Improving Evolutionary Testing in the Presence of Function-Assigned Flags

Stefan Wappler
Technical University of Berlin
Ernst-Reuter-Platz 7
10587 Berlin, Germany
stefan.wappler@tu-berlin.de

André Baresel
QVI Tech GmbH
Ernst-Augustin-Str. 12
12489 Berlin, Germany
andre.baresel@qvitech.com

Joachim Wegener
Berner & Mattner
E.-v.-Kreibig-Str. 3
80807 Munich, Germany
joachim.wegener
@berner-mattner.com

Abstract

Evolutionary structural testing, an approach to automatically generating relevant unit test cases, encounters difficulties when the tested software contains boolean variables. This issue, known as the flag problem, has been studied by many researchers. However, previous work does not address the issue of function-assigned flags which constitutes a special type of the flag problem that often occurs in the context of object-orientation. This paper elaborates on a new approach to the flag problem that can also handle function-assigned flags while being applicable to the conventional flag problem, as well. It relies on a code transformation that leads to an improved fitness landscape which provides better guidance to the evolutionary search. We present four case studies including a fitness landscape analysis and empirical results. The results show that the suggested code transformation improves evolutionary structural testing in the presence of function-assigned flags.

1. Introduction

Evolutionary structural testing [5, 8, 11, 9, 10] has been shown to be an effective approach to automatically generating test cases that achieve high code coverage. It interprets the task of test case generation as a search problem to be solved using an evolutionary algorithm.

However, it has turned out that the evolutionary search might fail if the software unit under test makes use of particular code constructs. One of these code constructs is the usage of flag (or boolean) variables within conditions. This issue is known as the flag problem and has been studied by various researchers (e.g. [2, 3, 1, 4, 6]).

However, although the efficacy of the proposed approaches could be demonstrated for various test objects, their applicability is limited to flag problems that do not involve function calls meaning that the flag value is assigned the return value of a function. We call this code construct *function-assigned flag*. Especially in the context of object-orientation, function-assigned flags occur very frequently.

The help of the previous approaches to the flag problem is very limited in these cases. Therefore, a more general approach is demanded which is able to deal with function-assigned flags.

This paper presents a novel approach to the flag problem that is also applicable when the flag value is assigned via a function call. It also applies to object-oriented code where the concept of using the boolean return value of a method call as a predicate is widely used. We describe a code transformation that replaces all boolean-type flag variables with real-type variables and introduces local fitness functions. We demonstrate that the approach is successful on the basis of four case studies using C and Java test objects. Finally, we present the results of experiments with the test objects of the case studies. These results show that the evolutionary test case generator was able to successfully generate test cases for the transformed programs while it was not for the original programs.

This paper is structured as follows: section 2 describes the main idea of evolutionary structural testing, section 3 introduces the issue of function-assigned flags in more detail, section 4 presents our new code transformation, and section 5 investigates four case studies. section 6 concludes the paper and highlights directions for future research.

2. Evolutionary Structural Testing

In the area of software testing, the process of generating relevant test cases for a given software unit is considered to be the most important task. Done manually, it is time-consuming and error-prone; hence, it is very costly. Evolutionary structural testing [5, 8, 11, 9, 10] is one approach to automating the process of test case generation. The task to generate appropriate input data that leads to the execution of a particular program element is formulated as an

0-7695-2984-4/07 $25.00 © 2007 IEEE
DOI 10.1109/TAIC.PART.2007.25

optimization problem which is tried to be solved using an evolutionary algorithm.

An evolutionary algorithm is a meta-heuristic optimization technique that mimics the principles of the Darwinian theory of biological evolution. Figure 1 shows the workflow of a simple evolutionary algorithm. At first, a set of candidate solutions for the given optimization problem is generated at random. This set is called *population* and will be modified iteratively later on in order to find an ideal solution. In the next step, the candidate solutions are evaluated using a *fitness function*. This function assigns each candidate solution a quality value which correlates to the ability of the candidate solution to solve the optimization problem. Since the creation of new individuals depends on the assigned fitnesses, the fitness function is the most critical part of an evolutionary algorithm and decides on the success of the optimization. Once the candidate solutions are evaluated, they undergo two modification steps if no ideal solution is found initially. These two modification steps are $crossover$ and $mutation$. For crossover, two candidate solutions are selected according to their fitness. Those having a better fitness are selected with a higher probability. The selected candidate solutions are recombined in order to produce two offspring candidate solutions that are similar to their "parents" and expected to solve the optimization problem in a better way. The offspring candidate solutions are then mutated meaning that some of its parts are slightly changed at random. Finally, the new set of candidate solutions is evaluated again. This cycle of crossover, mutation, and evaluation, is repeated unless a termination criterion, such as the ideal solution is found or particular resources are exhausted, is satisfied.

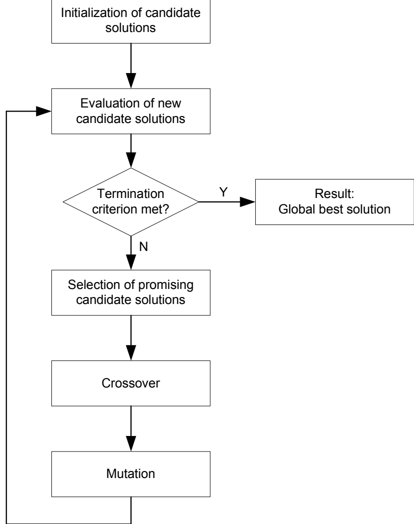

Figure 1. Workflow of an evolutionary algorithm

When applied to evolutionary structural testing, the evolutionary algorithm aims at generating a test case that covers a particular program element, the so-called *test goal*. Relevant program elements typically are statements or branches of the program under test, depending on the used test adequacy criterion. For instance, if the test case generator is supposed to provide a set of test cases that reach high branch coverage – as required by many industrial testing standards –, an individual evolutionary search for a covering test case for each program branch is carried out. The entirety of test cases generated by the evolutionary algorithm is then eventually provided as final test suite.

Each test goal requires the definition of an individual fitness function. It has proven of value to base the fitness functions on the distance of the execution flow produced by a candidate solution to the targeted program element. Two metrics have become popular for defining this distance: approximation level and branch distance [11]. Approximation level is defined in terms of the control flow graph of the program under test. It measures the number of potential problem nodes of the shortest path from the problem node to the targeted program element where the problem node is that node of the control flow graph at which execution diverged down a branch (the *critical branch*) which makes it impossible to ever reach the target. Branch distance relates to the condition assigned to the problem node. It expresses how "close" execution was to taking the opposite branch and hence avoiding the critical branch. For each relational operator occurring in a condition, a particular distance function will be applied [11]. For instance, in case of a condition `if( a==100 )` the distance function for the equality operator is defined as $d = |a - 100|$, mapped into the range $[0, 1]$. The evolutionary search tries to minimize the approximation level and branch distance where 0 indicates that the candidate test case covered the test goal of interest.

3. The Problem of Function-Assigned Flags

This section describes the issue of the flag problem in general and that of function-assigned flags in particular. First, an overview of the flag problem and the proposed solutions is given. Then, the problem of function-assigned flags is focused on. We also show that previous approaches do not handle function-assigned flags effectively.

3.1. The Flag Problem

The flag problem relates to program variables that are used in a boolean manner and cannot be accessed from outside the program. The latter is especially true for local variables. The following listing shows a simple flag problem:

```
int func(int a)
{
  int flag = 0;

  if( a == 0 )
    flag = 1;

  if( flag )
    return 0;
  else
    return 1;
}
```

In this program, the local variable $flag$ is used as a boolean variable that takes only the values 0 and 1. It cannot be accessed from outside the program since it is a local variable. However, seen in the context of evolutionary structural testing, the true branch of the condition `if( flag )` constitutes a test goal for which a covering test case is hard to find when using the traditional branch distance. This difficulty is due to the nature of the distance function which is a binary function in this case: either, $flag$ is true, then, the branch distance is 0.0; or, $flag$ is false and hence the branch distance is 1.0. The fitness function for the test goal would not provide any useful guidance to the evolutionary algorithm. Any input value different from 0 would receive the same (bad) fitness 1.0.

Various approaches have been proposed to address the flag problem: the flag removal algorithm of Harman et al. [4] relies on a code transformation that in principle substitutes the flag within a condition by the flag-defining expression. Bottaci [3] describes the general idea of using the distance of the conditions that control a flag assignment as a replacement distance for the distance of the condition that involves the flag. He introduces additional variables storing these intermediate distance values. Baresel and Sthamer [2] perform a static analysis in order to classify all occurring flag assignments into the categories $desired$ and $undesired$. They include the conditions controlling the assignments into the fitness calculation: the negated conditions controlling the undesired assignments are added to the flag condition which is used to calculate the branch distance. In [1], Baresel et al. deal with the issue of loop-assigned flags. They also classify the flag assignments according to desired and undesired. Additionally, they introduce local fitness functions and additional local variables. They reformulate the flag condition using the additional variables. Liu et al. [6] introduce a fitness variable for each flag variable - the value of which is calculated with the help of aggregation rules. They also classify the flag assignments according to desired and undesired. Whenever a flag condition is to be satisfied, the fitness value of the additionally introduced fitness variables is used.

McMinn and Holcombe [7] combine the idea of *chaining* and evolutionary testing to cope with the problem of internal variables and internal states, including the flag problem. A transitive static analysis identifies all statements that manipulate the variables controlling a flag assignment. Chains of such definitions are considered systematically to define the objective function that then rewards the execution of definition statements that eventually facilitate the desired flag assignment.

In general, most of the previous approaches incorporate the knowledge of what a desired and what an undesired flag assignment is into a code transformation or calculation rule. They also require the conditions controlling the calculation of a flag value to be present in the context of the considered function meaning that the information as to which condition controls a flag assignment is statically accessible. As we will see in the next section, these properties limit the applicability of the approaches when function-assigned flags are involved.

3.2. Function-Assigned Flags

A function-assigned flag is a flag which receives its value via a function call. The following two code fragments include both a simple function-assigned flag:

```
// fragment A
if( stack.isFull() )
  // target

// fragment B
int DOOR_CLOSED = is_door_closed();
if( DOOR_CLOSED )
  // target
```

Actually, no flag variable is involved in code fragment A. However, one can think of an implicit local variable that is assigned the return value of the $isFull$ method. Function-assigned flags occur relatively often in object-oriented programs. Using the boolean return value of a method call directly as a predicate is a popular programming practice. Function-assigned flags also occur in procedural programs, especially when the programming language offers the boolean data type. Code fragment B shows the integer variable $DOOR_CLOSED$ used in a boolean manner.

However, previous approaches dealing with the flag problem would not provide an improvement for evolutionary structural testing in these cases since they would not change the fitness landscape. For instance, the approach of [4] would transform code fragment B to the following:

```
int DOOR_CLOSED = is_door_closed();
if( is_door_closed() )
  // target
```

Besides the danger that the *is_door_closed* function possesses side effects and multiple calls might affect the state of the system – leading in the worst case to a change in the behavior of the system –, the transformed version does not improve the fitness landscape. The approaches based on the

knowledge of desired and undesired flag assignments would either not have such information available or need to perform a possibly transitive analysis exceeding the function boundary. However, in the context of object-orientation, an extended static analysis of flag variable assignments tends to be hindered due to the dynamic function binding (polymorphism), meaning that it is not always apparent which predicate is to be used as a replacement for the flag condition. This information is available at runtime of the program only when the effective method binding will be detected. For instance, consider the following two implementations of the $isFull$ method of two stack classes:

```
class SimpleStack implements IStack {
  ...
  boolean isFull() {
    if( elements.size() == MAX_SIZE )
      return true;
    else return false;
  }
}

class ExtendedStack implements IStack {
  ...
  boolean isFull() {
    if( keepInLimit() )
      if( elements.size() == MAX_SIZE )
        return true;
      else return false;
    else return false;
  }
}
```

The first class, $SimpleStack$, compares the number of currently stored elements to a maximum size bound in order to calculate the return value of method $isFull$. The second class, $ExtendedStack$, allows exceeding the maximum size unless it is configured to keep the size within the limit. With respect to code fragment A, it depends on the runtime type of variable $stack$ – which is supposed to be either $SimpleStack$ or $ExtendedStack$ – which predicates must be incorporated into the flag fitness calculation: in the first case, the predicate (elements.size()==MAX_SIZE) is relevant, whereas in the latter case, both predicates (keepInLimit()) and (elements.size()==MAX_SIZE) are relevant.

The applicability of the chaining approach to object-oriented software is hindered by dynamic binding and by the fact that it does not regard the object identities for which the definitions of relevant variables occur.

4. A New Code Transformation

Code transformations are a means to creating a modified version of a given program with the intention that the modified version is more suited for a particular purpose. In the context of evolutionary testing, several code transformations have been suggested. The purpose of these transformations is to provide better guidance to the evolutionary search for appropriate test data since the transformed program is supposed to imply a smoother fitness landscape that possesses fewer plateaus, less discontinuous changes and less local optima than the fitness landscape implied by the original program. A code transformation consists of the application of a set of transformation rules, so-called $tactics$. In the following, we describe the 3 tactics that constitute our suggested transformation.

4.1. Tactic 1: Branch Completion

This tactic completes all "invisible" else branches of all conditions and adds tautological flag assignments. Table 1 shows a sample program on the left. On the right, it shows the same program after tactic 1 has been applied. All branches have been completed and tautological assignments of the flag under question have been inserted. The intention of this tactic is to make a flag assignment occur regardless of the control path taken during execution of the program. This tactic in combination with tactic 3 ensures that a guiding distance value can always be calculated. The scope of this tactic are all conditions that control a flag assignment. If there are nested conditions, all parent conditions are also considered and their respective else branches are also expanded.

<table>
<tr><th>original program</th><th>tactic 1 applied</th></tr>
<tr><td><pre>
void
func1(int a, int b)
{
  int flag = 0;
  if(a == 0)
    flag = func2(b);

  if(flag)
    // target
}

int func2(int b)
{
  if(b==0)
    return TRUE;
  else
    return FALSE;
}
</pre></td><td><pre>
void
func1(int a, int b)
{
  int flag = 0;
  if(a == 0)
    flag = func2(b);
  else
    flag = flag;

  if(flag)
    // target
}

int func2(int b)
{
  if(b==0)
    return TRUE;
  else
    return FALSE;
}
</pre></td></tr>
</table>

Table 1. tactic 1

4.2. Tactic 2: Data Type Substitution

This tactic substitutes the data type $boolean$ (or int when used in a boolean manner) with the data type $double$ in all flag declarations. This comprises not only local variable declarations but also the return type of functions that are supposed to return a flag (boolean) value. The latter occurs

quite often with the object-oriented paradigm. For example, the use of $is*$ methods or $has*$ methods is relatively common.

The data type substitution is the most essential tactic of our code transformation. It assumes the value 1.0 to be “maximum true” and the value -1.0 to be “maximum false”. Positive values indicate that the flag is $true$ whereas negative values indicate that it is $false$. By convention, 0 is considered to be a positive value. The flag’s amount expresses how far it is away from being the opposite value. For example, a flag value of -0.5 indicates that the flag is $false$ and is 0.5 “away” from being true. The distance that the flag value represents is later used to calculate the fitness value. The gradual values (that substitute $true$ or $false$) produce a smooth fitness landscape which guides the evolutionary search well as opposed to the original fitness landscape with the large boolean plateau.

The data type substitution also requires that all conditions involving the flag are modified. In case of a comparison of the flag value to true, e.g. (flag == true), the predicate would be changed to (flag >= 0). Hence, the modified predicate makes use of a relational operator defined for real values instead of the equality operator defined for boolean values. Analogously, a comparison to false would be changed to (flag < 0). Note that short-hand predicates, such as $(flag)$ would be completed to $(flag == true)$ first before applying the modification.

The tactic also demands a special treatment of the negation operator. All occurrences of it are replaced by the $-$ operator which is the negation operator for real values. For instance, a condition if(!flag) would then be modified to if(−flag >= 0)[1].

4.3. Tactic 3: Local Instrumentation

The local instrumentation is intended to assign gradual distance values to the flag variables. Therefore, each right-hand operator of a flag assignment will be instrumented meaning that the actual right-hand expression is replaced by a call to a distance function. Table 3 shows the application of this tactic. As can be seen, the constants have been replaced by calls to function $dist$ which returns the distance for the expression passed to it. The angle brackets used for the second argument passed to the dist function should mean that the formal expression as well as the actual values of the concerned variables are passed. This short-hand notation will be used throughout the remainder of this paper. The call of $func2$ has been replaced by a call to function map.

The following listing shows the pseudo-code of the $dist$ function.

[1] The case of a zero flag value requires special treatment: the minimum step width value of the data type is added to the flag value before negation.

tactic 1 applied	+ tactic 2 applied
void	**void**
func1(**int** a, **int** b)	func1(**int** a, **int** b)
{	{
int flag = 0;	**double** flag = −1;
if(a == 0)	**if**(a == 0)
flag = func2(b);	flag = func2(b);
else	**else**
flag = flag;	flag = flag;
if(flag)	**if**(flag >= 0)
// target	*// target*
}	}
int func2(**int** b)	**double** func2(**int** b)
{	{
if(b==0)	**if**(b==0)
return TRUE;	**return** 1;
else	**else**
return FALSE;	**return** −1;
}	}

Table 2. tactic 2

tactic 2 applied	+ tactic 3 applied
void	**void**
func1(**int** a, **int** b)	func1(**int** a, **int** b)
{	{
double flag = −1;	**double** flag = −1;
if(a == 0)	**if**(a == 0)
flag = func2(b);	flag = map(
	func2(b),
	2);
else	**else**
flag = flag;	flag = dist(flag,
	<a==0>, 1);
if(flag >= 0)	**if**(flag >= 0)
// target	*// target*
}	}
double func2(**int** b)	**double** func2(**int** b)
{	{
if(b==0)	**if**(b==0)
return 1;	**return** dist(1,
	<b==0>,1);
else	**else**
return −1;	**return** dist(−1,
	<b==0>,1);
}	}

Table 3. tactic 3

```
double dist(double assignedValue, expression exp,
            int nestingLevel) {
  double distance = branch_dist(exp);
  distance = map(distance, nestingLevel);

  if( assignedValue < 0 )
    distance = -distance;
  return distance;
}
```

Initially, the conventional branch distance will be calculated based on the passed expression. This calculation depends on the applied relational operator; for each operator, a particular distance function is designed [11]. Then, the distance is mapped to a particular range using the map function. This function realizes the idea of interval bisection which can be regarded as the inversion of the approximation level approach. Interval bisection allows integrating multiple information into one real value. In our case this information consists of the actual branch distance of the condition that controls the flag assignment and the nesting level. The nesting level of a statement corresponds to the number of conditions that control this statement. We use the nesting level instead of the approximation level since the latter cannot be calculated for the local fitness function unambigously using static analysis. This is due to the dynamic function binding as mentioned in section 3.2. The number of nesting levels may differ from function call to function call depending on how many levels the called function possesses.

Formula 1 shows the relationship between the original distance (d_{orig}) and the resulting mapped distance (d_{mapped}) where l is the nesting level. The map function implements this formula.

$$d_{mapped} = sign(d_{orig}) \frac{1 + |d_{orig}|}{2^l} \quad (1)$$

We will explain the idea of interval bisection using an example. Test data for testing $func1$ consists of a pair (a, b) of integer input data. Table 4 shows the nesting level, the branch distance, the flag value, and a graphical representation of the absolute amount of the flag value (called $interval$) that the test inputs $(1, 1)$, $(1, 0)$, $(0, 1)$, and $(0, 0)$ would achieve when being used as inputs for $func1$. The

(a,b)	nesting level	branch distance	flag value	interval
(1,1)	1	0.0005	-0.5002	0 0.5002 1
(1,0)	1	0.0005	-0.5002	0 0.5002 1
(0,1)	2	0.0005	-0.3751	0 0.3751 1
(0,0)	2	0.0005	+0.3750	0 0.3750 1

Table 4. Example test inputs for $func1$ and the resulting flag value

test inputs $(1, 1)$ and $(1, 0)$ do not satisfy the first condition of $func1$, hence leading to the traversal of the alternative branch and achieving nesting level 1. The branch distance 0.0005, calculated by the appropriate distance function [11], indicates how close execution was to evaluating the first condition to $true$. The flag value is negative in these cases indicating a $false$ flag outcome. The corresponding intervals in table 4 show a solid lower half which can be regarded as a reserved area for higher nesting levels. The branch distance of 0.0005 was mapped to the upper half, resulting in the absolute flag value 0.5002. The test input $(0, 1)$ satisfies the first condition of function $func1$ and leads to the traversal of the alternative branch of the condition in function $func2$, hence achieving a nesting level of 2. The miss of the $true$ branch of this condition is taken into account by the branch distance of 0.0005. As the interval shows in this case, the lower part is bisected as compared to the first two intervals, and the branch distance is mapped into the upper one of these new halfes. Finally, test input $(0, 0)$ satisfies both conditions and leads to an assignment of $true$ to the flag variable. Therefore, the sign of the flag value is positive, indicating the $true$ value. The absolute value of the flag indicates how close execution was to avoiding the $true$ outcome.

5. Case Studies

In this section, we investigate four test objects in more detail in order to demonstrate the suggested code transformation. The first two case studies, simple flag and nested flag, include a simple flag problem. We include them to show that the transformation also works for simple flag problems. The third and fourth case studies, function-assigned flag and a real-world example, include function-assigned flags. In the following, we describe the test objects in short and present both the original program and the transformed program. For each program, we show the fitness landscape for the two test goals that a flag condition controls (the true branch (denoted as *target A*) and the false branch (denoted as *target B*)).

5.1. Simple Flag

The simple flag represents a single flag assignment controlled by an atomic condition. No function call is involved in the flag assignment. Table 5 shows the original program on the left, and the transformed program on the right. As the table shows, the fitness landscape for target A consists of a huge plateau that does not provide useful guidance as opposed to the smooth landscape produced by the transformed version.

The landscapes for target B are very similar: both possess exactly one peak at point 0, the only point where the

original program	transformed program

void simple_flag(int a)
{
  int flag = 0;
  if(a == 0)
    flag = 1;
 	
  if(flag)
    // target A
  else
    // target B
}
	void simple_flag_t(int a)
	{
	  double flag = −1;
	  if(a == 0)
	    flag = dist(1, <a == 0>, 1);
	  else
	    flag = dist(flag, <a == 0>, 1);
	
	  if(flag >= 0)
	    // target A
	  else
	    // target B
	}

Target A (fitness vs. a) · Target B (fitness vs. a)	Target A (fitness vs. a) · Target B (fitness vs. a)

Table 5. simple flag

test goal is not achieved. The landscape of the original program is not problematic since the false branch is easily reachable.

5.2. Nested Flag

The nested flag contains a flag assignment that occurs when two conditions are both satisfied. Table 6 shows the programs and fitness landscapes related to this test object. The fitness landscape of the original program consists of a single plateau with one hole at the optimum position $[0, 0]$ for target A. In contrast, the landscape of the transformed program provides helpful guidance to the search algorithm. It guides the search to first let variable a be 0, and afterwards let variable b be 0. Like for the simple flag, target B is unproblematic.

5.3. Function-Assigned Flag

This test object is similar to the nested flag test object. Here, the nested condition has been moved into an extra function called $callee$. Table 7 shows the programs and fitness landscape for this test object. It can be seen that the mapping of the returned fitness value leads to a gradual fitness landscape without plateaus. The function call does not imply any unfavorable information loss.

5.4. Real-World Flag Example

We applied the code transformation to class Stack written in Java. This class implements a simple stack allowing to add and remove elements and to query whether it is empty or full. The maximum capacity of the stack has been set to 10 elements which is an arbitrary choice. The following listing shows the source code of class Stack, followed by the corresponding transformed version.

```
public class Stack {
  private static int MAX_ELEMENTS = 10;
  private Object[] elements;
  private int freeIndex;

  public Stack() {
    elements = new Object[MAX_ELEMENTS];
    freeIndex = 0;
  }
  public void add(Object element) {
    if( isFull() ) throw new Exception();
    elements[freeIndex++] = element;
  }
  public Object removeTop() {
    if( isEmpty() ) throw new Exception();
    return elements[--freeIndex];
  }
  public boolean isFull() {
    if( freeIndex >= MAX_ELEMENTS ) return true;
    else return false;
  }
  public boolean isEmpty() {
    if( freeIndex == 0 ) return true;
    else return false;
  }
}


public class Stack_t {
  // attributes and ctor like original Stack

  public void add(Object element) {
    if( isFull() >= 0 ) throw new Exception();
    elements[freeIndex++] = element;
  }
```

original program

```
void nested_flag(int a, int b)
{
  int flag = FALSE;

  if( a == 0 )
  {

    if( b == 0 )
      flag = TRUE;


  }



  if( flag )
    // target A
  else
    // target B
}
```

transformed program

```
void nested_flag_t(int a, int b)
{
  double flag = −1;

  if( a == 0 )
  {
    flag = dist(flag, <a==0>, 1);
    if( b == 0 )
      flag = dist(1.0, <b==0>, 2);
    else
      flag = dist(flag, <b==0>, 2);
  }
  else
    flag = dist(flag, <a==0>, 1);

  if( flag >= 0 )
    // target A
  else
    // target B
}
```

original program		transformed program	
Target A (fitness, a, b)	Target B (fitness, a, b)	Target A (fitness, a, b)	Target B (fitness, a, b)

Table 6. nested flag

original program

```
int callee(int b)
{
  if( b == 0 )
    return TRUE;
  else
    return FALSE;
}

void fa_flag(int a, int b)
{
  int flag = FALSE;
  if( a == 0 )
    flag = callee( b );



  if( flag )
    // target A
  else
    // target B
}
```

transformed program

```
double callee_t(int b)
{
  if( b == 0 )
    return dist( 1.0, <b==0>, 1 );
  else
    return dist( 0.0, <b==0>, 1 );
}

void fa_flag_t(int a, int b)
{
  double flag = FALSE;
  if( a == 0 )
    flag = map( callee_t(b), 2 );
  else
    flag = dist( flag, <a==0>, 2 );

  if( flag >= 0 )
    // target A
  else
    // target B
}
```

original program		transformed program	
Target A (fitness, a, b)	Target B (fitness, a, b)	Target A (fitness, a, b)	Target B (fitness, a, b)

Table 7. function-assigned flag

```
  public Object removeTop() {
    if( isEmpty() >= 0 ) throw new Exception();
    return elements[--freeIndex];
  }
  public double isFull() {
    if( freeIndex >= MAX_ELEMENTS )
      return T.dist( T.TRUE,
        <freeIndex>=MAX_ELEMENTS>, 1 );
    else return T.dist( T.FALSE,
      <freeIndex>=MAX_ELEMENTS>, 1 );
  }
  public double isEmpty() {
    if( freeIndex == 0 )
      return T.dist( T.TRUE,
        <freeIndex==0>, 1 );
    else return T.dist( T.FALSE,
      <freeIndex==0>, 1 );
  }
}
```

Note that the transformation also modified the return types of the two methods. Also note that the methods provided by class T, which appears in the transformed methods, implement the same algorithms as described in section 4.3.

We do not present a visualization of the fitness landscapes since they are high-dimensional and are not based on Euclidean scales. However, the experimental results provided in the next section show that the transformation created fitness landscapes that helped finding covering test cases.

5.5. Experimental Results

We performed a series of experiments with the described case studies for the purpose of empirical validation. We used the DaimlerChrysler evolutionary testing system [11] to carry out the experiments for the test objects written in C. For the experiments with the Java test object, we used the DaimlerChrysler prototypic Java test system [10].

We ran two series of experiments for each test object: once, we used the original version and repeated the generation process 10 times; then, we used the transformed version with the same number of repetitions. We restricted the value range for integers to $[-10e5, +10e5]$.

Table 8 shows the results of the experiments. We only report on the true branch of the flag condition (target A; for Stack the first true branch of method add). The first column names the test objects. The second and fourth column shows the success rate (SR) of the 10 runs. Success rate is the quotient of the number of successful runs and the total number of runs (i.e. 10). The third and fifth column shows the average number of fitness function evaluations (FE). The value in parenthesis is the standard deviation. We configured the evolutionary search to terminate at least after 200 generations. In case of the procedural test objects, the evolutionary algorithm used 240 individuals within 6 subpopulations and a generation gap of 0.9. In case of the Stack test object, the search employed 50 individuals. These settings have turned out to be adequate during preceding experiments. They explain the difference between the worst-case evaluation values of the procedural and object-oriented test objects in the second column. As the table shows, the test

test object	original version		transformed version	
	SR	FE (σ)	SR	FE (σ)
simple flag	0	43224 (0)	1	2694 (919)
nested flag	0	43224 (0)	1	23661 (5794)
f.-assigned flag	0	43224 (0)	1	20800 (4904)
Stack	0	10000 (0)	1	1261 (786)

Table 8. experimental results

systems were not able to generate a test case for covering the interesting branches in case of the original programs. In contrast, a covering test case was found in all cases when the transformed program was being used.

We also analyzed the development of the fitness values over the generations for the interesting test goals in order to examine whether the code transformation enabled the evolutionary algorithm to actually perform an optimization, and, if it does, how the evolutionary search behaves. Figures 2 to 5 show the best fitness values over the generations for the 10 runs. The thick graphs represent the best fitness averaged over the 10 runs. It can be observed that the fitness values incrementally improve which indicates that an

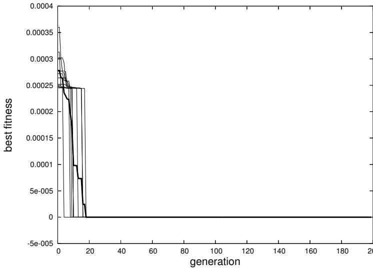

Figure 2. Fitness development for simple flag

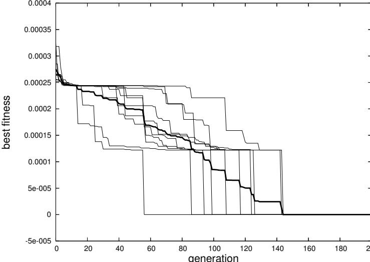

Figure 3. Fitness development for nested flag

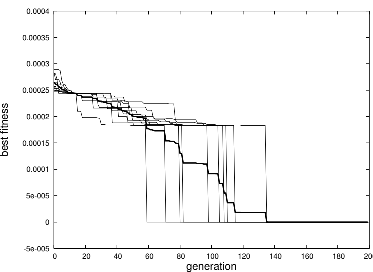

Figure 4. Fitness development for function-assigned flag

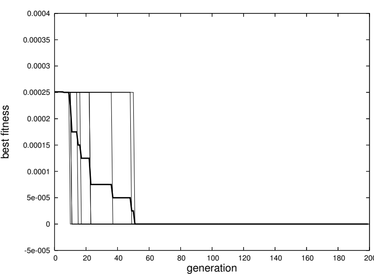

Figure 5. Fitness development for Stack

optimization process took place. In case of the Stack test object, the initial fitness improvements are very small and cannot be easily seen in the figure. The final jumps of each graph of the single runs are due to the change of the sign of the flag value: once the last relevant condition is satisfied, the sign of the flag value is inverted. Consequently, the distance function of the $>=$ operator returns the distance value 0, causing the best fitness to immediately change to 0.

6. Conclusion and Future Work

This paper presented a code transformation that improves evolutionary structural testing in the presence of function-assigned flags. Flags are an obstacle to the evolutionary search since they lead to plateaus in the fitness landscape that might not provide sufficient guidance in finding an optimum solution. One essential tactic of the transformation is to substitute boolean variables by double-type variables. Local fitness functions are used to provide gradual distance values. The suggested transformation is applicable to procedural and object-oriented software. On the basis of 4 case studies, we demonstrated the efficacy of the transformation. In experiments, the evolutionary search was able to find covering test cases for the transformed programs while it failed to do so for the original programs.

In general, we expect the presented transformation approach to be applicable to and helpful for the majority of flag problems, especially in the context of object-orientation. However, we have not investigated its suitability in case of neither serially-assigned flags nor loop-assigned flags in more detail. Additionally, the approach might benefit from further experimentation with more test objects and their analysis.

7. Acknowledgement

We would like to thank Harmen Sthamer for the valuable comments and discussions. This work was supported by EU grant 33472 (EvoTest).

References

[1] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, July 2004.

[2] A. Baresel and H. Sthamer. Evolutionary testing of flag conditions. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 2442–2454, July 2003. July 2003.

[3] L. Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, July 2002.

[4] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, July 2002.

[5] B. F. Jones, H. Sthamer, and D. E. Eyres. Automatic test data generation using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, Sept. 1996.

[6] X. Liu, H. Liu, B. Wang, P. Chen, and X. Cai. A unified fitness function calculation rule for flag conditions to improve evolutionary testing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*. ACM Press, 2005.

[7] P. McMinn and M. Holcombe. Evolutionary testing using an extended chaining approach. *Evolutionary Computation*, 14(1):41–64, 2006.

[8] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[9] P. Tonella. Evolutionary testing of classes. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 119–128, New York, NY, USA, 2004. ACM Press.

[10] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm. In *Proceedings of the IEEE World Congress on Computational Intelligence (WCCI-2006)*, pages 3193–3200, Vancouver, Canada, July 2006. IEEE Press.

[11] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(1):841–854, 2001.