DOI 10.12694/scpe.v14i1.827
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 14, Number 1, pp. 57–67. http://www.scpe.org © 2013 SCPE

# NEW PERFORMANCE ESTIMATION FORMULA FOR EVOLUTIONARY TESTING OF SWITCH-CASE CONSTRUCTS

GENŢIANA IOANA LAŢIU, OCTAVIAN AUGUSTIN CREŢ, AND LUCIA VĂCARIU[*]

**Abstract.** Evolutionary structural testing is a technique that uses specific approaches based on guided searches algorithms. It involves evaluating fitness functions to determine whether test data satisfy or not various structural testing criteria. For testing multi-way decision constructs the nested *If-Then-Else* structure and Alternative Critical Branches (ACBs) approaches are generally used. In this paper a new evolutionary structural approach based on Compact and Minimized Control Flow Graph (CMCFG) which uses two different formulas for evaluating the performance of test data, is presented. The CMCFG approach is derived from the concept of Control Flow Graph (CFG). Experiments on different *Switch-Case* constructs with different nesting levels have demonstrated that CMCFG yields significantly better results in finding test data which cover a particular target branch in comparison with the previous approaches.

**Key words:** Evolutionary structural testing, control flow graph, switch case structures, fitness function.

**AMS subject classifications.** 68N15, 68N19, 68N01

**1. Introduction.** The main idea behind the evolutionary testing process is to automatically generate test data through the use of optimizing search techniques [1]. The search space which corresponds to the evolutionary process is represented by the specific domains of the input variables of the software program under test. Evolutionary structural testing has been intensively used for automatically generating test data by many researchers. M. Harman and P. McMinn present in [9] a vast theoretical exploration of global search techniques embodied by Genetic Algorithms. Other approaches related to evolutionary testing with flag conditions are presented in [5], [16], and [4]. Different transformations techniques were applied and reported in the literature for Evolutionary Testing (ET) in order to improve the fitness function calculation, because a well-defined fitness function is essential for the efficiency of the evolutionary search process ([12], [8], and [13]).

The main constructs (sequence structures, selection structures and repetition structures) of a software program were studied and tested in the literature using evolutionary search techniques, but less work has been done on the *switch-case* constructs which are used to express multiple branch selection statements. This type of construct was studied in [15], where it was tested using the concept of Alternative Critical Branches (ACBs). ACBs consist of all case branches that can prevent the execution of the target branch. The ACBs consist of one element that is the alternative branch of the target branch if it is leaving a two-way decision node. Each control-dependent node has only one ACB assigned to it. All the ACBs with respect to the target branch constitute a set. It forms the Critical Branches Set (CBS) which is extended from the single critical branch concept. This concept refers to the branch which prevents the target branch to be reached when the current test data is executed. If any element from CBS corresponding to the target branch is taken, then there is no chance to generate test data which cover the target branch.

The focus in this approach is on the structural testing of multi-way decision statements, in particular on branch coverage. The ACBs are used for determining the approximation level, which is used by the fitness function formula that evaluates the performance of each individual. The approximation level has been calculated by subtracting one from the number of ACBs which are in the CBS and which are lying between the node from which the test data diverge away and the target node.

The rest of this paper is organized as follows: Section II describes the evolutionary testing methodology and the switch-case constructs. Section III describes different fitness function calculation approaches used for structural testing in case of switch-case constructs. Section IV presents the experimental results obtained for different level of imbrications and Section V presents the final conclusions and future work.

**2. Evolutionary testing methodology and switch case construct.** Evolutionary testing (ET) is a meta-heuristic approach by which test data can be generated automatically through the use of optimization search techniques. It is usually used for testing complex systems which may involve many components with correlated activity between them. The ET process tries to improve the effectiveness of the traditional testing process by transforming the testing objectives in search problems which will be solved using evolutionary

[*]Technical University of Cluj-Napoca, Computer Science Department, 26-28 George Baritiu street, 400027 Cluj-Napoca, Romania

algorithms. In the ET process the search space is represented by the variation domain of the input variables of the software under test, in which test data fulfill the specific test objectives. The ET process' phases are presented in Fig. 2.1:

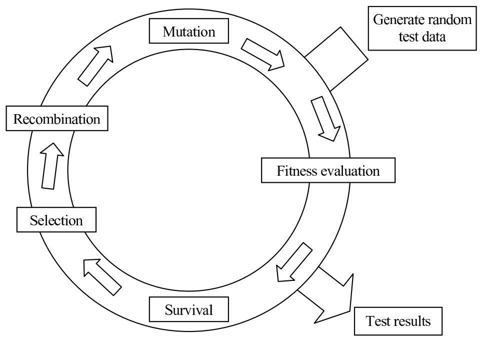


Figure 2.1. *Evolutionary testing process*

ET is used in many search problems in software testing, because it has a very good capacity of adapting itself to the system under test. The ET process is an iterative procedure which combines good test data in order to achieve better test data. ET was successfully reported in the literature and applied for different forms of testing, called: specification testing [14], unit testing [7], and extreme execution time testing [17].

During the ET process the initial test data are randomly generated. Each individual from the population represents the test data using which the test is executed. Test data take values from the domains of the software under test's input variables. The performance of each individual is evaluated and the fitness value corresponding to the current individual is determined. Next, the population members are selected with respect to their performance. The chosen individuals are subject to *crossover* and *mutation* processes to generate new individuals, called *offsprings*. *Crossover* is used to combines two parents to produce a new offspring, while *mutation* is used for randomly altering a gene value (for instance, switching from 1 to 0 in case of binary individuals) from the individual. By reuniting the new created offsprings with their parents a new population is formed. The evolutionary process repeats all the above described steps until the established testing criteria are met. Then the process stops and the best solution found will be the testing solution.

The goal of this research was to study the *switch-case* construct in the context of structural path oriented testing, aiming to find test data which executes a particular branch in a program that contains *switch-case* constructs. In order to automatically generate test data which trigger the execution of a particular branch of the program, every possible solution is evaluated with respect to the test objective. The *switch-case* construct is a multi-way selection control mechanism which is used as a substitute for the nested *if-then-else* structure. It is extensively used in software programs because it improves the readability of the source code and reduces repetitive coding. The general structure of a *switch-case* construct is presented in Fig. 2.2:

```
Switch (expression) {
Case expression: //some code
        Jump, return or break statement
Default: //some code
        Jump, return or break statement
}
```

Figure 2.2. *General switch-case conditional construct*

The *switch-case* construct, as presented in 2.2, gives the developer the possibility of choosing between many statements, by passing the flow control to one of the *case* statements within its body. The *switch* statement evaluates the expression and executes the *case* branch that corresponds to the expression's value. A *switch-case* construct can include any number of *case* statements. Each *case* statement is followed by an optional *break*, *return* or *goto* statement (called breaking statements). The breaking statements are used either to return a value and exit the *switch* body, or to break out of the *switch* construct when a match is found, or go to a specific location in the code.

If *break*, *return* and *goto* options are not present after a *case* statement then the control flow is transferred to the next *case* statement until it will meet one of the breaking statements. If an expression transmitted to the *switch-case* construct does not match any *case* statement, the control will go to the *default* statement. If no default statement exists, the control will go outside the *switch* body. A simple example of a *switch-case* construct is presented in Fig. 2.3.

```
Switch (x) {
        Case 'b': y = 'B'; break;
        Case 'f': y='F'; break;
        Case 'c': y='C'; break;
        Case 'a': y='A'; break; //Target branch
        Default: y='Z';
        }
```

FIGURE 2.3. *Simple switch-case conditional construct*

A previous work [15] has argued that for a particular case branch, the CBS should be constructed. This set is composed of all the case branches that cause the target to be missed. For instance, the CBS that corresponds to the target branch from the source code listed in Fig. 2.3 is composed by *case* 'b', *case* 'f', *case* 'c, and *default.* The target branch is definitely missed when the execution of test data diverges away at any branch within the CBS.

The fitness function used for evaluating each test data is calculated using the sum between two metrics:

1. The *approximation level.* This is calculated by subtracting 1 from the number of ACBs located between the node from which the test data diverge away and the target branch itself (in the example from Fig. 2.3, the branch that corresponds to case 'a').
2. The *branch distance.* This is calculated using the following expression: $|expr - C| + 1$, where *expr* is the value of the expression which appears after the *switch* keyword, and C is the constant value for the desired *case* statement. For both operands ($expr$ and $C$) the corresponding ASCII code for each character is used. The value 1 which appears at the end of the formula is the positive failure constant [14]. For instance, if $x$ = 'f', then the branch distance which corresponds to the target branch specified in Fig. 2.3 is $|102 - 97| + 1$.

The fitness value indicates how close the test data are to trigger the execution of the target branch located inside the *switch* statement.

## 3. Different fitness function calculation approaches for switch-case constructs.

**3.1. Fitness calculation based on nested if-then-else statements.** *Switch-case* constructs are considered to be equivalent to nested *if-then-else* statements with respect to the CFG. The *switch-case* construct presented in Fig. 2.3 is equivalent to the nested *if-then-else* construct shown in Fig. 3.1:

```
if (x=='b') {y='B';}
else if (x=='f') {y='F';}
else if (x=='c') {y='C';}
else if (x=='a') {y='A';} // Target branch which should be tested
else {y='Z';}
```

FIGURE 3.1. *Transformation of switch-case conditional construct in nested if-then-else statements*

The target branch for which test data should be generated is the case branch $x$=='a'. Each test data automatically generated by the ET process must be evaluated using the fitness function. The purpose of this function is to guide the evolutionary search process to find the test data that trigger the execution of the target branch. The fitness function evaluation represents the calculation of distances to the target branch.

In structural testing, previous work [2] has demonstrated that the fitness function having the expression illustrated in 3.1 correctly evaluates how close the test data are to cover the target branch:

$$F(test\_data) = Approx\_level + Normalized\_branch\_distance \tag{3.1}$$

The normalized branch distance is computed using formula 3.2 and indicates how close the test data are to take the alternative branch:

$$Normalized\_branch\_distance = 1 - 1.001^{-distance} \tag{3.2}$$

The *approximation level* represents the number of decision nodes lying between the decision nodes where the actual test data diverge away from the target branch itself. In Fig. 3.2, given $x$ = 'b' the control flow takes the Yes branch at node 1. The *approximation level* is 3. The branch distance is computed according to (2) using the values of the variables or constants involved in the conditions of the branching statement [2]. For the branching condition $x$ ='b' the branch distance is $|x - 98|$.

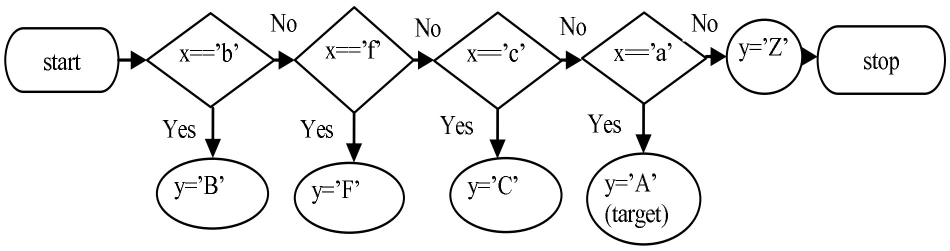


FIGURE 3.2. *CFG for a simple switch-case construct*

As shown in Fig. 3.2 each decision node is control-dependent on the previous decision nodes. For *switch-case* constructs represented as nested *if-then-else* statements, each *case* branch is dependent on the *case* branching node it leaves and all the *case* branching nodes located before it. For instance in Fig. 2.3 the branch corresponding to $x$=='a' is control dependent on $x$=='c', $x$=='f' and $x$=='b' branches. Considering that the target branch is the branch corresponding to case 'a', then the approximation level will be computed as follows:

- its value will be 0 if test data diverge away at condition node x=='a';
- its value will be 1 if test data diverge away at condition node x=='c';
- its value will be 2 if test data diverge away at condition node x=='f';
- its value will be 3 if test data diverge away at condition node x=='b'.

If the fitness function is computed for two specific values of the $x$ variable, 'c' and 'b', the corresponding *branch distances* are 2, respectively 1 if we consider the traditional approach for computing the *branch distance* based on relational predicates [11]. So for these two values ('c' and 'b') the *approximation level* equals 1 and 3 respectively. Considering that the fitness value is the sum between the *approximation level* and the *branch distance*, the fitness value for $x$=='c' equals 3 and the fitness value for $x$=='b' equals 4.

Taking into consideration that better test data have smaller fitness values, the value 'c' is considered to be better than the value 'b' because it is closer to 'a' (which constitutes the target branch). This choice is contrary to the traditional approach, because 'b' is closer to 'a'.

In conclusion the approach with nested *if-then-else* statements is not a perfect one because in the *switch-case* constructs the order in which the clauses are written is not important for the evaluation of the fitness function, while the nested *if-then-else* statements can induce significantly different fitness values depending on the order in which they are written. For instance, the fitness value for $x$ = 'c' is smaller than the fitness value for $x$ = 'b' even though 'b' is closer to 'a' than 'c' is. This approach is not guiding the evolutionary search algorithm in the correct direction, because the dependencies between *case* branches result in an inappropriate approximation level value.

**3.2. ACBs-based fitness function calculation.** The ACBs-based approach for fitness calculation assumes that all *case* branches in the *switch-case* construct are mutually exclusive in semantics [2]. A special CFG called *Flattened Control Flow Graph* (FCFG) is described in [2]. This graph is extended from the traditional

CFG, with the only difference that the switch node can have more than two successors, all of them being on the same level. In this graph each case branch is control-dependent only on the branching switch node. Fig. 3.3 shows the FCFG corresponding to the simple *switch-case* construct presented in Fig. 2.3:

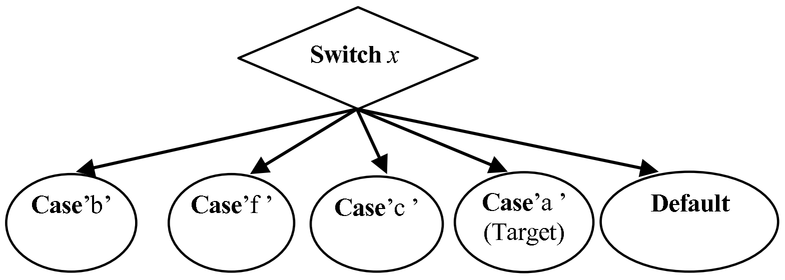


FIGURE 3.3. *Flattened Control Flow Graph for simple switch-case construct*

Based on the FCFG definition ([2]) each node has a set of control nodes on which it depends. This set constitutes the CBS. The target branch execution is definitely not triggered by the test data when the execution diverges away in any node from the CBS. When any node in the CBS is taken by the test data, then there is no chance that the target branch is covered. In the example from 3.3 the CBS attached to the target branch is composed by the following branch cases: 'b', 'f', 'c' and default. If the actual test data object executes one of the *case* statements from the CBS, it has no chance to execute the target branch.

With this concept of CBS and FCFG the approximation level metric (which is part of the fitness function expression) is calculated by subtracting 1 from the number of critical branches located between the node from which the test data diverge away and the target itself. The branch distance metric used for evaluating the test data uses the switch expression's value and the constant for the target branch.

Using this approach for the case when $x$ equals 'b' or 'c', the *approximation level* will be 0 and the *branch distance* will be $|98 - 97| + 1 = 2$ and $|99 - 97| + 1 = 3$, respectively. The fitness calculated based on 3.1 will be 2 when $x ==$ 'b' and 3 when $x ==$ 'c'.

For this simple case it is obvious that the ACBs-based fitness value is guiding the evolutionary search in a correct direction compared to the nested *if-then-else* approach: the fitness value for $x==$'b' is smaller than the one for $x==$'c'. If the simple switch-case construct becomes a more complex one, containing case statements without break options and one level of nesting, then it can look like in Fig. 3.4:

```
Switch (x) {Case 'b':  value= 'B';
                       break;
            Case 'k':  value='K';
                       break;
            Case 'c':  value='C';
                       break;
            Case 'a':  Switch (y) {Case 'h': value = 'H';
                                             break;
                                   Case 'i':
                                   Case 'n': value = 'N'; //Target
                                             break;
                                   Default: value = 'Z';}
            Default: value='W';}
```

FIGURE 3.4. *Complex switch-case conditional construct*

The corresponding FCFG for the *switch-case* construct with one nesting level presented in 3.4 is shown in Fig. 3.5:

For the complex *switch-case* construct illustrated in Fig. 3.4, if $x =$ 'b' and $y =$ 'h' the *approximation level* is 1 and the *branch distance* is $|1 - 0| + 1 = 2$. The total fitness function value is 3. If $x =$ 'a' and $y =$ 'h', then the *approximation level* is 0 and the *branch distance* is $|7 - 13| + 1 = 7$. The total fitness is 7. So the

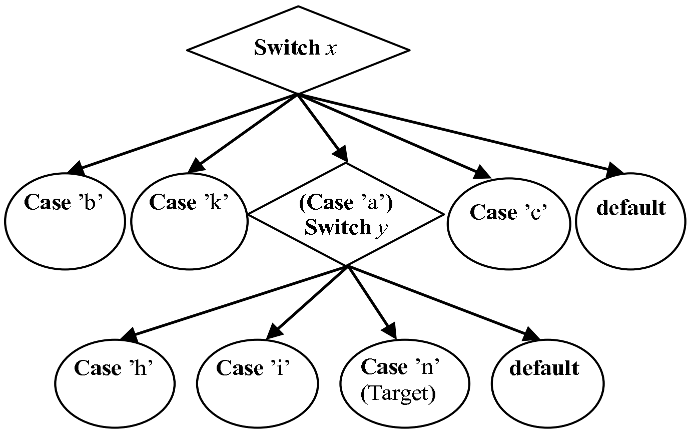


FIGURE 3.5. *FCFG for complex switch-case construct*

pair of values ($x$ = 'b', $y$ = 'h') has a smaller fitness value than ($x$ = 'a', $y$ = 'h'), even though the second pair of values is closer to the solution values ($x$ = 'a', $y$ = 'n'). So it is obvious that the fitness value calculation approach proposed in [15], which is based on ACB approach, misleads the evolutionary search process.

**3.3. Fitness calculation based on the CMCFG approach and Korel's distance formula.** To correctly guide the evolutionary search algorithm in the right direction we propose a new approach based on Compact and Minimized Control Flow Graph (CMCFG). As shown in Fig. 3.5 every switch node has as descendants several *case* branches. For the target branch, one or more *case* branches can lead to the target branch being not executed by the test data.

In the CMCFG approach each *switch* statement is represented on a different level. The *approximation level* is calculated based on the number of switch nodes from which we subtract 1. The numbering of the approximation level starts in the CMCFG in a top-down manner. As shown in Fig. 3.5, if test data diverge away from target branch at the first switch node, it will have an approximation level of 1, while if they diverge away from the target branch at the second switch node, it will have an approximation level of 0.

All the *case* branches which prevent the target branch from being executed are the *case* branches which have one of the following breaking options: *break*, *return* or *goto* statement. All these branches stop the execution of the *switch-case* structure and force the exit from this multi-way decision construct. All the *case* branches which don't have a *jump* or a *break* option are considered as not preventing the target branch to be missed and they are merged in the CMCFG graph with the next *case* branches which have a *break* option.

The CMCFG that corresponds to the complex *switch-case* construct presented in Fig. 3.4 is shown in Fig. 3.6. The node which corresponds to the *case* 'i' branch has no *break* or *return* statements and therefore it is merged with the node which corresponds to the *case* 'n' branch. In Fig. 3.6 the *case* 'n' node has resulted by merging *case* 'i' and *case* 'n' nodes. So it doesnt matter whether the test data is 'i' or 'n', because the target is prevented to be executed only when the *break* statement is encountered.

In CMCFG the *case* branches having no breaking options are not represented. Instead of these *cases*, the next *case* branch which has a *break* or a *return* statement is displayed. Compared to the approach based on critical branches, this one is more compact because it can be successfully used for modeling different type of *switch* constructs and the decision nodes which don't prevent the target to be covered are not present in the graph. The processing time of this new graph is smaller compared to the processing time for the FCFG, because the graph has fewer nodes.

The new proposed fitness function used for evaluating each test data is:

$$F(test\_data) = Approx\_level + \sum Normalized\_branch\_distance \tag{3.3}$$

The sum that appears in 3.3 refers to the sum of the normalized branch distances computed for each gene of the individuals using 3.4:

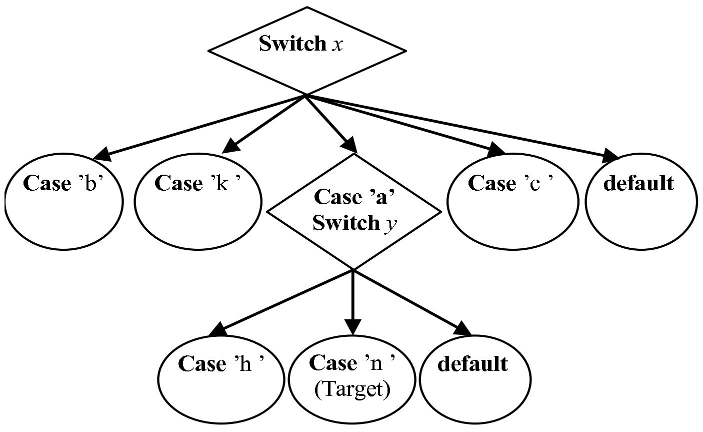


FIGURE 3.6. *CMCFG for complex switch-case construct*

$$Normalized\_branch\_distance = \frac{Branch\_distance}{Branch\_distance + 1} \tag{3.4}$$

When computing the fitness function, the normalized branch distance is chosen because the *approximation level* is more important than the *branch distance*. We use equation 3.4 for normalizing the *branch* distance according to the study presented in [3].

The *branch distance* is calculated using the functions based on relational predicates introduced by Korel in [10], for which the *switch* expression value and the target case value are used: $|switch\_expr - target\_case|$.

The test data values $x$ = 'b' and $y$ = 'h' will diverge away at the level of the node *case* 'b'; therefore the approximation level will be 1. The fitness function will be (|98 − 97| / |98 − 97| + 1) + (|104 − 110| / |104 − 110| + 1) + 1 = 2.35.

The second test data values $x$ = 'a' and $y$ = 'h' will diverge away at the level of the node case 'h'; therefore the approximation level will be 0. The fitness function will be (|104 − 110| / |104 − 110| + 1) = 0.85.

Taking into consideration that better test data always attain smaller fitness values, by comparing the previous pairs of test data, it is easy to find out that the second one is closer to the desired test data values ($x$ = 'a' and $y$ = 'n'). This means that the approach based on CMCFG and *branch distance*, introduced by Korel in [10], gives a better guidance to the evolutionary search process than the evolutionary approaches based on nested *if-then-else* and ACBs.

**3.4. Fitness calculation based on the CMCFG approach and Euclidian distance.** In the previous section the formula proposed by Korel [10] was used for calculating the branch distance. The fitness function used for evaluating each test data was composed of the *approximation level* and the normalized *branch distance*.

Another new fitness calculation approach based on CMCFG representation of the *switch-case* construct and the Euclidian distance is proposed hereinafter. It uses for evaluating each test data the CMCFG model for representing the entire *switch-case* construct in combination with the fitness function formula presented in 3.5:

$$F(test\_data) = Approx\_level + \sum Euclidian\_distance \tag{3.5}$$

The Euclidian distance formula is the most commonly used formula for calculating distances. It examines the square root differences between coordinates of a pair of objects. Considering two points in an $n$-dimensional search space, the Euclidian distance formula is shown in 3.6:

$$Euclidian\_distance = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + ... + (x_n - y_n)^2} \tag{3.6}$$

Formula 3.6 is adapted for calculating distance in evolutionary testing by making the following changes:

1. Each $x$ variable is the value of the node which is located along the target path in CMCFG. For instance, for the simple *switch-case* construct presented in Fig. 2.3 the $x$ variable from the Euclidian distance formula has the value 5.

2. Each $y$ variable form Euclidian distance formula represents the actual test data values. In case of the complex *switch-case* construct presented in Fig. 3.4 for a set of test data equal to ('b', 'h'), the $y$ values for 3.6 will be replaced with 98 and 104 (the ASCII codes for 'b' and 'h').

For the simple *switch-case* construct presented in Fig. 2.3, in case $y1$ == 'c' the Euclidian distance will be: $\sqrt{(97-99)^2}$ . The fitness value according with 3.5 for test data equal to 'c' is: $0 + 2 = 2$.

For the most complex *switch-case* construct presented in Fig. 3.4, in case of pair values (y1 == 'b', y2 == 'h') the Euclidian distance will be: $\sqrt{(97-98)^2 + (110-104)^2}$. The fitness function value according to 3.5 for the test values $x$ == 'b' and $y$ == 'h' is: $1 + 6.08 = 7.08$.

The performance for the new fitness calculation approach based on approximation level and Euclidian distance will be compared with the other three approaches (nested *if-then-else*, ACBs and CMCFG with normalized branch distance) in order to be able to find out easy which one is the best for generating test data.

**4. Experimental Results.** The experiments using the two new estimation performance formulas in conjunction with the CMCFG model were executed on eight different *switch-case* constructs having different nested levels - from 0 to 7. All these switch-case constructs were also tested using the nested *if-then-else* approach and the Alternative Critical Branches approach.

The software program used for testing the *switch-case* constructs was written in C# and all the experiments were performed using a system equipped with an Intel I3 processor running at 2.2 GHz, and Windows 7 Operating System.

For all the four approaches, ten runs were performed for testing each *switch-case* construct and the results were compared. For testing the switch-case constructs an evolutionary framework was designed and implemented in C#. The high level architecture of the software program used for generating test data which cover the target is presented in Fig. 4.1. It has a separate component which implements each evolutionary method described in Section III and three layers which together form an application able to automatically generate test data for a particular path from the software under test. The software program is a desktop application which has a very easy to use interface.

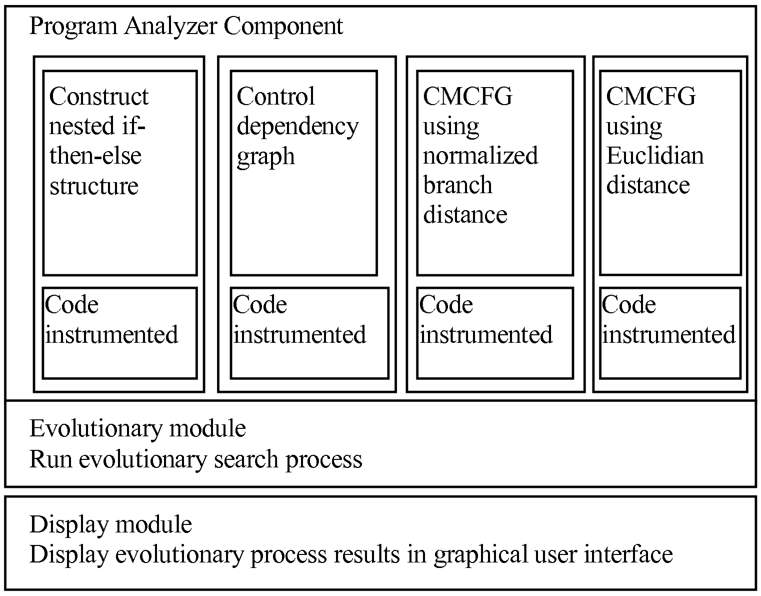

Program Analyzer Component
Construct nested if-then-else structure
Control dependency graph
CMCFG using normalized branch distance
CMCFG using Euclidian distance
Code instrumented
Code instrumented
Code instrumented
Code instrumented
Evolutionary module
Run evolutionary search process
Display module
Display evolutionary process results in graphical user interface

FIGURE 4.1. *High-level architecture of the evolutionary framework*

The software program used for experiments is composed of three parts: a static analyzer module, a module for running the evolutionary process and a module for displaying the graphical results.

The module that performs the static analysis consists of four sub-modules which build the nested *if-then-else* structures, or build the dependency graph flow and the CBS, or build the two CMCFG approaches (depending on which approach is to be executed). The static analyzer component instruments the code with the information needed for calculating the fitness function.

The module that executes the evolutionary process uses the data provided by the program analyzer module and runs the genetic algorithm which represents the evolutionary search method in the ET process. This module runs the evolutionary process for 100 generations and uses an initial population composed of 40 randomly generated individuals. Each individual from the population represents a different test data which are applied for testing the *switch-case* constructs.

The graphical module takes the results provided by the evolutionary module and displays them in a user interface. The best individual from each generation is displayed in a data grid. For the current generation, the software application displayed the individual genes values, the fitness function value and the computational time needed for each generation.

Table 4.1 presents the best run for each of the four evolutionary approaches out of ten runs for each. It shows that the iteration number at which the evolutionary algorithm is able to find test data which covers the target branch is smaller for the two CMCFG approaches compared to the ACB and nested *if-then-else* approaches.

Table 4.1
*Experimental results - the iteration number at which the solution is found*

| **Nested level** | **Evolutionary process** | | | |
|---|---|---|---|---|
| | IF-THEN-ELSE (nested if-then-else structure) | ACB (Alternative Critical Branches Approach) | CMCFG+ Normalized branch distance | CMCFG+ Euclidian distance |
| 0 | 27 | 18 | 7 | 12 |
| 1 | 90 | 56 | 30 | 36 |
| 2 | 98 | 65 | 40 | 46 |
| 3 | 100 | 79 | 52 | 59 |
| 4 | >100 | 83 | 60 | 72 |
| 5 | >100 | 91 | 68 | 76 |
| 6 | >100 | 96 | 80 | 88 |
| 7 | >100 | 100 | 89 | 95 |

The test data were generated for unstructured switch-case constructs having case branches with no break or return options. The processing time for the CMCFG-based methods was smaller compared to the processing time needed for the CBS-based and the nested if-then-else structures approaches.

The processing time strongly depends on the number of nodes in the control flow graph. If the CMCFG has one branch node less than the normal control flow graph, then from our experiments the processing time resulted to be significantly smaller compared to the processing time for a normal control flow graph. From the experiments that were run, it came out that for each nested level our proposed methods are faster with about 1 millisecond per iteration in comparison with other two approaches.

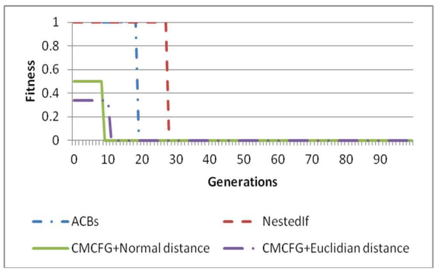
Fig. 4.2 shows the results obtained for each nested level of the tested *switch-case* construct. All four approaches are displayed on the same graphic in order to facilitate their comparison.

As shown in Fig. 4.2, the proposed CMCFG-based approaches which are using the normalized distance and the Euclidian distance for calculating fitness function converge faster than the two other approaches. In the previous figures one can notice that for each switch-case construct the CMCFG based approaches converge to 0 in a smaller number of generations in comparison with the other two evolutionary approaches.

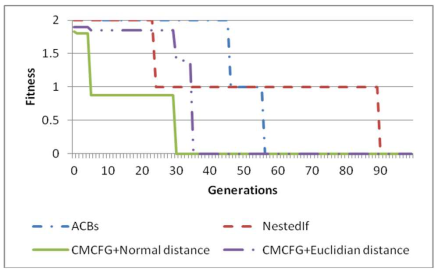
The nested *if-then-else* approach is not able to generate test data for the target branch in 100 generations for a *switch-case* construct with 4 nested levels. The CBA-based approach converges much slower in comparison with our CMCFG approach that is using the two different fitness function formulas that we proposed. This means that the fitness function formulas used in CMCFG and introduced in this paper improve the guidance of the testing process based on evolutionary searches, compared to the two other approaches that were also tested. The process improvement has been illustrated in Table I, which indicates the iteration at which each approach is able to find test data for the target branch.

## 5. Conclusions and Future Work.

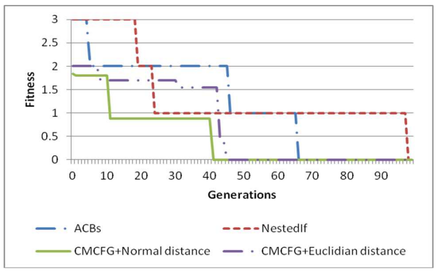
In this paper, the approach based on nested *if-then-else* constructs and the one based on ACBs have been pointed out to be problematic because of a poor guidance of the search algorithm. The new performance estimation formulas introduced in this paper are used in conjunction with a



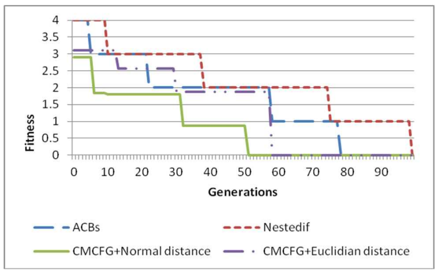
(a) Test data generation Switch-case construct with Nested level 0



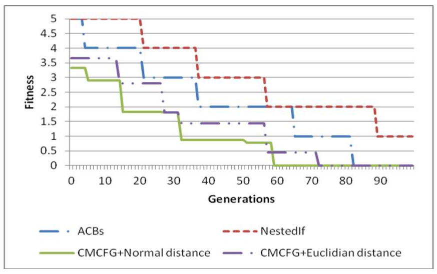
(b) Test data generation Switch-case construct with Nested level 1



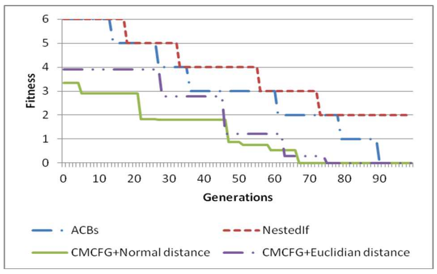
(c) Test data generation Switch-case construct with Nested level 2



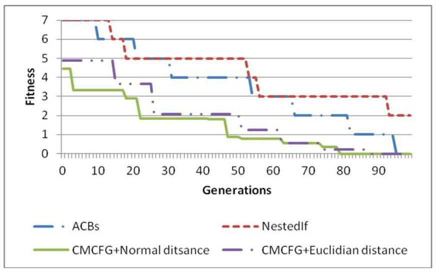
(d) Test data generation Switch-case construct with Nested level 3



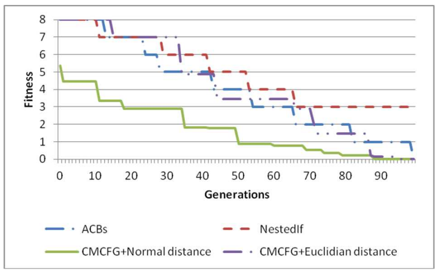
(e) Test data generation Switch-case construct with Nested level 4



(f) Test data generation Switch-case construct with Nested level 5



(g) Test data generation Switch-case construct with Nested level 6



(h) Test data generation Switch-case construct with Nested level 7

FIGURE 4.2. *Test data generation Switch-case construct with different Nested levels*

new representation model of *switch-case* constructs, which is the CMCFG.

Our proposed approaches using CMCFG representation in conjunction with normalized branch distance and Euclidian branch distance are able to find test data for a given target path in a smaller number of iterations in comparison with the other evolutionary approaches. From the practical experiments it came out that the fastest is CMCFG with normalized branch distance.

Table I illustrates that our proposed formulas for calculating fitness function are 40 iterations faster in finding test data than the *if-then-else* approach and 20 iterations faster than the ACB approach. Since the best evolutionary approach finds test data in a smallest number of iterations, Table 1 clearly shows that the best method is the CMCFG-based one with normalized *branch distance*. The two new fitness function formulas used in conjunction with the CMCFG approach are two original approach proposed, implemented and tested here.

Future work will involve using evolutionary algorithms for generating test data that cover a particular target branch in larger projects from the applicative area. In order to be able to completely automate test data generation process, a complete software framework should be implemented. This software framework should offer the human tester the possibility of choosing between different approaches and evolutionary algorithms, make suggestions concerning the best strategy to adopt for different classes of software programs etc.

## REFERENCES

[1] Phil Mcminn and Mike Holcombe. *The State Problem for Evolutionary Testing.* In *Genetic and Evolutionary Computation Conference (GECCO)*, Springer-Verlag, 2003, pp. 2488-2498

[2] AnkurPachauriand Gursaran. Program test data generation branch coverage with genetic algorithm: Comparative evaluation of a maximization and minimization approach. *International Journal of Software Engineering and Applications*, 3(1):207-218, January 2012.

[3] A. Arcuri. It does matter how you normalise the branch distance in search based software testing. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 205–214, 2010.

[4] André Baresel, David Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, pages 108–118, New York, NY, USA, 2004. ACM.

[5] André Baresel and Harmen Sthamer. Evolutionary testing of flag conditions. In *Proceedings of the 2003 international conference on Genetic and evolutionary computation: PartII*, GECCO'03, pages 2442–2454, Berlin, Heidelberg, 2003. Springer-Verlag.

[6] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[7] Nirmal Kumar Gupta and Mukesh Kumar Rohil. Using genetic algorithm for unit testing of object oriented software. In *Proceedings of the 2008 First International Conference on Emerging Trends in Engineering and Technology*, ICETET '08, pages 308–313, Washington, DC, USA, 2008. IEEE Computer Society.

[8] Mark Harman, Lin Hu, Robert M. Hierons, André Baresel, and Harmen Sthamer. Improving evolutionary testing by flag removal. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 1359–1366, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[9] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Trans. Software Eng.*, 36(2):226–247, 2010.

[10] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, August 1990.

[11] P. McMinn. Search-based software testing: Past, present and future. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 153–163, 2011.

[12] Phil McMinn, David Binkley, and Mark Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Trans. Softw. Eng. Methodol.*, 18(3):11:1–11:27, June 2009.

[13] Phil McMinn and Mike Holcombe. Evolutionary testing of state-based programs. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1013–1020, New York, NY, USA, 2005. ACM.

[14] Nigel Tracey, John Clark, and Keith Mander. Automated program flaw finding using simulated annealing. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '98, pages 73–81, New York, NY, USA, 1998. ACM.

[15] Yan Wang, Zhiwen Bai, Miao Zhang, Wen Du, Ying Qin, and Xiyang Liu. Fitness calculation approach for the switch-case construct in evolutionary testing. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, GECCO '08, pages 1767–1774, New York, NY, USA, 2008. ACM.

[16] Stefan Wappler, Andre Baresel, and Joachim Wegener. Improving evolutionary testing in the presence of function-assigned flags. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION '07, pages 23–34, Washington, DC, USA, 2007. IEEE Computer Society.

[17] Joachim Wegener and Matthias Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Syst.*, 15(3):275–298, November 1998.

*Edited by:* Marc Eduard Frîncu
*Received:* Sep 26, 2012
*Accepted:* Apr 8, 2013