Fitness Function Design to improve Evolutionary Structural Testing

André Baresel, Harmen Sthamer and Michael Schmidt
DaimlerChrysler AG, Research and Technology
Alt-Moabit 96a, 10559 Berlin, Germany, +49 30 39982 222
{andre.baresel;harmen.sthamer;michael.a.schmidt}@daimlerchrysler.com

Abstract

Evolutionary Structural Testing uses Evolutionary Algorithms (EA) to search for specific test data that provide high structural coverage of the software under test.

A necessary characteristic of evolutionary structural testing is that the fitness function is constructed on the basis of the software under test. The fitness function itself is not of interest for the problem; however, a well-constructed fitness function may substantially increase the chance of finding a solution and reaching higher coverage. Better guidance of the search can result in optimizations with less iterations, therefore leading to savings in resource expenditure.

This paper presents research results on suggested modifications to the fitness function leading to the improvement of evolutionary testability by achieving higher coverage with less resources. A set of problems and their respective solutions are discussed.

1 INTRODUCTION

Evolutionary testing designates the use of metaheuristic search methods for test case generation. The input domain of the test object forms the search space in which one searches for test data that fulfil the respective test goal. Due to the non-linearity of software (if-statements, loops, etc.), the conversion of test problems into optimization tasks mostly results in complex, discontinuous, and non-linear search spaces. The use of neighborhood search methods such as hill climbing are therefore not recommended. Instead, metaheuristic search methods are employed, e.g. evolutionary algorithms, simulated annealing or tabu search. In this work, evolutionary algorithms are utilized to generate test data, since their robustness and suitability for the solution of different test tasks has been proven in previous work. e.g. [19] and [17].

The only prerequisites for the application of Evolutionary Tests (ET) are an executable test object and its interface specification. In addition, for the automation of structural testing, the source code of the test object must be available to enable its instrumentation.

1.1 A BRIEF INTRODUCTION TO EVOLUTIONARY ALGORITHMS

Evolutionary algorithms represent a class of adaptive search techniques and procedures based on the processes of natural genetics and Darwin's theory of biological evolution. They are characterized by an iterative procedure and work in parallel on a number of potential solutions for a population of individuals. Permissible solution values for the variables of the optimization problem are encoded in each individual. An overview of evolutionary algorithms is presented in Figure 1 and a detailed description can be found in [20] and [12].

The fitness value is a numerical value that expresses the performance of an individual with regard to the current optimum so that different individuals can be compared. Usually a spread of solutions exists ranging in fitness from very poor to good. The notion of fitness is fundamental to the application of evolutionary algorithms; the degree of success in their application may depend critically on the definition of a fitness that changes neither too rapidly nor too slowly with the design parameters of the optimization problem. The fitness function must guarantee that individuals can be differentiated according to their suitability for solving the optimization problem.

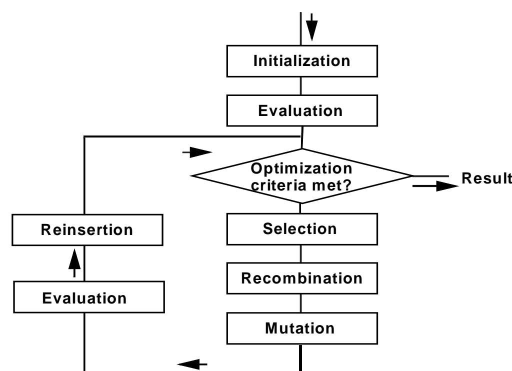


Figure 1: Overview of a typical procedure for evolutionary algorithms

Our experiments used a population of 300 individuals split into 6 subpopulations of 50 individuals. In order to combine multiple strategies, migration was introduced to permit an exchange of the best individuals between subpopulations at regular intervals. The subpopulations

also compete with each other. Strong ones receive more individuals, the others diminish in size. Details of the evolutionary settings are described in [20].

1.2 APPLICATION TO SOFTWARE TESTING

In order to automate software tests with the aid of evolutionary algorithms, the test goal itself must be transformed into an optimization task. This necessitates a numeric representation of the test goal, from which a suitable fitness function for evaluation of the test data generated may be derived. Different fitness functions emerge for test data evaluation according to which test goal is pursued. For structural testing, fitness functions may be based on a computation for each individual that indicates its distance from the desired program predicate execution [19] and [17]. For example, if a branching condition *"x==y"* needs to be evaluated as *True*, then the fitness function may be defined as $|x-y|$ (the fitness values are minimized).

Each individual within the population represents a test datum with which the test object is executed. For each test datum the execution is monitored, and the fitness value for the corresponding individual determined. It is important to ensure the test data generated are in the input domain of the test object.

2 STRUCTURE OF FITNESS FUNCTION

Structural testing is widespread in industrial practice and stipulated in many software development standards, e.g. [22], [23], [24], and [25]. The execution of all statements (*statement coverage*), all branches (*branch coverage*), or all conditions with the logical values *True* and *False* (*condition coverage*) are common test aims. The aim of applying Evolutionary Testing to structural testing is the generation of a quantity of test data, leading to the best possible coverage of the respective structural test criterion.

Whereas all previous work from other authors has concentrated on single selected structural test criteria (*statement-*, *branch-*, *condition* and *path-test*), DaimlerChrysler Research has generated a test environment to support all common control-flow and data-flow oriented test methods [20]. For this purpose, the structural test criteria are divided into four categories, depending on control-flow graph and required test purpose:

- node-oriented methods,
- path-oriented methods,
- node-path-oriented methods, and
- node-node-oriented methods.

The separation of the test into partial aims and the definition of fitness functions for partial aims are performed in the same manner for each category. Each partial aim represents a program structure that requires execution in order to achieve full coverage of the selected structural test criterion, i.e. each single statement represents a partial aim when using statement coverage criterion. For the Evolutionary Test, the test therefore has to be divided into partial aims. These depend on the specified structural test criteria. Identification of partial aims is based on the control-flow graph of the program under test.

As mentioned previously, the definition of a fitness function that represents the test aim accurately, and supports the guidance of the search, is conditional to the successful application of Evolutionary Tests. In order to define the fitness function, this research builds upon previous work dealing with branching conditions (among others [17], [8], and [18]). These are extended in [20] by introducing the idea of an approximation level. A more detailed definition of approximation level for node-oriented and path-oriented methods is provided in sections 2.1 and 2.2. These are the basis for the remaining node-path-oriented and node-node-oriented methods and, for this reason, the last two methods are not further discussed here.

2.1 NODE-ORIENTED

Node-oriented methods require the attainment of specific nodes in the control-flow graph. The *statement test* as well as the different variants of the *condition test* may be classified in this category. As regards condition testing ([10] and [2]), a special case applies for the fulfillment of the respective test criterion. In addition to the branch nodes, the necessary logical value allocations for the atomic predicates in the conditions must also be attained.

For node-oriented methods, partial aims result from the nodes of the control-flow graph. The objective of the Evolutionary Test is to find a test data set that executes every desired node of the control-flow graph. For the *statement test*, all nodes need to be considered; for the different variations of the *condition test*, only the branching nodes are relevant. Condition testing also requires the predicates of the branching conditions to be evaluated. In the case of the *simple condition test*, for example, the evaluation of each of the atomic predicates must be inventoried to represent *True* and *False* partial aims. In the case of the *multiple condition test*, all combinations of logical values for the atomic predicates form independent partial aims.

In node-oriented methods, the fulfillment of a partial aim is independent of the path executed in the control-flow graph. This has been taken into account by our fitness function. The fitness functions of the partial aims consist of two components. In addition to the calculation of the distance in the branching nodes; which specifies how far away an individual is from fulfilling the respective branching condition (compare [17], [8], and [18]); an approximation level is introduced as an additional element for the fitness evaluation of individuals:

```
Fitness = AL  +  DIST
AL:   approximation level (natural numbers)
DIST: normalized  local  condition  distance
(value range 0..1)
```

The approximation level enables the comparison of individuals that miss the partial aim in different branching nodes (details in [20] and [1]). It indicates how close the

executed path is, as compared to the required partial aim. This extension enables different paths through the program, to the desired partial aim, to be treated equally with respect to the fitness evaluation. Unlike previous work, it is unnecessary to select a specific path to a distinct node through the control-flow graph. In this approach, only the execution of the specific node is of relevance. The higher the attained level of approximation the better the fitness of the individual. This extends the idea stated in [17], where a small fixed value was added to the calculated distance for every executed node belonging to a path that leads to the target node. Therefore, individuals closer to the target node receive a higher fitness value as compared to those that branch away earlier.

2.2 PATH-ORIENTED

Path-oriented methods require the execution of certain paths in the control-flow graph. All variations of *path tests,* from the *reduced path test* [6] to *complete path coverage* [7], belong to this category. Therefore all paths through the control-flow graph, necessary to fulfil the chosen structural test criterion, are determined and identified as partial aims.

Establishing fitness functions for path-oriented test methods is much simpler than for node-oriented methods because the execution of a certain path through the control-flow graph forms the partial aim for the Evolutionary Test. Corresponding to the node-oriented methods, the fitness function for path-oriented methods consists of two components: approximation level and distance calculation.

The covered program path is compared to the program path specified as a partial aim in such a way that the length of the identical initial path section reflects the approximation level (compare [9]), and the last non-fitting condition is used for distance evaluation.

3 IMPROVING THE FITNESS FUNCTION

In Evolutionary Structural Testing, the fitness function is constructed on the basis of the software tested. The function itself is not of interest for the problem, the only goal is to find a test datum that fits a test criterion. A well-constructed function can:

- considerably increase the chance of finding the solution and reach a better coverage of the software under test and
- result in a better guidance of the search and thus in optimizations with less iterations.

Other work on designing fitness functions and the results of the optimization process can be found in [8]; this investigates the use of various distance functions. Hamming distance, reciprocal function and their influence on optimization performance are discussed. In [8], a decision was made in favor of the Hamming distance because the authors used genetic algorithms with a bit representation of all parameters in their approach. Evolutionary algorithms are used with integer and floating point representations and their corresponding mutation and recombination operators. For this reason Hamming distance is not investigated in this paper.

Modifying the distance function of branch conditions is only one possible mechanism for modifying the fitness function. In the following sections it is argued that more general alterations to the fitness function may lead to better results in Evolutionary Testing. This results in a higher chance of finding the solution or a better performance of the optimization process in general.

Evolutionary testing has been successfully used for complex functions (e.g. shown in [20]). The optimization problem often relies on some details, for this reason the authors present very simple examples for the discussion.

3.1 COMPOSED CONDITIONS AND NESTED BRANCHES

In the general solution proposed, the test object is instrumented in such a way that the semantics of the software under test is not changed by the added code. Special care is taken regarding side effects that might occur during the execution of branch conditions if short circuit evaluation is implemented by the compiler. To keep side effects unchanged, the fitness function should only take into account the results of the executed parts of the condition. This leads to a problem with optimization performance and, at worst, to a low chance of successful optimization. Example 1 presents an extreme situation in which the solution of an input, which evaluates the condition as *True*, can only be found by optimizing the test data for the atomic conditions one after the other.

```
if(a==0 && b==0 && c==0 && e==0 && f==0)
  {  /* ex 1 - to be executed */ }
```

Example 1: Composed condition and its problem with short circuit evaluation

There is a very low probability of coming across a solution by chance which fits all of these atomic conditions. When executing this example code, only the first parts of the condition are evaluated until one atomic condition is evaluated as *False*. Whenever an individual is found that fits one more atomic condition, the probability of finding a solution which also fits the next one decreases considerably. This is due to the fact that a better solution must be found by not changing those input parameter values that correspond to these first condition parts. The number of input parameters that should not be changed increases with each atomic condition that is executed in the desired way. The same problem occurs with nested *if-then-else* structures. Consequently, a better fitness function should be introduced which could compensate for this behavior. Resolution of this issue for nested *if-then-else* structures will now be discussed. As this example shows, a better guidance of the search takes into account all parts of the composed condition. This facilitates the optimization of individuals for all atomic conditions at the same time. The function is constructed by the summation of all atomic

condition distances. This increases the chance of more effective mutations and achieves a well-performing recombination.

For this, the test object has to be changed to enable evaluation of all atomic conditions for every test execution so that no short-circuit evaluation is performed. This is not a problem for side-effect-free conditions when an instrumentation code is added in front of the *if-statement*. If the test object does not allow this due to side effects, the removal of these side effects by program transformation as shown in [5] may improve evolutionary testability.

No side effects are present in Example 1. A fitness function built upon the distances of all atomic conditions fundamentally increases the chance of finding the right solution as shown in Figure 2.

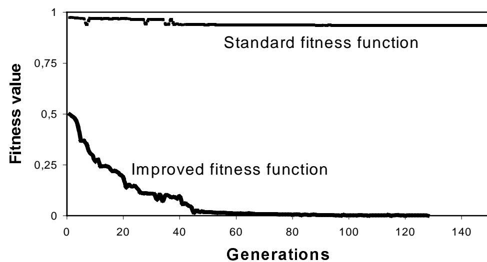


Figure 2: Optimization progress of the standard and improved fitness function

The fitness values of two test runs using the standard and improved fitness functions are displayed over the number of generations. The improved fitness function for complex conditions performs very well. It found a solution already after approximately 130 generations. However, using the standard fitness function, building only upon the evaluated atomic conditions, no individual was found within 1000 generations that achieved the partial aim.

Nested *if-then-else* structures may lead to the same behavior as the composed conditions. For nested *if-then-else* structures, shown in Example 2, the optimization is guided by the nested *if-conditions*. Static program analysis may identify the presence (or absence) of side effects. The knowledge of data dependencies makes it possible to pre-calculate where the value of a condition is fixed. In this case, the evaluation of conditions of inner *if-statements* may be performed earlier in the program under test in order to achieve the same improvements as described in Example 1.

```
If (a==0) {
   If (b==0) {
      If (c==0) {
      .../* ex1 */
```

Example 2: Code example for nested *if-statements*

A calculation of all conditions may be performed prior to the first *if-statement*. With this change of the fitness function, a search for a test datum that executes the statement *ex1* performs much better than a search with the fitness function solely on the basis of the executed program parts, since all conditions to be fulfilled are taken into account.

3.2 DEPENDENCIES WITHIN LOOP ITERATIONS

The fitness function that is used to optimize a test datum to execute a certain target node, as described in [11], takes control flow dependencies into account. These are all the branches of a program that lead to a part of the program from which the target node can no longer be reached, as shown in Figure 3. Loops have no special handling in this approach; this means that the evolutionary search of an input to traverse a target node within a loop has no guidance. Jones et al. [8] avoid this problem by unrolling the loop in the control-flow-graph for the fitness evaluation only.

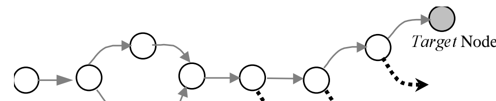


Figure 3: Target node with control dependencies

If the target node is inside a loop, every iteration produces another chance for traversing the target node as long as the loop is not exited. Missing the target node in one iteration has no effect on the fitness using control dependencies. Experiments showed that this hampers Evolutionary Testing efforts; since guidance to test data, resulting in a loop iteration where the execution is closer to the target node than others, is lacking. In many cases, this leads to a random search with a very low chance of finding a solution if the search space is large. Example 3 illustrates the problem.

```
for (idx=1;idx<=10;idx++)
{/* .... inner-pre-code ... */
 if (a==0) {
    if (b==0) {
    { /* Target Node – execute this*/ }
 /* .... inner-post-code .... */
}
```

Example 3: Dependencies in loops

In this simple example there is neither control dependence in the *pre-loop-code,* the *inner-pre-code*, nor in the *inner-post-code*. Only one branch has a control dependence for the selected target node; that is constructed by the loop header. This branch is executed when the counter *idx* is greater than 10. A fitness function building upon this information has no guidance to the target node; it gives the same poor fitness for all solutions that do not execute the target node. The search is therefore arbitrary since the search is not directed towards the execution of the target node.

A human tester would simply recognize additional information built by the nested *if-then* structure. This structure leads however, to no control dependence. This is due to the fact that if the target node in one iteration is missed, another chance presents itself in the next iteration. The solution proposed in this paper is to add dependencies

of one loop iteration to the fitness function. Whilst monitoring the execution of the test object, we can observe this information on all iterations and calculate a fitness from it. This may essentially improve the chance of finding a solution as shown in Figure 4.

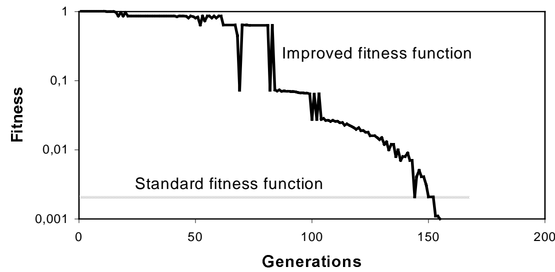


Figure 4: Optimization progress with regard to the control dependencies

Iteration control dependencies can be calculated by analyzing the control flow of one loop iteration. This leads to a set of branches that may miss the target node in a loop iteration. The approximation levels are also calculated for the additional branches. *Iteration control dependencies* are identified with the algorithm for control dependencies after removing the backward branches of a loop. The algorithm to distinguish control dependencies can be found in [16] (backward dominance).

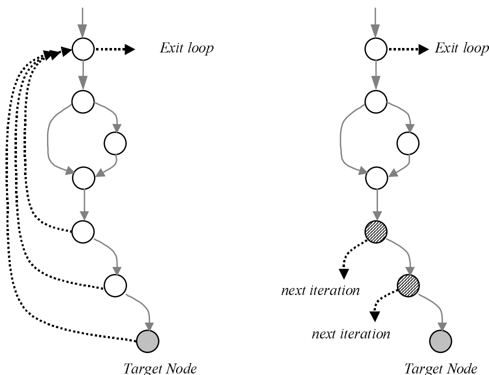


Figure 5: Control flow graph with *iteration control dependencies*

The figure shows the results for one example. The loop exit and backward branches have been highlighted in the left graph of Figure 5. In this example the target node is iteration-control-dependent from the two nodes where the backward branches begin as shown in the right graph.

When comparing the two fitness curves in Figure 4 one notices that, although the standard fitness function calculates a relatively good value from the beginning, whereas the other is relatively poor. This is because the standard fitness function is based on the local distance in *Exit loop*. This is not sufficiently meaningful for the target node and leads to stagnation. In contrast the improved fitness function also takes the conditions of both backward branches into account. Both fitness functions use the same evaluation principle, however the evaluations are carried out in different nodes.

The method used to estimate the backward branches for unstructured loops is not defined but first experiments using this approach are promising.

3.3 MEASURING PATH COVERAGE

The approach of applying evolutionary algorithms to path coverage cited in various publications ([9] and [18]) is to calculate the fitness of an individual by estimating the length of the first matching part of the target path and the actual executed path. This leads to a search where the solution is optimized for the branching conditions of the path in a stepwise manner. For example, when a solution is found for the first condition in the path, the next condition is considered. The poor behavior of an optimization, such as this, has been described previously in section 3.1.

A fitness function is introduced in which the length of all identical path sections is used as approximation level. This evaluation method is advantageous in that an individual diverging from the target path at the beginning, but covering the desired path towards the end, obtains a similarly high fitness value as compared to an individual covering the specified target path at the beginning, but diverging from it towards the end. The combination of two such individuals (recombination) may lead to a considerably better individual. In Figure 6 for instance, the execution of the first individual (covering the six nodes 1, 3, 4, 5, 7, and 8 on the target path), will obtain a high approximation level if all identical path sections are considered for the fitness evaluation. If only the first matching path section is measured, the second individual (covering five nodes 1, 2, 3, 4, and 7) will obtain a higher approximation level than the first one.

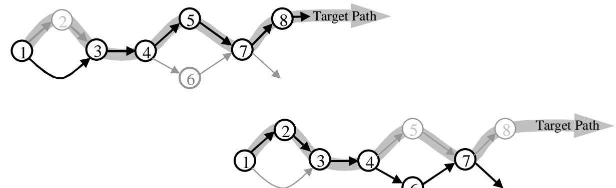


Figure 6: Execution of two individuals for a path-oriented test goal

The optimization goal is the gray path. Two possible execution paths have been highlighted. Example 4 illustrates the target path by *ex1*, *ex2* and *ex3*.

```
if (a==0) /* ex 1 */ ;
if (b==0) /* ex 2 */;
    else /* .... */;
if (c==0) /* ex 3*/ ;
    else return;
```

Example 4: Source code for path-oriented tests

Using the improved fitness function the optimization performs better in finding an input for the requested path that matches sub-paths because it always takes all conditions of the path into account. In contrast to this, other approaches, as described in many publications, optimize the solution condition by condition. The results are displayed in Figure 7.

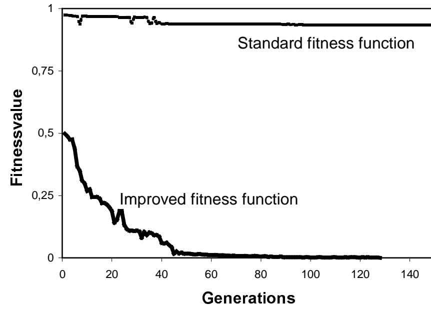


Figure 7: Optimization progress of path optimization

3.4 FITNESS OF DIFFERENT PATHS IN NODE ORIENTED TEST GOALS

For node-oriented test methods fitness evaluation is based on control dependencies of the target node. Measuring fitness is based on the point at which a control dependent node is evaluated incorrectly. An approximation level at this point is used to decide the closeness of the executed path to the target node.

A fitness evaluation was implemented that utilizes an approach level and a local condition distance. Approximation levels are calculated for all the nodes of a program that have a control dependence for the currently selected test goal. The control flow graph is examined for all possible execution orders of these special nodes. Based on this information, approximation levels are assigned. A more detailed description of approximation levels can be found in [20] and [1].

We were able to observe that in some instances the procedure of assigning the approximation levels, which lead to a well performing fitness function, was more complicated than just checking the execution order. This is further highlighted in Example 5.

```
  /* pre-code */
○ switch (a)
  {
○   case 1:
      if ( cond_1 ) return;
○     if ( cond_2 ) break;
      /* ... some code .... */
○   case 2:
      /* ... some code ... */
○     if ( cond_3 ) break;
      return;
  }
● /* TARGET NODE */
```

Example 5: Source code template to assign the approximation levels

Example 5 has three paths that do not execute the target node:

- path through "1 → 2 → return",
- "1→ 2 → 3 → 4 → 5 → return" and
- "1 → 4 → 5 → return"

The open question for designing the fitness function is the closeness of the branching node 2 to the target node, which can be compared in Figure 8.

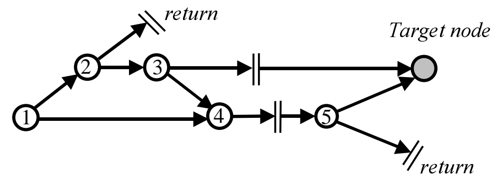


Figure 8: Control flow graph of Example 5

Node 5 gets the best approximation level since it leads directly to the target node if the branching condition is evaluated in the desired way. However, can node 2 achieve the same approximation level since one direct path to the target node exists? Or should this node obtain a different approximation level since another path leading to " 5 → return" exists? These two possibilities in assigning approximation levels to nodes will considerably alter the respective fitness function in the neighborhood of a solution that executes the target node. This may influence the performance of the search.

The above algorithms have been implemented on these two possible approximation level allocations in our structural test system. In this paper, they are called "optimistic" and "pessimistic" approximation levels since the level of node 2 is assigned on the basis of the direct path to the target node (optimistic), or to the path that misses the target node (pessimistic).

In order to check the behavior and performance of "optimistic" and "pessimistic" approximation levels, Example 5 has been tested in three versions with different conditions which can be seen in Example 6.

Version 1

```
cond_1: b>0 && b<4
cond_2: c>0 && c<10
cond_3: a == 1 && b==0 && c==0 && d==0
```

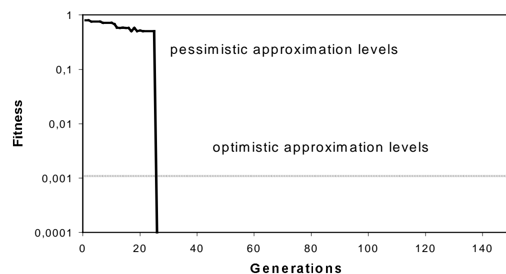
Version 2

```
cond_1: b==0
cond_2: c==2 && d==2
cond_3: b==0 && c==0 && d==0
```

Version 3

```
cond_1: b>0
cond_2: c==0 && d==0
cond_3: a==1 && b==0 && c==0
```

Example 6: Versions for "optimistic" and "pessimistic" approximation levels



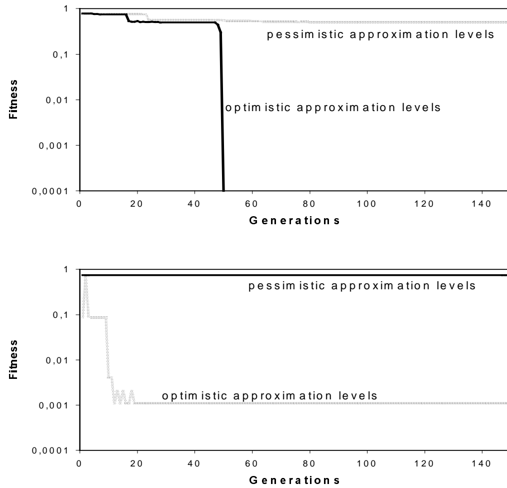


Figure 9: Optimization progress for the three different versions

The chosen expressions change the feasibility and chance of executing the different paths to the target node. As the figure shows, this leads to a different behavior of the fitness function based on *optimistic* versus *pessimistic* approximation levels. For version 3 no solution was found in the test. This is caused by no guidance of the search to execute the path which evaluates condition 2 to *True*. The authors suggest also using node 3 for fitness evaluation. Further research has to be done on this area, since from this point it cannot be decided which function should be used in general, or which analysis methods may identify the best performing fitness function. Another possibility is to use multi objective optimization, but this is not of the scope of this paper.

3.5 CODING OF INPUT SEQUENCES

For structural tests using EA, a test preparation module generates test driver code that maps the variable vector of the EA to parameters of the program under test. This was first introduced by [9]. Mapping can be designed for data structures, arrays and even for dynamic data structures e.g. lists and trees,

Support for dynamic arrays and lists has been implemented in our structural test system. Tests have shown that research on coding these structures is needed. This is because for a program under test, it is not only necessary to find special values in the correct sequences, but also to order them correctly. Finding the correct order with standard EA operators may be very difficult as the following example shows. Example 7 converts an input string of ten characters into a weekday.

```
// computation of day, month, year
// in a loop over the characters

If (year==1752) {
 if (month=9 && day>13)
   // handling of special dates..
}
```

Example 7: Source code illustrating sequence problem

The optimization tends to the solution string “9.9.1752”. This achieves a good fitness since the correct year and month are found and the day is close to the 13th. To obtain better results the EA should be able to insert characters. A new coding of array parameters is introduced, enabling the EA to move the elements of a sequence easily.

This paper argues that other EA operators may not be used because variable vectors often consist of sequences of more than one variable (e.g. arrays of structures), as well as other parameters that are not part of the sequence. It was therefore decided not to use scheduling operators because of the possible mixture of variable types.

With the introduction of an additional array which is used to encode the order of the elements, the scheduling problem was mapped back into a parameter optimization. Appending an ordering array to the interface of a test object may be performed automatically within test driver generation whenever a sequence type is detected.

The additional coding array transforms the search space. Despite the increase in the input dimensions, the search problem is more easily solved by EA. This is due to the additional dimensions that introduce more solutions to the problem. Following the introduction of coding, the performance of the test system improved, whereas previously a solution would have been generated simply by chance when the optimization initially generated some special test data. Figure 10 displays the results.

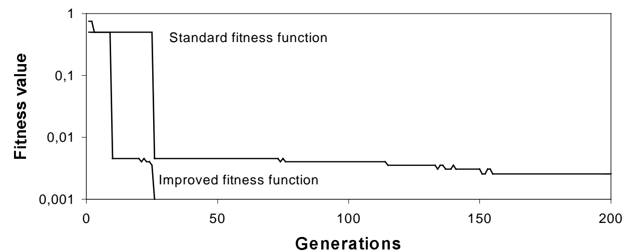


Figure 10: optimization progress

A real example of optimization of a sequence is shown in Table 1. Rows represent different individuals found during the optimization process. The last row displays one solution which is a string with a date of year 1752 in September after the 13th. The gray cells highlight the changes to the previous individual.

Character code	Ordering code	Resulting string
. 1 1 5 . 7 2 6 7 1	3 1 5 6 1 5 6 9 2 6	1 . 7 . 1 7 5 2
. 1 1 5 . 7 2 3 8 2	3 1 6 5 1 5 6 8 2 6	1 . 8 . 1 7 5 2
. 1 6 5 . 7 2 6 9 1	3 1 6 5 1 5 6 8 2 6	6 . 9 . 1 7 5 2
. 1 6 5 . 7 2 6 9 1	3 1 6 5 1 5 6 8 2 0	1 6 . 9 . 1 7 5 2

Table 1: The optimization process in detail; an overview of optimization steps (the best individuals are shown)

Results showed that additional coding may be of assistance for sequences in which the order is important. Consecutively, the values of an array must be optimized. This solution was tested and proved successful in

operation. Special EA operators for scheduling would also perform well if they were able to handle:

- order and parameter optimization at the same time and
- order optimization of sequence elements consisting of structures.

Further work needs to be carried out on complex structure coding, e.g. trees, where the order and the tree structure have to optimized. An example of this is when a partial aim is only executed when a special position in the tree holds a particular value.

4 CONCLUSION

The aim of this work is to enhance the construction of fitness functions in order to improve evolutionary testability by obtaining higher coverage with less resources.

The reason for the failure of an optimization is often difficult to analyze since the search space is usually very large and contains many dimensions. Therefore, visualization of optimization progress is problematic. Different problems and their various solutions have been discussed in this paper.

Poor optimization performance due to composed conditions, nested *if-statements*, and plausible improvement by the avoidance of short circuit evaluation have been discussed. This paper introduces fitness functions with an improved behavior for optimizing test data for target nodes in loops. An alternative method of calculating the fitness value for path coverage including its results on the optimization process is presented. Additionally, problems with approximation levels used to evaluate the closeness of an executed path to a target node are discussed. A search space transformation is introduced which uses a new coding of sequential input parameters.

Future work aims at further improvements to the evolutionary structural test. The idea of solving optimizations problems for difficult conditions as described in [3] is promising. The *chaining approach* introduced in the aforementioned paper may be used as problem transformation.

Further research has to be carried out in the areas presented when using ET for testing modules on a higher system level with sub modules and internal states.

Unstructured loops and loops with flags need more general transformation if the described approach cannot solve the optimization task. In this case a complete change of the program under test may be useful if it assists the discovery of test data for higher coverage of the original program.

References

[1] *Baresel, A.*: Automating structural tests using evolutionary algorithms,(German) Diploma Theses, Humboldt University of Berlin, Germany, *2000*.

[2] *Beizer, B.:* Software Testing Techniques, Van Nostrand Reinhold Company, *1983*.

[3] *Ferguson, Roger; Korel, Bogdan*: The Chaining Approach for Software Test Data Generation. ACM Transactions on Software Engineering and Methodology, Vol. 5 No.1, January 1996, pp.63-86.

[4] *Grochtmann, M., and Wegener, J.*: Test Case Design Using Classification Trees and the Classification-Tree Editor CTE. Proceedings of Quality Week '95, San Francisco, USA, 1995.

[5] *Harman, Mark; Hu, Lin; Munro, Malcolm; Zhang, Xingyuan*. Side-Effect Removal Transformation. IEEE International Workshop on Program Comprehension (IWPC 2001) Toronto, Canada

[6] *Howden, W.:* An Evaluation of the Effectiveness of Symbolic Testing. Software – Practice and Experience, vol. 8, pp. 381 – 397, *1978.*

[7] *Howden, W.:* Reliability of the Path Analysis Testing Strategy. IEEE Transactions on Software Engineering, vol. 2, no. 3, pp., *1976,* 208 - 215.

[8] *Jones, B.-F.; Sthamer: H.-H.; Eyres, D.*. Automatic structural testing using genetic algorithms. Software Engineering Journal, vol. 11, no. 5, pp. 299 – 306, *1996.*

[9] *Korel, Bogdan.:* Automated Test Data Generation. IEEE Transactions on Software Engineering, vol. 16 no. 8 pp.870-879; August 1990.

[10] *Myers, G.:* The Art of Software-Testing. John Wiley & Sons, *1979.*

[11] *Pargas, R., Harrold, M. and Peck, R.:* Test-Data Generation Using Genetic Algorithms. Software Testing, Verification & Reliability, vol. 9, no. 4, pp. 263 – 282, *1999.*

[12] *Pohlheim, H.*: GEATbx - Genetic and Evolutionary Algorithm Toolbox for Matlab. http://www.geatbx.com/, 1994-2001.

[13] *Pohlheim, H.*: Evolutionäre Algorithmen - Verfahren, Operatoren, Hinweise aus der Praxis. Berlin, Heidelberg: Springer-Verlag, 1999. http://www.pohlheim.com/eavoh/

[14] *Pohlheim, H, Wegener, J., Sthamer, H.*: Testing the Temporal Behavior of Real-Time Engine Control Software Modules using Extended Evolutionary Algorithms. in Computational Intelligence, VDI-Berichte 1526, Düsseldorf: pp. 61-66, 2000.

[15] *Ronald E. Prather and J.Paul Myers, Jr.*: The Path Prefix Software Testing Strategy, IEEE Transactions on Software Engineering, Vol.13 No. 7 July 1987.

[16] *Schaeffer, Marvin.*: A mathematical theory of global program optimization. Prentice-Hall Inc., 1973.

[17] *Sthamer, H.-H.*: *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD Thesis, University of Glamorgan, Pontypridd, Wales, Great Britain, 1996.

[18] *Tracey, N., Clark, J., Mander, K. and McDermid, J.:* An Automated Framework for Structural Test-Data Generation. Proceedings of the 13th IEEE Conference on Automated Software Engineering, Hawaii, USA, *1998.*

[19] *Wegener, J.; Sthamer, H.; Jones, B.; Eyres, D.*: Testing Real-time Systems using Genetic Algorithms. Software Quality Journal, vol. 6, no. 2, pp. 127 – 135, *1997.*

[20] *Wegener, J; Sthamer, H. ; Baresel, A. (2001)*: Evolutionary Test Environment for Automatic Structural Testing. Special Issue of Information and Software Technology, vol 43, pp. 851 – 854, 2001.

[21] *Wegener, J., and Grochtmann, M.*: Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing. Real-Time Systems, 15, pp. 275-298, 1998.

[22] Capability Maturity Model for Software, Software Engineering Institute, Carnegie Mellon University.

[23] IEC 65: A Software for Computers in the Application of Industrial Safety-Related Systems (Sec 122).

[24] RTCA/DO-178B Software Considerations in Airborne Systems and Equipment Certification.

[25] Vorgehensmodell zur Planung und Durchführung von Informationstechnik-Vorhaben des Bundesministeriums des Innern (Process Model for Planning and Realization of Information Technology Projects of the German Secretary of Interior).