Automatic generation of test input data for MC/DC test coverage

Zeina Awedikian
Soccer Lab
Ecole Polytechnique de Montreal
Airborne computer software need the approval of the Federal Aviation Administration (FAA).

Software should satisfy all criteria of the RTCA/DO–178B document entitled “Software Considerations in Airborne Systems and Equipment Certification”, that treats system safety assessment.

One of the objectives of DO–178B for level A software is to generate tests that achieve the modified condition / decision coverage (MC/DC) of the software structure.

Project objective:

- Automatically generate test data for the code under test in order to satisfy the MC/DC test criteria.
Structural coverage MC/DC

Modified Condition/Decision coverage:

- Do 178 definitions:
  - Condition: ‘A boolean expression containing no Boolean operators’.
  - Decision: ‘A Boolean expression composed of conditions and zero or more Boolean operators.’

- Each condition should be shown to affect independently the outcome of the decision.
- A minimum of n+1 test cases for a decision with n inputs.

Z = (A or B)

test cases:
(FF) → F,
(TF) → T,
(FT) → T

- All decisions outcome are tested at least once.
- All conditions outcome are also tested at least once: All blocks in the code are tested.
- Most important, for critical systems such as avionic systems:
The tester will be able to locate where exactly the error is, if any.
The purpose is to generate test data for the decision at node 6. System should:

1. Identify the decisions.
2. Generate test data that will traverse the control flow to get to node 6.
3. Then satisfy the MCDC criteria for node 6.

```c
int triangle (int a, int b, int c) {
    int d = a;
    if (a == 1 || (b <= 3 && d > 4)) {
        ++a;
        if (a > b + c) {
            d = b + c;
            if (d > 100)
                d = 100;
        }
    }
    return d;
}
```
In order to automatically generate data to satisfy all MC/DC test criteria for any high level code, we need to implement several modules first:

- Analysing the code
- Generating Branch Distance fitness function for decisions
- Code instrumentation
- Extracting decisions dependencies
- Generating MC/DC test cases
- Using a metaheuristic algorithm to generate test input data
Building the code syntax tree

To Generate the code AST tree
- Use a grammar and a parser to parse the code and build the AST tree (abstract syntax tree)
- In the tree, each node is identified by its type and linked to its children.

```c
int triangle (int a, int b, int c) {
  1  d = a;
  2  if (a == 1 || (b <= 3 && d > 4)) {
  3    a++;  
  4    if (a > b + c) {
  5      d = b + c;
  6      if (d > 100)  
  7        d = 100;
  8    }
  }
  8 return d;
}
```

```
```
Extracting decision structure

- Extracting the decisions from the code
  - Create a visitor that will go through the AST tree of the entire code.
  - Collect only the condition part of the decisions.

- Analysing the structure of this decision
  - Divide each decision into conditions.

- Build the decision structure tree

\[
\text{IfStatement} \quad \text{If} \quad A \quad \text{Equal} \quad 1 \quad \text{OR} \quad B \quad \text{Comparateur} \quad > \quad 4 \quad \text{AND} \quad D \quad \text{Comparateur} \quad < \quad 4 \quad \text{If} \quad A \quad \text{Operator} \quad ++
\]

Diagram:
- If
- A || (B && D)
- IfStatement
- Identifier: a
- Equal
- Identifier: 1
- OR
- Identifier: b
- Comparateur: >
- Identifier: 3
- AND
- Identifier: d
- Comparateur: <
- Identifier: 4
- Identifier: a
- Operator: ++
Project Phases

In order to automatically generate data to satisfy all MC/DC test criteria for any high level code, we need to implement several modules first:

- Analysing the code:
  - Generating Branch Distance fitness function for decisions
- Code instrumentation
- Extracting decisions dependencies
- Generating MC/DC test cases
- Using a metaheuristic algorithm to generate test input data
Our software engineering problem to generate test input data automatically is a search based software engineering problem, guided by a fitness function.

To make sure all pieces of high level code are tested at least once, all decisions (if, else, for, while...) should be validated to true at least once.

If all the test data generated made the decision false, we need to know which data is closer to make the decision = true.

```c
int triangle(a,b,c) {
    d = a;
    if ( a == 1 || (b <= 3 && d > 4))
        a++;
}
```

Test data:
- a = 3, b=2, c=-5 $\Rightarrow$ false
- a = 6, b=7, c=1 $\Rightarrow$ false

How far the current test data is from making the decision true?

Generate a “branch distance” fitness function for each decision.
### Generating branch distance fitness function (2)

**Fonction de fitness de Bottacci:**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Fitness if exp false</th>
<th>Fitness if exp. true</th>
<th>Fitness for IF then branch</th>
<th>Fitness for ELSE branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=b</td>
<td>F=abs(a-b)</td>
<td>F=0</td>
<td>( abs(a-b) )</td>
<td>( a!=b ? k : 0 )</td>
</tr>
<tr>
<td>a≠b</td>
<td>F=K</td>
<td>F=0</td>
<td>( a!=b ? 0 : k )</td>
<td>( a==b ? abs(a-b) : 0 )</td>
</tr>
<tr>
<td>a&lt;b</td>
<td>F=(a-b)+K</td>
<td>F=0</td>
<td>( a&lt;b ? 0 : a-b + k )</td>
<td>( a&lt;b ? a-b + k : 0 )</td>
</tr>
<tr>
<td>a≤b</td>
<td>F=(a-b)</td>
<td>F=0</td>
<td>( a&lt;=b ? 0 : a-b )</td>
<td>( a&lt;=b ? a-b : 0 )</td>
</tr>
<tr>
<td>a&gt;b</td>
<td>F=(b-a)+K</td>
<td>F=0</td>
<td>( a &gt; b ? 0 : a - b + k )</td>
<td>( a &gt; b ? a - b + k : 0 )</td>
</tr>
<tr>
<td>a≥b</td>
<td>F=(b-a)</td>
<td>F=0</td>
<td>( a &gt;= b ? 0 : a-b )</td>
<td>( a &gt;= b ? a -b : 0 )</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>b</td>
<td>F=min(f(a),f(b))</td>
<td>F=0</td>
</tr>
<tr>
<td>a &amp;&amp; b</td>
<td>F=f(a)+f(b)</td>
<td>F=0</td>
<td>fa + fb</td>
<td>min(fa,fb)</td>
</tr>
</tbody>
</table>

**Note:**
- a and b
  - integers, real, an expression, a condition, multiple conditions...
  - For Strings, we are investigating the notion of distance.
Generating branch distance fitness function (3)

In the visit of our decision, we collect 3 types of children.

- **ASTConditionalExpression**  `a && b`, `a || b`
- **ASTRelationalExpressions**:  `a < b`
- **ASTComparator** (relational expressions operators): `<`, `<=`, `>`, `>=`, `==`, `!=`, `+=`, `*=` ...

Results:

```
(cote1^2 == cote2 - 3 || (cote1 += 2) <= 4)
```

```
R0
```

| R0: R1 || R2 | F0 = min(F1,F2) |
|-----------|-------------|
| R1: side1^2 == side2 - 3 | F1 = ((side1^2 == side2 - 3) ? 0 : abs((side1^2)-(side2 - 3)) ) |
| R2: side1 += 2 <= 4 | F2 = (side1 += 2 <= 4) ? 0: ((side1 += 2)-(4)) |
public class triangle{
    ...
    public static int getType(float[] sides){
        ...
        if(side1 < 0 || side2 < 0 || side3 < 0){
            ret = ILLEGAL_ARGUMENTS;
        }
        else{
            int triang = 0;
            if(side1 == side2){
                triang = triang + 1;
            }
            if(side2 == side3){
                triang = triang + 2;
            }
            if(side1 == side3){
                triang = triang + 3;
            }
            if(triang == 0){
                if(side1 + side2 < side3 ||
                    side2 + side3 < side1 ||
                    side1 + side3 < side2){
                    ret = ILLEGAL;
                }
            }
        }
    }
}
In order to automatically generate data to satisfy all MC/DC test criteria for any high level code, we need to implement several modules first:

- Analysing the code
- Generating Branch Distance fitness function for decisions
- Code instrumentation
- Extracting decisions dependencies
- Generating MC/DC test cases
- Using a metaheuristic algorithm to generate test input data
Evaluation the fitness

- Generate the Branch Distance Fitness function for node 6. ✓
- Calculate it for case (3, 2, -5) ?

Problem:
- At node 0: (a,b,c) = (3, 2, -5), d = 3
- At node 3: b++ => b = 3
- At node 5: d = b + c => d = ?
- At node 6: d = ?

Solution:
- Add code to the original file to trace the execution of the program and collect the new values of the variables at each decision for each test datum.
- Instrument the code and produce a trace.

```c
int triangle (int a, int b, int c) {
    0   d = a
    1   if ( a == 1 || (b <= 3 && d > 4)) {
        2     if ( b < c )
        3       b++;  
        4     if ( a > b + c ) {
        5       d = b + c;
        6       if (d > 100)
        7           d = 100;
        8   return d;
    }
}
```
```java
public static void Calculate(int a, float b, float c) {
    if (a == 1 || (b <= 3 && c > 4)) {
        a = b++;
        if (b + 2 != c || b - 4 <= 4) {
            if (b < 0) {
                a++;
            } else {
                a --;
            }
        }
    }
}
```

1. **Code to be instrumented**

2. **Test datum as input**
   
   \( a = 2, b = 3, c = 5 \)

3. **Trace printed**

   <genes>2 3 5</genes>
   <line id='13'><variable name='a'>2</variable></line>
   <line id='13'><variable name='b'>3.0</variable></line>
   <line id='13'><variable name='c'>5.0</variable></line>
   <trace>13</trace>
   <line id='16'><variable name='b'>4.0</variable></line>
   <line id='16'><variable name='c'>5.0</variable></line>
   <trace>16</trace>
   <line id='17'><variable name='b'>4.0</variable></line>
   <trace>-17</trace>
In order to automatically generate data to satisfy all MC/DC test criteria for any high level code, we need to implement several modules first:

- Analysing the code
- Generating Branch Distance fitness function for decisions
- Code instrumentation
- Extracting decisions dependencies
- Generating MC/DC test cases
- Using a metaheuristic algorithm to generate test input data
Extracting decision dependencies

- Branch distance fitness function for 6 for test datum 1 is now computed.

**Problem**
- Did this test datum 1 allowed the execution to arrive to node 6?

**Solution**
- We need to extract the dependencies of node 6 and make sure that they were traversed in the trace file.

**Results**
- We use a module to transform the code to an xml file.
- We extract the dependencies from the xml and write it to an xml output file.

```
<?xml version="1.0"?>
<Dependencies>
  <FILE>Exemple.java</FILE>
  <FUNCTION>
    <NAME>Calculate</NAME>
    <LOC>
      <id>3</id>
      <DEPENDS-ON>
        <CTL>2</CTL>
      </DEPENDS-ON>
    </LOC>
    <LOC>
      <id>4</id>
      <DEPENDS-ON>
        <CTL>2</CTL>
      </DEPENDS-ON>
    </LOC>
    <LOC>
      <id>5</id>
      <DEPENDS-ON>
        <CTL>-4</CTL>
      </DEPENDS-ON>
    </LOC>
    <LOC>
      <id>6</id>
      <DEPENDS-ON>
        <CTL>-4</CTL>
      </DEPENDS-ON>
    </LOC>
    <LOC>
      <id>7</id>
      <DEPENDS-ON>
        <CTL>6</CTL>
      </DEPENDS-ON>
    </LOC>
  </FUNCTION>
</Dependencies>
```
<Dependencies><FILE>Exemple.java</FILE><FUNCTION><NAME>Calculate</NAME>
...<LOC><id>6</id><DEPENDS-ON><CTL>-4</CTL><CTL>2</CTL></DEPENDS-ON></LOC></FUNCTION></Dependencies>

<genes>2 3 5</genes>
<line id='2'><variable name='a'>2</variable></line>
<line id='2'><variable name='b'>3.0</variable></line>
<line id='2'><variable name='c'>5.0</variable></line>
(trace)2
<line id='4'><variable name='c'>5.0</variable></line>
 trace>2

<genes>2 10 4</genes>
<line id='2'><variable name='a'>2</variable></line>
<line id='2'><variable name='b'>3.0</variable></line>
<line id='2'><variable name='c'>5.0</variable></line>
<trace>2
<line id='4'><variable name='c'>5.0</variable></line>
 trace>4
<line id='6'><variable name='d'>3</variable></line>

Dependencies fitness = 2

Dependencies fitness = 1

Dependencies fitness = 0
In order to automatically generate data to satisfy all MC/DC test criteria for any high level code, we need to implement several modules first:

- Analysing the code:
- Generating Branch Distance fitness function for decisions
- Code instrumentation
- Extracting decisions dependencies
- Generating MC/DC test cases
- Using a metaheuristic algorithm to generate test input data
Generating MCDC

Pseudo-algorithm to generate MCDC test cases for a decision structure.

- Write a small grammar to interpret the input structure
- Write a parser that will create an AST graph
- Extract all variables from the expression
- \( N = \) number of variables
- Create a truth table \( TT \) of size \((N+1,2^N)\)
- Populate the truth table rows:
  - for each column, alternate true and false each \(2^{\text{column}}\)
- For each truth table row
  - Reset graph nodes values
  - Give the leaf variables the value in the current row
  - Evaluate the values of the entire graph nodes
  - Update the truth table output column

\[
\begin{array}{ccc}
A & B & C \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & \\
0 & 1 & 1 & \\
1 & 0 & 0 & \\
1 & 0 & 1 & \\
1 & 1 & 0 & \\
1 & 1 & 1 & \\
\end{array}
\]

\( V = A, B, C \)

\( N = 3 \)
Generating MCDC (2)

For each variable
- Create a set
- Search for the pair of rows where the value of this variable only changes, and the rest of the variables values are the same
- Compare output of these 2 rows
- If the outputs are opposite, add the 2 rows to the variable set

The test sets:
(A, 1->false, 5->true)  
(A, 0->false, 4->true)  
(A, 2->false, 6->true)  
(B, 1->false, 3->true)  
(C, 2->false, 3->true)

The table:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The output of rows 1 and 2 are opposite, so they are added to the variable set.
public class triangle {
    ...
    public static int getType( float[] sides )
    {
        ...
        37   if (side1 < 0 || side2 < 0 || side3 < 0)
        38       ret = ILLEGAL_ARGUMENTS;
        39     else {
            40       int triang = 0;
            41       if (side1 == side2)
            42           triang = triang + 1;
            43     }
            44       if (side2 == side3)
            45           triang = triang + 2;
            46     }
            47       if (side1 == side3)
            ...
                48           triang = triang + 3;
                ...
        }
        49   if (triang == 0)
        50       if (side1 + side2 < side3 || side2 + side3 < side1 || side1 + side3 < side2)
        51           ret = ILLEGAL;
        ...
    }
    ...

In order to automatically generate data to satisfy all MC/DC test criteria for any high level code, we need to implement several modules first:

- Analysing the code
- Generating Branch Distance fitness function for decisions
- Code instrumentation
- Extracting decisions dependencies
- Generating MC/DC test cases
- Using a metaheuristic algorithm to generate test input data
Search based software problem

- At this point, we are ready to start automatically generating the test input data.

- This will require a search of inputs, generated from the input parameter space (real parameters, integers parameter...).

- Many software engineering problems can be represented as search based software engineering problem, SBSE.

- In SBSE, we apply search techniques to search large search spaces, guided by a fitness function.

- In SBSE, researchers use metaheuristic algorithms such as genetic algorithms, simulated annealing, taboo search.
We used a genetic algorithm.

Insertion:
- The current generation is feed into the algorithm.
- At time 0, an initial random generation is created.

Fitness evaluation:
- The algorithm evaluates the **fitness function** for each individual of the current generation.
- Fitness function of one of the individuals is equal to 0? If yes, it returns the individual as a test datum for this test case.

End?
- All test cases are achieved?
- N is achieved? note that the decision couldn’t be tested.

Selection:
- The algorithm selects n individuals with the best fitness function. They will be used as parents of the next generation.
- \( \text{Generation}(i+1) = [\text{mutation}+\text{crossover}] \) of (Generation (i))

Recombination:
- It is called crossover in the case of the genetic algorithm: Crossing between individuals of the parents.

Mutation:
- Modify the value of 1 individual of the current parents.
Run instrumented code

Module: Code instrumentation

Collect information

Covered Decision points

Dependencies collected

Dependencies Fitness()

Target reached?

Bottacci Fitness() Of first diverging node

Module: Bottacci fitness generation

MCDC Fitness() Of the target decision

Module: MCDC test sets

Normalized Distance fitness

Add()

no yes

F =0

no yes

Mark MCDC test as done save the test datum

N max?

All MCDC tests are done? Or N max ?

Selection

Crossover

Mutation

Current population

Variables Values at nodes

Input code

Module: Dependencies extraction

no yes

yes

Stop
public static int getType( float[] sides )
{
...
34.  if (sides.length != 3) {
35.    ret = ILLEGAL_ARGUMENTS;
36.  } else {
37.    if (side1 < 0 || side2 < 0 || side3 < 0) {
38.      ret = ILLEGAL_ARGUMENTS;
39.    } else {
40.      int triang = 0;
41.      if (side1 == side2) {
42.        triang = triang + 1;
43.      }
44.      if (side2 == side3) {
45.        triang = triang + 2;
46.      }
47.      if (side1 == side3) {
48.        triang = triang + 3;
49.      }
50.      if (triang == 0) {
51.        if (side1 + side2 < side3 || side2 + side3 < side1
52.            || side1 + side3 < side2) {
53.          ret = ILLEGAL;
54.        } else {
55.          ret = SCALENE;
56.        }
57.      }
58.      } else {
59.        if (triang > 3) {
60.          ret = EQUILATERAL;
61.        } else {
62.          if (triang == 1 && side1 + side2 > side3) {
63.            ret = ISOCELES;
64.          } else {
65.            if (triang == 2 && side2 + side3 > side1) {
66.              ret = ISOCELES;
67.            } else {
68.              if (triang == 3 && side1 + side3 > side2) {
69.                ret = ISOCELES;
70.              } else {
71.                ret = ILLEGAL;
72.              }
73.            }
74.        }
75.      }
76.  ...

Results

For N = 20, n = 10

If at 43
Test set: false  Test data: 10.0 97.0 10.0 Test output: false
Test set: true  Test data: 84.0 10.0 10.0 Test output: true
1 generations were required.

Test set: false  Test data: 50.0 84.0 51.0 Test output: false
Test set: true  Test data: 2.0 80.0 80.0 Test output: true
1 generations were required.

If at 53
The algorithm wasn't able to generate the test data for 1 tests.
Test set: false  Test data: 85.0 35.0 85.0 Test output: false
20 generations were required.

Test set: false  Test data: 34.0 73.0 34.0 Test output: false
Test set: true  Test data: 34.0 34.0 34.0 Test output: true
20 generations were required.

The algorithm wasn't able to generate the test data for 1 tests.
Test set: false  Test data: 72.0 51.0 51.0 Test output: false
20 generations were required.

Test set: false  Test data: 75.0 75.0 21.0 Test output: false
Test set: true  Test data: 75.0 75.0 75.0 Test output: true
7 generations were required.
public static int getType( float[] sides )
{
...  
if (sides.length != 3) {
    ret = ILLEGAL_ARGUMENTS;
} else {
    if (side1 < 0 || side2 < 0 || side3 < 0) {
        ret = ILLEGAL_ARGUMENTS;
    } else {
        int triang = 0;
        if (side1 == side2) {
            triang = triang + 1;
        } if (side2 == side3) {
            triang = triang + 2;
        } if (side1 == side3) {
            triang = triang + 3;
        } if (triang == 0) {
            if (side1 + side2 < side3 || side2 + side3 < side1 || side1 + side3 < side2) {
                ret = ILLEGAL;
            } else {
                ret = SCALENE;
            }
        } else {
            if (triang > 3) {
                ret = EQUILATERAL;
            } else {
                if (triang == 1 && side1 + side2 > side3) {
                    ret = ISOCELES;
                } else {
                    if (triang == 2 && side2 + side3 > side1) {
                        ret = ISOCELES;
                    } else {
                        if (triang == 3 && side1 + side3 > side2) {
                            ret = ISOCELES;
                        } else {
                            ret = ILLEGAL;
                        }
                    }
                }
            }
        }
    }
}
Conclusion

- Running experiments on the same and on different input codes.
- Tune the number of generations and individuals per generation and run experiments.
- Compare with a random generator of input data to prove the effectiveness of the metaheuristic algorithm for our search.
Thank you

For further information:
zeina.awedikian@polymtl.ca
Soccer Lab
Ecole Polytechnique de Montreal
http://web.soccerlab.polymtl.ca/
References

- Bottacci L. « Predicate Expression Cost Function to Guide Evolutionary Search for Test Data ». Hull University, U.K.
- Hayhursts K. et al. (Mai 2001). « A practical Turioal on Modified Condition / Decision Coverage ». NASA.