A New Evolutionary Approach for the Structural Testing of Switch-case Constructs

Gentiana Ioana Latiu, Octavian Augustin Cret and Lucia Vacariu
Technical University of Cluj-Napoca, Computer Science Department, Bariiu Street, Cluj-Napoca, Romania

Keywords: Evolutionary Structural Testing, Control Flow Graph, Switch Case Constructs.

Abstract: Evolutionary structural testing uses specific approaches based on guided searches that involve evaluating fitness functions to determine whether test data satisfy or not various structural testing criteria. For testing switch-case constructs the nested if-then-else structure and Alternative Critical Branches (ACBs) approaches were used so far. In this paper a new evolutionary structural approach based on Compact and Minimized Control Flow Graph (CMCFG), which is derived from the concept of Control Flow Graph (CFG), is presented. Experiments on different levels of imbrications demonstrate that this new approach has significantly better results in finding test data which cover a particular target branch in comparison with the previous approaches reported in the literature.

1 INTRODUCTION

The main idea behind evolutionary testing process is to transform the test goal into an optimization problem that is solved using evolutionary algorithms (Wegener et al., 2001). The evolutionary process search space is represented by the domains of the input variables of the software program under test. Evolutionary structural testing has been intensively used for generating test data by many researchers. Harman and McMinn (2010) present a theoretical exploration of global search techniques embodied by Genetic Algorithms. Other approaches related to evolutionary testing with flag conditions are presented in Baresel and Sthamer (2003), Baresel et al., (2004), and Wappler et al., (2007). Different transformations were applied and reported in the literature for Evolutionary Testing in order to improve the fitness function calculation, because a well-defined fitness function is essential for the efficiency of evolutionary search process ((Harman, et al., 2002), (McMinn and Holcombe, 2005), and (McMinn et al., 2009)).

The main software programs constructs (loops, simple statements, if-then-else decision structures) were extensively tested in the literature using evolutionary algorithms. Less work has been done on the switch-case constructs which are used to express multi-way decisions and were studied in Wang, et al. (2008), where the switch-case construct was tested using the concept of Alternative Critical Branches (ACBs). ACBs consist of all case branches that can lead to a miss of chosen target branch when the target branch is leaving a switch node. The ACBs consist of one element that is the alternative branch of target if it is leaving a two-way decision node. Each control dependent node has assigned only one ACB. All the ACBs with respect to the target branch make up a set. The array of all the corresponding ACBs for the target branch forms the Critical Branches Set (CBS). This is extended from the single critical branch concept. If any element which is contained in CBS corresponding to target branch is taken, then there is no chance to cover the target branch. The focus in this approach is on structural testing of multi-way decision statements, in particular on branch coverage.

Our paper proposed a new evolutionary approach for testing switch-case constructs. The main idea of this approach is to generate a Compact and Minimized Control Flow Graph (CMCFG), derived from Control Flow Graph (CFG). The CFG is a directed graph where each node has at most two successors (Ferrante et al., 1987). Inside this new Compact and Minimized Control Flow Graph (CMCFG) each node can have more than two successors and all the case branches which correspond to the same switch node are on the same level. The case branches which don’t have any break or return options are merged with the next case
branches which have one of these options and thus a new, improved fitness function was proposed, tested and compared against the previous ones reported in the literature (Wang, et al., 2008).

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 and Section V presents the final conclusions and future work.

2 EVOLUTIONARY TESTING METHODOLOGY AND SWITCH-CASE CONSTRUCTS

Evolutionary testing (ET) is a meta-heuristic approach by which test data can be generated automatically using optimization search algorithms. The search space is represented by the variation domains of the input variables of the software under test, in which test data fulfill the specific test objectives. ET is generally used in many search problems in software testing, because it has a very good capacity of adapting itself to the system under test. The main steps of ET process are presented in Figure 1:

![Evolutionary Testing Process](image)

Figure 1: Evolutionary testing process.

ET was successfully applied for different forms of testing, namely: specification testing (Tracey et al., 1998), unit testing (Gupta and Rohil, 2008), and extreme execution time testing (Wegener and Grochtmann, 1998).

During the ET process the test data are initially randomly generated and take values from the domains of input variables of the software under test. Then the test data performance is evaluated based on the fitness function which represents a formalized version of the test objective. If the established testing criteria are met, then the process stops and the best solution found will be the testing solution, otherwise the test data will be modified using specific evolutionary operators and the process will restart by evaluating the new test data. The most important evolutionary operators used during ET process are crossover and mutation. Crossover is used to combine two parents to produce a new offspring. Mutation is used for altering a gene value from the chromosome (switching from 1 to 0 in case of binary chromosomes).

Based on the ET methodology the goal of this research was to study the switch-case construct in the context of structural testing, aiming to find test data which executes a particular branch in a program containing multi-way decision constructs. In order to retrieve the input data which triggers 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 software program source code and it reduces repetitive coding.

The general structure of a switch-case construct is presented in Figure 2:

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

Figure 2: General switch-case conditional construct.

The switch-case construct 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 which can be an expression of any type and executes the case branch that corresponds to the expression’s value. It can include any number of case statements. Each case branch is followed by an optional break, return or goto statement (named breaking statements). These statements are used either to break out of the switch construct when a match is found, or return a value and exit the switch body, or go to a specific location in the code.
If the optional statements break, return and goto are not present after a case branch then the control flow is transferred to next case branch until it will meet one of the breaking statements. If an expression passed to 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 switch-case construct is presented in Figure 3.

```c
Switch {x} {
    Case 1: y = 4; break;
    Case 5: y=30; break;
    Case 2: y=8; break;
    Case 0: y=0; break; // Target branch
    Default: y=1;
}
```

Figure 3: Simple switch-case conditional construct.

A previous work (Wang, et al., 2008) has argued that for a particular branch condition, the Critical Branches Set (CBS) should be defined. This array is composed by all case branches causing the target to be missed. The CBS which corresponds to the target branch from the source code listed in Figure 3 is composed by \{branch “case 1”, branch “case 5”, branch “case 2”, and branch “default”\}. The branch target is definitely missed when the execution of test data diverges away down any branch which is in CBS.

The fitness function used for evaluating each test data is calculated using the sum between two metrics: the approximation level and the branch distance. The approximation level is calculated by subtracting 1 from the number of ACBs which are between the node from which the test data diverges away and the target itself (the branch that corresponds to “case 0”). The branch distance is calculated using the following expression \(|expr - C| + 1\), where expr is the value of the expression which appears after switch keyword, C is the constant value for the desired case statement and 1 is the positive failure constant (Tracey et al., 1998). For example, if \(x = 10\), then the branch distance metric for the target branch specified in Figure 3 is \(|10 - 0| + 1 = 11\). The fitness value indicates how close the test data are to triggering the execution of the code located on the particular branch of the switch statement, which constitutes the target of the current evaluation.

### 3 FITNESS 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 constructs with respect to the Control Flow Graph (CFG). The switch-case construct presented in Figure 3 is equivalent to the nested if-then-else construct shown in Figure 4:

```c
If (x==1) {
    y=4;
} else if (x==5) {
    y=30;
} else if (x==2) {
    y=8;
} else if (x==0) {
    y=0; // Target branch
} else { y=12; }
```

Figure 4: Transformation of switch-case conditional constructs in nested if-then-else statements.

The target branch for which test data should be generated is the case branch corresponding to \(x = 0\). In order to be able to generate test data which cover this specific branch, every potential solution randomly generated by the evolutionary search process must be evaluated using a fitness function. The aim of the fitness function is to guide the evolutionary search to find the proper test data which execute the target branch.

In structural testing, previous work (Gursaran, 2012) has demonstrated that the fitness function having the expression illustrated in (1) evaluates how close the test object is to cover the target branch.

\[
\text{Fitness(test_data)} = \text{Approximation Level} + \ \frac{\text{Normalized Branch Distance}}{\text{Normalized Branch Distance}} \tag{1}
\]

The normalized branch distance is computed using (2) and indicates how close the test object is to take the alternative branch.

\[
\text{Normalized Branch Distance} = 1 - \text{Distance} \tag{2}
\]

The approximation level counts the number of decision nodes lying between the decision node where the actual test data diverge away from the target branch itself. In Figure 5 given \(x = 1\) the control flow takes the true branch at decision 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 (Gursaran, 2012). For the branching condition \( x = 1 \) the branch distance is \(|x - 1|\).

In conclusion the approach which uses nested if-then-else statements to represent switch-case constructs is not a perfect one because the fitness value for \( x = 5 \) is smaller than the fitness value for \( x = 1 \) even though 1 is much closer to 0 in comparison with 5. This approach is not guiding the evolutionary search algorithm in the correct direction, because the dependencies between case branches result in an inappropriate approximation value.

### 3.2 Fitness Calculation based on Alternative Critical Branches Approach

The approach for fitness calculation based on ACBs assumes that all case branches in the switch-case construct are mutually exclusive in semantics. A special CFG called Flattened Control Flow Graph (FCFG) is described in Gursaran (2012). This graph is extended from the traditional CFG, with the only difference that the switch node is allowed to have more than two successors. In this graph each case branch is control dependent only on the switch branching node.

Figure 6 presents the FCFG corresponding to the switch-case construct presented in Figure 3:

Based on the FCFG definition each node has assigned an array of control nodes on which it depends. The target branch is definitely missed when the execution of test data 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 test data cover the target branch. In the example shown in Figure 6 the CBS attached to the target branch is composed by: branch “case 1”, branch “case 5”, branch “case 2” 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 “case 0”.

With this proposed concept of CBS and FCFG the approximation level metric (that is part of the fitness function expression) is calculated by subtracting 1 from the number of critical branches situated between the node from which the test data
dervge away from target and the target itself. The target branch is the case branch for which the evolutionary algorithm should generate test data. The branch distance metric used for evaluating the test data uses the value which comes after the switch expression and the constant for the target branch.

Using this approach for the case when \( x = 1 \) or 5, the approximation level value will be 0 and the branch distance will be \( |1 - 0| + 1 = 2 \) and \( |5 - 0| + 1 = 6 \), respectively. The fitness calculated based on (1) will be equal to 2 in case \( x = 1 \) and equal to 6 in case \( x = 5 \).

For this simple case it is obvious that the fitness value based on ACBs approach is guiding the evolutionary search in a correct direction compared to the nested if-then-else approach, because \( x = 1 \) has a smaller fitness value in comparison with \( x = 5 \). 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 Figure 7:

![Figure 7: Complex switch-case conditional construct.](image)

The corresponding FCFG is shown in Figure 8:

![Figure 8: Flattened control flow graph.](image)

For the complex switch-case structure presented in Figure 7, if test data is composed by \( x = 1 \) and \( y = 7 \) the approximation level is 1 and the branch distance is \( |1 - 0| + 1 = 2 \). The total fitness function value is 3. If \( x = 0 \) and \( y = 7 \) then the approximation level is 0 and the branch distance is \( |7 - 13| + 1 = 7 \). The total fitness function value is 7. So the pair of values (\( x = 1, y = 7 \)) has a smaller fitness value than (\( x = 0, y = 7 \)), even though the second pair of values is closer to the solution values (\( x = 0, y = 0 \)).

So it is obvious that the fitness value calculation approach proposed in Wang, et al. (2008), which is based on ACBs approach, misleads the evolutionary search process.

### 3.3 Fitness Calculation based on CMCFG Approach

To correctly guide the evolutionary search algorithm in a correct direction we propose a new approach, CMCFG.

As shown in Figure 8 all the switch nodes have as descendants several case branches. For the target branch, one or more case branches can lead to the target branch being missed.

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 approximation level starts in CMCFG top-down. As shown in Figure 8, if test data derive away from target branch at first switch have an approximation level of 1, and if the test data derive away from target branch at the second switch 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 options: break, return or goto statement. All these branches stop the execution of the switch-case constructs and force the exit from this structure. For the case branches which don’t have a jump or break option, they 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 structure presented in Figure 7 is shown in Figure 9:

![Figure 9: Complex CMCFG graph.](image)
because the target is prevented to be executed only when break option is met.

![CMCFG for switch-case construct with 1 nesting level.](image)

Figure 9: CMCFG for switch-case construct with 1 nesting level.

In CMCFG the case branches which have no break, return or goto statements are not represented. Instead of these cases, the next case branch which has a break or a return statement is represented. Compared to the approach based on critical branches, this approach 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 in comparison with the processing time for FCFG, because the graph has fewer nodes.

The fitness function which evaluates each test data is presented in (3):

\[
\text{Fitness(test data)} = \text{Approximation Level} + \sum \text{Normalized branch distance} \quad (3)
\]

The sum that appears in (3) refers to the sum of the normalized branch distances computed for each gene of the individuals using (4):

\[
\text{Normalized branch distance} = \frac{\text{branch distance}}{\text{branch distance} + 1} \quad (4)
\]

In fitness function calculation the normalized branch distance is chosen because the approximation level is more important in comparison with branch distance. We use the equation (4) for branch distance normalization based on the study presented in Arcuri (2010). The formula used for our proposed fitness function is derived from (1). The sum of normalized branch distance allows the algorithm to converge faster. This formula was chosen based on some experimental practical trials made before.

The branch distance is calculated using the switch expression value and the target case value: \([\text{switch expr} - \text{target case}]\).

The test data values \(x = 1\) and \(y = 7\) will diverge away at node “case 1”; therefore the approximation level will be 1. The fitness function will be \((1 - 0) / (1 - 0 + 1) + (7 - 13) / (7 - 13 + 1) + 1 = 2.35\). The second test data values \(x = 0\) and \(y = 7\) will diverge away at node “case 7”; therefore the approximation level will be 0. The fitness function will be \((7 - 13) / (7 - 13 + 1) = 0.85\). The second test data object has a fitness function value smaller than the first test data object, which means it is closer to the desired test data values \((x = 0\) and \(y = 13\)). This means that the approach based on CMCFG gives a better guidance to the evolutionary search process in finding test data covering the target branch in comparison with the approaches based on nested if-then-else and ACBs.

4 EXPERIMENTAL RESULTS

Experiments using the new approach based on CMCFG 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 ACBs approach.

The tool used for testing the switch-case constructs was written in C# and all the experiments were performed using a PC having the following configuration: Intel I3 processor running at 2.2 GHz, and Windows 7 Operating System.

For all the three approaches ten runs were performed for testing each switch-case construct and the results were compared. The architecture of the software program used for generating test data to cover the target branch using the three approaches presented in Section III is presented in Figure 10:

![High-level architecture of the software program.](image)
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 three sub-modules which build the nested if-then-else structures, or build the dependency graph for ACBs approach, or build the CMCFG (depending on which approach is to be executed). The static analyzer 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 component and performs the evolutionary process. This is using genetic algorithms for evolutionary process. This module runs the evolutionary process for 100 generations and uses an initial population composed of 40 randomly generated individuals. For all three approaches the individuals are generated using Random class instance from .Net. This class uses a time-dependent default seed value. The default seed value is based on system clock and has finite resolution. Each individual from the population consists of a set of genes in which each gene corresponds to a data input variable of the program under test.

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

Figure 11 and Figure 12 show the user interface of the software program that was created for performing experiments. Figure 11 shows the settings area from the user interface of the software program, where the user can choose which evolutionary algorithm will be used for generating test data, which switch-case construct will be tested (selected as target) and also which fitness calculation approach will be used.

By checking “ACB” or “Nested” options, the ACBs approach or the nested if-then-else approach will be applied. If none of these options is checked then the CMCFG approach is applied by default. The parameters for the current evolutionary algorithms can be set up as well: population size, number of generations, individual length and also the selection method.

Figure 12 shows the results obtained for testing a switch-case construct which has the nested level 0. These results are obtained for a random run. The results table allows the developer to trace the current algorithm’s execution and displays all the important information: the generation number, the value of the best individual from the current generation, the fitness function for the best individual and the processing time for current generation in milliseconds.

Figure 11: Software application’s user interface.

For the run shown in Figure 12, the algorithm found test data which cover the target case branch at generation 4. The Current iteration column displays the current iteration of the evolutionary search process. The second column contains the best individual from the current generation. The column called Performance display the fitness function value corresponding to the best individual from the current iteration. The last column from the data grid display the Time needed for processing the entire population of 40 individuals.

Figure 12: Software application’s results view.

Table 1 presents the best run for each of the three 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 CMCFG approach compared to the two other approaches.

Table 1: Experimental results – the iteration number at which the solution is found.

<table>
<thead>
<tr>
<th>Nested level</th>
<th>IF-THEN-ELSE (nested if-then-else structure)</th>
<th>ACBs (Alternative Critical Branches Approach)</th>
<th>CMCFG (Compact and Minimized Control Flow Graph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>27</td>
<td>18</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>90 graph</td>
<td>56</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>98</td>
<td>65</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>&gt;100</td>
<td>79</td>
<td>52</td>
</tr>
<tr>
<td>4</td>
<td>&gt;100</td>
<td>83</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>&gt;100</td>
<td>91</td>
<td>68</td>
</tr>
<tr>
<td>6</td>
<td>&gt;100</td>
<td>96</td>
<td>80</td>
</tr>
<tr>
<td>7</td>
<td>&gt;100</td>
<td>100</td>
<td>89</td>
</tr>
</tbody>
</table>

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 method was smaller compared to the processing time needed for the ACBs approach and the nested IF-then-else constructs approach. 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 in comparison with the processing time for a normal control flow graph. From the experiments implemented was found out that for each nested level our proposed method managed successfully to find test data which cover the target branch in less number of iterations.

Figures 13 ÷ 20 show the results obtained for each nested level of the switch-case construct. All three approaches are displayed on the same figure in order to facilitate their comparison.

Figure 13: Test data generation for a particular case branch in nested level 0 – switch-case construct.

Figure 14: Test data generation for a particular case branch in nested level 1 – switch-case construct.

Figure 15: Test data generation for a particular case branch in nested level 2 – switch-case construct.

Figure 16: Test data generation for a particular case branch in nested level 3 – switch-case construct.

Figure 17: Test data generation for a particular case branch in nested level 4 – switch-case construct.
As shown in the previous Figures, the proposed CMCFG-based approach converges faster than the two other approaches. The nested if-then-else approach is not able to generate test data for a particular case in 100 generations for a switch-case construct with more than 3 nested levels. The ACBs based approach converges much slower in comparison with our proposed approach. This means that the fitness function formula used in this paper improves the guidance of the evolutionary search process, compared with the other tested approaches. The process was improved with approximation 15 iterations in comparison with ACBs approach and with approximation 50 iterations in comparison with nested if-then-else approach.

In Figure 21 there is displayed a comparison between the three approaches for generating test data which cover target branch in switch-case constructs which have the nesting level between 0 and 7. As it is shown for all tested switch-case constructs our approach was able to generate test data in less number of iterations in comparison with ACBs approach and nested if-then-else approach.

5 CONCLUSIONS AND FUTURE WORK

Evolutionary testing uses evolutionary search algorithms to generate test data that cover a particular path in a software program. 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. This paper introduced a new approach for calculating the fitness function for switch-case constructs which improves the evolutionary testing process.

For generating test data which cover a particular case branch in a switch-case construct the CMCFG approach was used. The solution was tested on switch-case constructs having different levels of nesting and which can also have case branches without break or return options.

The proposed improvements solve the problem of generating test data for a particular case branch in a switch-case construct faster with approximation 15 iterations in comparison with ACBs approach and with approximation 50 iterations faster than nested if-then-else approach. The representation of the switch-case construct as a CMCFG structure is an original approach proposed here. The formula used for fitness function is also an original metric.
A New Evolutionary Approach for the Structural Testing of Switch-case Constructs

proposed here which improves the guidance of the evolutionary search method.

Future work will involve using evolutionary algorithms for generating test data that cover a particular case branch in larger projects. Also Simulated Annealing and PSO algorithms will be implemented for testing switch-case constructs. A testing framework based on evolutionary algorithms could be designed and implemented, for completely automate the test data generation process.

REFERENCES


