Evolutionary Testing of Flag Conditions

Andre Baresel, Harmen Sthamer

DaimlerChrysler AG, RIC/SM Methods and Tools,
Alt-Moabit 96a, 10559 Berlin, Germany
{andre.baresel, harmen.sthamer}@daimlerchrysler.com

Abstract. Evolutionary Testing (ET) has been shown to be very successful in testing real world applications [16]. However, it has been pointed out [11], that further research is necessary if flag variables appear in program expressions. The problems increase when ET is used to test state-based applications where the encoding of states hinders successful evolutionary tests. This is because the ET performance is reduced to a random test in case of the use of flag variables or variables that encode an enumeration type.
The authors have developed an ET System to provide easy access to automatic testing. An extensive set of programs has been tested using this system [4], [16]. This system is extended for new areas of software testing and research has been carried out to improve its performance. This paper introduces a new approach for solving ET problems with flag conditions. The problematic constructs are explained with the help of code examples originally found in large real world applications.

1 Introduction to the Flag Problem

Evolutionary Structural Testing has to generate a set of test data for a given test object in order to obtain a high coverage of the program structures. The automatic process creates several test aims, which are tried to be executed in separate search processes. Test aims for statement coverage are the statements of the test object. Each search to solve a test aim is guided by a fitness function. This function defines numerically the proximity of a test datum to the current test aim.

The function uses the values examined at the condition statements during the program execution. For instance, the instrumentation of an equivalence condition on *a* and *b* will report the distance of a and b as fitness. A very small distance results in a very good fitness. This function guides the search to a equals b.

In the case of flag values, the fitness function is simply a Boolean function returning only one poor fitness value for all test data resulting in the undesired flag value. The Boolean fitness function does not guide at all the search to a desired solution and results in an evolutionary test behaving like a random test.

Figure 1 shows two fitness landscapes close to the solution of executing the test aim. On the left hand side a Boolean function created by flag variables is shown and, in contrast to that, the right hand side shows a local distance function formed by equivalence operators. The minimum of both functions is searched for. Whereas a solution for one value of the Boolean function is easily found, the solution leading to the other Boolean value is very hard to find. This is different for the function created for the equivalence operators. Values for this function can be easily optimized.

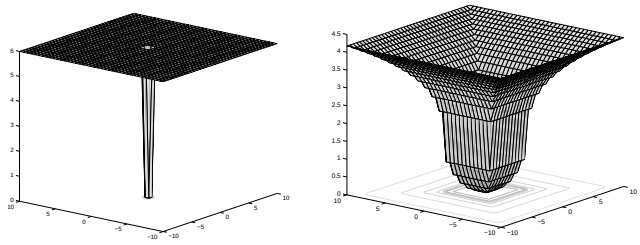

Figure 1: The diagram on the left shows a Boolean function and the diagram on the right shows a fitness landscape created by an equivalence operator of two double variables.

It is very common to use flag variables in real world applications. Even code generators e.g. for matlab / simulink / stateflow [7] create code containing flags. This creates the problem of a Boolean-like function, which does not guide the search for test data. It also occurs when using variables that encode only a small subset of the integer type. This often emerges for variables defined as enumeration types. The measured distance between enumeration type elements does not help to find the right input to fulfill a condition that compares elements of this type. In this case the fitness function returns values that do not direct the search to the solution because the function only expresses that right or wrong values have been monitored.

The authors would like to show that random search behavior can be avoided by using a new fitness function for flag conditions. This also works for enumeration type values. The improvement is using additional information about the flag assignments in a program. These assignments can be identified by static data flow analysis.

2 Short Overview on Evolutionary Testing

Evolutionary algorithms (EA) have been used for searching data for a wide range of applications. EA is an iterative search procedure using different operators to copy the behavior of biologic evolution. When using EA for a search problem it is necessary to define the search space and the objective function (fitness). The algorithms are implemented in the widely used tool box GEAtbx [12]. It consists of a large set of operators e.g. real and integer parameter, migration and competition strategies.

ET uses EA for automatic software testing. The different software test criterions formulate requirements for a test case set to be generated. The creation of such a test data set usually has to be carried out manually. Automatic software testing generates test data sets automatically, trying to fulfill the requirements in order to increase efficiency and achieve considerable cost reduction [15].

This paper covers the automation of structural testing which is a white box test defining the quality of a test case set on the basis of the structural coverage of the test object. Evolutionary structural testing defines test aims and fitness functions to transform the problem of generating a test case set into searches solved by EA[16]. The quality of the fitness function plays an important role for the success of this approach. Some research has carried out to improve the quality (e.g. [2], [9]).

ET works very well for many real world applications. However, '*interesting test objects*', where even an experienced software tester will have problems finding a full covering test data set, are still a challenge for the original ET. This is often due to certain constructs in the programs, which are not taken into account by the fitness.

3 The Approach to Testing Flag Conditions

This section introduces the idea of improving the fitness function by means of additional information in case of flag variables appearing in the code. The goal is to design a fitness function with a better guidance of the search, so that test data generation for flag conditions is improved and does not result in a random search.

The section is divided into subsections explaining the solution step by step. The paper ends with the analysis of a real world example and shows that the approach introduced helps gain full coverage of this test object.

3.1 Direct Assignment of a Boolean Value

The basic idea of the new approach will be explained first by using a simple example. The source code in Example 1 shows the assignment and the use of a flag variable within a nested if-then structure.

Example 1: Usage of a flag assignment

```
1: flag = false;                                        [ENTRY-NODE]
2: if (a==0) flag=true; /* flag assign */              [TARGET-NODE-1]
3: ...                  /* no other assignments to this flag*/
4: if (c == 4) {
5:    if (flag && b> c) /* contains a flag condition */
6:       /* test aim */                                 [TARGET-NODE-2]
```

The original ET approach will immediately reach a high coverage without, however, fully covering the structure. The last search is performed for the test aim at node 6. The fitness function for this test aim is based on the control dependencies, which are created by the if-statements of line 4 and 5. This provides the search with good guidance for reaching the if-statement in line 5. However, the local distance there is a Boolean function and does not direct the search.

The authors suggest a better fitness function, which makes use of the static analysis of the test object. The *use-definition-analysis* returns the assignments in the code, which have an influence on the flag condition. The estimation of the *use-definition-chain* ([1]) will return, that the flag assignment of line 2 is the only assignment influencing the flag use of line 5. This information is the basis for a new fitness function which first guides the search to test data, which execute the flag assignment in line 2, and goes to the flag condition of line 5.

A fitness function targeting two locations in the source code has been defined as a node-node-oriented fitness function in [16]. The test aim is split into two target nodes. The first target node is the assignment, the second one the original test aim.

After applying the new fitness function, the test data searched for has to execute the assignment as well as the test aim. It is now explained how this changes the fitness function applied to Example 1**.**

- The original approach searches for " c==4" and "flag=true && b>c".

  This is created by the control dependency of the if-statements of line 4 and 5. Because of the last condition containing a flag an ET behaves like a random search.

- The improvement searches for "a==0 " and "c==4" and "flag=true && b>c"

  The first part is created by the control dependencies for the flag assignment of line 2 and the second part by the control dependencies for the test aim. This new fitness function directs the search to "a==0" in the first instance, automatically resulting in fulfilling the flag condition.

The following shows a short overview of the evaluation for a node-node-oriented fitness**.** Evolutionary Structural Testing monitors the execution of the program under test. The monitoring result for each test data contains information on the execution path and the evaluation of the conditions during runtime. On the basis of this information a fitness value has to be calculated. The original ET approach has introduced the idea of decisive branches and approximation levels assigned to the control flow graph of the test object. The approximation level at a decisive branch expresses the global distance to the test aim, whereas a local distance can be used to compare solutions with the same level reached. A detailed description on this can be found in [16].

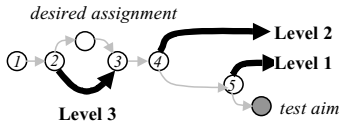

Figure 2: Control flow graph of Example 1 with highlighted decisive branches and annotated approximation levels

Approximation levels are assigned to all branches that create a control dependency for the test aim. The dependencies can be calculated by standard algorithms (see [13]). For the *node-node-oriented* fitness function, it is necessary to estimate the dependencies for progressing from *entry node* to *target node 1* (for target nodes see Example 1, right hand side) and the dependencies progressing from *target node 1* to *target node 2*.

The assignment of the levels is performed by analyzing the possible execution orders of the identified nodes. Nodes that come later in the execution path will achieve better approximation levels than nodes at the beginning of the paths.

control dependencies for “entry-node” to “target node 1”:
node 2
control dependencies for “target node 1” to “target node 2”:
node 4, node 5
execution order of the identified nodes and assigned approximation levels:
Node 2 Level 3
Node 4 Level 2
Node 5 Level 1

Figure 3 shows a graphical interpretation of the approximation levels comparing the original (left hand side) and the new fitness evaluation approach (right hand side).

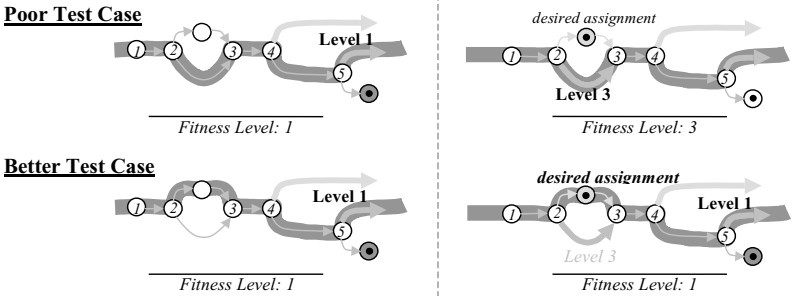

Figure 3: The original ET (left) estimates the same fitness for both test cases; the new approach (right) assigns a poor fitness for the path not executing the flag assignment (top)

Whereas in the original approach only one decisive branch can be executed per execution path and test aim (because of the definition of the control dependency), it is necessary in the new approach to make sure that the first decisive branch (e.g. top right path) defines the fitness. This is due to the concatenation of the decisive branches of the two target nodes. A complete definition of the fitness calculation rules will be provided later in Definition 1 after all details have been explained. Figure 4 shows the experiment results. Two scales are used in the diagram, since the fitness values are of different value ranges. The original fitness for the selected test aim ranges from 0 to 2, and the new fitness has an increased range of [0, 3].

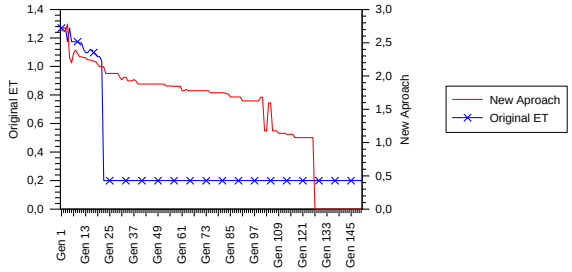

Figure 4: The diagram shows the fitness of the best individual created by the EAs over all generations; the fitness function of the new approach guided a search so that the solution was found in Generation 130, whereas the original ET stagnates at fitness 0,4.

With the new fitness function targeting the flag assignments the evolutionary test can be improved to reach a higher coverage for applications that use single flags with just one assignment. In the next steps the authors would like to show how this idea can be extended to solve more complicated flag uses and to provide a universal solution for any kind of flag definition and use.

3.2 Dealing with Undesired Flag Assignments

The previous subsection gave an example where the execution of the test aim depended on a desired assignment. It is certain that in real world applications program code can include assignments that should not be executed to fulfill a test aim.

A very simple source code showing this can be seen in Example 2. In order to execute the test aim a flag value "true" is necessary. For this reason, a solution that reaches the test aim cannot execute the flag assignment in line 2.

Example 2: Source code containing an undesired flag assignment

```
1: flag = true;
2: if ( a!=0 || c>5 ) flag=false;/* avoid this assignment execution */
3: switch (er) {
4: case 5:
5:   if (flag)
6:     /* test aim */
```

The original ET approach does not use information about the undesired assignment. For the original ET a high probability of the flag assignment execution results in a random search, because the fitness function does not take into account the condition at line 2 (it does not create a control dependency for the test aim). A static analysis of the program under test can be helpful in this situation. By using the information about undesired flag assignments it is possible to create a better fitness function. This function has to guide the search to *not execute* the identified flag assignment. The control

dependencies of the undesired assignment will be used to define the new function, but, in contrast to the solution explained for desired flag assignments, the fitness is now based on the branches not leading to the assignment.

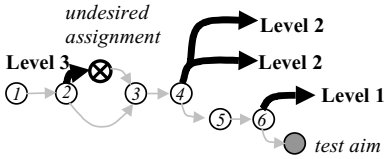

Figure 5: Control flow graph of the example with highlighted decisive branches. The branch leading to the undesired assignment is decisive and gets a poor approx. level.

After applying the new fitness function, the test data that is searched for must *not execute* the assignment and then traverse the test aim. It is explained next how this changes the fitness function applied to Example 2**.**

§ The original approach searches for "`er==5`" and "flag=true".

This is because of the control dependency created by the switch- and if-statements of line 3 and 5. The flag condition creates a Boolean fitness function.

§ The new approach is searching for "not (a!=0 || c > 5)" and "er==5" and "flag=true"

The first part is created by the control dependencies for the flag assignment of line 2 and the second part by the control dependencies for the test aim at line 6. This new fitness function allows only solutions with "not (a!=0 || c > 5)" in the first instance (this will automatically result in fulfilling the flag condition of the second term).

An experiment with the new fitness function is presented in Figure 6. Again, the fitness curves have been made comparable by using two scales. The optimization using the original approach stagnates at value 0,4 because it does not find a solution which results in a flag value "true".

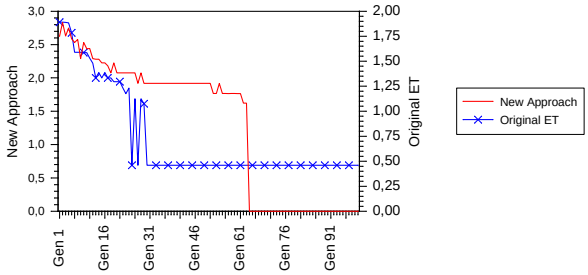

Figure 6: Diagram demonstrating the improvement of the search process; the original approach does not find a solution for the test aim (remains at fitness about 0,4)

As shown by means of this example it is also possible to improve the original ET approach for test objects containing undesired flag assignment.

3.3 Multiple Assignments to Flag Variables

The examples shown in the previous sections only use single flag assignments. The approach is now extended for test objects with multiple flag assignments. To achieve this, the idea of *include* and *exclude* lists for flag assignments is introduced. The explanations refer to Example 3. The test aim has been commented on and the desired and undesired flag assignments for this test aim have been marked in the source code.

Applying static analysis for the example will return three flag definitions where the execution of one assignment is not desired for covering the test aim. All desired as-

signments will be collected in a so-called *include-assignment* list and the undesired ones will be placed on the corresponding *exclude-assignment* list.

Example 3: Source code containing multiple flag assignments

```
1: flag = false;
2: If (a==0) flag = true;      /* execute this */
3: If (b==0) flag = true;      /* or execute this */
4: …
5: If (z > y) flag = false;    /* do not execute ! */
6: …
7: If (c==0) {
8:   If (flag) /* test aim: go here */
```

In order to obtain a solution with a higher structural coverage it is necessary for the fitness function to return good values for any test data that:

- are close to or execute any of the *include-assignments* and
- do not execute any of the *exclude-assignments*.

All *include-assignments* have to be treated equally since it cannot be decided which of the assignments creates an executable solution for the test aim. Even with these new requirements for the fitness calculation is it possible to reuse the idea of approximation levels. Assigning the right levels to the nodes in the control flow graph and some additional calculation rules will result in fitness values that meet the requirements.

To enable a fitness calculation for the suggested improvements, branches are additionally defined as *'decisive'* (execution has an effect on fitness calculation). These are branches *avoiding the execution of a desired flag assignment* (Figure 7 - highlighted branches of nodes 2 and 3) or branches *leading to an undesired flag assignment* (highlighted branch of node 3). If a path traverses one of those identified branches fitness has to be calculated because of the depending flag assignments.

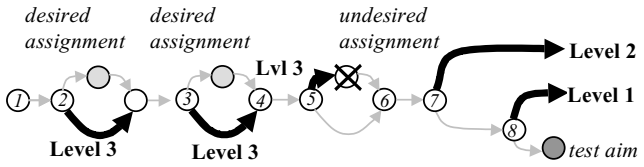

Figure 7: Decisive branches assigned with approximation levels

The introduction of additional decisive branches makes new fitness calculation rules necessary, because some execution paths might run through several decisive branches. This cannot happen in the original ET.

DEFINITION 1: FITNESS CALCULATION RULES

1. *first decisive branch temporarily determines the fitness value, fitness can only be changed again when rules 2 and 3 are enabled.*
2. *within loops the best level over all iterations establishes the fitness value*
3. *whenever a desired flag assignment is executed it fixes the temporary fitness value; only rule 4 can make this fitness invalid*
4. *whenever an undesired flag assignment is executed it reactivates rule 1 and resets the fitness calculation.*

Some examples illustrating the usage of the rules will be described next (all referring

to Example 3). If a test case misses the assignment of line 2 (rule 1 is enabled) and then executes the assignment of line 3, it will achieve a good fitness because of rule 3 (level 2). However, if the assignment of line 5 is executed afterwards, rule 4 is activated and fitness is set back to a poor value (level 3). A path that does not execute a decisive branch (highlighted in Figure 7) and does not fulfill the condition of node 7 is assigned a fitness value of level 2. Last but not least, a test reaching line 8 and not having executed any decisive branch before will meet the test aim.

It will be described next how the *include-* and *exclude lists* change the fitness function and how this affects the search process. Example 3 is used for descriptions.

§ The original ET searches for data fulfilling the condition "c==0" and "flag=true"

This is because of the control dependency created by the if-statements of line 7 and 8. As described previously, this fitness function will result in a random search.

§ The suggested improvement searches for a solution of the following term: "(a=0 || b=0) and not (z>y) and (c=0 && flag=true)"

The first part is created by the include-assignments of the example (line 2 & 3); the second part comes from the exclude-assignment of line 5, and the last part is created by the control dependencies of the test aim. The new fitness function only allows solutions with "(a==0 or b==0) and not (z>y)" in the first instance, which automatically fulfill the flag condition.

The new fitness function targets the flag assignments, in this way enabling the evolutionary test to find a high covering test data set for test objects that uses flags. The authors will show next that even *flag uses within loops* and *expressions in flag assignments* will no longer be a problem with this approach.

3.4 Flag Assignments Within Loops

As mentioned previously, applications can also assign flag variables within loops. ET generally has no problems with loops appearing in the test object source code. The method can search for test data fulfilling test aims outside and inside any kind of loops. For this reason the new approach also works if the flag assignment is placed within a loop. The short Example 4 demonstrates this case. The flag assignment of line 3 is placed within a *while-loop* and the flag use is at line 4.

Example 4: Flag assignment appearance within a loop

```
1: flag = false;
2: while (i<10)
3:   if (a[i]==0) flag = true;
4: if (flag)
5:   /* test aim */
```

In this situation, an evolutionary search for a solution of the test aim at line 5 requires an improved fitness function because of the flag condition at line 3. The calculation of the *include-assignments* and *exclude-assignments* will return just one *include-assignment* in line 3 (line 1 is an initialization that is always executed).

ET has no problem guiding the search to a test datum that executes the assignment of line 3. The new fitness function is the following logical term:

§ The suggested improvement is searching for a solution of the following term: "{*in-any-iteration* a[i]==0}" and "flag==true"

The operator "in-any-iteration" is created by the include assignments which is placed within the loop. The assignment needs to be executed only once.

This new fitness function improves the evolutionary test. However, using the new fitness function implemented via approximation levels does not solve the problem of undesired flag assignment appearing in loops. This is because of the incomplete use of the monitored information. With the appearance of *undesired-assignments* inside loops it is necessary to define a fitness function which numerically evaluates the "*in-all-iterations*" operator. In Example 4 it is the fulfillment of the condition:

"{*in-all-iteration* not a[i]==0 }

But an implementation using approximation levels with the rules of Definition 1 results in a fitness calculated on the basis of a transformed version of this term:

"not {in-any-iteration a[i]==0 }.

This is logically equivalent to the first but results in a fitness function which is calculated on the basis of only one iteration that fulfills the condition: *"a[i] = 0".*
Analyzing the results of ET has shown that this will lead to a random search. The reason is that the fitness values do not take into account that a test datum with just one iteration meeting the condition "a[i] equals zero" is better than a test datum executing more iterations meeting this condition. A different implementation is needed here.

Using the new approach the ET can also be improved in the case of loops appearing in the code. The new fitness function guides the search to execute desired flag assignments appearing within any kind of loop statements even if the loop is created by 'goto' statements. Only the appearance of *undesired-assignments* within loops still causes the flag problem for our implementation.

3.5 Boolean Expressions Assigned to Flags

The previous subsections have shown how to improve the fitness function if the flag is directly assigned to just one value. Real world applications often make use of assigning expressions to flags which are evaluated at runtime. Static analysis cannot distinguish which value these assignments return. Nevertheless, the approach can also handle this very common case. Example 5 shows one function code containing an assignment of an expression in line 1 and the use of the flag in line 3.

Example 5: Flag assigned by a Boolean expression

```
1: flag = (a==0) || (b>0 && b<5);
2: ...
3:  if (flag) /* test aim */
```

Running the original ET approach will perform poorly with a low coverage. There is only a slim chance that it will find a solution. An ET inserting the Boolean expression at line 3 will work without any problems. Unfortunately the transformation is sometimes a difficult task (more details in [6]). A more simple transformation can help obtain a test object which can be handled with the approach explained in the previous subsections. This transformation changes the assignment expression into an if-then-else construct which behaves completely equivalent to the original code. The transformed version is shown in Example 6. This code has just two direct value assignments to the flag variable.

Example 6: Transformed expression

```
1: if ( ( a == 0 ) || ( b > 0 && b < 5 ) )
     flag = true;   else   flag = false;
```

Applying the new approach does not actually require such a transformation as seen in Example 6. Simple function calls inserted into the Boolean expression will obtain the information necessary for the fitness calculation,

e.g. `flag=DistEqual(a,0) || DistGreater(b,0) && DistLess(b,5)`).

A fitness function using the data collected during the execution of the instrumented expression guides the search to a solution assigning the required flag value.

The approach in [5] describes a similar idea for guiding the search by data dependencies. All paths containing assignments relevant for the flag condition under test are generated and optimized in a sequence. However, our approach uses a single optimization considering all these paths simultaneously. [5] can solve the previously described flag problems, but cannot guide the search in case of Boolean expressions.

Up until now a general approach for guiding the search in the case of just one flag variable has been presented. The next subsection extends the applicability to a wider range of test objects. The use of more than one flag takes place in many code generated examples. However, it becomes more difficult to guide the search in such cases.

3.6 Using More Than One Flag Variable

The idea is extended for test problems with more than one flag variable in the condition that needs to be fulfilled. The next example shows an instance of this case.

Example 7: Source code showing the use of more than one flag variable

```
1 If (a==0) initialized = true;              /* execute this*/
2 If (b==0) has_been_fired = true;           /* and execute this */
3 ....
4 if (initialized)
5 {
6    If (b==3) has_been_fired = true;        /* or execute this */
7 ..
8    If (has_been_fired )
9       /* test aim */
```

The idea of this example is that the test aim is only executed if the two flags are assigned in the right way before reaching the test aim. In real world applications any combination of flag assignments within nested conditions and loops are conceivable. The fitness evaluation has to check in parallel for each flag appearing in the code whether or not it is assigned in the desired way. That is why the original approach using approximation levels does not work properly.

With the use of approximation levels and local distances the instrumentation can decide how close the test datum is to a flag assignment. This value (*temporary flag fitness*) can be stored together with the flag value at runtime. If the flag is assigned as necessary for the test aim this information is not used, but, if it is not, the temporary flag fitness will be used to calculate the overall fitness of the test datum. The fitness calculation is prepared by checking the control dependencies of all flag uses; estimating the *exclude-assignment* and *include-assignment* lists for each identified flag, calculating the decisive branches on the basis of the *include* and *exclude* lists, and assigning the *flag-approximation-levels* to the branches independently for each flag. The assigned *flag-approximation levels* are used to calculate temporary *flag fitness*. This fitness forms the basis for the improved fitness function for multiple flag uses.

DEFINITION 2: FITNESS CALCULATION RULES FOR MULTIPLE FLAGS

1. *temporary flag fitness is estimated as defined in the rules of Definition 1 for each flag appearing in the code*
2. *upon reaching a control-dependent condition containing flags it is checked whether or not the flags are assigned in the desired way. If the flags are assigned correctly the usual fitness calculation proceeds. Whenever one or more flags are not assigned correctly, fitness is calculated using the corresponding temporary flag fitness.*

Annotation. *In the case of multiple flags occurring in a condition we suggest calculating an overall fitness using all temporary flag fitness values. See* [2] *for the reasons for parallel condition optimization.*

The implementation of this approach is quite complex and the calculation of the approximation levels requires some extra processing time. However, when applying the original approach on examples using multiple flags, the coverage reached with the original ET is worse. A related paper to this idea is [3]. A complete solution needs further research.

3.7 Real World Example

The authors have tested the approach using a real world application extracted from software for a car-controlling unit. The function under test is responsible for regulating the internal states of the energy control of a small sub-function within body and comfort electronics of a car. The function has 100 Loc and an *if-then* nesting level of 5. It has 4 input parameters and internally uses two flags. For testing reasons the initial state and the input situation have been generated.

The original approach did not find a solution for the condition statements using the two flags. Within a test of about 320.000 individuals it has reached a branch coverage of about 90%. This happens with the standard settings of the ET-system using 300 individuals in 6 populations with competition enabled. A maximum of 200 generations per test aim were allowed. The authors ran the same test with a bigger population (700 individuals) and a later stopping criterion (max. 400 generations), but no improvements were noticeable.

Using the new approach no problems were distinguished. One of the flags was assigned by an expression which had a very low probability evaluating to 'true'. This was the reason why even more tests did not perform better using the original approach. The second flag was assigned within a nested if-then structure and tested later in a different nesting level. In the original ET approach there was only a very low chance of executing the flag assignment and the condition within one execution. The fitness function did not guide the search to this solution.

The new approach used the standard EA settings and covered the test object fully after 60 generations and testing 14500 individuals. This is 5 percent of the workload of the original approach and leads to 100 % branch coverage.

4 Conclusion

Evolutionary Testing uses metaheuristic search methods to automate software testing aspects. The occurrence of flag variables has been pointed out to be problematic because of a poor guidance of the search at conditions containing flags. The authors

introduce a new fitness function, which improves evolutionary structural testing in case of flag conditions. This function uses additional information on the flag assignments occurring in the test object. The solution is explained by using short code examples extracted from real world applications.

The introduced improvements cannot only solve the problem of flag variables, the authors argue that the test of sources containing enumeration type conditions, also known to be problematic, can be improved using the introduced approach.

By using a fitness function that guides the search to variable assignments as well as variable uses it has been shown that the flag problem can be solved. We believe that future research on sequence testing can reuse this idea. This is because in sequence testing state variables are assigned and used in different areas of a program and sometimes also in the different steps of a sequence. A fitness function guiding the search to the execution of state variable assignments and to the condition testing the state variable will perform better than the original ET.

Literature

[1] Appel, A. W.: Modern Compiler Implementation in C, Cambridge, New York: Cambridge University Press, 1998

[2] Baresel, A., Sthamer, H. and Schmidt, M.: Fitness Function Design to improve Evolutionary Structural Testing, Proceedings of the GECCO, New York, USA, July 2002

[3] Bottaci, L.: Instrumenting Programs With Flag Variables For Test Data Search By Genetic Algorithms, GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, p. 1337-1342, July 2003

[4] Buhr, K.: Complexity Measures for the Assessment of Evolutionary Testability (only german version). Diploma Thesis, Technical University Clausthal, 2001

[5] Ferguson, R.; Korel, B.: The Chaining Approach for Software Test Data Generation, Transactions on Software Engineering and Methodology, Vol. 5 No.1, pp.63-86, 1996

[6] Harman, M., Hu, L., Hierons, R., Baresel, A. and Sthamer,H.: Improving Evolutionary Testing by Flag Removal, Proceedings of GECCO, New York, USA, 9-13th July 2002

[7] http://www.mathworks.com/

[8] Harman, Hu, Hierons, Fox, Danicic, Baresel, Sthamer; Wegener: Evolutionary Testing Supported by Slicing and Transformation, 18th IEEE International Conference on Software Maintenance (ICSM 2002), 3 - 6 Oct. 2002. Montreal, Canada. Page 285.

[9] 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

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

[11] Michael, C.C,. Mcgraw, G, Schatz, M.A.: Generating Software Test Data by Evolution, IEEE Transactions on Software Engineering, vol. 27, No. 12 pp. 1085-1110; Dec. 2001

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

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

[14] Tracey, N., Clark, J., Mander, K. and McDermid, J.: An Automated Framework for Structural Test-Data Generation, Proceed. of the 13th IEEE Conf. on Automated SE, 1998

[15] Wegener, J., Sthamer, H., Baresel, A.: Application Fields for Evolutionary Testing, Eurostar 2001 Stockholm, Sweden, November 2001

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