The flag landscape.

right-hand side of Fig. 2 to the smoother landscape on the left-hand side of Fig. 2.

The "define code" section is the portion of code which assigns (possibly in several places) a value to the flag variable. This section of code starts with the first reaching assignment to the flag variable and finishes with the last reaching assignment to it before the associated use. If an assignment to flag is contained within a structured construct (such as an `if` or `while` statement), then the first (respectively, last) assignment is taken to be the first (respectively, last) statement of the outermost enclosing construct.

The "use" section is the predicate use of the flag variable which the assignments of the define section may reach. The algorithm developed here is capable of removing flags, provided that either the define section is not in a loop-body, or it is in the same loop-body as the use section.

The goal of flag variable removal is thus to transform programs with higher levels of flag variable use to lower forms (and ultimately to level 0, where the flag problem is not present). The following subsections consider each level of difficultly in turn, showing how the flag problem at that level can be reduced to a lower level by transformation.

5.1 Flag Removal at Level 0

Level 0 represents a program free from flag use and, therefore, no (further) transformation is required at level 0.

5.2 Flag Removal at Level 1

Level 1 is trivial to deal with. Because there is only a single assignment, a, to *flag* prior to the use of *flag* at p the right-hand side of a can be substituted for the use of *flag* at p. This substitution will always be possible due to the lack of intervening assignments to $REF(a)$ between a and p.

5.3 Flag Removal at Level 2

At level 2, the simple substitution of the right-hand side of the expression at a for the use at p is not possible because the meaning of the expression at a (potentially) is changed at each intervening assignment to a variable in $REF(a)$. To overcome this problem, the expression at a will be rewritten using temporary variables, to store the values of variables assigned to between a and p. The introduction of temporary variables is illustrated in Fig. 4.

5.4 Flag Removal at Level 3

The problem of producing a single assignment to *flag* from a sequence of assignments, which may contain many assignments to both *flag* and to other variables can be achieved using Amorphous Slicing [3], [12], which is guaranteed to produce a single assignment to *flag* from a straight line of assignments. An amorphous slice of a program, or program fragment, constructed with respect to a variable v, is a simplified program which preserves the effect of the original program on the final value of v.

Level 0	No use of *flag* at p
Level 1	*flag* used at p. Define section is a single prior assignment, a to the variable, *flag*. No assignments to $\mathbf{REF}(a)$ on any path between a and p
Level 2	Level 1 together with additional assignments to $\mathbf{REF}(a)$ between a and p.
Level 3	Level 2 but define section is a sequence of assignments. Some of these assignments are to *flag* and some (possibly) to other variables.
Level 4	Level 3 but define section is a (possibly singleton) sequence containing conditionals.
Level 5	The define section is in a different loop-body to the use section.

Fig. 3. Levels of flag problem for flag variable *flag* use at predicate p.

```
flag = (x>0) || ((y==x) && z<y) ;
...
x=2; y=y+1; /*no assignments to z */
...
if (flag) ...
```

(a)

```
Tempx = x; Tempy = y;
flag = (Tempx>0) || ((Tempy==Tempx) && z<Tempy) ;
...
x=2; y=y+1;
...
if (flag) ...
```

(b)

Fig. 4. Temporary variable introduction.

```
x= y+1;
y= x*2;
flag= x>y;
y= y + flag;
flag= flag || y==0;
x= y*x;
if (flag) ...
```

(a)

```
flag =
(y+1)>((y+1)*2) ||
(((y+1)*2)+(y+1)>((y+1)*2))==0;
```

(b)

```
T=y;
x = y+1 ;
y = x*2 ;
flag = x>y ;
y = y + flag ;
flag = flag || y ==0 ;
x = y*x;
if ((T+1)>((T+1)*2) ||
    (((T+1)*2)+(T+1)>((T+1)*2))==0)
```

(c)

Fig. 5. Amorphous slicing to produce a single assignment to the flag variable.

Step 1	Bush define code
Step 2	Blossom define code
Step 3	Use amorphous slicing to produce a single assignment to *flag* at each leaf.
Step 4	Convert define code to a conditional expression assignment to *flag*
Step 5	Simplify the expression assigned to *flag*
Step 6	Reduce from level 3 to level 2 using the algorithm of Section 5.4
Step 7	Reduce from level 2 to level 1 and 0 using the algorithms of Sections 5.3 and 5.2

Fig. 6. Algorithm for transforming level 4 into level 0.

For example, consider the code fragment in Fig. 5a. This is a straight-line sequence of assignments to the variables flag, x, and y. Amorphous slicing on the variable flag yields the single assignment in Fig. 5b. By the definition of an amorphous slice, this is an assignment which has the same meaning on the variable flag as the code in Fig. 5a. The right-hand side of this single assignment is an expression that captures the value of flag at the end of the define code. By replacing references to y in this expression, by the temporary variable T, the expression can be substituted for the use at the if statement, as depicted in Fig. 5c.

5.5 Flag Removal at Level 4

The algorithm for transforming from level 4 to level 0 is presented in Fig. 6. The essential idea is to "collect" the disparate assignments together and merge them into a single assignment statement. The effect is the same as the transformation for level 3. However, in the case of level four, the single assignment to the flag variable may require a conditional expression. The overall effect of transformation from level 4 to 0 is to produce a (possibly) conditional expression which captures the value stored in flag by the define code and to add sufficient assignments to allow this expression to be substituted for the use of the flag variable.

The production of the conditional expression from the define code is facilitated by two auxiliary transformations (called "bushing" and "blossoming" because of their effect on the tree-like structure of the program's CFG). The remainder of this section describes these two steps.

Bushing takes a loop free segment of code and produces a transformed program whose abstract syntax tree is of the form of a binary tree, in which edges may contain sequences of assignments. Blossoming takes such a binary tree and forces all the assignments to the leaves. The combined effect of bushing followed by blossoming is that a binary tree is produced with all assignments at the leaves. The internal nodes are predicates. Amorphous slicing performed on each of the leaves gives a single assignment at the leaf. Such a binary tree can be converted to a conditional expression. Bushing is illustrated in Fig. 7. The square-edged rectangles denote arbitrary assignments, while the round-edged rectangles denote arbitrary predicates.

The "then fold" transformation (Axiom 1 of Fig. 14) converts an if-then construct into an if-then-else construct, by copying the code that follows the original if-then into both the then and else branches. Bushing is achieved by repeatedly applying the "then fold" transformation until there are no if-then statements without corresponding else statements.

A bushed tree is further transformed by blossoming. A blossomed tree has the property that all assignment statements and only assignment statements are at the leaves of the tree and all internal nodes are predicates. Blossoming is illustrated in Fig. 8. Alphabetic letters are

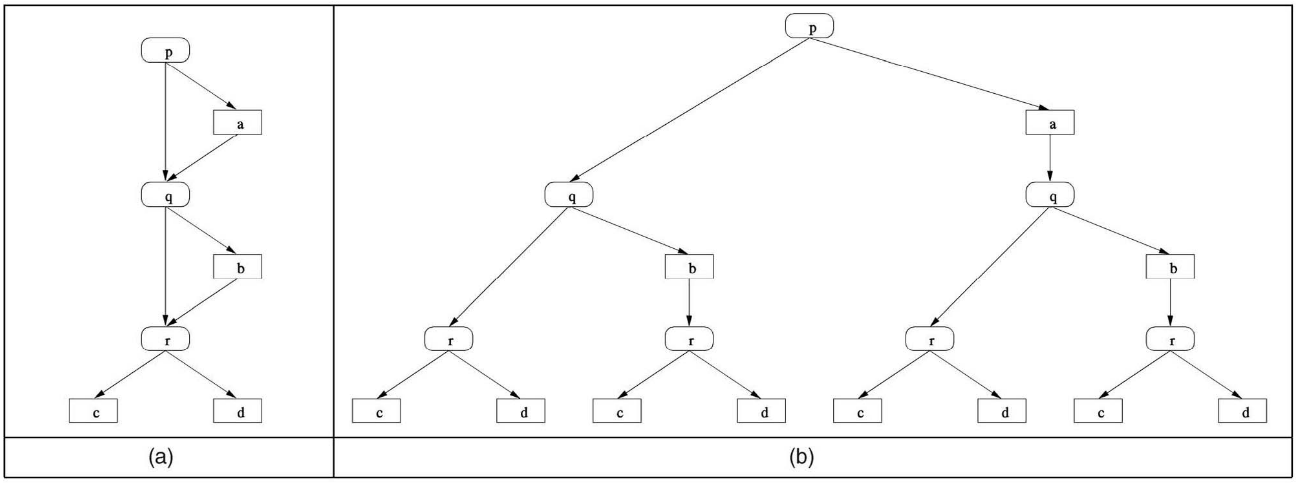


Fig. 7. Bushing a CFG.

used to label statements and predicates. Blossoming is achieved by a post order traversal of a bushed tree, applying Rule 5 of Fig. 14 at each assignment statement until it fails to apply (i.e., when there are no further predicates for the assignment to pass through). This approach is only applicable if expressions are side-effect free. In cases where expressions are not side effect free, then side-effect removal transformation [10] should be considered.

The leaves of the bushed and blossomed tree are sequences of assignments, which can be converted to single assignments to the flag variable using the algorithm for removing flags at level 3. Having done this, the tree is now a binary tree with single assignments to the flag variable at the leaves. Converting this to a conditional expression is straightforward.

A worked example, illustrating bushing and blossoming and how they facilitate transformation from level 4 to level 0 is given in Fig. 9. The figure shows the progress of transformation from top left to bottom right. The defined code (the initial three lines of code) are bushed and blossomed. Slicing the result using amorphous slicing on the variable `flag` produces a single assignment. Temporary variables, `Ta` and `Tb` are added to allow substitution of the expression assigned to `flag` by this single assignment. At the bottom of the figure, the effect of such a substitution is shown for a five line sequence of code. The first three lines are the define code and the last line is the use of `flag`, for which substitution is required. The fourth line is included merely to show that intervening assignments to relevant variables between define and use can be handled by the algorithm.

5.6 The Algorithm for Flag Removal is a Branch Coverage-Preserving Testability Transformation

The algorithm for removing flags involves several transformation steps which serve to produce a (possibly simplified) conditional expression which can be assigned to the flag variable. Each of these steps is traditionally meaning

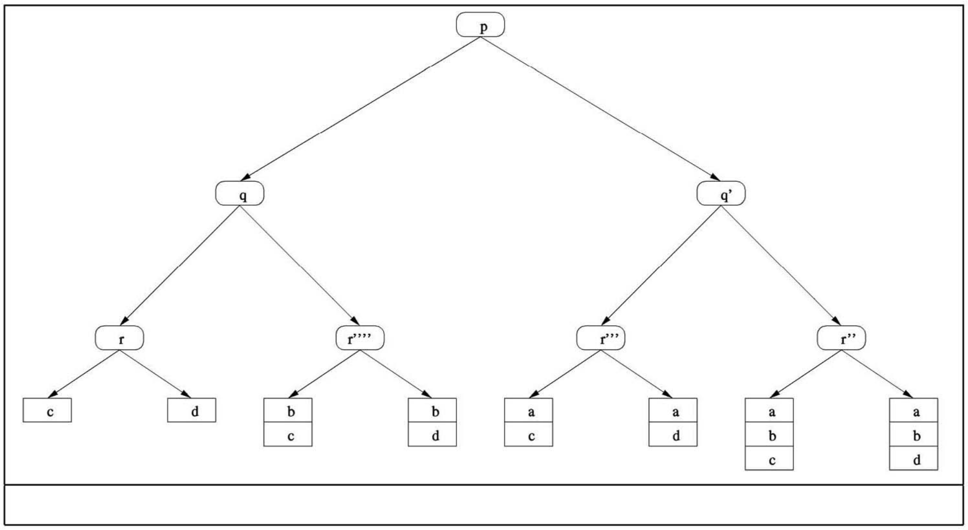
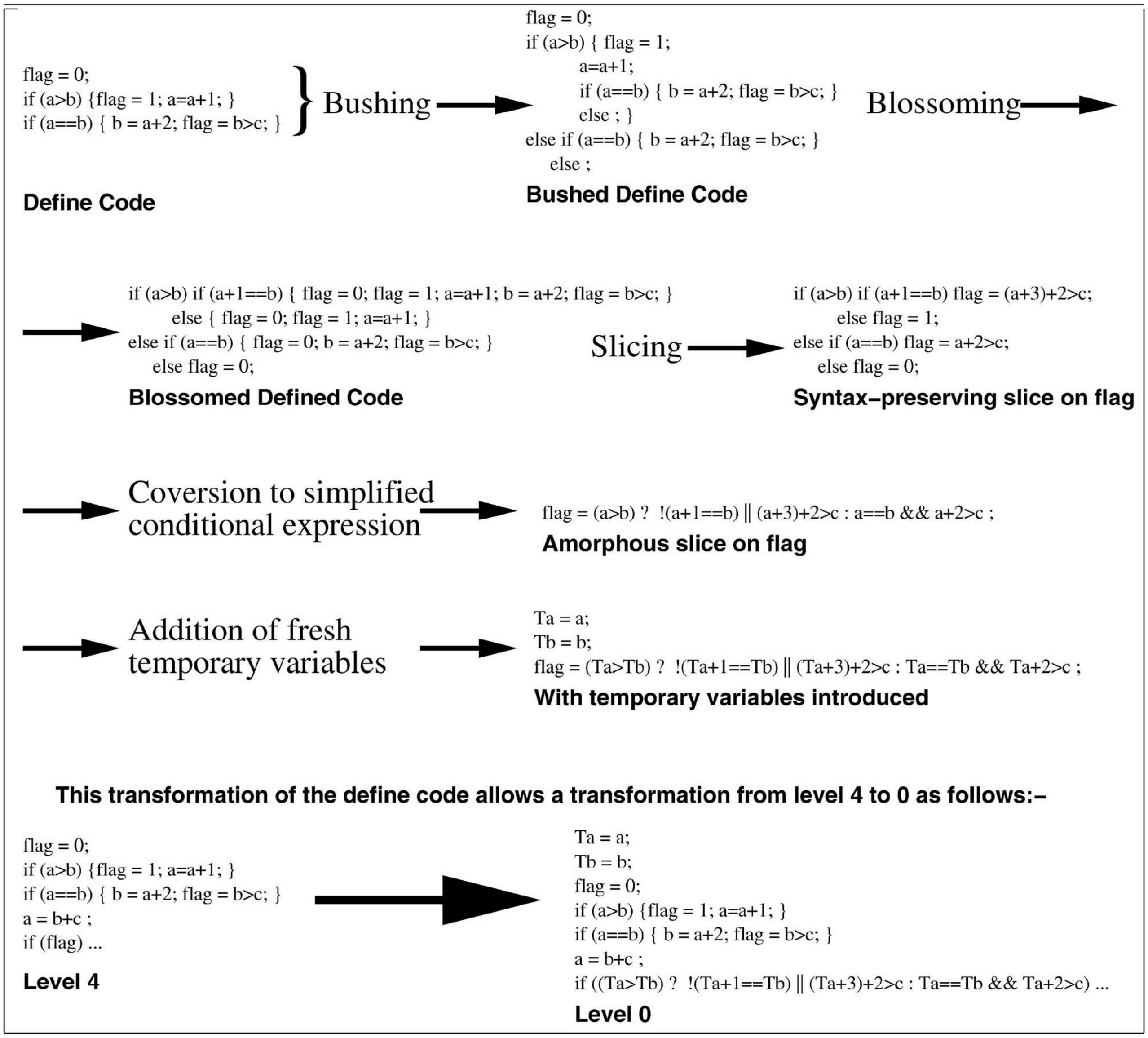


Fig. 8. Blossoming a CFG.



```
flag = 0;
if (a>b) {flag = 1; a=a+1; }
if (a==b) { b = a+2; flag = b>c; }
```

```
flag = 0;
if (a>b) { flag = 1;
        a=a+1;
        if (a==b) { b = a+2; flag = b>c; }
        else ; }
else if (a==b) { b = a+2; flag = b>c; }
    else ;
```

```
if (a>b) if (a+1==b) { flag = 0; flag = 1; a=a+1; b = a+2; flag = b>c; }
        else { flag = 0; flag = 1; a=a+1; }
else if (a==b) { flag = 0; b = a+2; flag = b>c; }
    else flag = 0;
```

```
if (a>b) if (a+1==b) flag = (a+3)+2>c;
        else flag = 1;
else if (a==b) flag = a+2>c;
    else flag = 0;
```

```
flag = (a>b) ?  !(a+1==b) || (a+3)+2>c : a==b && a+2>c ;
```

```
Ta = a;
Tb = b;
flag = (Ta>Tb) ?  !(Ta+1==Tb) || (Ta+3)+2>c : Ta==Tb && Ta+2>c ;
```

With temporary variables introduced
This transformation of the define code allows a transformation from level 4 to 0 as follows:–

```
flag = 0;
if (a>b) {flag = 1; a=a+1; }
if (a==b) { b = a+2; flag = b>c; }
a = b+c ;
if (flag) ...
```

```
Ta = a;
Tb = b;
flag = 0;
if (a>b) {flag = 1; a=a+1; }
if (a==b) { b = a+2; flag = b>c; }
a = b+c ;
if ((Ta>Tb) ?  !(Ta+1==Tb) || (Ta+3)+2>c : Ta==Tb && Ta+2>c) ...
```

Fig. 9. A worked example illustrating the flag removal algorithm.

preserving.[4] The overall effect upon the original program, however, is relatively minor: the assignment to the flag variable produced by amorphous slicing and bushing and blossoming is employed simply to obtain an expression (possibly a rather large and conditional one) which can then be substituted for the use of the flag variable.

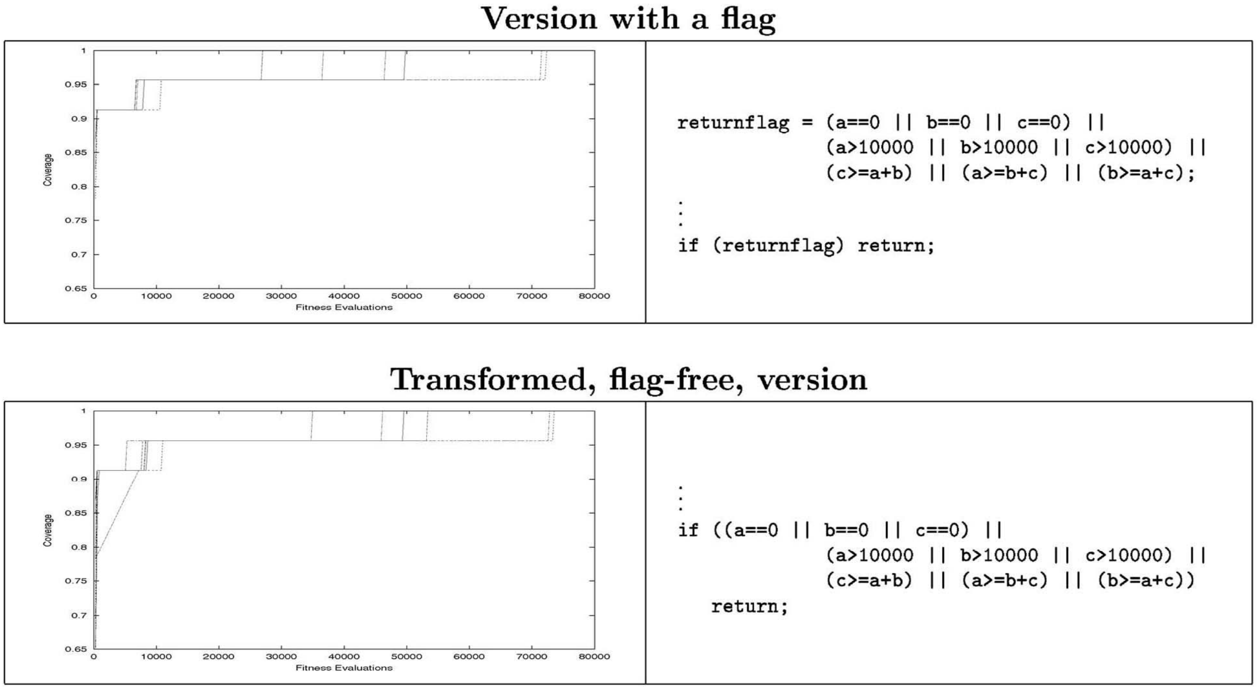
The ability to substitute "definition" for "use," relies upon the introduction of assignments to temporary variables. The algorithm retains the original flag assignment code. Therefore, the only modification is the substitution of "definition" for "use" of the flag variable itself and the associated insertion of assignments to temporary variables.

Substitution for an expression clearly has no effect upon the edges of the CFG and, thus, has no effect upon branch coverage adequate test sets. Therefore, the only price to pay, in terms of modification of the CFG, for the flag removal transformation is the addition of the straight line sequence of temporary variable assignments.

A straight-line sequence of assignments *does* add additional edges to the CFG of the program. However, these edges will not affect branch coverage adequate test sets. This is because the straight line sequence of assignments is inserted between two existing sequential statements, x and y. There must, by definition, be an edge in the CFG from x to y. The insertion of the assignment sequence introduces a path from x to y in place of this edge. However, the path will be traversed under precisely the same conditions as the original single edge from x to y.

Furthermore, as the assignments are to new temporary variables, the projection of the state onto the original program variables remains unchanged along this path. The state at node y in the transformed and original program will therefore always agree on all variables in the original

[4] A formal proof of this is beyond the scope of the present paper, but the reader can easily check, informally, that each step is reasonable, given the constraints for transformation, that predicates are side-effect free, that loops terminate and that other forms of abnormal termination do not occur.


Version with a flag

```
returnflag = (a==0 || b==0 || c==0) ||
             (a>10000 || b>10000 || c>10000) ||
             (c>=a+b) || (a>=b+c) || (b>=a+c);
:
:
if (returnflag) return;
```

Transformed, flag-free, version

```
:
:
if ((a==0 || b==0 || c==0) ||
             (a>10000 || b>10000 || c>10000) ||
             (c>=a+b) || (a>=b+c) || (b>=a+c))
   return;
```

Fig. 10. Results for the "Validity Check" Triangle program.

program, ensuring that the two program versions follow identical branches from statement y onward.

Since the transformation is traditionally meaning preserving and also introduces no additional branches, it is therefore possible to say that it is a branch preserving testability transformation. Indeed, it is possible to go further. The transformation is a reversible testability transformation because test data generated for the original program will also be branch adequate for the transformed program.

6 Empirical Evaluation of Flag Removal

The DaimlerChrysler Evolutionary Testing system [25] was used to generate test data for flag-based programs and these results were compared with those obtained from running the testing system with identical parameters on the transformed, flag-free versions of the programs. A full description of the system is beyond the scope of this paper. The reader is referred to other papers by the DaimlerChrysler team which describe their approach to evolutionary testing [2], [25].

This section presents four indicative experiments, which illustrate various incarnations of the flag problem and the effect of their removal upon evolutionary test data generation. The figures show the results obtained on the left-hand side together with the relevant fragments of the corresponding programs on the right-hand side. The program fragments are shown to illustrate the particular flavor of flag problem considered. However, when using the system, the human need not be aware either of the flag-free version of the program, nor indeed of the evolutionary process itself. The user simply submits a program (possibly with flags) and obtains a set of branch adequate test data automatically.

The results plot the coverage achieved (for six separate executions of the evolutionary testing system) against the number of fitness evaluations. They are therefore a measure of effectiveness against effort.

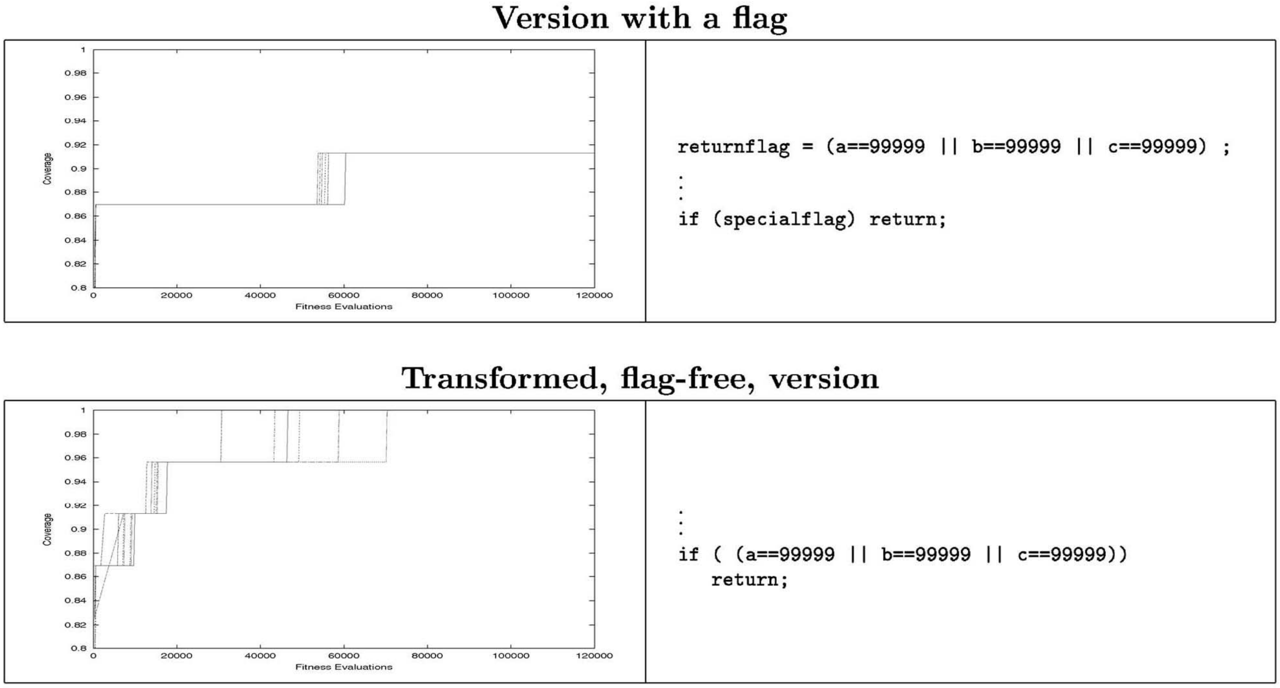
A test goal consists of attempting to optimize test data to cover a particular branch. The coverage for each trial therefore increases in steps, as each test goal is satisfied. In all examples, a test set which achieves full branch coverage exists (there are no infeasible branches).

6.1 Triangle Classification Program

The triangle classification program is widely used as a benchmark in software testing. The program has three variables (a, b, and c), which represent the side lengths of a figure. The goal of the program is to determine whether the three side lengths represent a triangle, and if they do, to categorize the triangle type.

Input values are double values within range -1,000 to 20,000 with a precision of 0.00001. This gives a search space size of approximately 10^{27}. Two versions of the triangle classification program were used in the experiment: a "Validity check" program and a "Special Value" program. These two variants of the triangle program illustrate the range of difficulty introduced by flags from none (Validity Check) through to severe (Special Value). The results for each variant are shown in Figs. 10 and 11.

In the "Validity Check" variant, the flag is assigned a value which represents a set of validity checks on inputs. There are many subcriteria (Boolean terms), many inputs which satisfy each subcriterion and many which fail to satisfy each. Therefore, the fitness landscape does not contain a *small* high fitness plateau. Furthermore, each of the subcriteria is also checked later on in the program by a separate conditional statement and so each subcriterion also



Version with a flag

```
returnflag = (a==99999 || b==99999 || c==99999) ;
:
:
if (specialflag) return;
```

Transformed, flag-free, version

```
:
:
if ( (a==99999 || b==99999 || c==99999))
   return;
```

Fig. 11. Results for the “Special Value Check” Triangle program.

forms a separate test goal. In this situation the presence of flags presents no difficulty.

By contrast the “Special Value” variant of the triangle program represents a difficult form of flag-based program. The flag variable is set to true by only very few inputs, creating a tiny plateau of high fitness. In such a situation, evolutionary testing degenerates to random testing.

The results show that for the “Validity Check” version of the program, the removal of flags makes practically no difference, with all trials reaching maximum fitness, and with all doing so with a similar spread of effort. On the other hand, the “Special Value” variant shows how bad the flag problem can be. After 40,000 fitness evaluations, none of the trial runs has risen above a coverage of 0.87 and after 120,000 evaluations none has risen above 0.92. No trial reached the maximum possible coverage. However, for the flag free version, after only 20,000 fitness evaluations, *all* of the trial runs have reached a coverage of more than 0.96 and after only 80,000 evaluations all have reached maximum possible coverage (1.0).

6.2 Calendar Program

The calendar program computes dates, but takes account of special days and date corrections which have taken place as a result of the British parliament’s “Calendar Act” of 1751 [5]. This act required that the date of 2 September, the following year was to be immediately followed by 14 September, a decision which caused much consternation and a demand for the return of the “stolen 11 days.” These “stolen days” form a special case in the calendar program which is denoted by a flag.

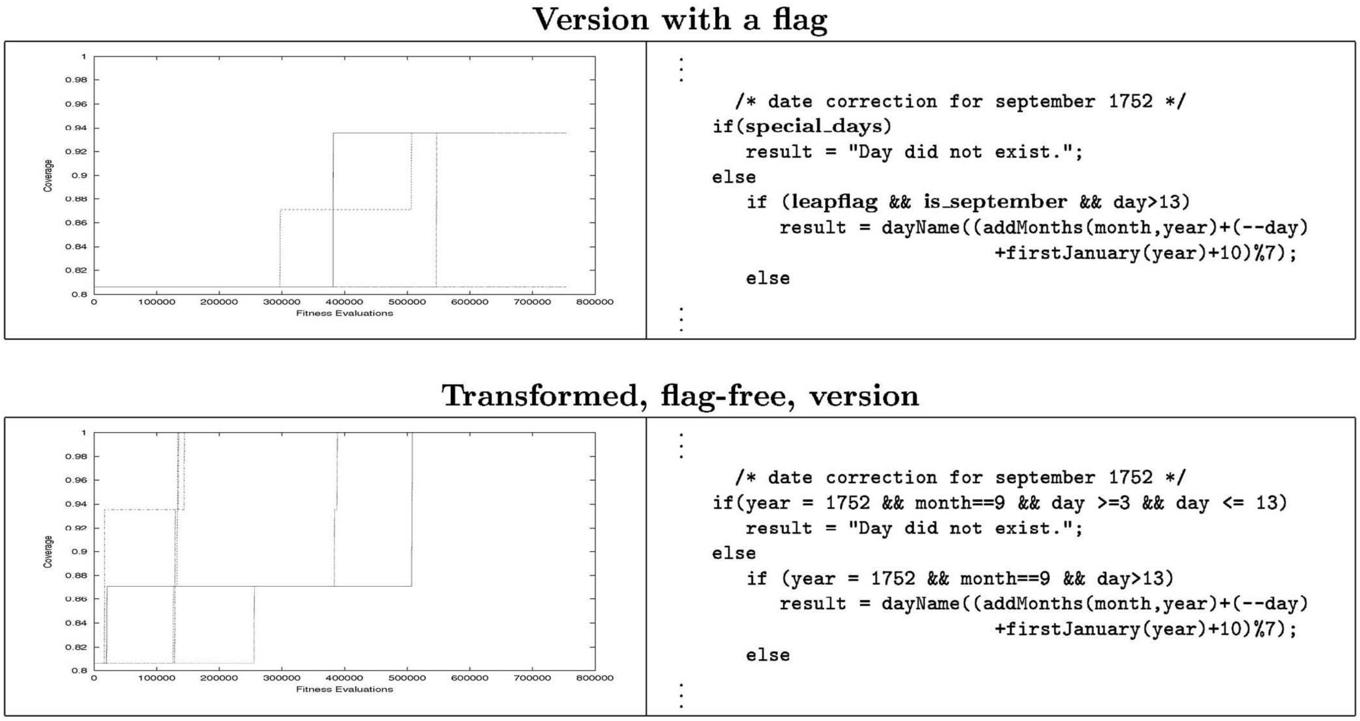
The calendar program is a typical flag-based program which tests for an “unusual” condition and sets the value of a flag according to this test. This is typical because flags often test for exceptional cases. That is, the value assigned is far more likely to take one of the two possible values than the other. In this case, the program takes eight character variables, each of which may take values within a range of 0 to 10. These are subsequently converted into the integer variables `day`, `month`, and `year` seen in the code-fragment quoted in Fig. 12. This gives a search space of approximately 8^{10}, with the flag representing any date within the “11 stolen day” period.

The results of evolutionary test data generation for the calendar program, together with the relevant fragments of code are shown in Fig. 12. The flag-free code has been simplified for readability. The actual transformed program produced by the flag-removal algorithm contains many temporary variables. Of course, the fact that the transformed version has poor readability is not an issue for this work (unlike most work on transformation) because the transformed program is not read by a human.

The results show that flag removal helps in this instance because all of the runs achieve the maximum possible fitness using the flag-free version, while none does so using the original program. It can also be seen from the growth of coverage for each run, that using the flag-free version both achieves higher coverage overall and achieves each level of coverage faster.

6.3 Line Covered by Rectangle Program

The `LineCoveredbyRectangle` program attempts to work out whether a line segment is completely covered by a rectangle figure or not. The program has eight input variables of short integer type within a range of -32,000 to +32,000. The total search space size is therefore $64,000^8$.



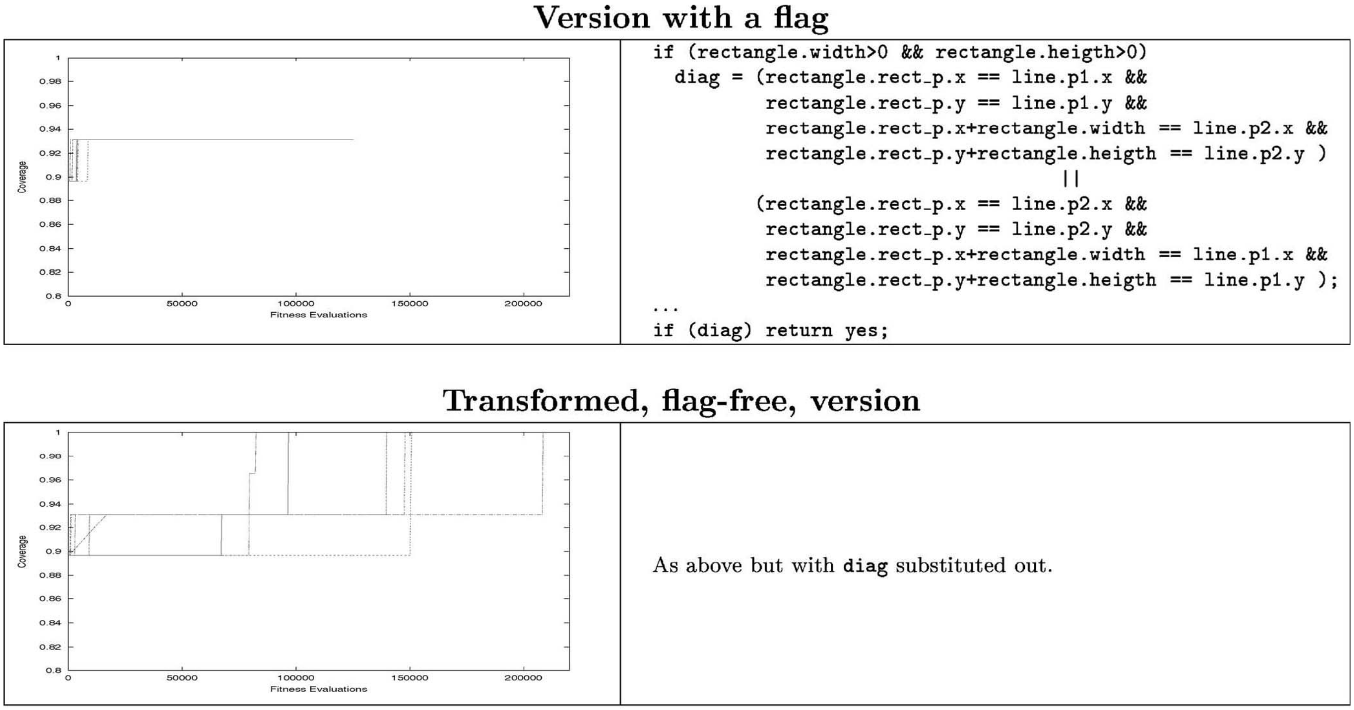
Version with a flag

```
⋮
    /* date correction for september 1752 */
  if(special_days)
     result = "Day did not exist.";
  else
     if (leapflag && is_september && day>13)
        result = dayName((addMonths(month,year)+(--day)
                          +firstJanuary(year)+10)%7);
     else
⋮
```

Transformed, flag-free, version

```
⋮
    /* date correction for september 1752 */
  if(year = 1752 && month==9 && day >=3 && day <= 13)
     result = "Day did not exist.";
  else
     if (year = 1752 && month==9 && day>13)
        result = dayName((addMonths(month,year)+(--day)
                          +firstJanuary(year)+10)%7);
     else
⋮
```

Fig. 12. Results for the leap year program.



Version with a flag

```
if (rectangle.width>0 && rectangle.heigth>0)
  diag = (rectangle.rect_p.x == line.p1.x &&
          rectangle.rect_p.y == line.p1.y &&
          rectangle.rect_p.x+rectangle.width == line.p2.x &&
          rectangle.rect_p.y+rectangle.heigth == line.p2.y )
                                ||
         (rectangle.rect_p.x == line.p2.x &&
          rectangle.rect_p.y == line.p2.y &&
          rectangle.rect_p.x+rectangle.width == line.p1.x &&
          rectangle.rect_p.y+rectangle.heigth == line.p1.y );
...
if (diag) return yes;
```

Transformed, flag-free, version

As above but with **diag** substituted out.

Fig. 13. Results for the line covered by the Rectangle program.

There is a special case when the line in question happens to be a diagonal of the rectangle. This is true for $32,000^4$ of the $64,000^8$ possible inputs. This means that the chance of stumbling across such a value "at random" is approximately one in 10^{20}.

The results of evolutionary test data generation for the Rectangle program, together with the relevant section of flag-containing code, are shown in Fig. 13. The results show that flag removal produces better results. In particular, observe that the final test goal (of moving from just under 0.94 coverage to the maximum possible coverage of 1.0) is abandoned in all of the trials of the version with flags and 100 percent coverage is always achieved for flag free version after 200K fitness evaluations.

Rule 1 (Push Assignment)

$$\frac{i_1 \neq i_2, i_2 \notin \mathbf{REF}(e_1), e_3 = \mathbf{SUB}(e_2, i_1, e_1)}{[\![i_1 = e_1;\ \ i_2 = e_2;]\!] \Rightarrow [\![i_2 = e_3;\ \ i_1 = e_1;]\!]}$$

Rule 2 (Unfold and Merge Assignment)

$$\frac{e_3 = \mathbf{SUB}(e_2, i, e_1)}{[\![i = e_1;\ \ i = e_2;]\!] \Rightarrow [\![i = e_3;]\!]}$$

Rule 3 (Push If)

$$\frac{e_2' = \mathbf{SUB}(e_2, i, e_1), [\![i{=}e_1;\ c]\!] \Rightarrow [\![c'\ \ i{=}e_1;]\!]}{[\![i{=}e_1;\ \texttt{if}\ \ (e_2)\ \ c]\!] \Rightarrow [\![\texttt{if}\ \ (e_2')\ \ c'\ \ i{=}e_1;]\!]}$$

Rule 4 (Push If-Else)

$$\frac{e_2' = \mathbf{SUB}(e_2, i, e_1),\ \ [\![i{=}e_1;\ \ c_1]\!] \Rightarrow [\![c_1'\ \ i{=}e_1;]\!],\ \ [\![i{=}e_1;\ \ c_2]\!] \Rightarrow [\![c_2'\ \ i{=}e_1;]\!]}{[\![i{=}e_1;\ \ \texttt{if}\ \ (e_2)\ \ c_1\ \ \texttt{else}\ \ c_2]\!] \Rightarrow [\![\texttt{if}\ \ (e_2')\ \ c_1'\ \ \texttt{else}\ \ c_2'\ \ i{=}e_1;\ \ ]\!]}$$

Rule 5 (Push into if)

$$\frac{p' = \mathbf{SUB}(p, i, e)}{[\![i = e; \texttt{if}\,(p)\ S\ \texttt{else}\ S']\!] \Rightarrow [\![\texttt{if}\,(p')\ i = e; S\ \texttt{else}\ i = e; S']\!]}$$

Axiom 1 (Then fold) $[\![\texttt{if}\,(p)\ S_1 \qquad S_2\ ]\!] \Rightarrow [\![\texttt{if}\,(p)\ \{S_1 S_2\}\ \texttt{else}\ S_2]\!]$

Fig. 14. Transformation rules.

7 Related Work

Testability has been defined by Voas and Miller [23], in terms of the Propagation, Infection, and Execution (PIE) framework. The PIE method measures testability in terms of the likelihood that an infection (a fault) is executed and subsequently propagated in an observable way. In this paper, the term testability is construed in the more general sense to mean "the ease with which test data can be generated." The flag problem for evolutionary testing can be thought of as an example of the high Domain to Range Ratio (DRR) problem which Voas 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. Voas observed that such situations (with high DRRs) could lead to fault masking, preventing the propagation component of the PIE framework. However, a high DRR is an insufficient requirement to cause a problem for evolutionary test data generation, as shown by the empirical study with flag variable; the problem only arises when the high DRR also has a range value for which there are very few corresponding domain elements. In search-based nomenclature, this corresponds to the "needle in a haystack" problem.

The transformations considered in this paper need not preserve functional equivalence. This is a departure from most work on program transformation, but it is not the first instance of nontraditional meaning preserving transformation in the literature. Weiser [26] introduced program slicing. A slice for a set of variables V and program point n consists of those parts of the program which potentially influence the values computed for variables in V at n. Those parts which can be statically determined to have no such effect are removed to form the slice. A slice is thus a syntactic projection of the original program that preserves a semantic projection of the program's semantics. As such, a slice can be thought of preserving an abstract interpretation of the program from which it is constructed.

Similarly, Dershowitz and Manna [9] and Feather [11] considered nonmeaning program transformations for program modification in corrective and adaptive maintenance. In these "evolution transforms," the idea is that the software is to be evolved using transformation and that this evolution necessarily does not preserve the (complete) meaning of the program. The semantics of the program are divided into several orthogonal components, or dimensions, allowing changes to semantics which affect one dimension, while leaving the others invariant. In this way, an evolution transform is a higher-level version of a slice, in which the slicing criterion is not a set of variables and a program point, but a set of dimensions which are to remain invariant.

The three novel aspects of testability transformation (with respect to traditional transformation) identified in Section 3 are helpful to clarify the similarities and differences between testability transformation and these other two variations on traditional transformation of slicing and evolution transforms.

- Testability transformations allow cotransformation of the test adequacy criterion. Unlike testability transformation, slicing and evolutionary transformations are not concerned with testing and, so, clearly have no such requirement.

- Testability transformations are disposable. Unlike testability transformation, evolution transforms produce results which are intended to act as replacements for the original program from which they are constructed. By contrast, a testability transformation is merely a means to an end; once the test data is generated from the transformed program, it is discarded. Like testability transformation, slicing is often used as a means to an end, rather than an end in itself.
- Testability transformations are not required to preserve even a projection of the original semantics. Unlike testability transformation, slicing and evolution transforms can be regarded as preserving abstractions of the original program's semantics. In practice, this theoretical difference may be less important than it seems. The authors have, as yet, only used fully meaning preserving transformations in the process of defining testability transformation algorithms.

8 Summary

This paper introduced testability transformation: a novel approach to program transformation, in which test data is generated from the transformed program but, for which the transformation process guarantees that the test data will be adequate for the original.

It has been shown that a testability transformation is novel because it is required neither to preserve the traditional meaning of the program, nor an abstraction of it. Rather, a testability transformation preserves a new meaning relating to preservation of adequate test data sets.

The approach was illustrated by the definition of a simple algorithm for flag removal and an empirical study which showed that this algorithm improves the performance of evolutionary test data generation, when the flag has relatively few input values which make it true (or relatively few that make it false).

Appendix
transformation Rules

The transformation rules used in this paper are defined in Fig. 14.

The rules are written in the form of a logical calculus. A rule of the form: $\frac{A}{B \Rightarrow C}$ can be interpreted as "If A holds then the fragment B can be transformed into the fragment C." The term $\mathbf{DEF}(e)$ denotes the defined [1] variables of e, where e is either an expression or a statement. The term $\mathbf{SUB}(e_1, i, e_2)$ returns the expression that results from substituting all occurrences of the variable i in the expression e_1, with the expression e_2.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[2] A. Baresel, H. Sthamer, and M. Schmidt, "Fitness Function Design to Improve Evolutionary Structural Testing," *GECCO 2002: Proc. Genetic and Evolutionary Computation Conf.,* pp. 1329-1336, July 2002.

[3] D.W. Binkley, "Computing Amorphous Program Slices Using Dependence Graphs and a Data-Flow Model," *Proc. ACM Symp. Applied Computing,* pp. 519-525, 1999.

[4] British Standards Institute, "BS 7925-1 Vocabulary of Terms in Software Testing," 1998.

[5] *Calendar Act,* Anno vicesimo quarto George II, cap. xxiii, 1751.

[6] G. Canfora, A. Cimitile, and A. De Lucia, "Conditioned Program Slicing," *Information and Software Technology Special Issue on Program Slicing,* M. Harman and K. Gallagher, eds., vol. 40, pp. 595-607, 1998.

[7] S. Danicic, C. Fox, M. Harman, and R.M. Hierons, "The ConSIT Conditioned Slicing System," *Software Practice and Experience,* to be published.

[8] J. Darlington and R.M. Burstall, "A Tranformation System for Developing Recursive Programs," *J. ACM,* vol. 24, no. 1, pp. 44-67, 1977.

[9] N. Dershowitz and Z. Manna, "The Evolution of Programs: A System for Automatic Program Modification," *Proc. Fourth Ann. Symp. Principles of Programming Languages,* pp. 144-154, 1977.

[10] 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 Trans. Software Eng.,* vol. 29, no. 7, pp. 665-670, 2003.

[11] M.S. Feather, "A System for Assisting Program Transformation," *ACM Trans. Programming Languages and Systems,* vol. 4, no. 1, pp. 1-20, Jan. 1982.

[12] M. Harman, D.W. Binkley, and S. Danicic, "Amorphous Program Slicing," *J. Systems and Software,* vol. 68, no. 1, pp. 45-64, Oct. 2003.

[13] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer, "Improving Evolutionary Testing by Flag Removal," *GECCO 2002: Proc. Genetic and Evolutionar Computation Conf.,* pp. 1359-1366, July 2002.

[14] M. Harman and B.F. Jones, "Search Based Software Engineering," *Information and Software Technology,* vol. 43, no. 14, pp. 833-839, Dec. 2001.

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

[16] B.F. Jones, H.-H. Sthamer, and D.E. Eyres, "Automatic Structural Testing Using Genetic Algorithms," *The Software Eng. J.,* vol. 11, pp. 299-306, 1996.

[17] C.C. Michael, G. McGraw, and M.A. Schatz, "Generating Software Test Data by Evolution," *IEEE Trans. Software Eng.,* vol. 12, pp. 1085-1110, Dec. 2001.

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

[19] R.P. Pargas, M.J. Harrold, and R.R. Peck, "Test-Data Generation Using Genetic Algorithms," *The J. Software Testing, Verification and Reliability,* vol. 9, pp. 263-282, 1999.

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

[21] L. Ramshaw, "Eliminating Goto's While Preserving Program Structure," *J. ACM,* vol. 35, no. 4, pp. 893-920, 1988.

[22] N. Tracey, J. Clark, and K. Mander, "The Way Forward for Unifying Dynamic Test-Case Generation: The Optimisation-Based Approach," *Proc. Int'l Workshop Dependable Computing and Its Applications (DCIA),* pp. 169-180, Jan. 1998.

[23] J.M. Voas and K.W. Miller, "Software Testability: The New Verification," *IEEE Software,* vol. 12, no. 3, pp. 17-28, May 1995.

[24] M. Ward, "Reverse Engineering through Formal Transformation," *The Computer J.,* vol. 37, no. 5, 1994.

[25] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary Test Environment for Automatic Structural Testing," *Information and Software Technology Special Issue on Software Eng. Using Metaheuristic Innovative Algorithms,* vol. 43, no. 14, pp. 841-854, 2001.

[26] M. Weiser, "Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method," PhD thesis, Univ. of Michigan, Ann Arbor, 1979.

Mark Harman is a reader in computing at Brunel University, UK. He has worked extensively on program slicing, transformation and testing, and, more recently, he founded the field of search-based software engineering. Dr. Harman is funded by the UK Engineering and Physical Sciences Research Council (EPSRC) and DaimlerChrysler, Berlin. He is program chair for several conferences and workshops including the 20th International Conference on Software Maintenance and the Search-Based Software Engineering Track of the Genetic and Evolutionary Computation Conference.

Lin Hu received the BSc degree in mathematics from Peking University and the MSc degree in applied mathematics from Beijing Institute of Technology, China. He is a research fellow with the Department of Information Systems and Computing, Brunel University, UK. His research interests are in the area of software engineering including program slicing, program transformation, and software testing.

Rob Hierons received the BA degree in mathematics (Trinity College, Cambridge), and the PhD degree in computer science (Brunel University). He then joined the Department of Mathematical and Computing Sciences at Goldsmiths College, University of London, before returning to Brunel University in 2000. He was promoted to full Professor in 2003.

Joachim Wegener studied computer science at the Technical University Berlin and obtained the PhD degree on the evolutionary testing of real-time systems at the Humboldt University of Berlin. This work gained him the Software Engineering Prize of 2002, awarded by the Ernst Denert Foundation. He is manager of software analysis and testing at DaimlerChrysler Research and Technology and played an important role in the development of the test system TESSY and the classification-tree editor CTE. Due to the close interconnection of research and development at DaimlerChrysler, he also acts as the quality assurance representative for development projects at Mercedes-Benz Passenger Car Development. He is a member of the SEMINAL Search-Based Software Engineering Network and heads the "Embedded Systems Testing" research group of the German Informatics Society.

Harmen Sthamer received a degree in electronics and communication from Polytechnic of Hannover, Germany, the MSc degree in electronic production Engineering, and the PhD degree in software technology from the University of Glamorgan, GB. Currently, he is working as a scientist in the Software-Technology Laboratory of DaimlerChrysler, Research and Technology. He is currently working on systematic and evolutionary software testing methods for the verification of software-based systems, e.g., embedded systems. He is author and co-author of several papers and has presented at national and international conferences. He is a member of the Seminal Network working on search-based software engineering.

André Baresel has finished his study at the Humboldt University of Berlin, Germany with a degree of Computer Science. Since 2000, he has worked as a research scientist in the Software Technology Research Department of DaimlerChrysler AG. His work is currently focused on the development and improvement of a tool environment for automatic test data generation using evolutionary algorithms. He frequently attends seminal network workshops. He is also involved in standardization activities within the area of testautomation for telematic systems using TTCN-3.

Marc Roper received an honours degree in computer science from the University of Reading and the PhD degree from the Council for National Academic Awards (CNAA). He is currently a senior lecturer in the Department of Computer and Information Sciences at the University of Strathclyde, Glasgow, UK. His research interests cover a range of software engineering topics and, currently, focus on the development of tools and techniques to support software engineers in analyzing and understanding large software systems. He is a member of the IEEE Computer Society.