Testability Transformation

seminar for ForTest Network

Mark Harman
Brunel University

Joint work with
Rob Hierons, Lin Hu, Marc Roper and Joachim Wegener
Overview

- Test Data Generation
- Problems for Evolutionary Test Data Generation
- Testability Transformation
- Two Examples
Automatic Test Generation

Generating good quality test data is hard
Knowing what good quality means is hard
I do not propose to answer that question today

Starting point: structural test adequacy criterion
Specifically that some path or branch is to be covered
Structural Test Data Generation

There are five possible methods:

- Human analysis
- Random Testing
- Symbolic Execution
- Constraint Solving
- Evolutionary Testing

This talk focuses on Evolutionary Testing
But testability transformation applies elsewhere too
Evolutionary Testing

To execute a branch:

Define a **fitness function** for the predicate

Fitness function guides a **search** for test input

This has been shown to work well

... but there are problems
Problems with Evolutionary Testing

Program **structure** inhibits the fitness function formation

Examples of structure problems include:

- Side effects
- Unstructured control flow
- Flag variables
We are testing to cover structure
... but the structure is the problem
So we transform the program
... and this alters the structure
So a question arises:

Are we still testing according to the same criterion?
We need to co-transform the test adequacy criterion
Informally

A **transformation** is a partial function on programs.

We need to pair the program and test adequacy criterion
- call this the **test pair**

A **testability transformation** is a partial function on test pairs such that...
Testability Transformation

Test data which is adequate for the transformed test pair is adequate for the original test pair.
Trivial Example

Informally, this is already done:

"Branch coverage is MC/DC coverage when we expand out if statements"

```java
if (a && b)  
    s1;
else s2;
```

```java
if (a)  
    if (b) s1;
else s2;
else s2;
```
More Formally

**Definition 1 (Testing-Orientated Transformation)**

Let \( P \) be a set of programs and \( C \) be a set of testing criteria. A program transformation is a partial function in \( P \to P \). A Testing-Orientated Transformation is a partial function in \( (P \times C) \to (P \times C) \).

**Definition 2 (Testability Transformation)**

A Testing-Orientated Transformation, \( \tau \) 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 \( c' \).
Reversible Testability Transformations

A testability transformation only guarantees that sufficient test data will be generated to meet the original test adequacy criterion.

A Reversible Testability Transformation guarantees that test data generated is necessary and sufficient:

Definition 3 (Reversible Testability Transformation)
A testability transformation, \( \tau \) is a Reversible Testability Transformation iff its inverse is a testability transformation.
Examples

We now look at two examples

The first is particular to Evolutionary Testing

The second is a general problem in test data generation

The first illustrates how the adequacy criteria may need to change during Testability transformation

The second illustrates the way Testability Transformation may lead to novel transformations
The Flag Problem

Flag variables → ‘coarse fitness landscape’

Possibly a large **plateau** of low equal fitness
Possibly a **small plateau** of high equal fitness
No guide from low to high
Can not find high plateau

Worst case:
Evolutionary testing degenerates to random testing
Flag Removal Transformation

... 

\[ n' = n; \]
\[ \text{flag} = (n' \%2 == 0)?0:(n' < 4); \]

... 

if (a[i] != '0' && (n' \%2 == 0)?0:(n' < 4)) 

...

\[ \text{VAST} \]
Nothing New

These are all **standard transformations**

But we require a **change** in the **adequacy criterion**

Depends upon the interpretation of 'node of the CFG'

But test data:

  which is adequate for **MC/DC on the transformed**

  is adequate for **branch on the original**
Unstructuredness

Unstructured control flow presents problems

Seek transformation to single-entry/single-exit

Such a transformation is always possible

(Note: Due to Cooper not Böhm and Jacopini)

Unfortunately the approach is to introduce flags

... and to massively alter the structure
Equivalence

Definition 4 (Functional equivalence)
Program $p$ is *functionally equivalent* to program $q$ if they always produce the same output for the same input.

Definition 5 (Path equivalence)
Program $p$ is *path equivalent* (or strongly equivalent) to program $q$ if, for all inputs, the sequences of test and actions performed by the two programs are identical.

For us, path equivalence seems a natural choice.
Path Equivalence is restrictive

Knuth and Floyd: ‘regular expression flowchart semantics’

Regular expression captures possible paths through flowchart

gotos cannot always be removed under path equivalence

$R$ describes paths through structured programs

Hopcroft showed that

$$\sigma_1(\tau_{1F}\tau_{2F}\sigma_2)^*(\tau_{1T}\sigma_3 | \tau_{1F}\tau_{2T})\sigma_4$$

is not in $R$. 
Diagrammatically

This does not preserve (strict) path equivalence
Connection

This means that branch coverage of the transformed program corresponds to branch coverage for the original.

So here we do not need to co-transform the adequacy criterion but new concepts of equivalence and new transformations.

Conjecture:
In theory, we never need to co-transform the adequacy criterion.
Disposable Transformations

We generate test data using the transformed program because it is easier
... then throw away the transformed program

Transformation as a means to an end not an end in itself

Do the transformations even need to preserve meaning?
Conclusion

Test data generation is hard
   ...anything which helps is good
Test data generation can be impeded by structure
   ... so transform the structure
To avoid throwing baby out with bath-water
   ... also transform adequacy criterion
This allows the application of transformation to testability
   ... and the generation of new transformations
Future Work

Other non-meaning preserving transformation

Transformation as a means to an end

Would like branch coverage preserving transformation

Variable dependence preserving too

Other Preserving?

Implementation

flags - some results
side effects - done but no results
restructuring - to do

Testability Transformation Conjecture: post transform to preserve adequacy?