Fitness Calculation Approach for the Switch-Case Construct in Evolutionary Testing

Yan Wang, Zhiwen Bai, Miao Zhang, Wen Du, Ying Qin, Xiyang Liu*
Software Engineering Institute, Xidian University
Xi'an, Shaanxi 710071, China
{ywangxd, baizhiwen, emilia.zhang, WenDuXD, qinyingxd, xiyangliu}@gmail.com

ABSTRACT

A well-designed fitness function is essential to the effectiveness and efficiency of evolutionary testing. Fitness function design has been researched extensively. For fitness calculation, so far the switch-case construct has been regarded as a nested if-else structure with respect to the control flow. Given a target embraced in a *case* branch, test data taking different *case* branches receive different approximation levels. Since the approximation levels received by test data do not evaluate their suitability accurately, the guidance provided by the existing approach to evolutionary search is misleading or lost. Despite the switch-case construct's wide use in industrial applications, no previous work has addressed this problem.

In this paper, a Flattened Control Flow Graph and a Flattened Control Dependence Graph for the switch-case construct are first presented, and a unified fitness calculation approach based on Alternative Critical Branches is proposed for the switch-case and other constructs. The concept of Alternative Critical Branches is extended from the single critical branch. Experiments on several large-scale open source programs demonstrate that this approach contributes a much better guidance to evolutionary search.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation, Measurement

Keywords

Switch-Case Construct, Fitness Function, Evolutionary Testing

1. INTRODUCTION

Generating test data manually is extremely tedious, costly and error-prone, but this activity is mandated by many standards such as U.S. RTCA's DO-178B. Evolutionary Testing (ET) is a technique with which test data can be generated automatically according to a test adequacy criterion. ET reformulates test data generation as an evolutionary search guided by the fitness of test data. The search space is the input domain of the software under test. ET has proved to be valuable for many test data generation problems, including specification testing [9], stress testing [5], finite state machine testing [3] and so on. To date, structural testing [1, 2, 4, 7, 10, 12, 16, 17, 19] is the most common form of evolutionary testing considered in the literature, and it remains a hot topic. A well-designed fitness function is crucial to the effectiveness and efficiency of evolutionary search. The more discriminating the fitness function, the more efficient the search. Literature reviews demonstrate the large volume of work concentrating on the design and implementation of fitness function [7, 8, 10, 14, 16, 17]. This paper focuses on structural testing, in particular branch coverage.

Similar to the if-else construct, the switch-case construct is widely used to express multi-way decision. To generate branch adequate test sets, the target may be embraced in a *case* branch. When the target branch is deeply nested in a structure involving the switch-case construct, random testing performs poorly or fails to find the desired test data. In this case, a directed search is required. ET is a promising directed search approach. In the field of ET, the switch-case construct is currently regarded as a nested if-else structure for fitness calculation. However, based on the control dependence graph and control flow graph of the nested if-else structure, test data cannot be evaluated appropriately, which causes the fitness to provide misleading guidance to evolutionary search.

In a switch-case construct, each constant after the *case* keyword is distinct. All *case* branches are mutually exclusive, and only one *case* branching node can be evaluated as true at one time. Therefore, if the condition of a particular *case* branch is satisfied, the conditions of other *case* branches will definitely not be satisfied. So there exists no control dependence among *case* branching nodes, and thus the control dependence of the target on other *case* branching nodes is meaningless and produces inappropriate fitness.

The inaccurate control dependence stems from the fact that the control flow graph of the nested if-else structure does not reflect the essential multi-way decision semantics of the switch-case construct. As a result, given a target embraced in a *case* branch, test data taking different *case* branches will receive different approximation levels which cannot accurately assess their suitability for the coverage of the target. The guidance for evolutionary search will be hampered or lost in severe cases on the basis of the existing fitness calculation approach. Despite the large volume of work on the design and implementation of fitness function, there has been no work focusing on the fitness calculation problem for the switch-case construct although it is widely used in industrial programs.

This paper proposes a unified fitness calculation approach for the

*Corresponding author

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.

switch-case and other constructs. The basic idea of this approach is to propose a new control flow graph for the switch-case construct, which reflects the multi-way decision semantics of the switch-case construct. All *case* branches are on the same level in the control dependence graph which is constructed from the new control flow graph, and there is no control dependence among *case* branching nodes. This approach produces appropriate fitness which evaluates the suitability of the test data accurately. This unified fitness calculation approach provides a much better guidance to evolutionary search. Encouraging results were obtained with the experiments on several large-scale open source programs.

The rest of this paper is organized as follows. Section 2 overviews evolutionary testing. Section 3 introduces the fitness calculation problem for the switch-case construct. Section 4 proposes a unified fitness calculation approach for the switch-case and other constructs. Section 5 describes the tool implementation and experiments on some representative open source programs. Section 6 discusses related work and Section 7 concludes.

2. EVOLUTIONARY TESTING

Evolutionary Testing is a meta-heuristic search technique for the automatic test data generation. ET reformulates the test data generation problem as an evolutionary search problem [6] guided by a fitness function. The given test object inputs are encoded as individuals in the evolutionary algorithm, and the search space is the inputs domain of the test object. The fitness function is tailored to search test data for the type of test that is being undertaken. The purpose of fitness function is to guide the evolutionary search into promising and unevaluated areas of the search space. The goal of the search is to find the global minimum of the fitness function, i.e. zero.

$$fitness = approximation\ level\ +\ normalize(dis) \quad (1)$$

$$normalize(dis) = 1 - 1.001^{-dis} \quad (2)$$

In structural testing, previous work [7] has argued that the fitness function is as illustrated in formula (1), where *approximation level* (also referred to as approach level) measures how close in structural terms an individual is to reaching the target, and branch distance *dis* indicates how close an individual is to reaching the alternative branch that should have been taken. Formula (2) [14] is applied to map *dis* into the range $[0, 1)$.

Approximation level is defined as the number of the target branch's control dependent nodes that were not encountered in the path executed by the individual [7, 10]. Control dependent nodes refer to the branching nodes on which the target is control dependent. The control dependence information is derived from the control flow graph of the program. For branches leaving a control dependent node, a critical branch is the edge which leads the execution path away from the target [13]. Once a critical branch is taken, it is impossible to execute the target. There is only one critical branch per control dependent node in the existing control dependence graph, so the relationship between critical branches and control dependent nodes is deemed to be one-for-one. Thus the approximation level is in fact, calculated by subtracting one from the number of critical branches lying between the node from which the individual diverged away from the target and the target itself [12]. Figure 1 is a slice of a control flow graph, where the search goal is the execution of node 6. The classification of branches is illustrated in Figure 1. The true branch from node 1, the false branches from node 4 and node 5 are all critical branches. Individuals driving the control flow down the false branch at node 5 receive an approximation level of zero, while individuals taking the false branch at node 4 receive an approximation level of one, and individuals taking the true branch at node 1 receive an approximation level of two.

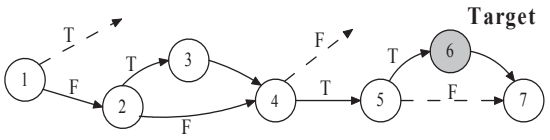

Figure 1: A control flow graph slice with node 6 as the target. Critical branches are shown with dashed lines.

At the point where the execution diverges away from the target down a critical branch, branch distance is calculated. Branch distance measures how close the predicate of the alternative branch is to be true. For example, if a branching condition $(x == y)$ needs to be evaluated as true, branch distance may be defined as $dis = |x - y| + K$ (K refers to a positive failure constant, K is designated as 1 in this work) [9]; if the branching condition $(x == y)$ needs to be evaluated as false, branch distance is defined as $dis = K$, namely 1 in this work.

3. FITNESS CALCULATION PROBLEM FOR THE SWITCH-CASE CONSTRUCT

The switch-case construct is a multi-way decision construct, and it is often used as a substitute for the nested if-else structure when testing a value against a set of constants. The constants following the *case* keywords must be distinct. Each *case* branch may or may not have a *break* (or *return*) statement. First we will consider the simplest switch-case construct with all *case* branches having a *break* (or *return*) statement except the *default* branch. A simple switch-case construct can be seen in Example 1.

Example 1. **A simple switch-case construct**

```
/* pre-code */
    switch(x){
1:    case 0:
2:      value = 1.0; break;
3:    case 2:
4:      value = 2.5; break;
5:    case 5:
6:      value = 1.5; break;
7:    case 1:
8:      value = 2.0; break;/* Target */
9:    default: value = 0.5; }
```

3.1 Nested If-Else Structure Based Fitness Calculation Approach

The target may be embraced in a *case* branch while generating branch adequate test sets. When the control flow diverges away from the desired *case* branch down other *case* branches, the target remains unexecuted, and it is then necessary to calculate the fitness. The switch-case construct is considered equivalent to a nested if-else structure with respect to the control flow. Example 2 is the nested if-else structure with the same control flow graph as Example 1. Figure 2 shows the control dependence graph of Example 1, which is constructed from the control flow graph of Example 1 [11]. As indicated in Figure 2, each *case* branch is control dependent on the *case* branching node it leaves and all *case* branching nodes situated before it. For example, branch 6 is control dependent on branching nodes 1, 3 and 5, while branch 8 is control dependent on branching nodes 1, 3, 5 and 7.

Considering a target *case* branch, the target will be missed when the execution diverges away down any other *case* branch. Individuals driving the control flow down different *case* branches will

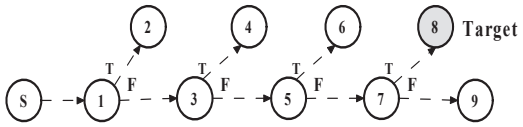

Figure 2: The control dependence graph of Example 1

receive different approximation levels, according to the definition of approximation level. In Example 1, the search goal is the execution of *case* branch 8. Given $x_1 = 0$, the control flow takes the true branch at node 1. The calculated approximation level is 3 and branch distance is 1. For $x_2 = 5$, the calculated approximation level is 1 and branch distance is also 1. The fitness values of x_1 and x_2 only differ in approximation level. Under the principle that better test data always attains smaller fitness, x_2 is assessed better than x_1 because of the produced fitness values. In Example 1, it is easy to find that, compared with x_2, x_1 is actually closer to the target with the branching condition $x == 1$. This is contrary to the result derived using the traditional approach. When the target is embraced in the switch-case construct, individuals cannot be assessed accurately with the existing fitness calculation approach.

Example 2. **The nested if-else structure of Example 1**

```
/* pre-code */
1:    if(x == 0)
2:      { value = 1.0; }
3:    else if(x == 2)
4:      { value = 2.5; }
5:    else if(x == 5)
6:      { value = 1.5; }
7:    else if(x == 1)
8:      { value = 2.0;/* Target */ }
9:    else { value = 0.5; }
```

Approximation level is calculated on the basis of the control dependence information of the target. Thus inappropriate fitness may result from the control dependence graph of the switch-case construct. In the existing control dependence graph, each *case* branch is control dependent on the *case* branching node it leaves and all *case* branching nodes situated before it. In a switch-case construct, no two *case* branches have the same branching condition. The conditions for the *case* branches cannot be simultaneously true. In the control dependency graph, once a certain *case* branching node is evaluated as true, all the other *case* branching nodes will be definitely evaluated as false. In Figure 2, if the true branch from branching node 1 is taken, other *case* branching nodes (e.g. branching nodes 3, 5 and 7) are certainly evaluated as false. So all case branches are mutually exclusive and there exists no control dependence among *case* branching nodes. The control dependence of the target on other *case* branching nodes is meaningless and creates inappropriate approximation levels as demonstrated.

However, the control dependence graph is constructed from the control flow graph. The existing control flow graph of the switch-case construct merely indicates the order of the comparison of between the value of the expression after the *switch* keyword and the constant after each *case* keyword. It does not reflect the fact that all *case* branches are mutually exclusive in semantics just as the true branch and the false branch from the *if* branching node are. It is the control flow graph that results in inaccurate control dependence information and inappropriate assessment of individuals.

Consequently, based on this inaccurate control flow graph, the existing fitness calculation approach misleads the evolutionary search. However, to date there has been no work addressing the fitness calculation problem for the switch-case construct.

3.2 A Testability Transformation Approach

As the previous discussion indicates, the control dependence of the target on other *case* branching nodes results in inappropriate approximation level. Meanwhile, if it is possible to place other *case* branches causing a miss of the target on the same level of the control dependence graph, individuals taking different *case* branches will receive the same approximation level. By adopting a testability transformation, a source-to-source program transformation seeking to improve the performance of test data generation techniques, this goal can be achieved [19]. The transformation process needs only to preserve the set of test adequate inputs, but not the meaning of a program. In other words, the transformed program must be guaranteed to execute the desired branch under the same initial conditions [19]. Testability transformations have been applied to many programs for branch coverage [16, 17]. Example 1 can be transformed into an if-else structure illustrated in Example 3 using a testability transformation.

Example 3. **A transformed version of Example 1**

```
/* pre-code */
1:    if(x == 1)
2:      { value = 2.0; /* Target*/}
      else{
3:      if(x == 0)
4:        { value = 1.0; }
5:      if(x == 2)
6:        { value = 2.5; }
7:      if(x == 5)
8:        { value = 1.5; }
9:      if(x != 0 && x!= 2 && x!= 5)
10:       { value = 0.5; /* corresponding to default branch */ }  }
```

In Example 3, the target branch (branch 2) is only control dependent on branching node 1. When the test data *x* drives the control flow away from the target down the false branch at branching node 1, regardless of which branch is taken among branches 4, 6, 8 and 10, the approximation level will be zero and branch distance will be $|x - 1| + 1$. Since the fitness is to be minimized, the closer the individual is to 1, the better the individual will be evaluated. The fitness values of individuals assess their distance to the desired test data accurately based on the transformed version of the switch-case construct.

The testability transformation approach solves the fitness calculation problem for the switch-case construct. Nonetheless, the drawbacks described below demonstrate that it is costly and complicated to transform a switch-case construct into the if-else structure to preserve the set of test adequate inputs. First, testability transformation for the switch-case construct is goal-oriented, which means the transformation is specific to a testing target. Whereas, in structural testing, especially branch coverage testing, each branch should be covered by the test criteria. Hence a further testability transformation of a program is required for each new target branch, resulting in additional cost, e.g. repeated compilation and linking. The cost will grow with the scale of the program under test. Second, a more troublesome problem occurs when *break* statements are omitted in the body of some *case* branches. The program is poorly structured and the control flow of the program becomes much more complicated in this situation. In Example 4 given in Section 4.1, the *break* statement is omitted in branch 3, and when the execution proceeds to branch 3, the target branch (branch 4) can still be reached. The transformation algorithm becomes intricate and difficult to construct in order to preserve the set of test adequate inputs. Finally, it is more difficult to construct the transformation algorithm in order to preserve branch adequate test sets, when there exists a definition of a variable after the *switch* keyword in a *case* branch, e.g., a definition of variable *x* in branch 3 in Exam-

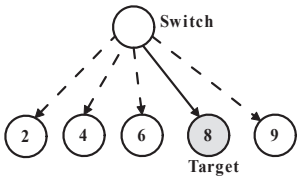

Figure 3: Flattened control flow graph of Example 1

ple 4. From the above discussion, it can be seen that it is necessary to seek a new approach to calculate fitness for the switch-case construct.

4. ACB BASED FITNESS CALCULATION APPROACH

As discussed in Section 3.1, all *case* branching conditions in the switch-case construct are mutually exclusive in semantics, just as the true branch and the false branch from the *if* branching node are. From this perspective, a new control flow graph for the switch-case construct is presented, namely Flattened Control Flow Graph (FCFG). In the traditional control flow graph, each branching node has at most two successors [11], while in FCFG, the switch branching node is allowed to have more than two successors. Figure 3 illustrates the FCFG of Example 1. As is shown, all *case* branches are successors of the switch node. Moreover, from the FCFG, a Flattened Control Dependence Graph (FCDG) can be constructed naturally. The FCDG of Example 1 appears the same as the FCFG shown in Figure 3. In FCDG, each *case* branch is control dependent on the switch branching node, and there exists no control dependence among *case* branches. As for the target *case* branch, when the control flow is driven down other *case* branches by different individuals, resulting in the target being missed, those individuals will receive the same approximation level.

4.1 Alternative Critical Branches

As shown in Figure 3, all *case* branches leave the switch branching node. For a target *case* branch, more than one *case* branch can lead to the target being missed. All *case* branches causing the target to be missed should be treated as critical branches according to the definition of critical branch [13]. That is to say, the number of critical branches leaving the switch branching node is often greater than one with respect to a target. In Example 1, if branch 2, 4, 6 or 9 is taken, the execution will diverge away from the target (branch 8), so all four branches should be considered critical to branch 8. The switch branching node has more than one critical branch, while other branching nodes (i.e. *if*, *while*) have only single critical branch per branching node for the target. The existing fitness calculation approach is based on single critical branch per branching node so it is not effective for programs with the switch-case construct. It is necessary to develop a new concept to extend the traditional critical branch.

The essence of the testability transformation approach is to classify all *case* branches into two categories. Any *case* branch causing the target to be missed falls into the first category and others form the second category. In Example 3, branch 4, 6, 8 and 10 (corresponding to branch 2, 4, 6 and 9 in Example 1) belong to the first category, while branch 2 (corresponding to branch 8 in Example 1) falls into the second category.

Inspired by the testability transformation approach, we classify *case* branches into two categories in the same way, without making any transformation. Once a branch in the first category is taken, the target will clearly be missed. In other words, all *case* branches in the first category are critical branches with respect to the target and they constitute a set of critical branches. We propose that Alternative Critical Branches (ACB) represents this set of critical branches. Considering a branch n, ACB(n) denotes all the branches which are from the same branching node that branch n leaves and can lead the execution path away from branch n. When branch n leaves a *switch* branching node, ACB(n) consists of all *case* branches that can lead to a miss of branch n. For a *two-way* branching node, ACB(n) has only one element that is the alternative branch of branch n, namely a critical branch in the literature [13]. The formal definition of ACB is given in Definition 1.

Definition 1. **Alternative Critical Branches (ACB)**

$$\text{ACB(n)} = \begin{cases} \{\text{all case branches leading to a miss of branch n}\} & \text{n leaving a } \textit{switch} \text{ branching node} \\ \{\text{the alternative branch of branch n}\} & \text{n leaving a } \textit{two-way} \text{ branching node} \end{cases}$$

There is only one ACB per branching node, relative to the traditional single critical branch per branching node. The concept of ACB is a natural extension to the traditional single critical branch. Individuals driving the control flow down any branch in ACB(n) obtain the same approximation level value.

For the switch-case construct with a *break* (or *return*) statement in each *case* branch, any branch except branch n, if taken, will lead to a miss of branch n, so all *case* branches except branch n compose ACB(n). In Example 1, branches 2, 4, 6 and 9 compose ACB(8). The *break* statement may be purposely omitted by programmers. In Example 4, the execution of branch 3 does not cause the target to be missed. Therefore, branch 3 should not be included in ACB(4).

Example 4. **A switch-case construct with break statements omitted in a case branch**

```
/* pre-code */
    switch(x){
1:    case 0: z+=2;break;
2:    case 3: value = 1.5;break;
3:    case 2: y++; x+=2;
4:    case 1: value = 2.0;break; /* Target */
5:    case 4: value = 2.5;break;
6:    default: value = 0.5; }
```

Given a branch n, ACB(n) can be constructed using Algorithm 1. If branch n leaves a two-way branching node, ACB(n) consists of only one element, namely the alternative branch of branch n (lines 16-17). If branch n leaves a switch branching node, ACB(n) is constructed on the basis of the *case* branches information. The *case* branches information (e.g. branch ID, the existence of a *break*/*return* statement) is obtained by static analysis and stored in the data structure *caseArray*. In Algorithm 1, all *case* branches situated between the first *case* branch and the closest preceding *case* branch with a break statement of branch n are treated as elements of ACB(n) (lines 2-11). Then all *case* branches succeeding branch n are also included in ACB(n) (lines 12-15). Consider Example 4 with the target being the execution of branch 4. Using Algorithm 1, ACB(4) = {branch 1, branch 2, branch 5, branch 6}. When the control flow takes any branch in ACB(4), the target is missed.

Algorithm 1: Alternative Critical Branches (ACB) Construction Algorithm

Input: *n* : a branch
caseArray: array of all *case* branches with the same sequence as that in the switch-case construct

```
    cdg: the Control Dependency Graph of the program
Output: ACB(n)
Initial: S= φ , ACB= φ
    S is a temporary set
1:  if n leaves a switch branching node in cdg then
2:    for each case branch b in the caseArray
3:    begin
4:      S=S ⋃ {b}
5:      if n==b then
6:        break
7:      endif
8:      if b contains a break or return statement then
9:        ACB=ACB ⋃ S, S= φ
10:     endif
11:   end
12:   for each case branch b behind n in the caseArray
13:   begin
14:     ACB=ACB ⋃ {b}
15:   end
16: else
17:   ACB= { the alternative branch of n in cdg }
18: endif
19: return ACB
```

4.2 A Unified Fitness Calculation Approach

As far as the nested structure is concerned, there is generally more than one control dependent node for the target. Each control dependent node has a corresponding ACB, so there often exist several ACBs for the target. The target is definitely missed when the execution diverges away down any branch in those ACBs, so they should be recognized in the calculation of approximation level. All ACBs with respect to the target also form a set. To distinguish this from the concept of ACB, we propose the notion of Critical Branches Set (CBS) in Definition 2.

Definition 2. **Critical Branches Set (CBS)**: For branch n, CBS(n) is the set of all the corresponding ACBs of branch n's control dependent nodes. When any branch in any ACB contained in CBS(n) is taken, there is no chance to cover branch n.

Example 5 has three levels of nesting, with two levels of nested switch-case structures, and the corresponding FCDG is shown in Figure 4. When any branch in ACB(9) is taken, the target (branch 9) will be missed. Furthermore, branch 9 is control dependent on branch 6 and branch 2. If the control flow diverges away down any branch in ACB(6) or ACB(2), the target will also be missed. So ACB(6) and ACB(2) should also be included, and thus CBS(9) = {ACB(2), ACB(6), ACB(9)}, where ACB(2) = {branch 12}, ACB(6) = {branch 3, branch 4, branch 11}, and ACB(9)={branch 7, branch 10}. When control flow is driven down any branch in any ACB contained in CBS(9), the target will be missed.

Example 5. **Example with nested structure**

```
/* pre-code */
1:    if(x > 0){
2:        switch(y){
3:        case 1:
4:        case 8: value = 1.0;break;
5:        case 6: x+=2;
6:        case 2: switch(c){
7:            case 'a': value = 2.0;break;
8:            case 'k': x++;
9:            case 'c': value = 3.0; break; /* Target */
10:             default: value = 1.5
              }
11:       default: value = 0.5;
          } }
12:   else  { /*some code*/ }
```

With the proposed concepts of ACB and CBS, the approximation level in formula (1) is calculated by subtracting one from the number of ACBs lying between the node from which the individual diverged away from the target, and the target itself.

Meanwhile, branch distance in formula (1) is calculated when the execution diverges away down any branch in any ACB contained in the target's CBS. When the execution diverges away at a switch branching node, the branch distance is $|expr - C| + 1$, where $expr$ is the value of the expression after the *switch* keyword, and C is the constant for the desired *case* branch, which can be obtained by the static analysis of the program. When the execution diverges away at other branching nodes, branch distance is calculated using the traditional method [9].

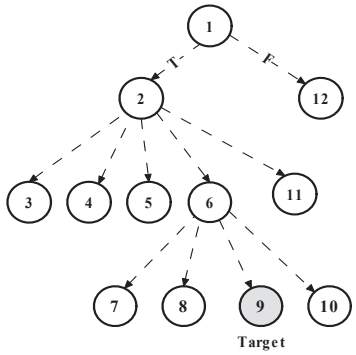

Figure 4: Flattened control dependence graph of Example 5

Consider Example 5. Given test data (x = 2, y = 2, c = 'a'), the execution will diverge away down branch 7 which is included in ACB(9). Using our unified fitness calculation approach, approximation level is 0 and branch distance is $|'a' - 'c'| + 1 = 3$. For test data (x = 2, y = 1, c = 'a'), approximation level is 1 and branch distance is $|1 - 2| + 1 = 2$.

5. EMPIRICAL STUDIES

5.1 Tool implementation

The high level architecture of our tool is shown in Figure 5. As introduced in Section 4.2, approximation level relies on the CBS of the target, and branch distance is related to variable values. Therefore, approximation level can be calculated by comparison of the execution trace with the CBS of the target. Once the execution encounters any branch in any ACB which is in the target branch's CBS, approximation level is calculated by subtracting one from the number of ACBs in CBS lying between the node from which the test data diverged away from the target and the target itself. To calculate approximation level in this way, execution tracing code is instrumented into the program under test. Branch distance can be calculated by instrumented code using the equation given in Section 4.2. The tool consists of two parts, Static Analysis module and Evolutionary Search module.

1. Static Analysis. We employ *ANTLR* (an open source compiler generator) [18] to generate a *C Parser* to extract the Abstract Syntax Tree (AST) of the source code. Then fitness calculation code is instrumented by the *Fitness Instrumentor* while traversing the AST. Control dependency information and *case* branches information such as the constant for each case branch are obtained through the *Program Analyzer* for the calculation of fitness.

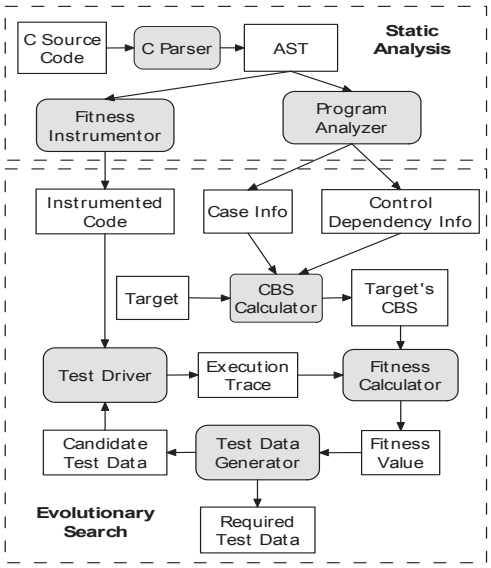

Figure 5: The high level architecture of our tool

2. Evolutionary Search. The target's CBS is calculated by *CBS Calculator*, which repeatedly invokes the ACB construction algorithm given in Algorithm 1 based on the control dependence nodes of the target. Evolutionary algorithm is applied to the *Test Data Generator*, which generates candidate test data for the *Test Driver*. The *Test Driver* repeatedly invokes the instrumented program under test . The execution trace is obtained after each run, with which the fitness value is calculated by the *Fitness Calculator*. The search terminates only when the required test data has been found or the maximum number of fitness evaluations has been reached.

5.2 Experiments

The switch-case construct is widely used in industrial programs, and the fitness calculation problem studied here forms a significant proportion of evolutionary testing. To validate our fitness calculation approach for the switch-case construct, experiments were conducted on 6 representative functions, coming from 4 open source projects in current use. Further details can be found in Table 1.

qemu-0.8.2 is an open source machine emulator, which can run an unmodified target operating system in a virtual machine. **gdb-6.6-2** is a widely used debugger, and **xen-3.1.0** is an x86 virtual machine monitor which allows multiple operating systems to share hardware. **a2ps-4.13** is a text to postscript converter. The selection of a test object was decided using the following three criteria: the number of branches embraced in the switch-case constructs, the maximum levels of nesting and the number of code lines. In all examples, the target branch is embraced in a switch-case construct, and the *break* statement may be deliberately omitted by programmers. Nested switch-case structures are also included in our experiments. As shown in Table 1, there are 290 branches embraced in the switch-case constructs in the four test objects. Experiments were conducted on the 290 branches.

For simplicity, ET using the fitness calculation approach based on the nested if-else structure is referred to as ET with nested, while ET using the fitness calculation approach based on ACB is referred to as ET with ACB. In experiments, we also adopted random testing. The random testing algorithm simply constructs 100,000 random inputs for each branch. Experiments were run on an Intel Dual

Table 1: Test Objects Details

Test Object/Function	Lines of Code	No. of Branches	Maximum levels of nesting
qemu-0.8.2			
console_putchar	225	87	7
gdb-6.6-2			
Slot_inst16b_decode	50	22	4
sh3_supply_register	85	32	4
xen-3.1.0			
OPLWriteReg	256	100	5
init_centaur	143	22	3
a2ps-4.13			
ps_escape_char	100	27	3
Total	859	290	

2.0GHz PC with 2Gb of RAM. 200 individuals were used per generation. The crossover probability is set to 0.3, and the mutation probability is set to 0.02. Both ET with nested and ET with ACB terminate after 100,000 fitness evaluations if test data has not been found. All the search algorithms were repeated 100 times for each branch. Success rate, average number of fitness evaluations, and standard deviation were used as metrics to examine the effectiveness of the three approaches.

Experimental results are shown in Table 2. Of the 290 branches given in Table 1, 205 branches (71%) were covered by all search methods, (e.g. *init_centaur* branch 2, *ps_escape_char* branch 21 shown in Table 2), leaving 85 hard-to-cover branches for which ET is required. Those branches are nested within more than one conditional statement. This experimental result is basically consistent with the result of a theoretical and empirical analysis done by Harman et al. [2]. Table 2 lists 25 branches of the 85 branches as examples.

As presented in Table 2, it is obvious that the success rate of ET with ACB approach is much higher than that of the other two approaches. Our fitness calculation approach demonstrates a significant success rate for the switch-case construct. Table 2 further demonstrates that the success rate was improved for several branches (e.g. branches 12, 13, 14, and 34 in *console_putchar*, branch 16 in *Slot_inst16b_decode*) by more than 15% over ET with nested. Moreover, there are three branches (branches 64, 69, and 73) in function *console_putchar* that ET with ACB covered, while ET with nested failed on each occasion in the 100,000 fitness evaluation limit. ET with nested encountered severe difficulties for the three branches due to the existence of three levels of nested switch-case construct. Those branches are indirectly control dependent on the seven *case* branching nodes in the first switch-case structure using ET with nested. The control dependence is meaningless and brings about inappropriate approximation level as discussed in Section 3.1, so the fitness values received by test data are inaccurate in determining their suitability for the coverage of the target and provide misleading guidance to the evolutionary search. With the increase in the levels of nested switch-case construct, the impact is aggravated, resulting in ET with nested's failure in generating test data for those branches. In our approach, for those branches with no control dependence on the seven *case* branching nodes in the FCDG, the fitness can evaluate test data accurately and provide much better guidance, so ET with ACB achieves a much higher success rate.

Additionally, the average number of fitness evaluations required by ET with ACB approach is much smaller than that of ET with nested. Our approach also effectively contributes to the number

Table 2: Experimental Results. AE is the average number of fitness evaluations, SD is the standard deviation

Test Object/Function (Branch ID)	Random Testing Success Rate (AE/SD)	ET with nested Success Rate (AE/SD)	ET with ACB Success Rate (AE/SD)
qemu-0.8.2			
console_putchar(11)	33% (32,679/16,036)	90% (10,605/9,476)	100% (4,890/2,568)
console_putchar(12)	37% (28,679/15,842)	70% (3,014/2,658)	100% (2,236/1,493)
console_putchar(13)	40% (12,679/6,569)	56% (1,797/1,243)	100% (2,030/1,601)
console_putchar(14)	41% (22,132/10,489)	64% (2,546/2,552)	99% (1,686/898)
console_putchar(34)	0%	68% (19,710/8,027)	95% (12,658/6,574)
console_putchar(64)	0%	0%	70% (20,407/11,086)
console_putchar(69)	0%	0%	71% (22,981/12,996)
console_putchar(73)	0%	0%	69% (21,453/12,894)
console_putchar(81)	0%	10% (8,638/8,056)	83% (13,367/7,236)
console_putchar(83)	10% (20,688/10,357)	80% (2,764/2,951)	99% (1,326/924)
gdb-6.6-2			
Slot_inst16b_decode(16)	43% (39,106/23,354)	71% (11,298/3,883)	93% (5,643/3,074)
Slot_inst16b_decode(19)	50% (45,512/28,105)	83% (12,781/6,744)	100% (5,967/3,184)
sh3_supply_register(16)	0%	100% (688/662)	100%(493/281)
sh3_supply_register(29)	0%	100% (943/912)	100% (678/497)
sh3_supply_register(36)	0%	100% (1,075/985)	100% (801/467)
sh3_supply_register(41)	0%	100% (1,100/1,136)	100% (818/541)
xen-3.1.0			
OPLWriteReg(63)	32% (49,204/21,802)	89% (6,172/5,916)	100% (3,506/1,728)
OPLWriteReg(64)	20% (53,871/26,386)	85% (7,764/6,852)	100% (4,327/2,128)
OPLWriteReg(67)	29% (48,772/25,296)	88% (5,762/5,551)	100% (3,239/1,827)
OPLWriteReg(68)	21% (52,924/23,867)	84% (6,806/6,336)	100% (3,801/1,879)
OPLWriteReg(71)	30% (47,978/22,348)	90% (5,802/5,484)	100% (3,106/1,581)
OPLWriteReg(72)	20% (56,034/28,485)	83% (6,512/5,733)	100% (3,685/1,667)
OPLWriteReg(75)	31% (48,867/23,973)	89% (6,201/5,979)	100% (3,461/1,716)
OPLWriteReg(76)	22% (55,046/24,614)	83% (6,891/6,201)	100% (4,077/2,007)
init_centaur(2)	100% (296/187)	100% (1,228/1,170)	100% (1,119/839)
init_centaur(5)	45% (56,691/32,143)	85% (25,112/10,298)	100% (10,838/3,490)
a2ps-4.13			
ps_escape_char(21)	90% (27,860/18,344)	95% (10,282/9,711)	100% (8,018/6,417)

of fitness evaluations required. In many instances, such as with numerous branches in *OPLWriteReg*, ET with nested performs almost 1.5 times as many fitness evaluations as ET with ACB. Furthermore, branches in functions *console_putchar* and *Slot_inst16b_decode* presented in Table 2 are embraced in a switch-case structure which is nested within two outer switch-case constructs, the advantage of our approach becomes more noticeable due to the increase of the levels of nested switch-case construct. ET with ACB generally performs almost half the number of fitness evaluations, compared to ET with nested. There are only two branches (branches 13 and 81 in function *console_putchar*) for which ET with nested performs fewer fitness evaluations. This is because the first generation of ET is randomly generated. When the required test data or data close to the required data is generated in the first generation, then the target will be covered with fewer fitness evaluations, otherwise ET with nested will find difficulty in generating test data due to the misleading guidance provided by the fitness calculation approach based on the nested if-else structure. So ET with nested has much lower success rates and fewer fitness evaluations for the two branches, compared to our approach.

Furthermore, as shown in Table 2, ET with ACB has much smaller standard deviations than ET with nested. ET with nested has more than twice the standard deviation of our approach for most branches, which indicates that our approach performs more stably than ET with nested due to the much better guidance provided by our fitness calculation approach.

In conclusion, the empirical study results provide evidence to support that our approach outperforms the fitness calculation approach based on the nested if-else structure with respect to the branches embraced in the switch-case structure. As a well-designed fitness function is crucial to the effectiveness and efficiency of evolutionary search, there is no misleading control dependence among *case* branching nodes in our approach, then fitness values calculated can evaluate test data accurately, providing more effective guidance to evolutionary search. Our approach has a higher success rate, performs fewer fitness evaluations and is more stable than ET with nested. Additionally, with the increase of the levels of the nested switch-case structure, advantages of our approach are more noticeable.

6. RELATED WORK

Miller and Spooner [20] were the first to dynamically generate floating point test data for paths, applying the numerical maximization technique. Xanthakis et al. [21] were the first authors to apply evolutionary computation to test data generation for the execution of paths. This work has been extended by various authors for structural test data generation[4, 7, 9, 2, 1, 19]. McMinn [15] gave a detailed survey of search based test data generation in 2004. Multi-object branch coverage was first proposed to search based test data generation by Harman et.al. in 2007 [1].

The fitness function is crucial to the performance of the search based test data generation technique. Intensive research into the design and implementation of the fitness function has been undertaken [7, 10, 9, 14, 19]. In 2001 and 2002, Wegener et al. [7] and Baresel et al. [10] introduced approximation level into the fitness function. Fitness function for structural testing has been designed and optimized for specific program constructs [8, 14, 16, 17, 16].

Nevertheless, despite the large body of work, the present paper is the first to investigate the fitness calculation problem for the switch-case construct. It seems that this problem can be handled

by testability transformation, a promising approach to improve the performance of the test data generation technique [19]. Testability transformation has been applied to many program constructs [16, 17] in the evolutionary testing. However, the cost of this application for the switch-case construct may be prohibitive. Our approach is inspired by testability transformation, without making any transformation.

7. CONCLUSIONS AND FUTURE WORK

This paper introduces a unified fitness calculation approach for the switch-case construct on the basis of Alternative Critical Branches. The concept of Alternative Critical Branches is extended from the single critical branch per branching node, enabling multiple critical branches per branching node to be taken into consideration when calculating fitness. An experimental assessment of the performance of our approach, compared with the fitness calculation approach based on the nested if-else structure and random testing, demonstrates the efficiency and effectiveness of our approach.

A programmer may choose to implement a conceptual switch-case construct using a nested if-else structure, similar to that presented in Example 2, which would exhibit the same problem. This type of nested if-else structure can be identified by examining the predicates in the nested conditionals and transformed into the switch-case constructs, then solved by this proposed approach.

Future work will also deal with unstructured programs using the ACB based fitness calculation approach. Branches containing *return* or *break* statements that lead to the target being missed are considered critical and compose some critical branch sets with respect to the target. Therefore, we expect that this approach could also be applied to unstructured programs.

8. ACKNOWLEDGEMENTS

This work is sponsored by National Advanced Research Project of China, and National Advanced Research Funding Project of China.

9. REFERENCES

[1] M. Harman, K. Lakhotia, and P. McMinn. A multi-objective approach to search-based test data generation. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07)*, pp.1098-1105, London, England, 2007

[2] M. Harman and P. McMinn. A Theoretical & Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation. In *International Symposium on Software Testing and Analysis (ISSTA'07)*, to appear, 2007.

[3] K. Derderian, R. Hierons, M. Harman, and Q. Guo. Automated Unique Input Output sequence generation for conformance testing of FSMs. In *The Computer Journal*, 49(3):331-344, 2006.

[4] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085-1110, 2001.

[5] L. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'05)*, pp.1021-1028 , Washington, USA, 2005.

[6] J. Clarke, J. J.Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. In *IEE Proceedings - Software*, 5(1):161-175, June 2003.

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

[8] X. Liu, H. Liu, B. Wang, P. Chen, and X. Cai. A unified fitness function calculation rule for flag conditions to improve evolutionary testing. In *Proceedings of the International Conference on Automated Software Engineering (ASE'05)*, pp.337-341, California, USA, November 2005.

[9] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis (ISSTA'98)*, pp.73-81, Florida, USA, 1998.

[10] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02)*, pp.1329-1336. New York, USA, July 2002.

[11] J. Ferrante, K. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. In *ACM Transactions on Programming Languages and Systems*, 9(3):319-349, 1987.

[12] P. McMinn and M. Holcombe. Hybridizing evolutionary testing with the chaining approach. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'04)*, pp.1363-1374. Seattle, USA, June 2004.

[13] R. Ferguson and B. Korel. The chaining approach for software test data generation. In *ACM Transactions on Software Engineering and Methodology*, 150(3):63-86, January 1996.

[14] P. McMinn and M. Holcombe. Evolutionary testing of state-based programs. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'05)*, pp.1013-1020. Washington DC, USA, June 2005.

[15] P. McMinn. Search-based software test data generation: A survey. In *Software Testing, Verification and Reliability*, pp.105-156, 2004.

[16] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02)*, pp.1351-1358. New York, USA, 2002.

[17] P. McMinn, D. Binkley, and M. Harman. Testability Transformation for Efficient Automated Test Data Search in the Presence of Nesting. In *Proceedings of the Third UK Software Testing Workshop*, pp.165-182, UK, 2005.

[18] T. Parr. ANTLR - ANother Tool for Language Recognition. http://antlr.org/.

[19] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, Andre Baresel, and Marc Roper. Testability Transformation. In *IEEE Transactions on Software Engineering*. 30(1): 3-16, 2004.

[20] W. Miller and D. Spooner. Automatic generation of floating-point test data. In *IEEE Transactions on Software Engineering*. 2(3):223-226, 1976.

[21] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing. In *In International Conference on Software Engineering and its Applications*. pp.625-636, Toulouse, France, 1992.