Hybridizing Evolutionary Testing with the Chaining Approach

Phil McMinn and Mike Holcombe

Department of Computer Science, The University of Sheffield,
Regent Court, 211 Portobello Street,
Sheffield, S1 4DP, UK
{p.mcminn, m.holcombe}@dcs.shef.ac.uk

Abstract. Fitness functions derived for certain white-box test goals can cause problems for Evolutionary Testing (ET), due to a lack of sufficient guidance to the required test data. Often this is because the search does not take into account data dependencies within the program, and the fact that some special intermediate statement (or statements) needs to have been executed in order for the target structure to be feasible. This paper proposes a solution which combines ET with the Chaining Approach. The Chaining Approach is a simple method which probes the data dependencies inherent to the test goal. By incorporating this facility into ET, the search can be directed into potentially promising, unexplored areas of the test object's input domain. Encouraging results were obtained with the hybrid approach for seven programs known to originally cause problems for ET.

1 Introduction

Evolutionary Testing (ET) (see Ref. [1]) uses evolutionary algorithms to search for software test data. For white-box testing coverage criteria, the execution of each uncovered structure - for example a program statement or branch - is taken individually as the goal of the search. However, the coverage of certain structures in certain types of programs can cause problems for ET, due to a lack of sufficient guidance to the required test data. Often this is because the technique does not take into account data dependencies between statements in the program. In order for the target structure to be feasible, special intermediate statements may need to have been executed. For instance, the outcome of a target branching condition may be dependent on a variable having a special value, which is only set in a special circumstance - for example a special flag or enumeration value denoting an unusual condition; a unique return value from a function call indicating an error has occurred; or a counter variable only incremented under certain situations. If the intermediate assignment is particularly unusual, it will not be executed by chance. Without specific knowledge of such dependencies, the evolutionary search tends to stagnate and fail. The problem of flag variables in particular has received much interest from researchers [2,3,4]

K. Deb et al. (Eds.): GECCO 2004, LNCS 3103, pp. 1363–1374, 2004.

- however there has been little attention with regards to the broader problem as described.

This paper proposes a solution which combines ET with the Chaining Approach. The Chaining Approach [5,6] is a structural test data generation technique based on local search. If the local search fails to find test data which directly executes the target, data flow analysis is used to identify intermediate statements, or *events*, which can decide whether the target will be reached or not. By instead focusing on test data that executes a *chain* of events, the search can be directed into potentially unexplored, yet promising areas of the test object's input domain. In the hybrid approach, evolutionary algorithms are applied for the test data search. However, a chaining mechanism is employed whenever a "problematic" test goal is encountered. Encouraging results were obtained with initial experiments carried out on seven test objects known to originally cause problems for ET.

2 Evolutionary Testing (ET)

ET (see Ref. [1]) uses evolutionary algorithms to search the input domain of a test object for desired test data. Individuals of the search are simply input vectors to the test object. The fitness function depends on the type of test data required.

White-box testing coverage criteria demand that all program structures of a certain type - for example statements or branches - are exercised. ET can automate the derivation of test data for this purpose by searching for input vectors which will execute each specific structure. In more developed approaches [7,8], the fitness function is made up of two components. The first component is referred to as the *approximation level*, or, (perhaps more appropriately) the *approach level*. This metric assesses how close an individual was to reaching the target on the basis of its execution path through the program's control structure. Central to this is the notion of a *critical branch* (also referred to in the literature as a *decisive branch*). A critical branch is simply a branch which leads to the target being missed. Once such a branch has been taken, failure to reach the target has essentially been "decided". For improved handling of structures nested within loops [8], a branch that misses the target within a loop iteration is also counted as critical. For the example of Figure 1, where the goal of the search is the execution of node 6, the set of critical branches includes the true branch from node 1 and the false branch from node 4. The false branch from node 5 is also treated as critical, since it misses node 6 within a loop body. The approach level for an individual is calculated by subtracting one from the number of critical branches lying between the node from which the individual diverged away from the target, and the target itself. For the execution of node 6, individuals taking the false branch at node 5 receive an approach level of zero, individuals diverging away down the true branch at node 4 receive an approach level of one, and so on.

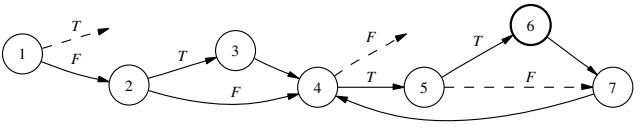


Fig. 1. An example control flow graph for calculating approach levels, with respect to the target - node 6. Critical branches are indicated with dashed arrows

At the point at which control flow diverged away from the target down a critical branch, a *branch distance calculation* is computed. This is the second component of the fitness function. This value indicates how close the alternative branch was to being taken. For example if the false branch is taken from node 5, and the branching condition at this node is `(x == y)`, the distance for the true branch is calculated using the formula `abs(x - y)` [9]. The overall (minimizing) fitness function is zero if the target structure is exercised, otherwise, a fitness value is computed for the individual on the basis of the approach level and the branch distance calculation d mapped into the range $0 \leq d < 1$:

$$approach\_level + d \tag{1}$$

3 The Problem

ET has been shown to be an effective method for structural test data generation [7]. However, the approach performs poorly for programs with certain characteristics. One problem can arise as a result of data dependencies within the program, where the target structure requires the prior execution of certain intermediate statements in order for it to be feasible. Since such statements do not affect whether the target structure will be reached in terms of control flow through the program, they are effectively ignored for the purposes of computing fitness information. If such statements are only executed under "special" circumstances rather than by chance, the target structure might not be covered.

Take the example of Figure 2. The `inverse` function finds the multiplicative inverse of an input `x`. To avoid a division by zero error, zero is simply returned when the divisor is zero. If the goal of the search is to execute node b, it is necessary to execute the true branch from node a - requiring a zero return value from the `inverse` function. This is in turn passed the input value `x` as an argument. However, the zero input value required to execute the target is unlikely to be found by chance, simply because it represents a very small portion of the overall input domain. Furthermore, the fitness function actually leads the search away from this value for all other input values - since as the value of `x` increases, the result of the `inverse` function decreases. This deception can be seen in a plot of the fitness function landscape (Figure 4a).

```
    double function_under_test(double x)
    {
(a)     if (inverse(x) == 0)
(b)         // target
    }
```

```
    double inverse(double d)
    {
(c)     if (d == 0)
(d)         return 0;
        else
(e)         return 1 / d;
    }
```

Fig. 2. C example resulting in a deceptive fitness function into order to execute node e (control flow graph node identifiers appear in brackets to the left of program statements)

The fitness landscape can also be flat for large areas of the input domain, giving the search little guidance at all. It is well known that this problem can be caused by the existence of flag and enumeration variables in branch predicates [2,3,4]. A flag is simply a variable that is true or false. When such a variable is used in a branch predicate, plateaux form in the fitness landscape corresponding to the true and false values of the flag. This can be seen with the example of Figure 3. The target of the search is to execute the statement corresponding to node f, requiring the true branch from node e to be taken. The branching condition at node e involves the variable `flag`, which is only true when all the variables of an inputted array are zero. However no guidance is provided to this input vector, since for all other vectors the branch distance calculation result, using the false flag value, is the same. The resulting plateau can be seen in a plot of the fitness landscape (Figure 5a). It can easily be seen from the program that the search simply needs to execute the path up to e where the assignment at node d is avoided on each iteration of the loop. However, ET is not aware of this assignment, and consequently does not recognize what needs to be done to prevent it - that is the optimization of all array values to zero.

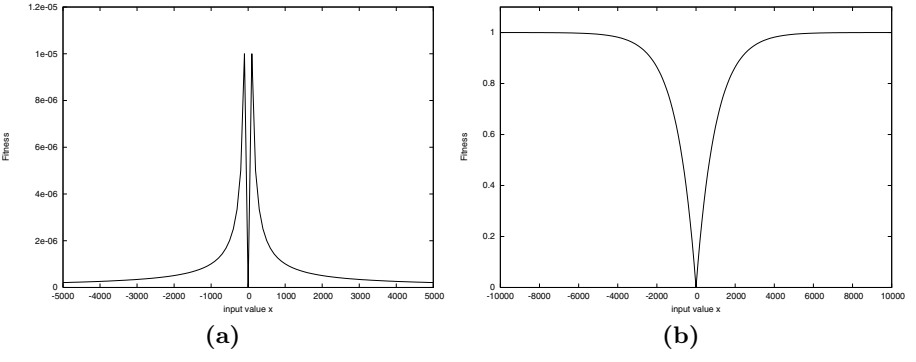
A similar problem can occur with enumeration variables. Again, two plateaux form in the fitness landscape when such variables are involved in branch predicates - one for the "correct" value, and one for all other values. Since enumerations are not ordinal types, no value from the enumeration can be deemed to be "closer" to the desired value.

A number of solutions have been proposed for the particular problem of flags. Harman et al. [3] present an approach to the problem of flags which transforms the flag out of the program. A different type of transformation procedure needs to be applied depending on the characteristics of the test object. No work has yet been published applying these transformations to programs with loops. Bottaci [2] deals with a special case. Baresel et al. [4] present an alternative approach that analyses the program, and selects a suitable fitness function from a library depending on features found within the program (e.g. a flag involved in a loop). It is stated that the method struggles to avoid undesirable assignments to flags in loops. It is claimed the method extends to enumerations, but no experimental results are presented. Whilst these solutions apply different "rules" each time

```
    void flag_avoid_loop_assignment(int a[10])
    {
(a)     int flag = 1;
        int i;
(b)     for (i = 0; i < 10; i++)
        {
(c)         if (a[i] != 0)
(d)             flag = 0;
        }
(e)     if (flag)
(f)         // target
    }
```

Fig. 3. C example requiring the avoidance of a flag assignment contained in a loop in order to execute node f (control flow graph node identifiers appear in brackets to the left of program statements)

a different program characteristic is encountered, incorporation of the Chaining Approach represents a general and elegant solution to flags and the broader problem as described.

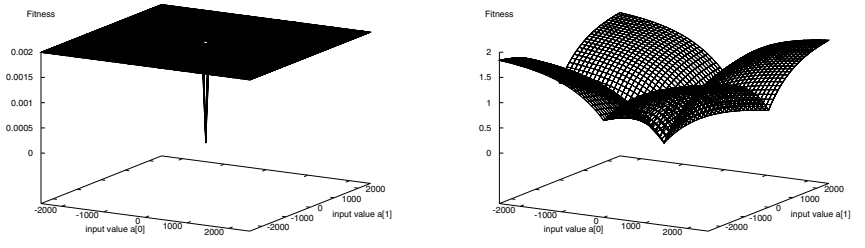


(a) **(b)**

Fig. 4. Fitness landscapes for the deceptive example (Figure 2). (a) shows the landscape for the fitness function used by ET (equivalent to that of the initial event sequence with the hybrid approach). (b) shows the landscape of the fitness function for the successful event sequence with the hybrid approach

4 The Chaining Approach

The Chaining Approach [5,6] is an alternative test data generation technique, based on a local search method known as the "alternating variable method". If a critical branch is taken, the search method attempts to change the flow of



(a) **(b)**

Fig. 5. Fitness landscapes for the flag example, plotted for two array elements, assuming all other elements are zero. (a) shows the landscape for the fitness function used by ET (equivalent to that of the initial event sequence with the hybrid approach). (b) shows the landscape of the fitness function for the successful event sequence with the hybrid approach

control by searching for input values using the branch distance calculation of the alternative branch as a cost function. In practice this is performed by successively finding local minima for each input variable until the alternative branch is taken, or no further improvement can be made. At this point the search declares failure. Any move which results in an critical branch being taken earlier in the execution path is disallowed.

Being based on local search, the alternating variable method frequently encounters problematic test goals with non-trivial search landscapes. Typically a "problem" branching node is encountered, for which the search can not change control flow so that a critical branch is not taken. In an attempt to circumvent this problem, the Chaining Approach employs its backup plan - the construction of "event" chains - or *event sequences*. Event sequences force the consideration of certain statements leading up to the target structure, which may influence the outcome at the problem branching node. Such statements are identified by the use of data flow analysis. Using this mechanism, the search can be directed into potentially unexplored but promising areas of the input domain.

The crux of the chaining mechanism is best explained with an example. For the flag program of Figure 3 the execution of branching node e as true is problematic. Prior events are identified by finding the last definitions of variables used at the problem node. For node e, this is the variable `flag`, last defined at nodes a and d. Two event sequences are constructed, one demanding the execution of a before e, and the other requiring instead d before e. Node a of course, is the event that causes node e to be executed as true. However, this is only the case if the value of `flag` is preserved from this point up to node e - i.e. that node d is avoided on each iteration of the loop. Associated with each event is a *constraint set* of variables that should not be modified until the next event. The variable `flag` is inserted into this constraint set because its modification

destroys the effect of the assignment at node a. If node d is accidentally taken, the search uses the branch distance calculation of the false branch from this node in order to change the flow of control. In this way, the focus of the search changes from one which attempts to make `flag` true, into one which specifically searches for an array where each element is zero - which in turn results in `flag` being true.

Formally, an event sequence is described as a sequence of events $< e_1, e_2, \cdots e_k >$ where each event is a tuple $e_i = (n_i, C_i)$ where n_i is a program node and C_i is the set of variables referred to as the constraint set [5]. The initial event sequence for the flag example is simply $< (f, \phi) >$, with the event sequences constructed at the next "level" being denoted as follows:

$$\begin{array}{ll} (1) & < (a, \{flag\}), (e, \phi), (f, \phi) > \\ (2) & < (d, \{flag\}), (e, \phi), (f, \phi) > \end{array}$$

Of course, the addition of node d is unhelpful, and if the latter event sequence is tried, the search for test data will fail. Such an event sequence is described as *infeasible*.

The generated event sequences are organized in a tree. At the root of the tree is the initial event sequence. The first level contains the event sequences generated as a result of the first problem node. In more complicated examples, further problem branching nodes could be encountered on route to executing some new event inserted into the sequence. In such instances the Chaining Approach backtracks further, and looks for last definition statements for variables used at these new problem nodes. The additional event sequences are added to the tree, which is explored to some specified depth limit.

5 A Hybrid Approach

In the hybrid approach, evolutionary algorithms are used for the test data search, with a chaining mechanism employed for "problematic" test goals. The search for test data for the initial event sequence is equivalent to the search conducted by the original ET approach. However, if the search fails, the best overall individual is taken, and its problem branching nodes are identified. These are the branching nodes at which critical branches were taken. Using data flow information with regards to the first problem branching node, new event sequences are built, with separate evolutionary searches conducted in an attempt to try find test data for each new event sequence.

For an event sequence S of length l, the fitness for an input vector $\mathbf{x}$ is computed using the formula:

$$\sum_{i=1}^{l} fitness(e_i) \tag{2}$$

where e_i is the ith event in the event sequence, and $fitness(e)$ is calculated for an event $e = (n, C)$ as follows:

1. If control flow diverged down a critical branch with respect to n, add the result of Equation 1 for node n to the fitness
2. For each definition node executed for each variable $v \in C$ violating the definition clear path required until the next event in the sequence (if one exists); add d; the branch distance for the alternative branch at the last branching node, mapped into the range $0 \leq d < 1$

Figure 5 depicts the fitness landscape for the initial event sequence for the flag example (equivalent to the search conducted by the original ET approach), alongside that of the event sequence $< (a, \{flag\}), (e, \phi), (f, \phi) >$. As can be seen, the latter landscape offers more guidance to the search, and is more amenable to the discovery of the required test data. As an example, the fitness function is computed as follows. With values of the array being $(1000, 1000, \ldots)$ (where all elements are zero unless stated), node e is executed twice, and the branch distance of the false branch from node c on the first two iterations of the loop are used in accumulating the fitness, according to rule 2. For array values of $(500, 1000, \ldots)$, the situation is the same, but the branch distance of the false branch from node c in the first iteration is smaller, and therefore the overall fitness is less. For array values of $(0, 1000, \ldots)$ only the false branch distance in the second iteration is needed. When all array values are zero, the event sequence is successfully executed and the overall fitness is zero.

For the deceptive example of Figure 2, the fitness landscape for the initial event sequence can be seen alongside that of $< (d, \{inverse\}), (a, \phi), (b, \phi) >$. As can be seen, the latter landscape is more conducive to the discovery of the required test data. With an input value of 500, nodes b and e are unexecuted. The approach levels and branch distances for these nodes are used to compute the fitness, according to rule 1. With an input value of 100, these nodes are still unexecuted, but the branch distances are smaller at the point at which control flow diverged away, being closer to zero, and therefore the overall fitness is smaller. At zero, all nodes are successfully executed.

In our work we have extended the data flow analysis utilized by the original Chaining Approach in order to find further program nodes that can influence the outcome at the problem node. The original Chaining Approach considers only last definitions of variables used at the current problem node. In our work, if this does not lead to the execution of the problem node in the desired way, last definitions of variables used at these last definitions are also considered. In practice this is performed by maintaining an *influencing set* of variables which can still affect the outcome at the current problem branching condition. (Unfortunately space restrictions prevent a full discussion of this algorithm here).

6 Results

Experiments were performed using different programs. Each contained a specific statement that could not be easily covered by ET, due to a specific "low probability" statement sequence required to be followed before the target is reached.

Table 1. Average fitness evaluations and running times for the experiments. The average number of fitness evaluations is the average sum of evaluations used by each evolutionary search for each event sequence for a test object.

	Average No. of Fitness Evaluations		**Average Search Time (s)**	
	200 generations	50 generations no improvement	200 generations	50 generations no improvement
Counter	547,064	213,546	157.9	67.1
Deceptive	63,684	23,907	10.2	4.8
Enumeration	314,551	105,523	48.6	18.0
Flag Assignment	121,772	56,687	21.3	11.2
Flag Loop Assignment	109,274	49,147	29.5	14.0
Flag Avoid Loop Assignment	81,592	45,562	23.7	14.7
Multiple Flag	228,892	83,867	39.3	16.2

Each problem statement was used as the target of the search using the hybrid approach. Although each program is of a relatively simple nature, the technical difficulties that will be encountered by ET are the same whether they are embedded in programs large or small. The actual size of a program is not directly responsible for increasing the complexity of the problem.

ET parameters applied by other authors in the field [3,10] were used for the evolutionary searches - namely 300 individuals per generation, comprising of 6 subpopulations with 50 individuals each; competition and migration across subpopulations; utilization of linear ranking; and a selective pressure of 1.7. Each experiment was repeated 50 times. The chaining tree was explored to a maximum depth of 5.

In the first run of experiments, searches were terminated after 200 generations. However it was found that for unsuccessful or infeasible event sequences, the search tended to stagnate well before the final generation was reached. Conversely, it was noted that for successful event sequences, test data could generally be found with an improvement on the previous best fitness value occurring within the last 50 generations. In an attempt to improve efficiency by saving on unnecessary fitness evaluations, a second run of the experiments was performed where each evolutionary search was terminated after 50 generations of no improvement. The results for the experiments are recorded in Table 1. The timing information reflects overall search times using a 1.3GHz PC with 512Mb of RAM. As can be seen, the latter termination criterion yielded the best performance, in some cases cutting search times by a third. This was achieved without comprising the reliability of the result. The following briefly describes each program and problem target statement, with further discussion of the results obtained.

Counter. This program is similar to the flag program of Figure 3 ("Flag Avoid Loop Assignment"), except a counter variable is used which keeps a total of the inputted ten array elements that are equal to zero. The target statement is executed if at least half of the array elements are zero. The range of the integers

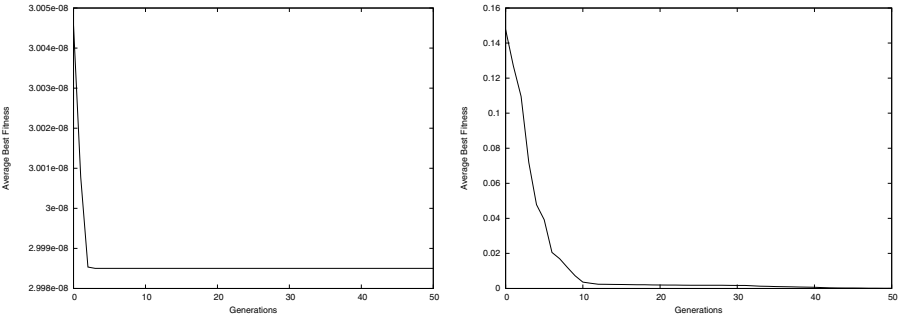


Fig. 6. Average best fitness plots for the deceptive fitness function program. (a) shows the search using the initial event sequence, with early stagnation. (b) shows the progress of the search for the successful event sequence

of the array was -15,000 to 15,000 giving a search space size of approximately 6×10^{44}. In all trials, eleven event sequences required consideration in order to find the required test data.

Deceptive. This is the program of Figure 2, with the target statement being that of node *b*. This experiment was carried out using an input range for `x` of -50,000 to 50,000 with a precision of 0.001, giving a search space size of approximately 10^8. The initial search stagnates early (Figure 6a). Event sequences are generated as outlined in Section 4. In all trials, the search succeeded with the event sequence that attempts to execute node *b* first (Figure 6b).

Enumeration. This decides whether three inputted color intensity values (integers in the range 0 to 255) represents one of the colors in an enumeration. A problem node occurs in the function under test when the color must be black. The search space size of approximately 1.6×10^7. The hybrid approach generated test data in all runs of the experiment. In five runs, the search fortuitously found test data for the initial event sequence. For all other trials, seven event sequences had to be considered before the one yielding test data could be found.

Flag Assignment. This function takes two double values. In searching for test data for one of the program statements, a flag has to be true. The flag is initially set to false, and is only set true when the first input value is zero. With an input range of -50,000 to 50,000 for each input variable, with a precision of 0.001, the search space size is approximately 10^{16}. The hybrid approach successfully generated test data in all trials, requiring the consideration of three event sequences.

Flag Loop Assignment. This program is similar to the program of Figure 3, except the flag which must be true for the target statement to be executed is set within the loop body when one or more of the ten inputted array values are zero. The range of the integers of the array was -15,000 to 15,000 giving a search space size of approximately 6×10^{44}. The hybrid approach generated test data in

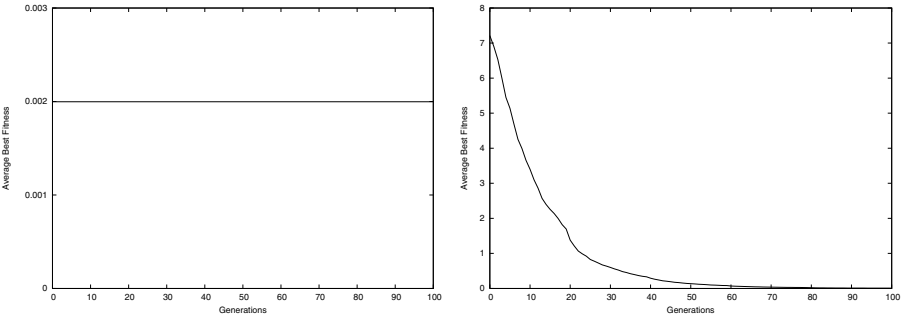


Fig. 7. Average best fitness plots for the flag avoid loop assignment program. (a) shows the search using the initial event sequence, which fails to improve on the best fitness of the initial generation. (b) shows the progress of the search for the successful event sequence

all trials. Generally three event sequences were evaluated, although in six runs, the search fortuitously found test data for the initial event sequence.

Flag Avoid Loop Assignment. This is the program of Figure 3, with the target statement being that of node f. Again, the range of the integers of the array was -15,000 to 15,000 giving a search space size of approximately 6×10^{44}. The initial search fails to improve on the best fitness value of the first generation (Figure 7a). Event sequences are generated as outlined in Section 4. The test data search for the event sequence that attempts to avoid node d was successful 94% of the time (Figure 7b).

Multiple Flag. This program involves two flags, both of which must be true in order to execute the target statement. The program takes two double input variables. Both flags are true if the first and second inputs are zero. However if the second double value is 1, the second flag is reset to false. Both double input ranges were -50,000 to 50,000 with a precision of 0.001 giving a search space size of approximately 10^{16}. The hybrid approach generated test data in all trials, evaluating five event sequences.

7 Conclusions

ET can often fail to test data for certain white-box test goals for types of programs, due to the lack of guidance provided by the search. This can occur when the target structure has data dependencies on previous statements which are only exercised under special circumstances. This paper argues that ET can be improved by incorporating ideas from the Chaining Approach, which forces the consideration of "events" leading up to the target structure. By instead searching for test data that executes an *event sequence*, the search can be directed into potentially untried, yet promising areas of the program's input domain. Experiments were carried out on seven programs. These programs - involving flags,

enumerations, counters and special function call return values - cause problems for the original ET approach. With the hybrid approach, test data could be successfully generated. Much of the literature in this area has solely concentrated on flag problems. The hybrid approach is general enough to tackle a wider range of programs which cause problems that are not just the result of flags.

Acknowledgements. This work is funded by DaimlerChrylser AG. The deceptive program (Figure 2) is an adaptation of an example presented by Mark Harman at the SBSE workshop in Windsor, UK, September 2002.

References

1. P. McMinn. Search-based software testing: A survey. *Software Testing, Verification and Reliability*, To appear, 2004/5.
2. L. Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1337 – 1342, New York, USA, 2002. Morgan Kaufmann.
3. 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 2002)*, pages 1359–1366, New York, USA, 2002. Morgan Kaufmann.
4. A. Baresel and H. Sthamer. Evolutionary testing of flag conditions. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science 2724*, pages 2442 – 2454, Chicago, USA, 2003. Springer-Verlag.
5. R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.
6. B. Korel. Automated test generation for programs with procedures. In *International Symposium on Software Testing and Analysis (ISSTA 1996)*, pages 209–215, San Diego, California, USA, 1996.
7. J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.
8. A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1329–1336, New York, USA, 2002. Morgan Kaufmann.
9. N. Tracey, J. Clark, and K. Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications*, pages 169–180, 1998.
10. A. Baresel, H. Pohlheim, and S. Sadeghipour. Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science 2724*, pages 2428 – 2441, Chicago, USA, 2003. Springer-Verlag.