Evolutionary Testing of State-Based Programs

Phil McMinn[*] and Mike Holcombe
Department of Computer Science,
University of Sheffield,
Regent Court, 211 Portobello Street,
Sheffield, S1 4DP, UK
{p.mcminn,m.holcombe}@dcs.shef.ac.uk

ABSTRACT

The application of Evolutionary Algorithms to structural test data generation, known as Evolutionary Testing, has to date largely focused on programs with input-output behavior. However, the existence of state behavior in test objects presents additional challenges for Evolutionary Testing, not least because certain test goals may require a search for a *sequence* of inputs to the test object. Furthermore, state-based test objects often make use of internal variables such as boolean flags, enumerations and counters for managing or querying their internal state. These types of variables can lead to a loss of information in computing fitness values, producing coarse or flat fitness landscapes. This results in the search receiving less guidance, and the chances of finding required test data are decreased.

This paper proposes an extended approach based on previous works. Input sequences are generated, and internal variable problems are addressed through hybridization with an extended *Chaining Approach*. The basic idea of the Chaining Approach is to find a sequence of statements, involving internal variables, which need to be executed prior to the test goal. By requiring these statements are executed, information previously unavailable to the search can be made use of, possibly guiding it into potentially promising and unexplored areas of the test object's input domain. A number of experiments demonstrate the value of the approach.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

Terms: Verification

Keywords: Evolutionary Testing, Chaining Approach, automated test data generation, state-based programs

[*]Corresponding author

1. INTRODUCTION

To date, the generation of test data by means of Evolutionary Algorithms, known as Evolutionary Testing, has been largely concentrated on the structural testing of individual program functions with input-output behavior [12]. Test data is generated for atomic function calls. However, functions and components at higher system levels can store internal data, and can exhibit different behaviors depending on the state of that data. This presents additional challenges to the Evolutionary Testing method. The first issue is the generation of input *sequences* to the test object, simply because some program structures may require the test object to be put into some state in order for them to be reachable. For example, program statements popping a value from a stack would not normally be feasible until the stack is in a non-empty state. Furthermore, state-based test objects often make use of internal variables such as boolean flags, enumerations and counters for managing or querying their internal state. These types of variables can lead to a loss of information in computing the fitness function, producing coarse or flat fitness landscapes. As a consequence, the search receives less guidance, and the chances of finding the required input sequences are decreased.

The approach proposed by this paper extends previous work of Baresel *et al.* [3] which allows input sequences to be generated, and the authors' own work [14, 15] addressing the problem of internal variables through hybridization with an extended *Chaining Approach*. The Chaining Approach was originally developed by Ferguson and Korel [7, 8, 11]. The basic idea of the approach is to identify a sequence of statements, involving internal variables, which need to be executed prior to each individual target structure. By requiring these statements are executed, information previously unavailable to the search can be made use of, possibly guiding it into potentially promising and unexplored areas of the test object's input domain. In this way, the chances of finding input sequences to troublesome structural targets may be improved.

2. EVOLUTIONARY STRUCTURAL TESTING

Structural testing coverage criteria demand that test data be found to execute all program structures of a certain type, for example all statements 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.

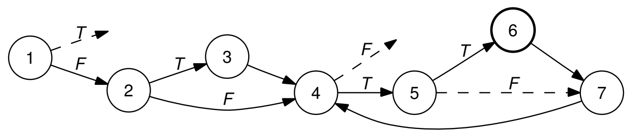


Figure 1: An example control flow graph for calculating approach levels with respect to a target node - node 5. Critical branches are indicated with dashed arrows

Real-valued encodings are used, with individuals directly representing input vectors to a function of the program currently under test [19]. In more developed approaches [5, 19] the fitness function is made up of two components. The first component is the *approach level* (sometimes referred to in the literature as the approximation level). The approach level metric assesses how close an input vector is to reaching some structural target on the basis of the execution path it takes through the program's control structure. Central to this is the notion of a *critical branch* (also referred to in the literature as *decisive branch*). A critical branch is simply a program branch which leads to a miss of the current structural target for which test data is sought. Once such a branch is taken through the program's control structure for some input vector, failure to reach the target has essentially been "decided". Take the example control flow graph of Figure 1. Suppose the goal of the search is to find test data for execution of the program statement corresponding to node 6. Critical branches include the true branch from node 1 and the false branch from node 4, because if either branch is taken, node 6 cannot be reached. For improved handling of structures nested within loops, some approaches [5, 17] count branches leading to a miss of the target within a loop iteration as critical. In the example, therefore, the false branch from node 5 is also treated as critical with respect to node 6.

The approach level for an individual (an input vector) 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. At the point at which control flow takes a critical branch for an 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 example if the false branch were taken from node 4, and the branching condition at this node is `(x == y)`, the branch distance for taking the alternative true branch is computed using the formula `abs(x - y)` (see reference [18] for a full list of formulas for different condition types). The branch distance d is normalized using the following function [1]:

$$normalize(d) = 1 - 1.001^{-d} \tag{1}$$

This value is added to the approach level to make up the total fitness value:

$$approach\_level + normalize(d) \tag{2}$$

3. GENERATION OF TEST DATA FOR STATE-BASED TEST OBJECTS

The procedure described in the previous section generates input vectors for test objects with input-output behavior. This section describes two methods for generating test data for state-based test objects. The first generates input sequences for test objects with several callable functions, with the second method extending the first through hybridization with an extended Chaining Approach, in order to overcome internal variable problems.

3.1 Method 1 - Sequence Generation

The method described in the last section generates input vectors for atomic function calls. Test objects with states, however, may require input sequences to be found so that structures dependent on the state are reachable. The input sequence may involve several different functions. For example in the case of a stack module, statements removing an element from the top of a stack in the "pop" function can not be covered until something has been pushed onto the stack via the "push" function.

Tonella [16] generates input sequences using Evolutionary Algorithms for the structural testing of classes. The encoding used is relatively complex, incorporating constructor calls for object creation, and method calls to the objects concerned. The focus of this paper, however, is generation of test data for programs written in the procedural paradigm. Baresel *et al.* [3] generate input sequences for states for single function test objects where the encoding used for single function calls - as described in the last section - repeated *num_calls* times, in order to represent a sequence of function calls of length *num_calls*, the value of *num_calls* being set for each test object by the tester. The individual now has several possible opportunities to cover the desired structure. The fitness value of the sequence is simply the value of Equation 2 for the function call which was closest to executing the target structure. In other words, the value of Equation 2 is computed for each function call $i, i \leq num\_calls$ and stored in $fitness_i$, with the smallest value of $fitness_i$ used as the final fitness value.

Method 1 of this paper extends the work of Baresel *et al.* for test objects with multiple functions. As a result, the encoding is more complicated, formed from a generic function call sub-encoding, which represents the possible call to any of the functions of the test object. This sub-encoding takes the form of a function identification number, and a universal parameter vector, which maps in a different way to the call signatures of the functions in the test object. An example of its construction can be seen in Figure 2. First, positions are assigned in the vector which correspond to the arguments of the first test object function encountered. The parameters `i`, `j` and `k` map to the first three positions, which are reserved for double, integer and integer types respectively. The parameters of the remaining functions are then mapped into this vector where possible. The integer parameter `p` of function 2 maps into position 2. The double parameter `q` maps into position 1. New positions are assigned for any parameters that do not map into the current vector. Therefore, a new position 4 is added to the end of the vector for the remaining double parameter `r` of function 2. As can be seen from the figure, when function signatures vary in terms of the numbers of variables of each different type there is some enforced redundancy.

Function signatures:

```
void function1(double i, int j, int k)
void function2(int p, double q, double r)
```

Generic encoding for a function call:

Function ID	**Position 1** double argument (1)	**Position 2** integer argument (1)	**Position 3** integer argument (2)	**Position 4** double argument (2)

Mapping of encoding to function 1:

1	`i`	`j`	`k`	*ignored*

Mapping of encoding to function 2:

2	`q`	`p`	*ignored*	`r`

Figure 2: Generating a generic encoding and mapping the universal parameter set to individual function call signatures

The sub-encoding is then repeated *num_calls* times for a sequence of length *num_calls*. In the case where the state-based test object only has one function, the function identification is dropped, and the encoding becomes identical to that of Baresel *et al.*

3.2 Method 2 - Sequence Generation with an Extended Chaining Approach

3.2.1 The Problem of Internal Variables

The use of internal variables in the conditions of programs can result in a degree of "information loss" when computing the branch distance measure, producing coarse or flat objective function landscapes for structures within the program. This in turn results in the search receiving less guidance, making it difficult for the search to find the required test data. The degree of difficulty depends on the type of internal variable and the form of assignments to it that appear in the program. Some internal variables may only result in a small amount of information loss, which may not affect the success of search. However, in extreme cases, such as in the case of boolean flag variables, almost all useful branch distance information is lost [2, 4, 6, 9, 10]. This is because the flag can only have one of two values - true or false, which in turn means the branch distance will only have one of two values - one or zero. This results in the formation of two plateaux in the fitness landscape; the first corresponds to the "one" distance, or all inputs which do result in coverage of the target structure, and the second corresponding to the "zero" distance, corresponding to the desired test data. No guidance is provided to the search as to how to navigate from one plane to the other. This is true in the example of Figure 3. The plateau corresponding to the false value of the flag can clearly be seen in Figure 4. The flag is only true when the input value of `i` is zero. However, because of the plateau, the search is not provided with any direction as to how to find this value.

State-based test objects contain internal variables in order to manage the state, which can cause problems for the search in a similar way to that described above. This work incorporates an extended Chaining Approach to help overcome this problem, as already successfully applied to internal variable problems for input-output programs [15].

```
Node
 (s)    void flag_example(int i)
        {
 (1)        int flag = 0;
 (2)        if (i == 0)
 (3)            flag = 1;
 (4)        if (flag)
 (5)          // target node
 (e)    }
```

Figure 3: Flag example code

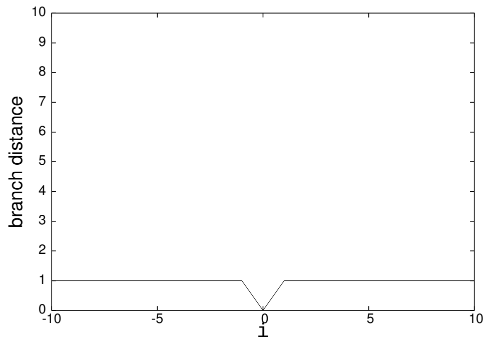


Figure 4: Node 4 - true branch distance landscape

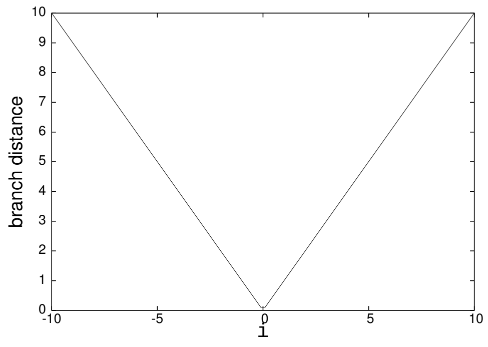


Figure 5: Node 2 - true branch distance landscape

3.2.2 The Chaining Approach

The Chaining Approach was originally developed by Ferguson and Korel [7, 8, 11], utilizing a local search method. It is of interest to this work because it incorporates a "backup" strategy when the search for test data fails, for example due to a coarse or flat fitness landscape caused by the use of internal variables. The basic idea of the Chaining Approach is to identify a sequence of program nodes, involving internal variables, which may need to be executed prior to the target structure. In doing this, further information can be made available to the fitness function and greater guidance can be provided to the search. Consider the example of Figure 3 again. If the search took into account the fact that node 3 should be executed prior to node 4, in order to set the flag to the required true value, branch distance information based on the true branch predicate at node 2 could be used. This information is far more conducive to finding the required test data (Figure 5). This is precisely the strategy the Chaining Approach employs, using the concept of an *event sequence*.

An event sequence is a sequence of events $\langle e_1, e_2, ...e_k \rangle$, 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". The constraint set is simply a set of variables that must not be modified until the next event in the sequence. Initially, the event sequence consists of the start node s and the target nodes. For the example of Figure 3 the initial event sequence is:

$$\langle (s, \emptyset), (5, \emptyset) \rangle$$

The search attempts to find test data for the initial event sequence, but is likely to fail due to a lack of guidance caused by the use of the flag variable. When a search fails, the Chaining Approach identifies the node where the flow of execution diverges away from the intended path. This node is called the *problem node*. Node 4 is identified as a problem node, because the flow of execution at node 4 can not be altered so that the false critical branch is avoided, with the alternative true branch taken instead. *Last definition* nodes of variables used at the problem node are then identified. Definitions of the flag variable prior to node 4 are to be found at nodes 1 and 3. New event sequences are then generated. In the example, two newly generated sequences are formed, corresponding to the two last definitions. The first requires the execution of node 1 before nodes 4 and 5:

$$\langle (s, \emptyset), (1, \{flag\}), (4, \emptyset), (5, \emptyset) \rangle$$

The second requires the execution of node 3 before nodes 4 and 5:

$$\langle (s, \emptyset), (3, \{flag\}), (4, \emptyset), (5, \emptyset) \rangle$$

The variable $flag$ is inserted into the constraint set of the events corresponding to nodes 1 and 3, to avoid any redefinition of the variable before node 4, which would destroy the effect of the last definition.

The first new event sequence is infeasible, and the search for test data fails. However, in attempting to find test data for the second new event sequence, the true branch distance information at node 2 can be made use of by the search, and test data is likely to be found.

The Chaining Approach generates more event sequences if further problem nodes are encountered. Event sequences are arranged in a tree, with child sequences developed from each parent sequence containing the problem node. The process repeats until test data is found or a certain depth in the tree has been reached, or a number of event sequences have been considered. More details can be found in references [7, 8, 11].

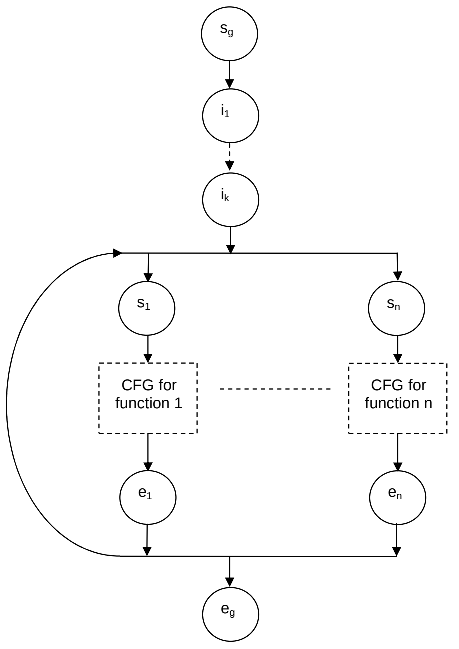


Figure 6: Global control flow graph

3.2.3 Description of the State-Based Hybrid Method

The state-based hybrid method (Method 2) supplements Method 1 with an extended Chaining Approach.

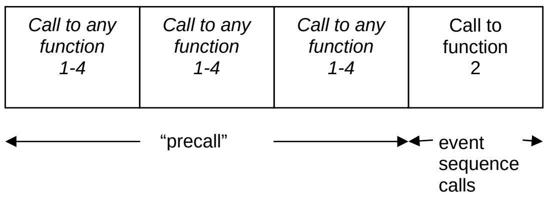
For the purposes of finding last definitions for the chaining algorithm, a global control flow graph is formed from the individual control flow graphs of each function in the test object. A special global "entry" start node s_g is created, with a set of initializing nodes $i_1...i_k$, corresponding to the initialization of any variables global to functions in the test object. Edges lead from s_g through each initializing node, and from the final initialization node i_k to the start node of the control graph of each individual function of the test object. Further edges lead from each function's end node to every function's start node.

The original chaining algorithm of Ferguson and Korel [7] is extended in several ways. The extended algorithm features an extended event sequence generation algorithm. This uses the concept of an "influencing set" to identify all variables that can have an influence on the problem node via some program path. Event sequences are generated on the basis of assignments to these variables. The original approach only considers last definition assignments to variables involved in conditions at the problem node. Furthermore, the extended algorithm contains a "return to problem node" capability. If the problem node is encountered more than once by the original chaining approach, it declares failure. The hybrid approach has the option to generate more event sequences which may aid the test data generation process, and overcome the problem node. Finally, the extended algorithm can handle conditions using logical AND and OR connectives, which the original approach could not. Further details can be found in reference [13].

The encoding of individuals for the evolutionary algorithm is split into two parts. The initial part, referred to as the "precall" allows the evolutionary algorithm to freely choose which functions are to be called, and thus is identical to the encoding of Method 1. The functions to be called for the remaining part are determined by the nodes in the event sequence. The second part of the encoding is referred to as the "event sequence calls" section, and is reserved for the function calls that need to be performed in order to execute each event node (except events corresponding to s_g through to i_k, which will always be executed in the first call) in the event sequence. Since the functions to be called are known, no function identification number or universal parameter set is required, and the encoding for each function call is simply based on the parameters to that function. As an example, take the following event sequence, for the execution of the branch (6, 7) for an arbitrary test object with 4 different functions:

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

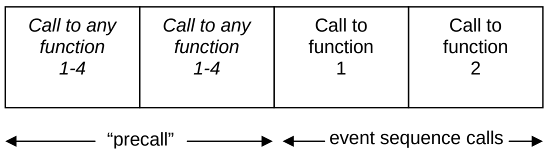
Suppose nodes 6 and 7 lie in function number 2. The initial sequence length is set at 4. The encoding therefore stipulates any function can be called in positions 1-3, whilst function 2 has to be called in position 3:



The length of the precall sequence shrinks as events are added before the original problem node. Suppose node 6 is a problem node, and node 5 is inserted before it in the event sequence:

$$< (s_g, \emptyset), (5, \{var\}), (6, \emptyset), (7, \emptyset) >$$

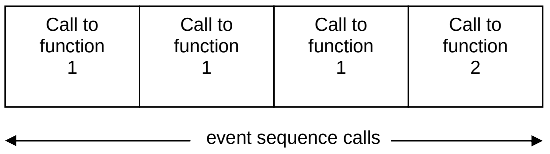
Suppose node 5 lies in function 1. The precall shrinks to a length of 2:



If more functions are required before the initial problem node than the number of calls in the precall, the precall disappears. Suppose the following event sequence is generated:

$$< (s_g, \emptyset), (5, \{var\}), (5, \{var\}), (5, \{var\}), (6, \emptyset), (7, \emptyset) >$$

The precall section no longer exists, leaving an encoding that consists of function calls for the event sequence only:



The length of the precall for an event sequence $precall\_len$ is therefore computed using the formula:

$$precall\_len = max(0, init\_seq\_len - init\_prob\_event\_pos)$$

If the event sequence is the initial event sequence, the value of $init\_seq\_len$ equals $num\_calls - 1$ (i.e. num_calls as used in Method 1 and set by the tester) and $init\_prob\_event\_pos$ is zero. Otherwise, $init\_seq\_len$ is based on the best individual for the initial event sequence test data search - being the call number in the sequence where target is closest to being executed. $init\_prob\_event\_pos$ is the position of the event in the current sequence that was the first problem node identified from the initial event sequence.

Events can also be inserted between the initial problem node and the event or events corresponding to the target structure. In this case the precall length remains unchanged, but the number of function calls required to execute the event sequence can increase. The functions required to be executed in the event sequence calls portion of the encoding are found by using the following simple algorithm, which takes each pair of adjacent events $e_i = (n_i, C_i)$ and $e_{i+1} = (n_{i+1}, C_{i+1})$ in the event sequence. If n_{i+1} is not reachable from e_n via an acyclic path within the same function, the function parameters are added to the event sequence function calls portion of the encoding.

The fitness function is identical to that of McMinn *et al.* [15], defined for an event sequence E of length l:

$$\sum_{i=1}^{l} fitness(e_i) \tag{3}$$

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:

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

2. For each definition node $def(v)$ 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 $def(v)$'s execution.

4. EXPERIMENTAL STUDY

An experimental study was conducted using both methods described in the previous section. Full branch coverage was attempted for ten state-based test objects written in the C language. Details of each test object can be found in Table 4. "Anomaly Detector" is a small module which monitors a stream of incoming data. Its purpose is to determine whether incoming data values are inconsistent with regards to a history of past entered values. "Array Difference" consists of one function, which takes an array of ten integers as a parameter. The function returns true if the current inputted array contains the same values as for the array inputted in the last call of the function. "Postcode" checks whether stream of inputted characters represents a UK postcode. "Sliding Window" is an implementation of the sliding

Table 1: Test object details

Test object	Lines of code	Branches	Loops	Max. nesting level	Functions (public)
Anomaly Detector	59	14	2	3	3 (3)
Array Difference	31	12	2	3	1 (1)
Postcode	170	50	0	10	5 (1)
Sliding Window	127	24	3	3	10 (4)
Smoke Detector	40	14	0	2	1 (1)
Sortcode	97	26	0	4	4 (1)
Stack	51	8	0	1	5 (5)
Tel. Number	80	22	0	4	1 (1)
Vending Machine	112	26	4	4	5 (4)
Industrial Example	425	128	0	10	1 (1)

window network protocol. "Smoke Detector" models a small controller for a smoke detector. "Sortcode" validates a UK bank sortcode, whilst "Telephone number" validates a telephone number. "Stack" implements a small stack, whilst the industrial example is a car controller module, kindly provided by DaimlerChrysler Research and Technology. For source code listings (excluding the industrial example), see reference [13].

4.1 Experimental Setup

For the evolutionary searches, 300 individuals were used per generation. Each generation was split into 6 subpopulations, with competition and migration across subpopulations. Linear ranking was used with a selection pressure of 1.7. Individuals are recombined using discrete recombination, and mutated using real-valued mutation. For Method 1, searches were terminated after 200 generations.

For Method 2, with chaining capabilities, an evolutionary search takes place for each event sequence, resulting in potentially several searches for each structure. Here, evolutionary searches are terminated after 50 generations of no improvement in the best fitness value (see reference [15]). The chaining tree was explored in a breadth-first fashion, terminating after the consideration of 200 event sequences if no test data could be found.

Each method was repeated ten times for each test object and branch. For each method and test object, the coverage obtained and the "success rate" was measured. The success rate measures the percentage of repetitions that resulted in the required test data being found. Coverage measures the percentage of branches for which at least one of the ten repetitions of the method was successful.

The value of *num_calls* used in the experiments for each test object were as follows: Anomaly Detector (45); Array Difference (2); Postcode (10); Sliding Window (6); Smoke Detector (10); Sortcode (10); Stack (45); Telephone Number (15); Vending Machine (5).

Table 2: Method 1 search stagnation

Test object	Average generation of last improvement
Anomaly Detector	1
Array Difference	1
Postcode	45
Sliding Window	1
Smoke Detector	7.3
Sortcode	10.5
Stack	1
Tel. Number	31.4
Vending Machine	3.1
Industrial Example	9.7

4.2 Results

Figure 7 shows success rates and coverage levels for each method and test object. In all cases Method 2 - with chaining capabilities - achieved the same or higher success rate and coverage. For Method 1, 100% coverage is only achieved with one test object - Sliding Window. For the other test objects, the presence of internal variables inhibited the search. Method 2, with chaining capabilities, was able to overcome some of these problems, achieving 100% coverage for seven of the ten test objects. These results only come with more effort - Method 2 generally performs a higher number of test data (fitness) evaluations per branch (Figure 8) in order to search each event sequences. However, Table 4.1, shows that unsuccessful searches for Method 1 stagnate early, and would be unlikely to find required test data even if the termination criterion were changed so that a comparable number of evaluations could be performed.

For Method 2, some branches were not covered due to the fact that the chaining tree became too large for all feasible event sequences to be explored. The 200 event sequence limit on chaining tree exploration was reached on 5,10 and 15 occasions for the Postcode, Sortcode and Telephone Number test objects respectively. For the Smoke Detector test object, experiments were re-run ten times with a termination criterion of 100 event sequences for chaining tree exploration. Test data was generated with a 100% success rate. The original limit of 200 event sequences was broken on 12 occasions, reaching a maximum of 622 event sequences for one particular branch. Furthermore, since event sequences can consist of several nodes, the method is particularly susceptible to evolutionary search problems involving nested statements or those dependent on composed conditions. These difficulties were originally observed by Baresel *et al.* [5]. Once input data is found for one or more of the conditions required to reach some program node, the chances of finding input data that also fits subsequent conditions decreases. This is because a solution for these later conditions must be found without violating any of the earlier conditions, since all the conditions can not be evaluated at once. As a result, the search performs poorly. For composed conditions, a similar effect occurs due to the use of the short-circuiting `&&` and `||` operators. In this way, test data generation was hindered for the Postcode, Sortcode, Telephone Number and industrial example, which have high levels of nesting or many composed conditions. The Postcode test object in particular has a high number of nested nodes, and suffered the most problems. Two branches were found to be infeasible for the industrial example.

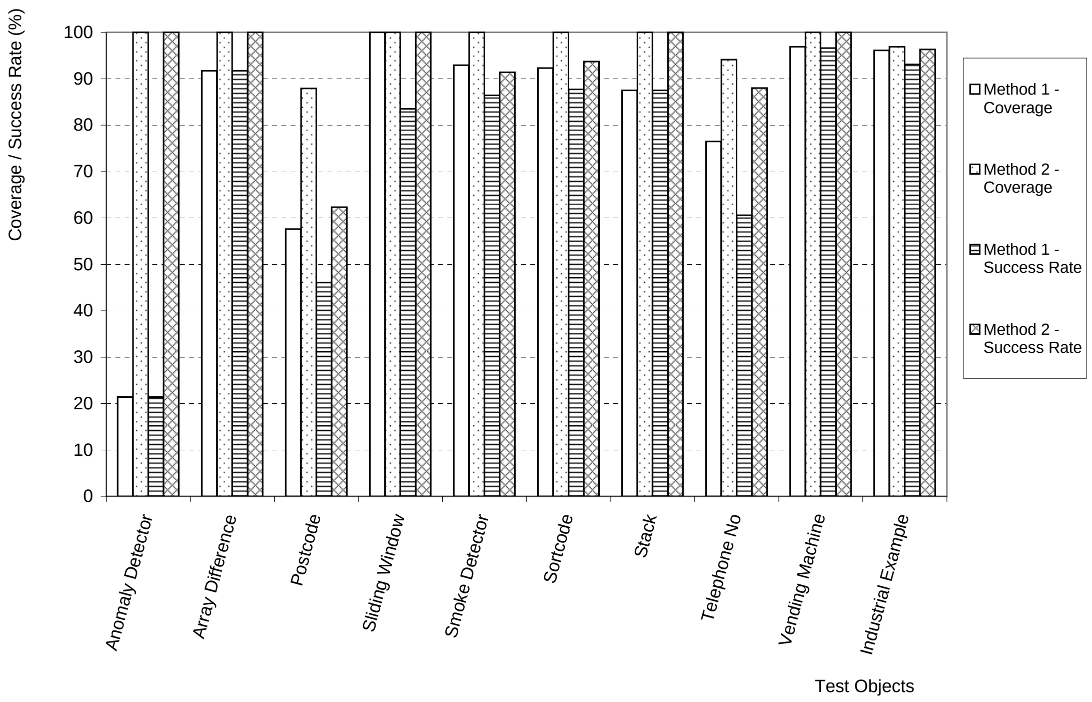


Figure 7: Coverage and success rate

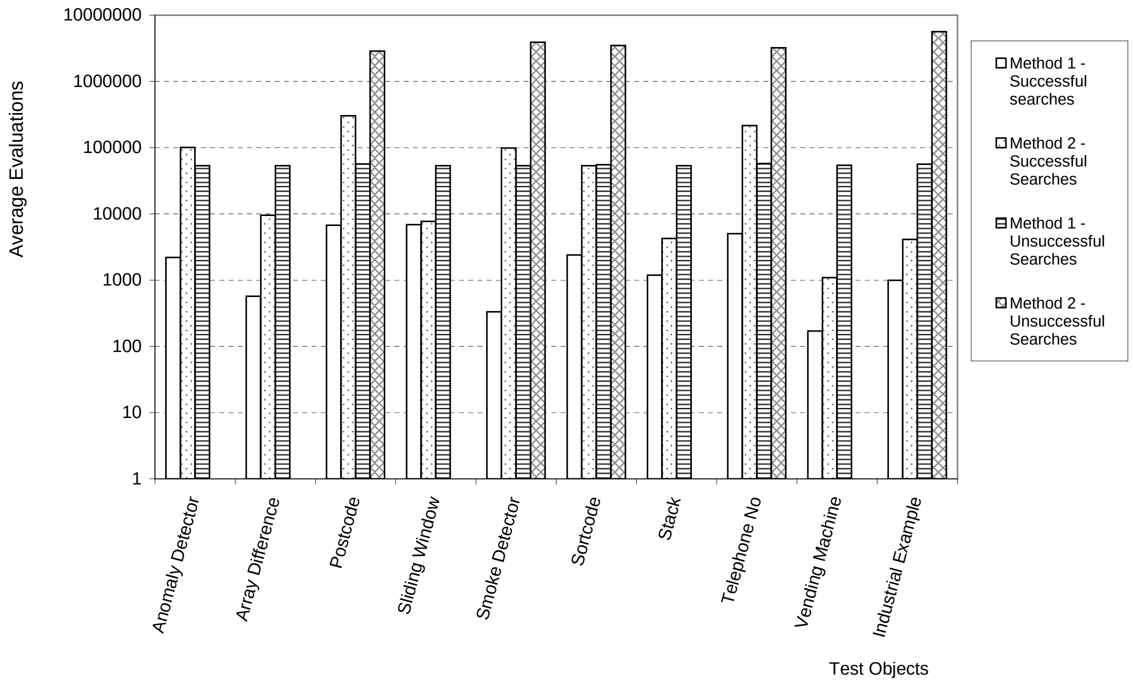


Figure 8: Average number of evaluations

5. CONCLUSIONS AND FUTURE WORK

This paper has investigated the evolutionary testing of state-based, procedural test objects. A method was proposed for the generation of input sequences, in order to accommodate the coverage state-dependent structures which can not be covered using traditional techniques tailored for input-output functions. This was then extended by hybridization with an extended Chaining Approach, in order to overcome internal variable problems which prevent the search receiving adequate guidance to the required test data.

Experiments were performed with ten state-based test objects, including one industrial example. In all experiments performed, the hybrid method achieved higher coverage levels and success rates. Certain structures still could not be covered, however. In some cases full exploration of the chaining tree was not possible. Furthermore, the method was inhibited by problems resulting from high levels of nesting in programs and short-circuiting in composed conditions. Future work will look at addressing these problems. The problem of nesting and composed conditions could be dealt with through the use of a Testability Transformation [10]. A possible solution is outlined in reference [13]. The problem of chaining tree size might be tackled by performing more rigorous program analysis in order to rule out certain event sequences which can not lead to the discovery of the required test data [13].

6. ACKNOWLEDGEMENTS

This work was funded by DaimlerChrylser Research and Technology. The authors would like to thank DaimlerChrysler for providing the industrial example used in the experiments.

7. REFERENCES

[1] A. Baresel. Automatisierung von strukturtests mit evolutionren algorithmen. Diploma Thesis, Humboldt University, Berlin, Germany, July 2000.

[2] A. Baresel, D. Binkley, M. Harman, and B. Korel. 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, 2004. ACM.

[3] 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 vol. 2724*, pages 2428 – 2441, Chicago, USA, 2003. Springer-Verlag.

[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 vol. 2724*, pages 2442 – 2454, Chicago, USA, 2003. Springer-Verlag.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

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

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

[14] P. McMinn and M. Holcombe. The state problem for evolutionary testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724*, pages 2488–2497, Chicago, USA, 2003. Springer-Verlag.

[15] P. McMinn and M. Holcombe. 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, 2004. Springer-Verlag.

[16] P. Tonella. Evolutionary testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 119–128, Boston, USA, 2004. ACM Press.

[17] N. Tracey. *A Search-Based Automated Test-Data Generation Framework for Safety Critical Software*. PhD thesis, University of York, 2000.

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

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