Evolutionary Testing Using an Extended Chaining Approach

P. McMinn p.mcminn@dcs.shef.ac.uk
University of Sheffield, Department of Computer Science, Regent Court,
211 Portobello Street, Sheffield, S1 4DP, UK

M. Holcombe m.holcombe@dcs.shef.ac.uk
University of Sheffield, Department of Computer Science, Regent Court,
211 Portobello Street, Sheffield, S1 4DP, UK

Abstract

Fitness functions derived from certain types of white-box test goals can be inadequate for evolutionary software test data generation (Evolutionary Testing), due to a lack of search guidance to the required test data. Often this is because the fitness function does not take into account data dependencies within the program under test, and the fact that certain program statements may need to have been executed prior to the target structure in order for it to be feasible.

This paper proposes a solution to this problem by hybridizing Evolutionary Testing with an extended Chaining Approach. The Chaining Approach is a method which identifies statements on which the target structure is data dependent, and incrementally develops chains of dependencies in an *event sequence*. By incorporating this facility into Evolutionary Testing, and by performing a test data search for each generated event sequence, the search can be directed into potentially promising, unexplored areas of the test object's input domain.

Results presented in the paper show that test data can be found for a number of test goals with this hybrid approach that could not be found by using the original Evolutionary Testing approach alone. One such test goal is drawn from code found in the publicly available `libpng` library.

Keywords

Evolutionary Testing, Chaining Approach, evolutionary algorithms, structural test data generation.

1 Introduction

Evolutionary Testing (Jones et al., 1996; McMinn, 2004; Pargas et al., 1999; Wegener et al., 2001; Xanthakis et al., 1992) uses evolutionary algorithms to search for software test data. For white-box testing criteria, each uncovered structure - for example a program statement or branch - is taken as the individual target of a test data search. With certain types of programs, however, the approach degenerates into a random search, due to a lack of guidance to the required test data. Often this is because the fitness function does not take into account data dependencies within the program under test, and the fact that certain program statements need to have been executed prior to the target structure in order for it to be feasible. For instance, the outcome of a target branching condition may be dependent on a variable having a special value that 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 that an error has occurred, or a counter variable only incremented under certain conditions. Without specific knowledge of such dependencies, the fitness landscape may contain coarse, flat, or even deceptive areas, causing the evolutionary search to stagnate and fail. The problem of flag variables in particular has received much interest from researchers (Baresel et al., 2004; Baresel and Sthamer, 2003; Bottaci, 2002; Harman et al., 2002), but there has been little attention with regards to the broader problem as described.

This paper proposes a solution which hybridizes Evolutionary Testing with an extended Chaining Approach. If the search fails to find test data which directly executes the target, the Chaining Approach performs data flow analysis to identify intermediate statements which may determine whether the target will be reached or not. Such intermediate statements are referred to as *events*. The search then focuses on executing a sequence of events in order to find test data to execute the target. By incorporating this facility into Evolutionary Testing, the evolutionary search can be directed into potentially unexplored, yet promising areas of the test object's input domain. Results from an experimental study confirm this. For eight test objects, test data can be found for a particular test goal with the hybrid approach that could not be found by using the original Evolutionary Testing approach alone. One such test goal is drawn from code found in the publicly available `libpng` library.

This paper is organized as follows. Section 2 describes some basic concepts. Section 3 introduces Evolutionary Testing, and the standard method of deriving fitness functions from white-box test goals. Problems of inadequate guidance are explained with examples. Section 4 introduces the Chaining Approach. Section 5 describes the proposed hybrid approach, including computation of the fitness function for the execution of event sequences, as well as improvements to the chaining algorithm. Section 6 details the experimental study and results. Section 7 concludes the paper.

2 Basic Concepts

A control flow graph (CFG) of a program is a directed graph $G = (N, E, s, e)$ where N is a set of nodes, E is a set of edges, and s and e are unique entry and exit nodes to the graph. Each node $n \in N$ corresponds to a statement in the program, with each edge $e = (n_i, n_j) \in E$ representing a transfer of control from node n_i to n_j. Nodes corresponding to decision statements (for example an `if` or `while` statement) are referred to as *branching nodes*. In the example of Figure 1, nodes 2, 4, and 6 are branching nodes. Outgoing edges from these nodes are referred to as *branches*. The branch executed when the condition at the branching node is true is referred to as the *true branch*. Conversely, the branch executed when the condition is false is referred to as the *false branch*. The predicate determining whether a branch is taken is referred to as a *branch predicate*. The branch predicate of the true branch from branching node 2 in the program of Figure 1 is `a == 0`. The false branch predicate is `a != 0`.

An *input vector I* is a vector $I = (x_1, x_2, \ldots x_k)$ of input variables to a program. The domain of an input variable x_i, $1 \leq i \leq k$, is the set of all values that x_i can take on. The domain of a program is the cross product $D = D_{x_1} \times D_{x_2} \times \ldots \times D_{x_k}$ where each D_{x_i} is the domain of the input variable x_i. A *program input* $\mathbf{x}$ is a single point in the k-dimensional input space D, $\mathbf{x} \in D$.

A *path* through a CFG is a sequence $P =< n_1, n_2, \ldots n_m >$ such that for $i, 1 \leq i < m, (n_i, n_{i+1}) \in E$.

A *definition* of a variable v is a node which modifies the value of v, for example an

```
CFG Node
  (s)      void flag_example(int a, int b)
           {
  (1)          int flag = 0;

  (2)          if (a == 0)
  (3)              flag = 1;

  (4)          if (b != 0)
  (5)              flag = 0;

  (6)          if (flag)
               {
  (7)              // target
               }
  (e)      }
```

Figure 1: Example containing a flag, resulting in a flat fitness landscape

assignment or input statement. In the program of Figure 1 the variable `flag` is defined at nodes 1, 3 and 5.

A *use* of a variable v is a node in which v is referenced, for example in assigning a value to another variable, or appearing as part of a branching condition. In the program of Figure 1, the variable `flag` is used at node 6.

A *definition-clear path* with respect to a variable v is a path in which v is not defined. In the example of Figure 1, the path $< 4, 6 >$ is definition-clear with respect to the variable `flag`, but $< 4, 5, 6 >$ is not, since `flag` is defined at node 5.

3 Evolutionary Testing

White-box testing coverage criteria demand that test data be found to execute all program structures of a certain type, for example all CFG nodes or all branches. Evolutionary algorithms can automate the derivation of test data for this purpose by searching the input domain of the program in question. Real-valued encodings are used, with individuals directly representing input vectors to a function of the program currently under test (Wegener et al., 2001). The fitness function to be minimized is derived from the current structure or test goal of interest (Wegener et al., 2001; Baresel et al., 2002), and is made up of two components - the *approach level* and the *branch distance*.

The approach level metric assesses how close an input vector is to covering the current structure of interest on the basis of the execution path taken through the program's control structure. Central to this is the notion of a *critical branch*. A critical branch is simply a branch which leads to the structural target of interest being missed in a path through the program. Critical branches are also referred to in the literature as *decisive branches*, because once such a branch is taken through the program's control structure, failure to reach the target has essentially been 'decided'. For example in Figure 2, each of the false branches from nodes 1, 2 and 3 are critical branches. The approach level for an individual (see Figure 2) 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.

At the point where control flow took a critical branch for some individual, the *branch distance* is calculated. The branch distance reflects how close the alternative branch was to being taken, and is computed using the values of the variables or constants involved in the predicates used in the conditions of the branching statement. For

```
CFG Node
  (s)     void example(int a, int b, int c, int d)
          {
  (1)       if (a >= b)
            {
  (2)         if (b <= c)
              {
  (3)           if (c == d)
                {
                  // target
            ...
```

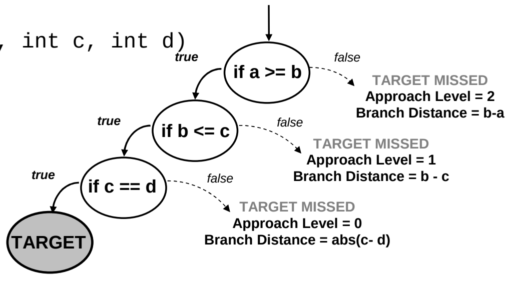


Figure 2: Calculating the fitness function for Evolutionary Testing

example, if the false branch were taken from node 3 in Figure 2, the branch distance for taking the alternative true branch is computed using the formula `abs(c - d)`. The closer the values of `c` and `d`, the smaller the branch distance value, and the closer the branch is to being executed as true. A full list of formulas for different types of branching condition can be found in Tracey *et al.*(Tracey et al., 1998). The branch distance *dist* is normalized using the following function (Baresel, 2000):

$$normalize(dist) = 1 - 1.001^{-dist} \quad (1)$$

This value is added to the approach level to make up the complete fitness value for an input vector and the current structural target of interest:

$$approach\_level + normalize(dist) \quad (2)$$

3.1 Problems with the Approach

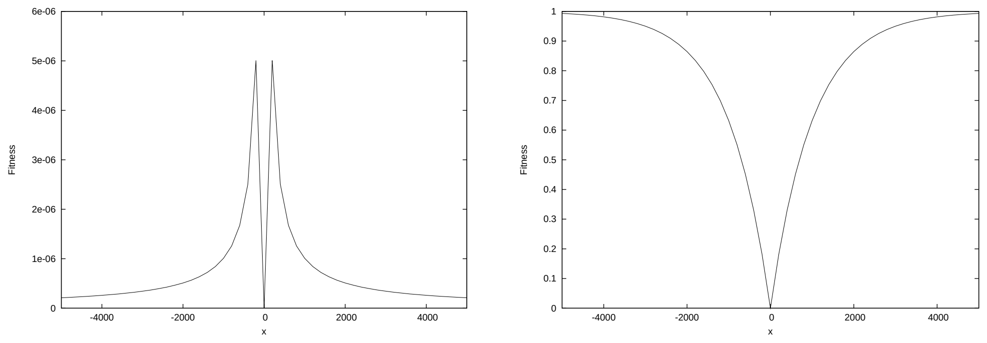
Evolutionary Testing has been shown to be an effective method for structural test data generation (Wegener et al., 2001). However, the approach degenerates into a random search for programs with certain characteristics. One problem can arise as a result of data dependencies within the program, where the target structure requires execution of earlier statements in order for it to be feasible. Due to the fact that these statements do not affect whether the target will be reached in terms of control flow through the program, they are ignored for the purposes of computing the fitness function. If such statements are only exercised under 'special' conditions rather than by chance, and if the search is not given guidance to their execution, the target structure is unlikely to be covered.

Take the example of Figure 3. The `inverse` function finds the multiplicative inverse of an argument `d`. 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 `2` in the function `function_under_test`, it is necessary to execute the true branch from node `1`. This requires a zero return value from the `inverse` function, and hence a zero input value `x` as an argument to the function `function_under_test`. 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 input domain, and there is no explicit fitness information to guide the search to it. In actual fact, the fitness function leads the search away from the zero input, 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 in Figure 4a.

```
CFG Node
  (s)      double function_under_test(double x)
           {
  (1)          if (inverse(x) == 0)
               {
  (2)              // target
               }
  (e)      }

           double inverse(double d)
           {
  (3)          if (d == 0)
  (4)              return 0;
               else
  (5)              return 1 / d;
           }
```

Figure 3: Example resulting in a deceptive fitness landscape



(a) (b)

Figure 4: Fitness landscapes for the 'Deceptive' test object (Figure 3). (a) Fitness landscape of the initial event sequence. (b) Fitness landscape of the final event sequence

The fitness landscape for a test goal 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 (Baresel et al., 2004; Baresel and Sthamer, 2003; Bottaci, 2002; Harman et al., 2002). A flag is simply a variable that is true or false. When flag variables are used in branch predicates, plateaux form in the fitness landscape, which correspond to the true and false values of the flag. This can be seen with the example of Figure 1. The test goal is the statement corresponding to node 7, which requires the true branch from node 6 to be taken. However, the branching node 6 involves the flag variable `flag`, which is only true when both input variables to the function are zero. No guidance is provided to this input vector, because for all other vectors the branch distance using the false value of the flag is the same. The resulting plateaux can be seen in a plot of the fitness landscape in Figure 5a. It can be easily seen from the program that the path which executes nodes 3 and avoids node 5 should be taken, in order for the variable `flag` to be true. However no guidance is provided to the search to direct it to the execution of this path.

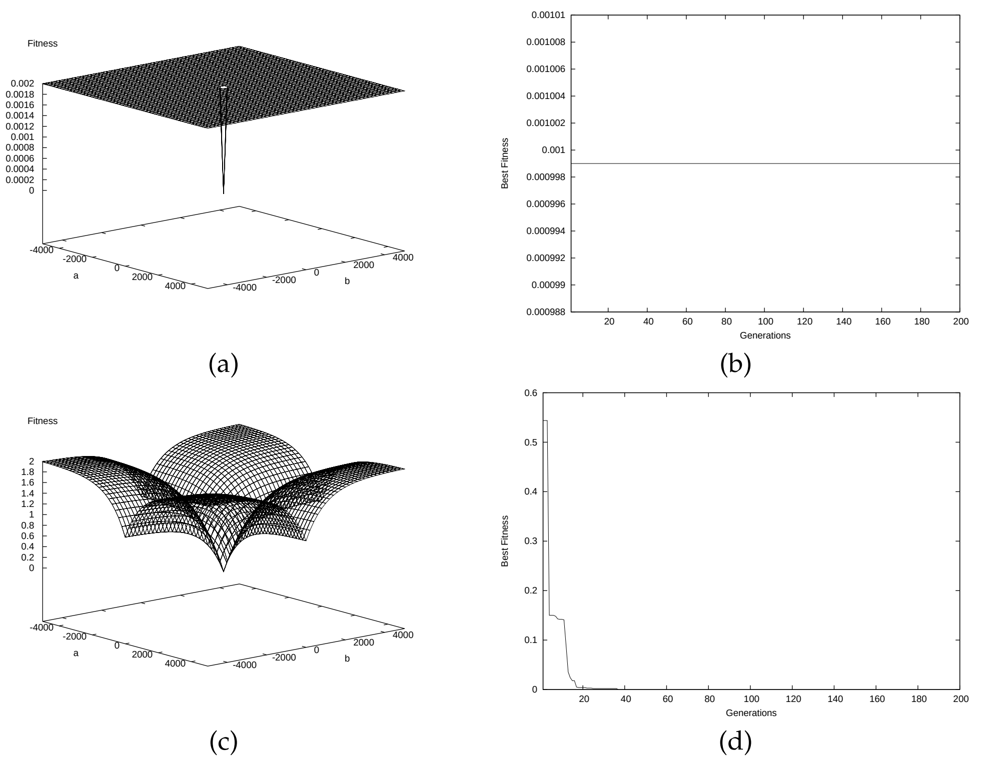


Figure 5: Fitness landscapes and best fitness plots for the 'Flag' test object (Figure 1). (a) Fitness landscape of the initial event sequence. (b) Best fitness plot for a test data search using the initial event sequence. (c) Fitness landscape of the final event sequence. (d) Best fitness plot for a test data search using the final event sequence.

4 The Chaining Approach

The Chaining Approach (Ferguson and Korel, 1996a; Ferguson and Korel, 1996b; Korel, 1996) is an alternative structural test data generation technique based on a local search method known as the 'alternating variable method'. The alternating variable method works to adapt a single input vector to find the required test data for some structure. If a critical branch is taken, branch distance calculations are used to guide the modification of the input vector until test data can be found, or no further improvement in branch distance values can be made. If the branch distance cannot be minimized to zero, test data will not have been found, and the search declares failure.

Recognizing search failure may be due to data dependencies within the program, the Chaining Approach employs a 'backup' strategy through the construction of *event sequences*, which may lead to execution of the target. An event sequence can be thought of as an abstract path. An event simply refers to the execution of a program node. Event sequences force the consideration of certain statements leading up to the target structure, identified using data flow analysis. By conducting a new search for each event sequence, new branch distance information can be employed to direct the search into originally unexplored, potentially promising areas of the input domain.

4.1 Basic Concepts for the Chaining Approach

An *event sequence* is 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 a set of variables referred to as a constraint set (Ferguson and Korel, 1996a). The constraint set is a set of variables that must not be modified until the next event in the sequence. That is to say, a definition-clear path must be executed between two events e_i and e_{i+1} with respect to each variable v in C_i.

The following event sequence $< (s, \emptyset), (3, \{flag\}), (7, \emptyset) >$ is an event sequence referring to nodes in the example of Figure 1. It requires that the start node s is executed, followed by the execution of node 3. Node 7 must then be reached, but by avoiding any reassignment to `flag`. This means the false branch must be taken from node 4.

An event sequence is feasible if input data exists on which the event sequence can be successfully executed, otherwise it is said to be infeasible. The event sequence $< (s, \emptyset), (3, \{flag\}), (7, \emptyset) >$ is feasible, however the event sequence $< (s, \emptyset), (5, \{flag\}), (7, \emptyset) >$ is not.

A *problem node* refers to a branching node for which the search can not find inputs so that the flow of execution is changed in order for some alternative, preferred, branch to be taken.

The set of nodes that can have an immediate effect on a problem node is the set of *last definitions* of variables used at that problem node. A last definition i is a program node that assigns a value to a variable v which may potentially be used by a node j. For the node to qualify as a last definition, a definition-clear path must exist between node i and node j with respect to v. The set of last definitions for variables used at node 6 is therefore the set $\{1, 3, 5\}$. A definition-clear path with respect to `flag` exists from node 1 to node 6 via the false branches from nodes 2 and 4, whilst a definition-clear path exists from node 3 to node 6 with respect to `flag` through the false branch from node 4.

4.2 The Chaining Process

The Chaining Approach begins with an initial sequence E_0 which contains the start node s and the target node. Both events have empty constraint sets. In the example of

Figure 1, the target is node 7. The initial event sequence is therefore:

$$E_0 = <(s,\emptyset),(7,\emptyset)>$$

Input data may not be found to take the true branch from node 6 so that node 7 is executed, due to the existence of a flag variable in the predicate at node 6. The flag is only true when `a` and `b` are zero. However guidance to these values is not provided by the true branch distance at node 6. Therefore, node 6 is declared as a problem node and is inserted before node 7 into the event sequence:

$$<(s,\emptyset),(6,\emptyset),(7,\emptyset)>$$

Last definition nodes for node 6 are then identified - nodes 1, 3 and 5. Three new event sequences are now generated, each demanding the execution of one of these nodes before node 6 in the sequence:

$$\begin{aligned} E_1 &= <(s,\emptyset),(1,\{flag\}),(6,\emptyset),(7,\emptyset)> \\ E_2 &= <(s,\emptyset),(3,\{flag\}),(6,\emptyset),(7,\emptyset)> \\ E_3 &= <(s,\emptyset),(5,\{flag\}),(6,\emptyset),(7,\emptyset)> \end{aligned}$$

The inserted events are formed using a last definition node and a constraint set formed from the variable defined at the node. The addition of the last definition variable into the constraint set specifies that it will not be modified again until the problem node is encountered, ensuring the effect of that last definition on the problem node is not destroyed. In this case, the constraint set contains the variable `flag`. In event sequence E_1 this means, for example, that the false branch must be taken from nodes 2 and 4 in order to prevent `flag` being redefined before node 6. It is unlikely that E_1 will lead to the discovery of the required test data. It adds no branch distance information for the purposes of the search, and the paths taken using it are likely to follow those of E_0. E_3 will not lead to the required test data, as it mandates the setting of `flag` to false. E_2 requires `flag` to be set to true at node 3. This requires node 2 to be executed as true, and so the search can use the branch distance information at this node to find a value of `a` in order for this to happen. This branch distance information explicitly directs the search to the zero value of the `a` variable. Such guidance was not available from the branching condition at node 6, which depends only on the flag variable. Furthermore, node 5 must be avoided, as it redefines `flag`. In order for this to occur, branch distance information at node 4 is used so that the false branch is taken. By explicitly requiring that `b` is zero, further guidance is provided to the search that was not previously available.

In general then, the Chaining Approach begins with an initial event sequence E_0 containing only the start node s and the target node t - $<(s,\emptyset),(t,\emptyset)>$. The test data search may fail to find inputs to execute the event sequence, with the flow of execution diverging down an unintended branch at some node p_1. Node p_1 is declared as a problem node and is inserted into the event sequence - $<(s,\emptyset),(p_1,\emptyset),(t,\emptyset)>$. For the problem node p_1, the set of last definition nodes $lastdef(p_1)$ are found for the set of variables used at p_1. For each last definition $d_i \in lastdef(p_1)$, a new event sequence is generated containing an event associated with that last definition:

$$\begin{aligned} E_1 &= <(s,\emptyset),(d_1,\{def(d_1)\}),(p_1,\emptyset),(t,\emptyset)> \\ E_2 &= <(s,\emptyset),(d_2,\{def(d_2)\}),(p_1,\emptyset),(t,\emptyset)> \\ \ldots \\ E_N &= <(s,\emptyset),(d_N,\{def(d_N)\}),(p_1,\emptyset),(t,\emptyset)> \end{aligned}$$

The constraint set associated with each last definition d_i in E_i is a one element set $def(d_i)$ that requires a variable defined by d_i is not modified between d_i and p_1.

The Chaining Approach selects one of the event sequences and tries to find inputs for which it is successfully executed. If such an input is found, then test data to execute the test goal has been found. If not, new event sequences may be generated. For example, in trying to find inputs to execute E_1, a new problem node p_{1_1} may be encountered before d_1 can be executed. If this is the case, p_{1_1} is inserted into the sequence:

$$< (s, \emptyset), (p_{1_1}, \emptyset), (d_1, \{def(d_1)\}), (p_1, \emptyset), (t, \emptyset) >$$

Last definitions of variables are then found for p_{1_1}, and new events are generated and inserted into a new set of event sequences:

$$\begin{array}{lcl} E_{1_1} & = & < (s, \emptyset), (d_{1_1}, \{def(d_{1_1})\}), (p_{1_1}, \emptyset), (d_1, \{def(d_1)\}), (p_1, \emptyset), (t, \emptyset) > \\ E_{1_2} & = & < (s, \emptyset), (d_{1_2}, \{def(d_{1_2})\}), (p_{1_1}, \emptyset), (d_1, \{def(d_1)\}), (p_1, \emptyset), (t, \emptyset) > \\ \ldots & & \\ E_{1_N} & = & < (s, \emptyset), (d_{1_N}, \{def(d_{1_N})\}), (p_{1_1}, \emptyset), (d_1, \{def(d_1)\}), (p_1, \emptyset), (t, \emptyset) > \end{array}$$

Generated event sequences are organized in a tree structure. The first level of the tree contains the event sequences generated as a result of the first problem node, with subsequent levels formed if further problem nodes are encountered. Event sequences are explored in the tree in a depth-first fashion to a maximum depth limit.

4.3 Formal Generation of an Event Sequence

The general strategy for generating event sequence using the original Chaining Approach can be formally described as follows (Ferguson and Korel, 1996a). Let $E =< e_1, e_2, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_m >$ be an event sequence. Suppose the test data search finds input data to partially execute the event sequence up to event e_i, with a problem node p encountered between events e_i and e_{i+1}. Let d be a last definition of problem node p. A new event sequence is generated from E by inserting two events into sequence: $e_d = (d, def(d))$ and $e_p = (p, \emptyset)$. Event e_p is always inserted between events e_i and e_{i+1}. In general, however, event e_d may be inserted in any position between e_1 and e_{k+1} in the event sequence. Therefore, the following event sequence is generated:

$$E' =< e_1, e_2, \ldots, e_{k-1}, e_k, e_d, e_{k+1}, \ldots, e_{i-1}, e_i, e_p, e_{i+1}, \ldots, e_m >$$

Insertion of new events into the sequence may require modification of existing constraint sets in the sequence. The constraint set of the event corresponding to the problem node is simply the same as the constraint set of the prior event in the sequence:

$$C_p = C_i \tag{3}$$

Before the insertion of the new event, variables in C_i could not be modified between e_i and e_{i+1}. The above step ensures this is still the case, by maintaining that these variables will not be modified between e_p and e_{i+1} either.

The constraint set C_d for the event e_d is formed from the variable defined at the event node, $def(d)$, merged with the variables of the constraint set of the previous event, C_k:

$$C_d = C_k \cup def(d) \tag{4}$$

In a similar fashion to Equation 3, this rule maintains consistency of the event sequence by ensuring that variables in C_k are not modified between e_k and the new event e_d. However, the variable $def(d)$ might still be modified between e_{k+1} and e_p, ruining the effect of the last definition. The final step prevents this by adding $def(d)$ to each constraint set for each event from e_{k+1} up to but not including e_p:

$$\forall j, k+1 \leq j \leq i, C_j \cup def(d) \tag{5}$$

5 A Hybrid Approach

This section introduces the proposed 'hybrid approach'.

5.1 Test Data Search

The alternating variable method adapts one input vector using branch distance information alone. However, evolutionary search maintains populations of solutions, and therefore needs to be able to make global comparisons with respect to test inputs and event sequences. Thus, a more sophisticated fitness function is required. Therefore, the fitness value of an input vector **x** for an event sequence E of length l is computed using the following fitness function, which is to be minimized (McMinn and Holcombe, 2004):

$$fitness = \sum_{i=1}^{l} fitness(e_i) \tag{6}$$

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

Rule 1. If the event node n_i (to be executed after the event node of e_{i-1} and before e_{i+1}) is missed, add the result of Equation 2, where $approach\_level$ is the approach level for node n_i, and $dist$ is the branch distance of the alternative branch where execution diverged away from n_i.

Rule 2. For each definition node def executed for each variable $v \in C_i$ violating the definition-clear path required until e_{i+1}, add the normalized branch distance for the alternative branch at the last branching node that led to def's execution using Equation 1.

5.1.1 Example

Recall the example of Figure 1 and the event sequence:

$$< (s, \emptyset), (3, \{flag\}), (6, \emptyset), (7, \emptyset) >$$

Take the input `(a=10, b=20)`. The first event, e_1 is the start node and is always executed. However for e_2, the second event, node 3 is missed, with the false branch from node 2 taken. The approach level is zero and the branch distance is $abs(10 - 0) = 10$. Therefore $0 + normalize(10)$ is added to the overall fitness according to rule 1.

Furthermore, the constraint set of e_2 is violated, since node 5 is executed, which redefines the value of `flag`. The branch distance of the alternative false branch from node 4 is $abs(20 - 0)$. Therefore $normalize(20)$ is added to the overall fitness according to rule 2. Node 6 of the third event e_3 is successfully reached, but node 7 of the forth event e_4 is missed, since the false branch from node 6 is taken. The approach level for node 7 of e_4 is 0 and the branch distance is $abs(1-0)=1$. Therefore $0+normalize(1)$ is added to the overall fitness according to rule 1:

$$\begin{aligned} fitness &= fitness(e_1)+fitness(e_2)+fitness(e_3)+fitness(e_4) \\ &= 0+(0+normalize(10)+normalize(20))+0+(0+normalize(1)) \\ &= 0.0307 \end{aligned}$$

The fitness landscape of this event sequence is smooth, providing good guidance to the required input vector, as seen in Figure 5c. This is in contrast to the original fitness landscape (Figure 5a).

Note that fitness function for standard Evolutionary Testing and the initial event sequence of the hybrid approach are identical. The initial event sequence consists of just the start node and the target node. The start node is always executed, and its event has no constraint set. Therefore the fitness function is the fitness of target node event, calculated using rule 1, which is the same as the fitness function for standard Evolutionary Testing.

This of course also means that it is unlikely for the hybrid approach to perform any worse than standard Evolutionary Testing. The search for the initial event sequence could be viewed as performing the standard Evolutionary Testing approach, with the chaining mechanism brought into effect when this search fails.

5.2 Identification of Problem Nodes and Variables Used at the Problem Node

The evolutionary search works to minimize the fitness function. If a fitness value of zero is found, test data to execute the test goal will also have been found. If this is not the case, further event sequences are generated using the first problem node encountered by the best individual found during the search.

The test data corresponding to the best individual may result in an execution path which diverges away from the intended path at several points - resulting in several potential problem nodes. However, as for the original Chaining Approach, only the first problem node in the sequence is used to find last definitions and generate new event sequences.

5.3 Extended Event Sequence Generation Using Influencing Sets

The original Chaining Approach inserts new events which are last definitions for variables used at a problem node. However there is potentially a greater set of variables that can affect the outcome at the problem node. For example the values of `a` and `b` influence the value of `x` and thus have an intermediate effect on the outcome of the following `if` statement:

```
x = a + b;

if (x > 0) {
  // ...
}
```

```
CFG Node
           typedef enum {FALSE, TRUE} bool;

  (s)      void check_errors(int r1, int r2)
           {
  (1)          bool error1   = FALSE;
  (2)          bool error2   = FALSE;
  (3)          bool shutdown = FALSE;

  (4)          if (r1 == 0)
  (5)              error1 = TRUE;

  (6)          if (r2 == 0)
  (7)              error2 = TRUE;

  (8)          shutdown = error1 && error2;

  (9)          if (shutdown)
               {
  (10)             // target
               }
  (e)      }
```

Figure 6: Example with multiple flags, and an intermediate assignment

This can cause problems for certain test objects. Consider the example of Figure 6. The target is node 10. Node 9 is a problem node. The branching condition uses a flag, and the fitness landscape is flat for the entire input domain, other than for the required input vector. The generated event sequences are:

$$\begin{aligned} E_1 &= <(s,\emptyset),(3,\{shutdown\}),(9,\emptyset),(10,\emptyset)> \\ E_2 &= <(s,\emptyset),(8,\{shutdown\}),(9,\emptyset),(10,\emptyset)> \end{aligned}$$

E_1 is infeasible. E_2 is feasible, but node 9 remains problematic, since it merely requires node 8 to be executed. This node is always executed, and no new information is added to the fitness function, whose landscape is still flat. Further event sequences need to be generated to guide the search to the assignments at node 5 and 7, but with the original Chaining Approach this is not possible.

Furthermore, take the example of Figure 7. The target is node 6, which is only executed when half of the inputted integer array values are zero. Node 5 becomes a problem node. It depends on a counter variable which is incremented in a loop when an array value is found to be zero. The fitness landscape contains flat steps down to the required test data, providing coarse levels of guidance (Figure 8a). Further event sequences are generated, these being:

$$\begin{aligned} E_1 &= <(s,\emptyset),(1,\{counter\}),(5,\emptyset),(6,\emptyset)> \\ E_2 &= <(s,\emptyset),(4,\{counter\}),(5,\emptyset),(6,\emptyset)> \end{aligned}$$

E_1 is infeasible. E_2 is not infeasible, but requires that the counter variable is incremented only once. This is unlikely to be enough to ensure that input data can be found - the chances of four more array values being zero in a large input domain being small. The fitness landscape still contains ridges (Figure 8b), and node 5 is still a problem

```
CFG Node
            #define THRESHOLD  5
            #define SIZE      10

  (s)       void counter(double a[SIZE])
            {
  (1)           int counter = 0;
                int i;

  (2)           for (i = 0; i < SIZE; i++)
                {
  (3)               if (a[i] == 0)
  (4)                   counter ++;
                }

  (5)           if (counter == THRESHOLD)
                {
  (6)               // target statement
                }
  (e)       }
```

Figure 7: Example resulting in a coarse fitness landscape - the 'Counter' test object

node. A further increment `counter` should not be inserted between the events corresponding to nodes 4 and 5, because `counter` appears in the constraint set for the event corresponding to node 4. Furthermore, a further increment could not appear between nodes s and 4, as the definition is not a last definition, since the last definition of `counter` already appears in the sequence at node 4.

To handle these problems an extension is made to the event sequence generation algorithm, using the concept of *influencing sets*. Given a program node and some path to the problem node, an influencing set consists of all variables that could potentially affect the outcome at the problem node. The event sequence generation process is forced to consider definitions for all variables that can potentially affect the problem node, allowing event sequences to be generated that were not possible with the original approach.

The recursive algorithm for the extended event sequence generation approach can be seen in Figure 9. Paths are explored backwards from the problem node. The influencing set is adapted according to the path taken. For a newly identified problem node, the influencing set is simply the set of variables involved in evaluated, but unsatisfied conditions, at the problem node. Beginning with the current problem node sn, the initial influencing set I, and the event prior to the problem node event in the event sequence $e = (n, C)$, the algorithm traces its way in a backwards manner through the nodes of the program. The set $prev\_nodes$ is simply the set of program nodes connected to the current node by an outgoing edge. Each node pn in $prev\_nodes$ is analyzed.

Firstly, the algorithm checks to see if pn is the same as the prior event node n. If this is the case, and the variable defined at n ($def(n)$) is contained in the influencing set, the influencing set is modified by removing $def(n)$ and by adding the uses of n ($uses(n)$). This is because no prior definition of $def(n)$ can now affect the outcome at the problem node, since n is itself a last definition. However, the variables used at n can affect the problem node because they are used in the assignment to $def(n)$. The procedure then recurses using the event node en as the current node sn, the new influencing set and the new prior event in the sequence.

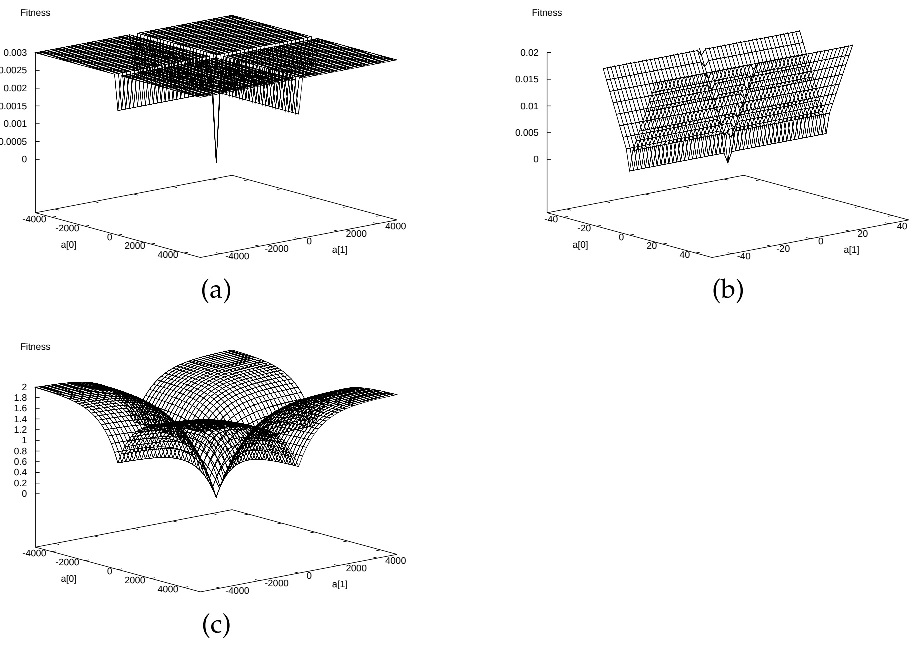


Figure 8: Fitness landscapes for the 'Counter' test object (Figure 7), plotted for the first two values of the array. The next three values are fixed at zero, with the final elements fixed at a non-zero value. (a) Fitness landscape of the initial event sequence. (b) Fitness landscape of an intermediate event sequence that mandates only one increment of the counter. (c) Fitness landscape of the final event sequence

```
**Let** $E$ be the original event sequence from which new event sequences are required

**Let** $S$ be a global set of search points, where a search point is a tuple $sp = (sn, I, e)$,
where $sn$ is a program node, $I$ is an influencing set of variables, and $e = (n, C)$ is
an event in the original event sequence $E$

**Procedure** $generate\_event\_sequences$(**In:** a search point, $sp = (sn, I, e = (n, C))$)
    **Let** $prev\_nodes$ be the set of control flow graph nodes connected to $sn$
    by an outgoing edge
    **If** $sp \notin S$
        $S \leftarrow S \cup sp$
        **Repeat**
            **Let** $pn$ be a program node, $pn \in prev\_nodes$
            $prev\_nodes \leftarrow prev\_nodes - \{pn\}$
            **If** $pn = n$
                **If** $def(pn) \in I$
                    $I \leftarrow I - \{def(pn)\}$
                    $I \leftarrow I \cup uses(pn)$
                **End If**
                $generate\_event\_sequences((pn, I, prev\_event(E, e)))$
            **Else If** $\forall v \in C, v \neq def(pn)$
                **If** $\exists v \in I, v = def(pn)$
                    **If** $reachable(pn, e)$
                        $create\_new\_event\_sequence(pn, E, e)$
                    **End If**
                    $generate\_event\_sequences((pn, I - \{def(pn)\}, e))$
                **Else**
                    $generate\_event\_sequences((pn, I, e))$
                **End If**
            **End If**
        **Until** $prev\_nodes = \emptyset$
    **End If**
**End Procedure**

$def(n)$ returns the variable defined at program node $n$ (or $\emptyset$ if one is not defined)

$uses(n)$ returns the set of variables used by a program node $n$

$reachable(pn, e)$ checks if a node $pn$ can be reached from another node $n$ of an event
$e = (n, C)$ without violation of the constraint set $C$

$create\_new\_event\_sequence(pn, E, e)$ creates a new event sequence for the next level
of the tree from a definition node $pn$, the original event sequence $E$ and the event $e$
after which the new event should be inserted

$prev\_event(E, e)$ returns the event prior to the event $e$ in an event sequence $E$
```

Figure 9: Recursive procedure for generating event sequences using influencing sets

If pn is not the prior event node, the algorithm checks the constraint set of the preceding event in the event sequence. If pn defines any variables in the constraint set, this particular line of enquiry terminates, since the path to the next event is not definition-clear.

If pn does not define any variable in the constraint set, but instead defines a variable in the influencing set, then a qualifying definition node has been found. A new event sequence can be generated using pn, if pn is reachable from en via some definition clear path with respect to C. This new event sequence is generated following the rules of the original approach (Equations 3-5).

If none of the above cases are true, the procedure recurses using the new node pn as the current node sn, along with the unmodified values of I and e.

Finally, a global data structure of 'search points' ensures that only acyclic program paths are considered between adjacent events in the event sequence, and that the algorithm terminates.

5.3.1 Examples

It is now demonstrated how influencing sets and the extended event sequence generation procedure has practical benefit.

Recall how further event sequences would be desired from the event sequence for the example of Figure 7:

$$E_2 =< (s, \emptyset), (4, \{counter\}), (5, \emptyset), (6, \emptyset) >$$

The problem node is node 5, and the influencing set is $\{counter\}$. In searching for program nodes back from node 5, node 4 is encountered. An event for node 4 appears in the event sequence. The definition variable is removed from the influencing set:

$$\begin{array}{rcl} I & \leftarrow & I - def(4) \\ & \leftarrow & \{counter\} - \{counter\} \\ & \leftarrow & \emptyset \end{array}$$

leaving an empty set. Uses at node 4 are added:

$$\begin{array}{rcl} I & \leftarrow & I \cup uses(4) \\ & \leftarrow & \emptyset \cup \{counter\} \\ & \leftarrow & \{counter\} \end{array}$$

Last definitions can be sought for variables in the influencing set from node 4. These last definitions include node 1, and node 4 on the previous iteration of the loop:

$$\begin{array}{rcl} E_{2_1} & = & < (s, \emptyset), (1, \{counter\}), (4, \{counter\}), (5, \emptyset), (6, \emptyset) > \\ E_{2_2} & = & < (s, \emptyset), (4, \{counter\}), (4, \{counter\}), (5, \emptyset), (6, \emptyset) > \end{array}$$

Event sequences can now be generated that include five instances of node 5, and therefore five increments of the counter. When this happens, the fitness landscape is smooth, providing good guidance to the required test data (Figure 8c). In this example the influencing set is effectively unchanged - the variable `counter` is defined and used in the same statement, and so is removed and then re-added. However, return to the example of Figure 6. The target is node 10. Node 9 is a problem node. The current event sequence is:

$$E_2 =< (s, \emptyset), (8, \{shutdown\}), (9, \emptyset), (10, \emptyset) >$$

Recall that E_2 is feasible, but assume node 9 is still problematic. In event sequence generation, node 8 is encountered tracing backwards from node 9. The influencing set is modified to remove the definition of $shutdown$ and add the uses - $\{error1, error2\}$. This means that further event sequences can be generated:

$$\begin{array}{rcl} E_{2_1} & = & < (s, \emptyset), (5, \{error1\}), (8, \{shutdown\}), (9, \emptyset), (10, \emptyset) > \\ E_{2_2} & = & < (s, \emptyset), (7, \{error2\}), (8, \{shutdown\}), (9, \emptyset), (10, \emptyset) > \end{array}$$

Assuming node 9 remains problematic, the following event sequence will be generated from both E_{2_1} and E_{2_2}:

$$< (s, \emptyset), (5, \{error1\}), (7, \{error1, error2\}), (8, \{shutdown\}), (9, \emptyset), (10, \emptyset) >$$

This event sequence requires nodes 5 and 7 to be executed before node 8, which in turn assures that the true branch is taken from node 9, and that node 10 will eventually be executed.

6 Experimental Study

An experimental study was designed to feature test goals that cause problems for Evolutionary Testing, and to try them with the proposed hybrid approach. The experimental study featured one real world test object, and seven synthetic test objects. The real world test object is drawn from the `libpng` publicly available open-source graphics file format library (libpng, 2005). The seven synthetic test objects feature test goals with flat, coarse and deceptive landscapes in loop and loop-free code. They are thus ideal for testing the hybrid approach.

6.1 Test Objects

This section describes the test objects and the input domain sizes used. The source code for test objects, where not found in this paper, can be found in McMinn (McMinn, 2005).

The `libpng` library - `png_init_read_transformations` function. The core code with respect to the chosen test goal of this function can be seen in Figure 10. It belongs to a file containing a set of functions which can be called by an application to handle data read in from a PNG graphics file. The target node is node 8, which depends on branching node 7 being evaluated as true. This requires the flag variable `k` to be false. Thus, the assignment to the flag at node 6 within the loop from node 4 must be avoided. Event sequences must therefore be generated in order to provide the search with this information. The function is passed a structure type as an input argument, within which five variables are relevant to the test goal. The variable `color_type` is an unsigned character of range 0-255. The variables `screen_gamma` and `gamma` are floats. A range 0-10 was used with a precision of 0.1. The variable `num_trans` is an unsigned

```
CFG Node
  (s)      void png_init_read_transformations(png_structp png_ptr)
           {
               // ...

  (1)          int color_type = png_ptr->color_type;

               // ...

               if ((color_type == PNG_COLOR_TYPE_PALETTE
                    && png_ptr->num_trans != 0)
                    && (fabs(png_ptr->screen_gamma * png_ptr->gamma - 1.0)
  (2)                  < PNG_GAMMA_THRESHOLD))
               {
                   int i,k;
  (3)              k=0;
  (4)              for (i=0; i<png_ptr->num_trans; i++)
                   {
                       if (png_ptr->trans[i] != 0
  (5)                       && png_ptr->trans[i] != 0xff)
  (6)                     k=1; /* partial transparency is present */
                   }

  (7)              if (k == 0)
  (8)                  png_ptr->transformations &= (~PNG_GAMMA);
               }

               // ...
  (e)      }
```

Figure 10: Example from the `libpng` PNG graphics file format reference library

short. A range of 0-30 was used. The array `trans` was of size 30, with its elements unsigned characters, and the full range of 0-255 used for each. Thus the input domain of the test object for the experiment, and therefore the search space size, is approximately 1.4×10^{80} possible input vectors.

Counter. The 'Counter' test object can be seen in Figure 7 and was described in Section 5.3. The inputted array is of ten doubles in the range -15,000 to 15,000, with a precision of 0.1 the search space size is approximately 6×10^{54}.

Deceptive. This program can be seen in Figure 3 and was described in Section 3.1. The input value of `x` lies in a range of -15,000 to 15,000, with a precision of 0.1, the search space size is approximately 1.5×10^{5}.

Enumeration. This decides whether three inputted colour intensity values (integers in the range 0 to 255) represent one of the colours in an enumeration. The target statement is executed when the inputs represent the colour black. However plateaux occur in the objective function landscape for the standard evolutionary approach, and the initial event sequence of the hybrid approach, due to the use of a variable that is of an enumerated type. The search space size is approximately 1.6×10^{7}.

Flag. This is the function of Figure 1, which was described in Section 4.2. With an input range of -15,000 to 15,000 for both input variables of integer type, the search space size is approximately 2×10^8.

Flag Loop Assignment. This program takes an array of ten integer values, along with an additional integer variable. A flag is initially set to false, but becomes true when one or more of the array values is zero. This assignment occurs within a loop body. When the flag is true, the target statement is executed. Due to the use of the flag, the fitness landscape consists of flat regions. The range of the integers of the array was -15,000 to 15,000 giving a search space size of approximately 8.6×10^{44}.

Flag Avoid Loop Assignment. This program also takes an array of ten integer values. A flag is initially set to true, but is set to false in a loop body which iterates through the array if any of the array values are not equal to zero. Consequently the search landscape is made up of one large plateau except for the point of the required test data. The range of the integers of the array was -15,000 to 15,000 giving a search space size of approximately 6×10^{44}.

Multiple Flag. This program can be seen in Figure 6 and was described in Section 3.1. With an input range of -15,000 to 15,000 for both input variables, the search space size is approximately 2×10^8.

6.2 Experimental Setup

Each of the test objects were put to the test with the hybrid approach.

The Genetic and Evolutionary Algorithm Toolbox (GEATbx) (Pohlheim, 2005) was used to perform the evolutionary searches. Search parameters employed by other authors in the field of Evolutionary Testing were used (Harman et al., 2002; Baresel et al., 2003), namely 300 individuals per generation, split into 6 subpopulations starting with 50 individuals each. Linear ranking is utilized, with a selection pressure of 1.7. Real-valued encodings are used. Competition and migration is employed across subpopulations. Individuals are recombined using discrete recombination, and mutated using real-valued mutation. Each experiment with each program version was repeated twenty times.

Two different termination criteria were used for the evolutionary searches. The first criterion terminates searches after 200 generations if no solution has been found, and is referred to as 'TC1'. The second terminates the search if there has been no improvement in the best fitness value found over the last 50 generations, and is referred to as 'TC2'. This criterion can therefore extend the search past the 200 generations limit, providing there has been an improvement in the best objective function value.

The experiments were repeated 20 times for each test goal and termination criterion. The maximum chaining tree depth for the chaining mechanism was set at 10.

6.3 Results

Table 1 shows the success rate of the searches comparing the search for the first event sequence (i.e. the search equivalent to using the original Evolutionary Testing approach) against searches for all generated event sequences. Test goals were only satisfied a small number of times using the first event sequence alone, showing that the test objects did indeed cause problems for the original Evolutionary Testing approach. After the consideration of all event sequences, test data was found 100% of the time, with the exception of the `libpng` test object, for which 5% of searches had not found test data by the time the search termination criterion was met. The table also shows the effects

Table 1: Successful searches for each test object, comparing searches using Evolutionary Testing / the first event sequence, and the hybrid approach ('all generated sequences') with different termination criteria. 'TC1' terminates searches after 200 generations if no solution has been found. 'TC2' terminates the search if there has been no improvement in the best fitness value found over the last 50 generations

Test Object	Evolutionary Testing / First event sequence		All generated sequences	
	TC 1	TC 2	TC 1	TC 2
`libpng`	0%	0%	95%	95%
Counter	0%	0%	100%	100%
Deceptive	0%	0%	100%	100%
Enumeration	80%	45%	100%	100%
Flag	0%	0%	100%	100%
Flag Loop Assignment	5%	0%	100%	100%
Flag Avoid Loop Assignment	0%	0%	100%	100%
Multiple Flag	0%	0%	100%	100%

of the different termination criteria. It shows that the original static limit of 200 generations is more likely to result in test data being found for the initial event sequence, but for the hybrid approach overall, both criteria fared the same. Table 2 shows the average number of fitness evaluations (test object executions) for both termination criteria. This table clearly shows that the 50 generations of no improvement criterion is more efficient - in some cases decreasing the number of fitness evaluations by a third, without compromising the ability of the hybrid approach to find test data.

The results with respect to each individual test object are discussed below.

`libpng`. The evolutionary search always fails for the initial event sequence, demonstrating the need for chaining with this test object. Figure 11a shows how the search stagnates as a result of the flag for an example search. The event sequence $< (s, \emptyset), (3, \{k\}), (7, \emptyset), (8, \emptyset) >$ (refer to Figure 10) is generated, and the search is successful in 19 out of 20 cases for both termination criteria, resulting in a 95% success rate (Figure 11b).

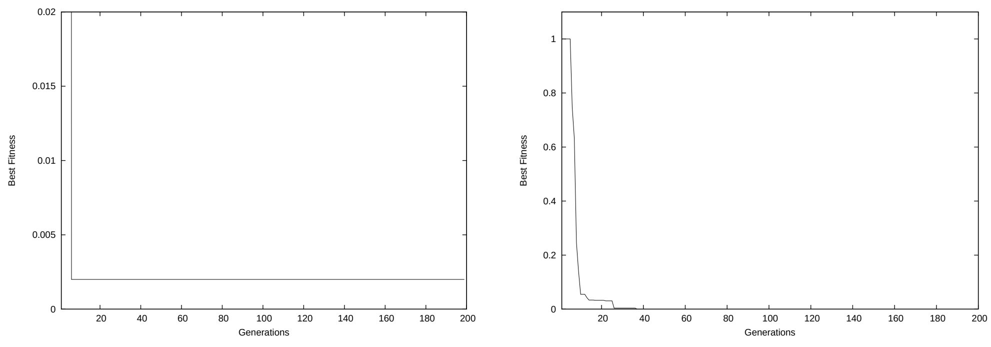
Counter. The search for test data fails for the initial event sequence, due to the coarse fitness landscape. A new event sequence is generated to increment the counter once. However this landscape still contains ridges, and the search struggles. An event sequence is then generated to increment the counter the required number of times. This search is successful.

Deceptive. The search for the initial event sequence always fails, the search stagnating early. Event sequences are generated, and the search succeeds when node 4 (refer to Figure 3) is attempted first, since the fitness landscape provides unequivocal guidance to the required test data - as can be seen in Figure 4b.

Enumeration. Test data can be found, although not always reliably, using the initial event sequence / original Evolutionary Testing approach. When this search fails however, the chaining mechanism ensures that test data is always found.

Table 2: Average fitness evaluations for each test object against different search termination criteria. 'TC1' terminates searches after 200 generations if no solution has been found. 'TC2' terminates the search if there has been no improvement in the best fitness value found over the last 50 generations

Test Object	Termination Criterion	
	TC1	TC2
`libpng`	78,819	45,123
Counter	290,461	131,253
Deceptive	59,571	19,435
Enumeration	54,075	37,373
Flag	61,094	20,594
Flag Loop Assignment	64,534	31,499
Flag Avoid Loop Assignment	82,266	41,766
Multiple Flag	61,094	20,594



(a) (b)

Figure 11: Best fitness plots for the `libpng` test object. (a) Best fitness for a test data search using the initial event sequence. (b) Best fitness plot for a test data search using the final event sequence.

Flag. Test data is not found using the initial event sequence, with the search failing to make any progress at all (Figure 5b). When the event sequence is generated executing node 3 and avoiding node 5, test data is found (Figure 5d).

Flag Loop Assignment. Test data is found for one search using standard Evolutionary Testing / the initial event sequence. However test data is only reliably found after the chaining mechanism is employed.

Flag Avoid Loop Assignment and Multiple Flag. For these test objects, test data is never found with the initial event sequence, but always found once the chaining mechanism is employed.

7 Conclusions and Future Work

Evolutionary Testing can often fail to find test data for certain white-box goals, due to a lack of guidance provided by the fitness function to the search. This can occur when the target structure has data dependencies on previous statements in the program. These statements must be executed for the target to be feasible, but are only done so under special circumstances. This paper shows that Evolutionary Testing can be improved through hybridization with an extended Chaining Approach. This hybrid approach incorporates and extends the idea of generating *event sequences* from the Chaining Approach, which can help provide increased guidance to the search to direct it to the execution of ignored statements, and thus unexplored but potentially promising areas of the test object's input domain.

An experimental study was performed with a test object drawn from the publicly available `libpng` PNG graphics file library, and seven synthetic test objects. The seven synthetic test objects featured different types of landscape that are problematic for evolutionary test data generation, i.e. flat, coarse and deceptive landscapes in loop and loop-free code and were ideal for testing the hybrid approach. In all cases it was shown that the hybrid approach could find test data where the original approach either could not or was inconsistent. This is an improvement over current work in this area, which has solely concentrated in the area of flat fitness landscapes as the result of flag variables.

Future work needs to establish how the approach scales up to larger systems, and the number of event sequences that can be reasonably handled. Some initial applications to state-based systems have been carried out (McMinn, 2005; McMinn and Holcombe, 2005).

Acknowledgements. This work was funded by DaimlerChrysler Research and Technology. The deceptive example is originally due to Mark Harman (presented at the Search-Based Software Engineering workshop in Windsor, UK, 2002).

References

Baresel, A. (2000). Automatisierung von strukturtests mit evolutionren algorithmen. Diploma Thesis, Humboldt University, Berlin, Germany.

Baresel, A., Binkley, D., Harman, M., and Korel, B. (2004). Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 43–52, Boston, Massachusetts, USA. ACM.

Baresel, A., Pohlheim, H., and Sadeghipour, S. (2003). 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 vol. 2724*, pages 2428 – 2441, Chicago, USA. Springer-Verlag.

Baresel, A. and Sthamer, H. (2003). Evolutionary testing of flag conditions. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724*, pages 2442 – 2454, Chicago, USA. Springer-Verlag.

Baresel, A., Sthamer, H., and Schmidt, M. (2002). 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. Morgan Kaufmann.

Bottaci, L. (2002). 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. Morgan Kaufmann.

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

Ferguson, R. and Korel, B. (1996b). Generating test data for distributed software using the chaining approach. *Information and Software Technology*, 38(5):343–353.

Harman, M., Hu, L., Hierons, R., Baresel, A., and Sthamer, H. (2002). Improving evolutionary testing by flag removal. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1359–1366, New York, USA. Morgan Kaufmann.

Jones, B., Sthamer, H., and Eyres, D. (1996). Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306.

Korel, B. (1996). Automated test generation for programs with procedures. In *International Symposium on Software Testing and Analysis (ISSTA 1996)*, pages 209–215, San Diego, California, USA.

libpng (2005). `libpng` - PNG reference library,
http://www.libpng.org/pub/png/libpng.html.

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

McMinn, P. (2005). *Evolutionary Search for Test Data in the Presence of State Behaviour*. PhD thesis, University of Sheffield.

McMinn, P. and Holcombe, M. (2004). Hybridizing evolutionary testing with the chaining approach. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004), Lecture Notes in Computer Science vol. 3103*, pages 1363–1374, Seattle, USA. Springer-Verlag.

McMinn, P. and Holcombe, M. (2005). Evolutionary testing of state-based programs. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005)*, pages 1013–1020, Washington DC, USA. ACM.

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

Pohlheim, H. (2005). GEATbx - Genetic and Evolutionary Algorithm Toolbox, http://www.geatbx.com.

Tracey, N., Clark, J., Mander, K., and McDermid, J. (1998). An automated framework for structural test-data generation. In *Proceedings of the International Conference on Automated Software Engineering*, pages 285–288, Hawaii, USA. IEEE Computer Society Press.

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

Xanthakis, S., Ellis, C., Skourlas, C., Le Gall, A., Katsikas, S., and Karapoulios, K. (1992). Application of genetic algorithms to software testing (Application des algorithmes génétiques au test des logiciels). In *5th International Conference on Software Engineering and its Applications*, pages 625–636, Toulouse, France.