Open Problems in Testability Transformation

Mark Harman
CREST: The Centre for Research on Evolution, Search and Testing,
King's College London, Strand, London, WC2R 2LS.

Abstract

Testability transformation (tetra) seeks to transform a program in order to make it easier to generate test data. The test data is generated from the transformed version of the program, but it is applied to the original version for testing purposes. A transformation is a testability transformation *with respect to a test adequacy criterion if all test data that is adequate for the transformed program is also adequate for the untransformed program.*

Testability transformation has been shown to be effective at improving coverage for search–based test data generation. However, there are many interesting open problems. This paper presents some of these open problems. The aim is to show how testability transformation can be applied to a wide range of testing scenarios.

1 Introduction

Automating test data generation is hard, but vitally important to reduce the cost of testing. One of the reasons this is such a hard problem lies in the astonishing variety of programs and test adequacy criteria for which test data generation must be applied. Testability transformation aims to attack this problem head on by transforming both programs and test adequate criteria into forms more amenable to automated test data generation. The approach has proved successful for several problems in search based testing.

The purpose of this paper is to demonstrate that it can be far more widely applied and to suggest some of the open problems. Many of these open problems would make ideal topics for programmes of doctoral study or for the more adventurous masters student seeking a project that combines program transformation and software testing.

Traditional program transformation [14, 52] refers to the automated process of altering a program's syntax without changing its semantics. The term 'semantics' is typically taken to refer to the functional properties of the program. Traditionally, transformation is permitted to change non-functional properties, such as speed of execution and resource consumption but it is not permitted to alter the input–output relation denoted by the original (untransformed) program.

Testability transformation is very different. The goal of the transformation is to ease the generation of test data. Though the test data is to be applied to the original program, it is generated from the transformed version. Therefore, the important semantic (and, indeed, also non-functional) properties to be preserved are contained, not in the input-output relation of the program, but in the goals of the testing process.

Typically, these goals are captured by a test adequacy criterion. The transformed program may denote *any* input output relation, not necessarily that of the original. The test data generated from the transformed program will be applied to the original, so changes in traditional input–output semantics will be irrelevant. However, the transformation must guarantee that any adequate set of test data generated from the transformed program, will also be adequate for the original program.

This difference in the properties to be preserved by the transformation process has profound implications for the nature of the transformation, making testability transformation theoretically interesting from a program transformation standpoint.

However, from the practical testing standpoint, it is more important that the application of testability transformation can improve test data generation techniques. It may improve either the efficiency or effectiveness of test data generation (or both). Testability transformation has been applied to several problems in search based testing, notably the flag problem [3, 4, 7, 26] and the nested predicate problem [41, 42].

To illustrate the startling non–functional properties of testability transformation, consider the scenario in which a program is to be tested to generate branch adequate test data. Suppose there is some computation along a predicate–free path in the program that affects the output produced but not the subsequent paths followed. For example, a sequence of side-effect free output statements that do not affect the subsequent path traversed.

Such a sequence of statements can be removed. Testability transformation allows this removal because any test data generated for branch coverage of the transformed program will also achieve the same level of branch coverage for the original program. Though the transformed program will not have the correct output, its branch execution behaviour remains invariant.

Why one would want to perform such a 'crazy' transformation? The answer lies in the behaviour of the automated test data generation system. Automated test data generation is computationally expensive and technically difficult. It is computationally expensive because it typically involves many trial executions of the program under test. It is technically complex because (at least for structural testing) it is necessary to account for many subtle possible behaviours. By removing parts of the program we may make the testability transformed program execute faster, thereby improving the efficiency of test data generation. By replacing semantically intricate behaviour with simpler behaviour we may be able to guide the test data generation algorithm towards more valuable test cases.

It might appear that testability transformation can only be applied to structural test adequacy criteria; surely transformation is inherently 'white box'? While it is true that the technique requires the source code of the program in order to perform the transformation, this does not mean that the goal of the process *necessarily* involves a white box testing process.

For example, consider the problem of generating test data for worst cases execution time, for which search based techniques have been applied [46]. Suppose there is a large constant-time computational task that is always performed at the end of the execution of the program under test. Since the task is constant-time in all executions, and it is always executed, it may be removed.

This will clearly change the behaviour of the program under test, but it will perfectly preserve the partial order we are interested in. That is, the rank order of each possible input to the program (ordered by the execution time they denote) will remain unchanged. However, all execution times will be reduced, thereby improving the performance of the test data generation technique.

As can be seen, the transformation possibilities of testability transformation are wide and varied. This paper presents some of the open problems in testability transformation, indicating some possible lines for research that may lead to extensions and improvements in the theory and practice of testability transformation.

Testability transformation is not the first instance of non-traditional meaning-preserving transformation. Previous examples include Weiser's slicing [59] and the "evolution transforms" of Dershowitz and Manna [16] and Feather [18]. However, both slicing and evolution transforms do preserve some *projection* of traditional meaning of the program under transformation and so they are less of a radical departure, compared to testability transformation.

For instance, it is possible to conceive of a formulation of both slicing and of evolution transforms in terms of abstract interpretation [11, 12], since both are projection–preserving. However, there are testability transformations that preserve a semantics that is neither more abstract nor more concrete than the original programs from which they are constructed. There are formulations of testability transformation for which the allowable transformations are neither subsets nor supersets of the transformations allowed by standard semantics. For instance, the program

```
if (x>y) ; else ;
```

can be transformed to the empty program, while preserving conventional semantics, but this simple optimization transformation is not permitted as a branch–adequacy–preserving testability transformation. This means that testability transformation does not preserve a more abstract semantics than conventional transformation.

However, the program

```
if (x>y) x=1; else x=2;
```

can be (branch–coverage–preserving) testability transformed to

```
if (x>y) ; else ;
```

but this transformation clearly does not preserve the conventional semantics of the program. Therefore, testability transformation is not more concrete than conventional transformation.

This example reveals that testability transformation cannot be explained by abstract interpretation; it is neither an abstract interpretation of conventional semantics, nor is conventional semantics an abstraction of it. Rather, testability transformation preserves an entirely new form of meaning, derived from the need to improve test data generation rather than the need to improve the program itself.

Testability transformation was first introduced by Harman et al. [26, 27]. Since then it has been applied to a number of problems in search based testing [3, 22, 28, 37, 41]. The primary purpose of this paper to demonstrate that there remain many interesting open problems in testability transformation that remain to be tackled in the literature.

Section 2 provides some basic definitions, making the paper self–contained. The remainder of the paper provides a set of open problems in different areas of testing. For some of these, there is merely a statement of the problem in broadest terms. For others, there are some examples of possible solutions.

2 Definitions

This section briefly reviews definitions of a testability transformation, in sufficient detail to make the paper self contained. Additional details and examples of successful applications of testability transformation can be found elsewhere [27].

Definition 1 (Testing-Oriented Transformation)
Let **P** be a set of programs and **C** be a set of test adequacy criteria[1]. 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, **C** is any criterion to be satisfied during testing. In this paper, **C** is used to refer to the overall criterion, which may be composed of a set of instances. Instances will be denoted by lower case c. For instance, branch coverage is a possible choice for **C**, while a particular instance, c, might be the set of branches to be covered in some program p.

Definition 2 (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'.

For some criterion, c, a c–preserving testability transformation guarantees that the transformed program is suitable for testing with respect to the original criterion.

Definition 3 (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 example, consider the program

```
x=1; y=z;
if (y>3) x=x+1;
    else x=x-1;
```

This program can be transformed to

```
if (z>3) ;
    else ;
```

Such a transformation does not preserve the effect of the original program on the variables `x` and `y`. However, it does preserve the set of sets of inputs that cover all branches. It also preserves the set of sets of inputs that achieve statement coverage. Therefore, the transformation is a branch–adequacy preserving testability transformation. It is also a statement–adequacy preserving testability transformation.

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

3 Exception Raising

Search based testing has been applied to the problem of raising exceptions. This work was pioneered by John Clark and his colleagues at the University of York [55, 56]. The idea is similar to that for branch coverage. The target is to raise an exception. Conceptually, this can be thought of as modifying the program to guard the exception–raising statement with a predicate and then attempting to cover the branch, the traversal of which leads to the exception being raised.

Indeed, such a 'little tweak' is an instance of a testability transformation itself. That is, we can reformulate exception raising as branch coverage, thereby allowing existing techniques (for branch coverage) to be applied to the new problem of exception raising.

However, the applications of exception raising are rather different to those of branch coverage and this raises some interesting open problems and possibilities.

3.1 Can we transform a program so that inputs that form near neighbours to exception raising cases also raise exceptions?

Exceptions may be hard to raise. After all, they are, by their very nature, conditions that are *exceptional* and, therefore, unlikely to occur. This means that there may be very few inputs that raise an exception, making the task of automatically identifying such inputs rather difficult.

One solution (applicable to any search based approach) would be to transform the program so that inputs that are near neighbours of those that raise exceptions will also raise exceptions. Notice that it is not necessary to *know*, at compile time, what these inputs are in order to transform a program such that near neighbours also raise exceptions.

This is one of the strengths of the testability transformation approach; it is possible to transform a program to make it behave in a desired manner, without knowing *precisely* how that behaviour would manifest itself, nor the inputs that would cause the manifestation to occur.

Having made near neighbours exception–raisers, the program will be more amenable to search. For example, we could use global search to locate inputs that raise *some* exceptions, possibly those that we introduced ourselves in the testability transformation. Having located these, we can then use a local search to try to identify the test inputs that raise the exception for which we were originally concerned.

3.2 Can we decompose hard exception conditions into a series of easier conditions?

Exceptions may be hard to raise because inputs cannot be found that make them occur. A decompositional approach could be adopted. Suppose the condition is re-written in Conjunctive Normal Form (CNF).

This may lead to a set of conjuncts all of which capture necessary conditions to raise the exception. Now each of these can form a separate program, from which we seek to find an input that raises the condition. Rather than simply attempting to generate a single input that raises the condition captured by each conjunct, we shall seek to find a set of such inputs.

One key measure of the hardness of a test data generation problem is the number of inputs that cause the desired behaviour to occur, or perhaps, more accurately, the ratio of the number of behaviour–causing inputs to the number of behaviour–avoiding inputs; the domain–to–range ratio of Voas [57].

By definition of CNF, each conjunct will be satisfied by at least as many inputs as the original exception condition, so we cannot produce a harder test data generation problem by this testability transformation. Therefore, by forming the CNF of the exception condition, we will have decomposed a hard test data generation problem into a set of potentially easier problems.

After we have generated a set of test inputs that raise each of the conjuncts, we can now see whether there is an intersection. Where there is, we have a solution to the original problem. Even partial overlap between these sets denotes a partial solution to our original problem and may provide valuable information to the tester. Furthermore, we can use the inputs in any partial intersection as seeds for further searches to satisfy the remaining unsatisfied conditions.

Testing for exception raising is also complicated by the fact that we may be attempting the impossible. That is, we may be attempting to raise an exception that captures a state to which the program should not arrive; defensive programming may have been used to check for such a rogue state and to raise an exception should it be reached. It may be that the program cannot reach this state; the testing process is being used to provide a measure of confidence in correct behaviour. In this situation the testability transformation to CNF can provide more feedback and insight to the tester. For instance, the tester can see which of the necessary conditions *can be* satisfied. These may indicate potential weaknesses that may materialize as failures should the code be re–used in a different context.

Of course, CNF is only one possible decomposition. Another natural candidate would be a transformation of the exception condition to be raised into Disjunctive Normal Form (DNF). In this way, we obtain a set of sufficient conditions in order to raise the original exception. We could then try separate searches using each disjunct, safe in the knowledge that should *any* produce a solution, then that solution would be sufficient to cause the original exception to be raised. This is, in essence, the idea behind the 'species per path' approach to evolutionary test data generation [43].

4 Temporal Testing

Temporal testing seeks to locate test cases with worst or best cases execution time. This is important for systems with hard timing constraints, for example real time and embedded systems. There has been work on search based techniques for finding test cases with worst case execution time [46]. This work requires many repeated executions of the software under test in order to locate worst and best cases.

4.1 Can we transform a program to linearly decrease the execution time for all inputs?

To help with temporal testing, we seek transformations that may reduce execution time. The testability transformation framework admits far greater flexibility than existing efficiency optimization transformations [1] would permit, because it allows programs to be transformed in ways that do not preserve functional equivalence.

Indeed, existing optimization transformations may be inapplicable in many cases, since they may not *linearly* reduce execution time across all inputs. That is, a transformation 'linearly' affects execution time iff the execution time of the transformed program is a linear function of that for the untransformed program for every input. It is a linear reduction iff it is a linear transformation that reduces execution time in every case. Linear temporal testability transformation preserves the ordering of execution times for each input.

Such transformations may not be easy to construct, but it may be possible to find them in some domains. If we could locate such linear execution time reduction transformations, this would help to improve the efficiency of test data generation for temporal testing. Reducing execution time by a constant factor across all inputs would allow for faster generation of test cases for worst case execution time and best case execution time alike. The effect of this transformation is depicted schematically in Figure 1(a), in which the grey execution time profile is produced from black profile by the application of a linear transformation on execution times.

4.2 Can we use static analysis to rule out certain paths guaranteed not to be worst/best case?

Static analysis can be used to identify paths that are guaranteed not to lead to worst/best case execution time [21]. For these paths, the transformed version of the program can be aborted, thereby reducing its execution time. Hitherto, the literature has yet to report on experiments with such a mixed static/dynamic approach. It may form an attractive example of a blended analysis [17]. The effect of this transformation is depicted schematically in Figure 1(b), which is produced from the execution time profile in Figure 1(a) by the application of a linear transformation on execution times.

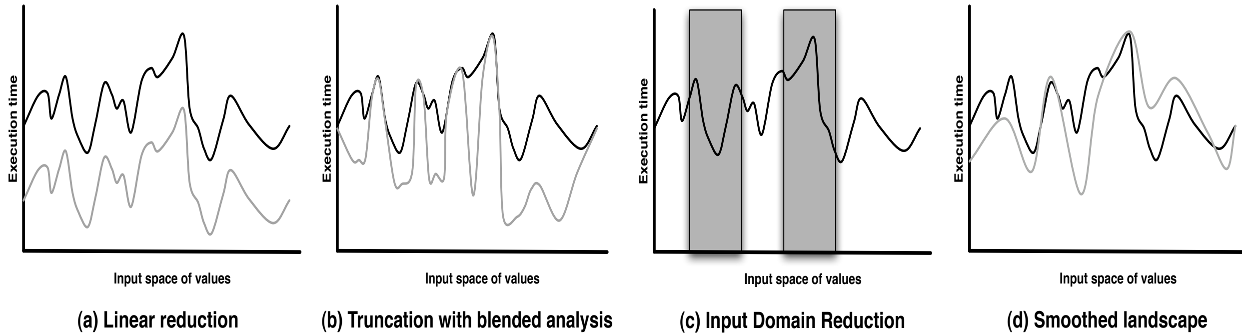

Figure 1. Transforming temporal properties. The dark lines indicate the original execution profile. The grey lines in parts (a),(b) and (d) denote the execution profile of the transformed program. The grey shaded areas in part (c) indicate the reduced area in which the search will take place after domain reduction.

Static analysis can also be used to determine parts of the input space that cannot contribute to computations that lead to the extremes of execution times. In this way, it is possible to reduce the size of the search space; to reduce the domain in which the search is performed, thereby improving its performance. Domain reduction has been applied to structural testing [24], where it has been shown to improve the efficiency of structural test data generation. There is no reason why it cannot also be applied to temporal testing. The effect of this transformation is depicted schematically in Figure 1(c), in which only the greyed out regions of the search space are searched in the reduced version of the program.

4.3 Can we apply transformations to smooth execution time?

Unlike many test objectives, the fitness function for temporal testing leads to highly 'jagged' search landscapes; a small change in input can lead to a dramatic change in execution time. Since we seek the worst case, any execution times which are not worst, could be 'smoothed' to create a better guide towards the most optimal case. The effect of this transformation is depicted schematically in Figure 1(d), in which the grey execution time profile is the smoothed version of the black execution profile.

5 Mutation Testing

A mutant is produced by a syntactic change that mimics the effect of a fault. The idea is both to generate and also to assess test data. Test data is mutation adequate if it kills all mutants generated. A mutant is killed by a test case if the mutant behaves differently when executed on the test case compared to the original.

A test set that is mutation adequate is able to find each and every one of the faults seeded by each of the mutants and that, therefore, it would also be good at finding real faults. This observation can be used to assess a test set (how many mutants can it kill?) or to generate one (seeking to kill as many as possible).

Mutation testing has been studied for many years [2, 10, 15, 35, 36, 50, 60]. The technique is theoretically strong because it is able to subsume other test adequacy approaches. It has also been found, empirically, to be effective at delimiting test suites that are as good or better at finding faults than other techniques [49].

However despite its theoretical power and practical appeal, mutation testing suffers from two problems that, hitherto, have left it largely unapplied in industry. There are a large number of possible mutants, making the technique expensive. There is also the problem of equivalent mutants; mutants that are syntactically different but semantically identical to the original program. The problem of the large number of mutants has largely been addressed by techniques for smarter generation of mutants, such as selective mutation [8, 48], but the equivalent mutant problem remains unsolved.

The equivalent mutant problem is particularly pernicious because it is not possible to define an algorithm to remove from consideration, all equivalent mutants. The problem of determining whether a mutant is equivalent is sadly reducible to the functional equivalence problem, which is known to be undecidable. There are techniques based on static analysis [32, 47] that seek to detect equivalent mutants. However, there will always remain some possibility that some of the mutants that are currently unkilled are simply unkillable because they are equivalent and have gone undetected as such.

This means that we must either weed out equivalent mutants by hand, with all the expense that this labour–intensive approach would entail, or try to avoid generating them in

the first place. Most of the existing work on the equivalent mutant problem has started from the assumption that equivalent mutants *will* be created and that, therefore, we need techniques to detect them.

Testability transformation offers an alternative approach; we can seek to transform the program so that it is *less likely* that some targeted mutation testing algorithm will generate equivalent mutants in the first place. We can also seek transformations that produce transformed programs from which is it simply not possible to create certain forms of equivalent mutant. The next two subsections consider these two possibilities in a little more detail.

5.1 Can a program be transformed so that it is less likely to produce equivalent mutants?

The `ABS` mutation operator takes an arithmetic expression and applies the unary `abs` function to it. The `abs` function returns the absolute value of the expression to which it is applied. Naturally, this will lead to an equivalent mutant if it is applied to an expression that always takes a non–negative value in every possible execution. Sadly, there are very many program expressions that are, indeed, never negative and so the `ABS` mutation operator is one of the operators that tends to generate a large number of equivalent mutants. An archetypal expression is `y=x*x`, which is guaranteed to be non–negative.

Testability transformation may prove to be applicable to this problem. Perhaps some occurrences of assignments of such always–positive expressions can be replaced with expressions that are transformed to be possibly negative. Care will then be required to transform the uses of such transformed assignments so that they behave identically with respect to mutation adequacy. Perhaps only some occurrences of always–positive values can be removed, while others must be retained to ensure mutation adequacy preservation.

Some expressions may be replaced by more complex equivalent expressions, in which all subexpressions are guaranteed to have the ability to take negative values. While the overall expression is still always positive (and will therefore lead to an equivalent mutant), the ratio of equivalent mutants to non–equivalent mutants will be reduced.

A bolder approach would be to seek to eradicate the possibility that an equivalent mutant could be created. This produces an appealing transformation based route out of 'Turing's swamp' of undecidability. While the detection of all equivalent mutants is undecidable, it may still be possible to produce a transformed program from which only non-equivalent mutants will be constructed.

Notice that such a transformation–based approach does not reduce to the halting problem because it may be conservative; it may transform parts of the program that cannot possibly lead to equivalent mutants. This would mean that it could not necessarily be used to detect equivalent mutants,

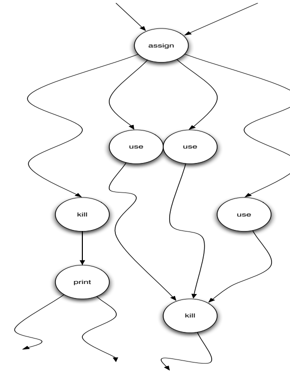

Figure 2. Schematic representation of an example where mutation of the assignment statement at the top of the CFG will be an equivalent mutant. Any path that leads to a pint cannot retain the value assigned because of intervening kills.

but only a superset of mutants that contains all equivalent mutants. Such a superset could be arbitrarily large and, in the worst case, may therefore yield no information regarding the set of mutants that are definitely equivalent.

Though this means that the approach cannot be reduced to the detection problem (which is known to be undecidable) it would not harm the applicability to mutation testing so long as the transformation is mutantion-adequacy preserving and the overall effect is to render equivalent mutants impossible. This remains a hard, but exciting possibility. Even partial fulfillment of such an ambitious goal could be extremely valuable.

5.2 Can a program be transformed so that certain equivalent mutants are avoided?

Another common cause of equivalent mutants is a mutation to a variable, the value of which is subsequently overwritten before the value is output. Such a mutant cannot cause a change that will lead to the mutant being killed, because any value assigned at the mutant point is overwritten before it has a chance to influence the output. A schematic illustration of this situation is depicted in Figure 2. Notice that this kind of equivalent mutant is killed by week mutation, but not by strong. Mutants that are equivalent even for weak mutation (such as those considered in Section 5.1) are much easier for a human to detect, because they only require consideration of local before and after states. Therefore, it is these weakly killable equivalent mutants that we should really concentrate upon.

In previous work, Hierons et al. [32] showed that program dependence analysis can be used to detect some equivalent mutants created in this manner. The approach could

be extended to form a simple testability transformation that would remove parts of the program that could potentially lead to equivalent mutants. This can be achieved using slicing [6, 25, 54], so there exists relatively mature technology that can be used to achieve such a testability transformation.

That is, a simultaneous slice should be constructed for the set of slicing criteria that include all output statements. Any code removed by this slicing process will be code that, when mutated, will have no effect of the output and that will, therefore, be guaranteed to produce an equivalent mutant. Because the slice preserves all code required to perform the output of the original program, the slice will be a mutation–adequacy preserving testability transformation.

There are also situations where traditional slicing may be insufficiently powerful to remove the code that could lead to an equivalent mutant. For these programs, a testability transformation, specifically tailored to removing the possibility for equivalent mutants will be required.

For example, consider the fragment below:

```
if (p)
  { x=x+1;
    y=x*2;
    q=1; }
  else
  { y=x-1;
    q=0;
  }
/* the original value of x is lost */
x = 1;
if (q==1)
   print(x);
   else
   print(y);
```

The first assignment in this program, `x=x+1`, does not affect the output, but it will be included in any slice on the output, because slicing algorithms cannot *determine* that the assignment cannot influence the output. There are many similar examples; one can construct programs with assignments to local variables in which it can be proved that these assignments do not affect the output, but for which slicing will not remove them; the semantics are simply too subtle to be captured.

Any attempt to define tools able to reason at this detailed semantic level is likely to lead to tools that are computationally very expensive. However, it is not necessary for a testability transformation algorithm to be able to *detect* such equivalent–mutant–causing statements. All the algorithm has to do is to *remove* any potential candidates through transformation; if some non-candidates are removed in the process, then this is fine, so long as any such removal is mutation–adequacy preserving. Defining such transformation algorithms is far easier than developing approaches for the detection of equivalent mutants.

For instance, examples such as the one above can be addressed using amorphous slicing [23], which attempts to transform a program to reduce its size, by substituting the effects of assignments. Amorphous slicing with respect to output statements will have the effect of reducing the size of the program (thereby reducing the number of mutants created). It will also have the (extremely attractive) spin–off benefit that assignments to local variables will be 'rolled into' the assignments which follow them, thereby reducing the potential for the creation of equivalent mutants. Applying this to the program fragment above (slicing at the two output statements) produces:

```
if (p)
  { y=(x+1)*2;
    q=1; }
  else
  { y=x-1;
    q=0;
  }
if (q==1)
   print(1);
   else
   print(y);
```

The amorphous slicing process has removed the assignment `x=x+1` (folding it into the assignment to `y` which follows). It is not possible to generate equivalent mutants from this assignment, because it is no longer present in the transformed program. Furthermore, notice how amorphous slicing substitutes the constant value assigned to `x` into the predicate that uses it. This reduces the number of mutants created (from the transformed program compared to the original). However, it does not affect mutation adequacy, because any mutation of `x=1` in the original would be coupled to another mutant to the output statement `print(x)`.

6 State Variable Problems

State variables are variables whose value exists outside a single call to the system under test. For instance, a static variable has a value that persists after the conclusion of a function call. Consider the Search Based Testing (SBT) technique used by Wegener et al. [58]. For this technique, calling a function under test will not be able to cover branches that are controlled by predicates, the outcome of which is dependent upon a static variable. This is because such SBT approaches call the function once only, and with the (implicit) assumption that there exists a combination of the parameters' values that will cause execution of any chosen target branch.

This implicit assumption is invalid in the presence of static variables. However, merely recognising that a static variable forms a part of the search space is insufficient to solve the problem. It is not possible to choose (at call time) the value to be assigned to a static variable in the same way that it is possible to choose values for the parameters to the call. The static variable's value may only be assigned indirectly, by calling functions that assign it a value. In this way, the problem is to determine which functions to call, how many calls to make and with which parameters in order to cause the desired predicate to receive the values that cause it to traverse the, as yet, uncovered branch of interest.

The problem of state variables has been addressed by several authors [44, 45, 61]. This section considers the way in which the problem could be formulated as a testability transformation problem. Hitherto, this possibility remains unexplored in the literature on the state variable problem.

6.1 Can a program with state variables be transformed into an equivalent without?

The problem of catering for behaviour controlled by state variables can be handled directly using testability transformation. A simple example will serve to show how the problem of determining the number of calls and associated parameters required for a testing problem can be transformed into a problem of determining inputs to a more traditional testing problem involving loops. The transformation is not complex and could easily be automated. The result is a more tractable testing problem.

The transformation also renders the problem in a form expressed in terms of loops. Since many of the problems in test data generation involve determination of loop invariants, bounds and constraints, it is likely that this will be the topic of much research.

It makes sense to convert all test data generation problems into canonical forms, wherever possible, so that research on solutions can focus on simple archetypal examples, in the knowledge that the techniques developed will be widely applicable. This argument is nothing more than Landin's argument concerning programming language 'syntactic sugar', adapted for test data generation [39].

Consider the simple state variable problem below:

```
static int x = 0;

void foo() {x=x+1;}

void bar()
{ foo();
 if (x > threshold)
    /* target */ ;
}
```

In this program, the value of `Threshold` is some (compile time unknown) constant. Calling the function `bar()` will not necessarily execute the target branch. Indeed, it is unlikely to do so. All irrelevant details not pertinent to the state variable problem have been removed to create this (artificial, illustrative) example. The program has no input, so there is no input space to search. The only factor under the tester's control is the number of times the function `bar()` is called. The function affects a static variable through calls to `foo()` and the tester cannot know how many times the function needs to be called in order to achieve the value that will cause the target to be executed.

This is the simplest illustration of the state variable problem. The variable `x` plays the role of state variable. In more complex examples, the function `bar()` would need to be called several times, and perhaps with particular parameters. However, these more complex examples of the state variable problem could also be addressed using the testability transformation approach outlined below.

Suppose we transform this program to more conventional test data generation scenario, in which the problem of determining the number calls to `bar()` becomes, instead, a problem of determining the value of a variable, `i`:

```
static int x = 0;
int targethitflag ;

void foo() {x=x+1;}

void bar'()
{ foo();
 if (x > threshold)
    /* target */ targethitflag = 1;
}

void main()
{ int i = 0;
      targethitflag = 0;

  while (i<MAX && !(targethitflag))
     { bar'(); i = i+1; }
}
```

In this transformed program, `MAX` is some resource constraint limit that prevents the test data generation algorithm from taking too long on this particular test data generation problem; it can be set to an arbitrarily large value. Notice that the transformed program transforms the search problem to one that, in this simple case, *automatically* finds the correct value for the number of calls required to `bar()`.

Of course, a more complex example would involve parameters as well as state variables. Let us now turn to such a more complex example. Consider the slightly more complex version of the program under test below

```
static int x = 0;

void foo(int z) {if (z>0) x=x+1;}

void bar(int v)
{ foo(v+1);
  if (x > threshold)
    /* target */ ;
}
```

In this version of the problem, the test data generation algorithm has to recognise that it needs to call `bar` sufficiently many times, but also that on each occasion `bar` is called, it needs to pass a value that is greater than 1. This program would be transformed into an imperative test data generation problem with an input space of unknown length; the unknown number of calls to `bar` is essentially transformed into an unknown input length.

Unbounded input lengths can be handled using, for example, a messy GA (for search based approaches [40]), or by treating the input as a sequence data structure (for CUTE/DART approaches [20, 53]). For search based testing, the transformed program would be:

```
static int x = 0;
int targethitflag ;

void foo() {if (z>0) x=x+1;}

void bar'(int v)
{ foo(v+1);
 if (x > threshold)
    /* target */ targethitflag = 1;
}

void main()
{ int i = 0;
      targethitflag = 0;
  int v;

  while (i<MAX && !(targethitflag))
     { scanf("%d",&v);
       bar'(v); i = i+1; }
}
```

7 Subsumption Relations

There is a well-known subsumption hierarchy for test data generation [5, 9]. For example, if a test set covers all branches, then it certainly covers all statements: branch adequacy is said to subsume statement adequacy. It would be natural to speculate about the subsumption relationships that exist between testability transformation formulations. That is, the definition of an adequacy criterion delimits the set of transformations that can be performed in testability transformation. This set of transformations is different for different criteria.

A natural way to define subsumption is in terms of the sets of transformations that are permitted by each formulation of an adequacy–preserving testability transformation.

Definition 4 (Allowable Mappings)
Let $M(C)$ be the set of mappings (from programs to programs) that are allowed by some C–preserving testability transformation τ. That is,

$$\{m \in M \mid m = (p, p') \wedge \exists c \in C.\exists \tau.\tau(p, c) = \tau(p', c)\}$$

For example, if the test adequacy criterion C is branch coverage, then one possible mapping in $M(C)$ is the pair

(if (e) $x = 1$; else $x = 2$;, if (e); else;)

That is, the transformation that takes a program of two branches that are themselves branch–free, and removes the two branches. A test adequacy criterion C subsumes another C' if C allows only a subset of the transformations allowed by C'. In this situation, we know that any transformation that works for C will also work for C'. More formally

Definition 5 (Subsumption)
C subsumes C' iff $M(C) \subseteq M(C')$.

Interestingly, branch coverage neither subsumes nor is subsumed by statement coverage according to Definition 5 above. To see that branch coverage does not subsume statement coverage. Consider the transformation below.

```
if (x>y) x=1; else x=2; ⇔
       if (x>y); else;
```

This is a branch–adequacy preserving testability transformation, but it is not a statement-adequacy preserving testability transformation. That is, the set of sets of inputs that cover all branches in the transformed program contains the set of sets of inputs that cover all branches in the original program. Therefore, we can generate our test inputs from the transformed program and yet apply the test set so–generated to the original.

To see that statement coverage does not subsume branch coverage, consider the transformation below:

```
if (x>y); else; ⇔ ;
```

This is a statement–adequacy preserving testability transformation, but it is not a branch–adequacy preserving testability transformation. That is, there are no statements in the program under test and so any set of test cases covers

all statement. Indeed, even the empty set of test cases vacuously covers all statements. We can therefore transform this program to *any* program. Such a transformation will be a statement–adequacy preserving testability transformation. However, there are two branches in the program under test and to achieve branch coverage a test set must contain at least two test inputs. The set of programs to which the program under test may be branch–adequacy preserving testability transformed is therefore more restricted for this program.

There exist several possible candidates for the formulation of the subsumes relationship among testability transformations. For instance, we could ask

> "If a transformation preserves branch coverage, are all branch adequate test sets generated from it, also statement adequate for the original?"

We are checking whether the conventional testing subsumes relationship also holds for test data generated from testability transformed programs. More formally,

Definition 6 (Conventional Subsumption Preservation)
Let C_1 and C_2 be conventional test adequacy criteria, such that C_1 subsumes C_2. Let τ be any C_1 preserving testability transformation. If, for all programs p, and instances, c_1 of C_1. $\tau(p, c_1) = (p', c_1)$ implies that the transformation from (p, c_2) to (p', c_2) is also a testability transformation, then (C_1,C_2) will be referred to a 'conventional'.

For this formulation of the subsumes relationship, all criteria pairs are conventional. That is, for any possible adequacy criteria C_1 and C_2, if C_1 subsumes C_2 then test data adequate for C_1, obtained from a C_1 preserving testability transformed program, will be adequate for C_2 also. This follows directly from the definitions of subsumes and C–preserving testability transformation. Suppose p' is a program obtained from p by a C_1 preserving testability transformation and that T is a set of test data that satisfies C_1 when applied to p'. By definition of C_1–preserving testability transformation, T must be C_1 adequate for p and, since C_1 subsumes C_2, T must also be C_2 adequate for p.

8 Probabalistic Testability Transformation

Korel et al. [37] introduced a transformation that speculatively removes code from a program for which test data generation has proved difficult. The test data generation method in this case is the chaining method [19]. The interesting aspect of this work is that the transformations are not even truly testability transformations according to Definition 2, because they do not necessarily preserve test adequacy. McMinn et al [41] also speculatively remove control flow in at attempt to improve search based testing in the presence of predicate nesting.

In this case, the transformation is performed as a 'last hope' attempt to generate test data to cover a hard–to–cover branch, when all else has failed. From the transformed program, it may be possible to generate test data that is branch adequate for the transformed program, but which fails to be equally branch adequate for the original.

However, there is a belief that it is *likely* that test data generated from the transformed program has a better than random chance of covering the desired branches. This probabilistic argument remains implicit in the work of Korel et al. There is no attempt to formalise, nor to reason about, the probability that test data will be generated with a certain likelihood. However, it is demonstrated empirically that the approach does, indeed, lead to better performance, thereby providing empirical evidence to support the suggestion that the aggressive transformation does lead to greater probability of test adequacy.

This work raises the possibility of relaxing the definition of a testability transformation. Instead of requiring the transformation to contain the 'adequacy semantics' of the original, we simply require it to mimic important aspects of the original's 'adequacy semantics' with a certain probability.

In order to formalize this probabilistic testability transformation it is helpful to define formal notation for the boolean predicate test for adequacy:

Definition 7 (Adequacy)
$Adequate(p, T, c)$ iff test set T is adequate for program p according to test adequacy criterion c.

Definition 8 (Strong Probabilistic)
A Testing-Oriented Transformation, τ is a *Strong Neutral Probabilistic Testability Transformation* with respect to probability π ($0 \leq \pi \leq 1$) iff for all programs p, and criteria c,

$$\frac{\mid \mathcal{T}' \cap \mathcal{T} \mid}{\mid \mathcal{T}' \mid} \geq \pi$$

where $\tau(p, c) = (p', c')$, $\mathcal{T} = \{T \mid Adequate(p, T, c)\}$ and $\mathcal{T}' = \{T' \mid Adequate(p, T', c)\}$

This is the 'neutral' formulation of probabilistic testability transformation because it makes no assumptions about the techniques that will be used to generate test data. Therefore it make a 'neutral' assumption that the test generation technique is equally likely to generate any of the possible test sets from all possible test sets that are adequate for the transformed program.

Loosely speaking, it requires that there is at least a π chance that test sets that are adequate for the transformed program are also adequate for the original. Other non-neutral formulations are possible, in which the test data generation technique is taken into account. Such an algorithm–

biased formulation would require that the transformation increases the chances that the algorithm will generate an adequate test set from the transformed program compared to the original.

It is relatively easy to prove that this formulation of testability transformation is a relaxation of the standard definition (Definition 2). That is, suppose we choose $\pi = 1$. This would yield the standard definition of testability transformation from Definition 8 of a Strong Neutral Probabilistic Testability Transformation. The only way in which a transformation could be a Strong Neutral Probabilistic Testability Transformation with $\pi = 1$ is, by definition, if

$$\frac{\mid \mathcal{T}' \cap \mathcal{T} \mid}{\mid \mathcal{T}' \mid} \geq 1$$

which means we must have $\mathcal{T}' \cap \mathcal{T} = \mathcal{T}'$, so $\mathcal{T}' \subseteq \mathcal{T}$. However, by definition, this means that $Adequate(p, T', c) \Leftrightarrow Adequate(p, T', c)\}$. So test data which is adequate for the transformed program is adequate for the original, so any Strong Neutral Probabilistic Testability Transformation for $\pi = 1$ is a testability transformation according to Definition 2.

The formulation above is also strong, in the sense that it requires that the testability transformation increases the probability of generating adequate test data from all programs. An alternative, weaker, formulation also exists. It could be that the transformation is 'weak' in the sense that it is targeted at a subset of all programs, only guaranteeing to improve adequate test data generation chances for those important programs for which it is designed.

Definition 9 (Weak Probabilistic)
A Testing-Oriented Transformation, τ is a *Weak Neutral Probabilistic Testability Transformation* with respect to probability π ($0 \leq \pi \leq 1$) iff there exists a non empty set of programs P, such that for all $p \in P$, and criteria c,

$$\frac{\mid \mathcal{T}' \cap \mathcal{T} \mid}{\mid \mathcal{T}' \mid} \geq \pi$$

where $\tau(p, c) = (p', c')$, $\mathcal{T} = \{T \mid Adequate(p, T, c)\}$ and $\mathcal{T}' = \{T' \mid Adequate(p, T', c)\}$

8.1 Can we find practical ways to exploit probabilistic testability transformation?

Korel et al. [37] defined a testability transformation that is able to increase the chances that the chaining rule will generate test data that is adequate for branch coverage for hard–to–cover branches, but this was only demonstrated empirically for a few cases. It remains a challenge to develop a more provably correct approach to probabilistic testability transformation. Such an approach should be able to provide a definition of the probability that test adequacy is more likely with the transformed program. This may prove to be a hard goal, because of the subtle interplay between the programming features possible, the adequacy criteria and the test data generation algorithm.

In the absence of any formally proven probabilistic testability transformations, there remains the more empirical problem of generating algorithms that can improve the chances for adequate test data generation (at least informally). Given the inherent uncomputability of most test data generation tasks, it seems likely that there would be benefits in such a probabilistic approach.

9 Stress Testing

In stress testing, the goal is to reveal conditions under which the software performance may degrade suddenly and to find the values of input combinations that cause this. Due to the emergent properties of some complex interactive distributed systems, it can happen that the search for such inputs is not merely a search for the level of use that causes critical break down. It can be that certain properties of the input (for relatively low load) can also cause critical breakdown in the level of service provided.

It is a hard problem to formulate a search criterion to capture these stress–causing inputs. However, there has been work on search based approaches for automation of stress testing and so it remains an open question as to whether this hard problem can be made any easier by testability transformation. Two open problems naturally suggest themselves:

1. Can a system be transformed to make potential stress more likely?

2. Can the system be transformed so that the space of inputs has larger basins of attraction for search based stress testing?

10 Directed Random and Concolic Testing

Concolic testing [53] and Directed Automated Random testing (DART) [20] form constraints for test data generation which are put to a linear constraint solver. These techniques have received much recent interest. However, the current approach uses the constraint solver lp_solve, which is itself, an implementation of an optimization algorithm for constraint solving, based on classic OR techniques. It cannot handle real–valued constraints (so CUTE, for example, cannot properly handle programs with floating point numbers), nor constraints which involve non–linear terms. This raises a natural question as to whether a program can be transformed to remove or reduce the prevalence of such non–linear constraints and to transform the use of floats to integers in a branch preserving manner.

11 Testing FSMs

There has been much work on test data generation for Finite State Machines (FSMs), but relatively little work on transformation for FSMs. There has been some initial work by Hierons et al. [33], who suggested the possibility of improving test data generation techniques for Finite State Machines using a testability transformation approach. More work is required to explore the possibilities.

There are extended FSMs, such as X machines [31, 34], for which testing is particularly powerful, because of the guarantees it is able to provide concerning testing adequacy. These models require certain design–for–test criteria to be met. A natural transformation based approach would be to seek to transform an extended Finite State Machine into an X machine. This would allow testing to take advantage of the improved testability of such machines. The investigation of such FSM testability transformations remains an open problem.

12 Specification Based Testing

Occasionally, we are fortunate enough to have a formal specification, from which to generate test cases [29, 30]. Even when a formal specification is not available, it is possible that there may be some model of the system under test. Model based approaches to system development are increasingly prevalent in industry and so the presence of some form of model of the system under test, formal or semi formal, is becoming more likely. Models are often written to help generate test data. It makes sense, therefore, to consider the extent to which a model (or specification) could be transformed to improve the chances of test data generation.

It is convenient to define modelling notations in such a way that testability transformation is relatively easy. Rather than attempting to write the model in a way that is both suitably abstract and also suitable for testing, the engineer could simply concentrate on getting the abstraction right and then use transformation to render the model in a form more suited to test data generation.

In this way, testability transformation would be following a parallel path of development to functional programming; define a language rich in applicable transformations and concentrate on getting the abstraction right, leaving the details to the transformation engine. Testability transformation for declarative modelling languages would be one way in which testability transformation could 'return to its roots' in the transformation literature [13, 14, 38, 51].

13 Conclusion

Test data generation is notoriously hard. Recent work (including that one search based testing) has made progress towards the ultimate goal of fully automated test case design. However, the techniques that are being developed are often hampered by features of the programs under test. Testability transformation provides a way to extend the applicability of these techniques and to increase their effectiveness and efficiency. Testability transformation is an instance of the application of a well-used engineering principle:

> If the problem we are trying to solve is inherently too hard, then develop techniques to transform the problem to make it more amendable to the tools and techniques available.

14 Acknowledgments

This is a single author paper, since it is a brief account of the topics and open problems raised by the author's keynote at the first Search Based Testing Workshop, in Lillehammer, Norway, April 2008. However, though it is a single author paper, there are many other researchers with whom I have worked who deserve significant credit for helping to form the ideas outlined in the paper.

I have worked on testability transformation with André Baresel, David Binkley, John Clark, Sebastian Danicic, Robert Hierons, Lin Hu, Bogdan Korel, Kiran Lakhotia, Phil McMinn, Marc Roper and Shin Yoo.

In particular, the idea of extending testability transformation to situations where the conditions outlined in Section 2 must be relaxed (described in Section 8) came form work with Bogdan Korel and Phil McMinn and my many conversations with them have also certainly contributed significantly to the development of these ideas.

The initial work on testability transformation was funded by the EPSRC project TeTra — Testability Transformation (GR/R98938), which ran from 2003 to 2006. More details concerning the TeTra project, including pointers to the literature are available on the TeTra website at

`www.dcs.kcl.ac.uk/staff/linhu/TeTra`

Current work on Testability Transformation is supported by the EU project EvoTest — Evolutionary Testing (IST-33472) and by the EPSRC project SEBASE — Software Engineering By Automated SEarch (EP/D050863). The SEBASE project website maintains a repository of all papers on Search Based Software Engineering, including those on Search Based Testing:

`www.sebase.org`

I am grateful to my partners and collaborators in the these projects for the many conversations we have had on Search Based Software Engineering, in general, and Search Based Testing in particular.

References

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

[2] D. Baldwin and F. Sayward. Heuristics for determining equivalence of program mutations. *Research Report 276, Department of Computer Science, Yale University*, 1979.

[3] A. Baresel, D. W. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop–assigned s: A testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 108–118, Omni Parker House Hotel, Boston, Massachusetts, July 2004. Appears in Software Engineering Notes, Volume 29, Number 4.

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

[5] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

[6] D. W. Binkley and M. Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

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

[8] L. Bottaci and E. S. Mresa. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing, Verification and Reliability*, 9(4):205–232, Dec. 1999.

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

[10] T. A. Budd. Mutation analysis: Ideas, examples, problems and prospects. In *Proceedings of the Summer School on Computer Program Testing*, pages 129–148, Sogesta, June 1981.

[11] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, Aug. 1992.

[12] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. *ACM SIGPLAN Notices*, 31(1):178–190, Jan. 2002.

[13] J. Darlington and R. M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

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

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

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

[17] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, pages 118–128. ACM, 2007.

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

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

[20] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In V. Sarkar and M. W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223. ACM, 2005.

[21] J. Gustafsson. WCET 2007 - report from the WCET tool challenge 2006 ideas for the WCET tool challenge 2008. In C. Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy, July 3, 2007*, volume 07002 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[22] M. Harman, A. Baresel, D. Binkley, R. Hierons, L. Hu, B. Korel, P. McMinn, and M. Roper. Testability transformation — program transformation to improve testability. In R. Hierons, J. Bowen, and M. Harman, editors, *Formal Methods and Testing LNCS 4949*, chapter 11. Springer, 2008. to appear.

[23] M. Harman, D. W. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, Oct. 2003.

[24] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener. The impact of input domain reduction on search-based test data generation. In *ACM Symposium on the Foundations of Software Engineering (FSE '07)*, pages 155–164, Dubrovnik, Croatia, September 2007. Association for Computer Machinery.

[25] M. Harman and R. M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[26] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. 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.

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

[28] R. Hierons, M. Harman, and C. Fox. Branch-coverage testability transformation for unstructured programs. *The computer Journal*, 48(4):421–436, 2005.

[29] R. Hierons, M. Harman, and H. Singh. Automatically generating information from a Z specification to support the classification tree method. In 3rd *International Conference of B and Z Users (ZB2003)*, pages 388–407, Turku, Finland, June 2003. Springer. LNCS 2651.

[30] R. M. Hierons. Testing from a Z specification. *Journal of Software Testing, Verification and Reliability*, 7:19–33, 1997.

[31] R. M. Hierons and M. Harman. Testing conformance of a deterministic implementation against a non-deterministic specification. *Theoretical Computer Science*, 323(1-3):191–233, 2004.

[32] R. M. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[33] R. M. Hierons, T.-H. Kim, and H. Ural. On the testability of SDL specifications. *Computer Networks*, 44(5):681–700, 2004.

[34] M. Holcombe. What are X-machines? *Formal Asp. Comput*, 12(6):418–422, 2000.

[35] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8:371–379, 1982.

[36] K. N. King and A. J. Offutt. A FORTRAN language system for mutation-based software testing. *Software Practice and Experience*, 21:686–718, 1991.

[37] B. Korel, M. 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, Nov. 2005.

[38] J. Kort, R. Lämmel, and J. Visser. Functional transformation systems. In 9th *International Workshop on Functional and Logic Programming (WFLP'2000)*, Benicassim, Spain, Sept. 2000. Online proceedings at `http://www.dsic.upv.es/~wflp2000/`.

[39] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, Mar. 1966.

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

[41] P. McMinn, D. Binkley, and M. Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering and Methodology*. To appear.

[42] P. McMinn, D. Binkley, and M. Harman. Testability transformation for efficient automated test data search in the presence of nesting. In *UK Software Testing Workshop (UK Test 2005)*, Sheffield, UK, Sept. 2005.

[43] P. McMinn, M. Harman, D. Binkley, and P. Tonella. The species per path approach to search-based test data generation. In *International Symposium on Software Testing and Analysis (ISSTA 06)*, pages 13–24, Portland, Maine, USA., 2006.

[44] P. McMinn and M. Holcombe. The state problem for evolutionary testing. In *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 2488–2498, Berlin, 12-16 July 2003. Springer-Verlag.

[45] P. McMinn and M. Holcombe. Evolutionary testing of state-based programs. In H.-G. Beyer and U.-M. O'Reilly, editors, *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1013–1020. ACM, 2005.

[46] 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.

[47] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4:131–154, 1994.

[48] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5:99–118, 1996.

[49] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evalutation of data flow and mutation testing. *Software Practice and Experience*, 26:165–176, 1996.

[50] A. J. Offutt and R. Untch. Mutation 2000: Uniting the orthogonal. In W. E. Wong, editor, *Mutation Testing for the New Century (proceedings of Mutation 2000)*, pages 45–55, San Jose, California, USA, Oct. 2001. Kluwer.

[51] H. Partsch. *The CIP Transformation System*, pages 305–322. Springer, 1984. Peter Pepper (ed.).

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

[53] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In M. Wermelinger and H. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272. ACM, 2005.

[54] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[55] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis (ISSTA 98)*, pages 73–81, March 1998.

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

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

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

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

[60] M. R. Woodward. Mutation testing - its origin and evolution. *Information and Software Technology*, 35:163–169, 1993.

[61] Y. Zhan and J. A. Clark. The state problem for test generation in simulink. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1941–1948, Seattle, Washington, USA, 8-12 July 2006. ACM Press.