Automated Test Generation using CBMC

Rui Gonçalo

CROSS Project
Computer Science Department
University of Minho

December 2012
Summary

1 Software Testing
2 Coverage
3 Automated Test Generation
4 Model Checking
5 CBMC
6 Goals
7 Conclusion
Software Testing \textsuperscript{[1]}

“Observation of a program in execution under controlled conditions”

John Rushby in *Automated Test Generation and Verified Software*
Software Testing

“controlled conditions”

Assignment to the input variables

Allows the tester to verify the behavior of the program
Software Testing

Assignment to the input variables

↓

Test cases
Example of a test case

Test case 1:
(x = 0, y = 0)

Test case 2:
(x = 4, y = 0)

```c
int func (int x, int y)
{
  int a = 0;
  if (x > 3 || y == 1)
    a = x + y;
  else
    if (x == y)
      a = x;
  a++;
  return a;
}
```

Test case 1:
(a = 1)

Test case 1:
(a = 5)
Test Generation

Generation of test cases

Remains a largely manual process in software industry

Entails high costs and time consuming.
Automated Test Generation

A process able to generate test cases in an automatic way is mandatory, to decrease the efforts of the testing phase.

How many test cases?
Coverage

Test coverage measures the percentage of source code points that a testing process reaches.

↓

Which source code points?
Coverage [2]

Depending on the source code points:

A. Statement Coverage
B. Decision Coverage
C. Condition Coverage
D. Decision/Condition Coverage
E. Modified Condition/Decision Coverage
Coverage

if ( a == 0 && b > 3 )

Condition Condition

Decision
**Statement Coverage**

Every statement has been invoked at least once.

S#1 if (x > 1 && y == 0)
S#2 a = x + y;
S#3 if (x == 2 || y > 1)
S#4 b = x - y;

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>S#1</th>
<th>S#2</th>
<th>S#3</th>
<th>S#4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Statement Coverage

If the programmer used the **or** operator, in the first decision, by mistake, the test case would not notice!

S#1 if (x > 1 || y == 0)
S#2 a = x + y;
S#3 if (x == 2 || y > 1)
S#4 b = x - y;

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>S#1</th>
<th>S#2</th>
<th>S#3</th>
<th>S#4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

✓
Decision Coverage

Every **decision** has taken all possible outcomes at least once.

```c
if (x == 2 || y > 1)
a = x + y;
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>TRUE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
Decision Coverage

The effect of the second condition is not tested!

```
if (x == 2 || y > 1)
a = x + y;
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>Decision</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>TRUE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
Condition Coverage

Every condition has taken all possible outcomes at least once.

```
if (x == 2 || y > 1)
a = x + y;
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Cond#1</th>
<th>Cond#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
Automated Test Generation using CBMC

### Condition Coverage

The decision is always TRUE!

```java
if (x == 2 || y > 1)
a = x + y;
```

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>Cond#1</th>
<th>Cond#2</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
**Condition/Decision Coverage**

Every **condition** and **decision** have taken all possible outcomes at least once.

```c
if (x == 2 || y > 1)
    a = x + y;
```

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>Cond#1</th>
<th>Cond#2</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
Condition/Decision Coverage

The independent effect of the conditions is not tested!

```
if (x == 2 || y > 1)  
a = x + y;

if (x == 2 || y > 1)  
a = x + y;

if (x == 2 && y > 1)  
a = x + y;
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Cond#1</th>
<th>Cond#2</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
Modified Condition/Decision Coverage

Every condition in a decision must be shown to independently affect the decision’s outcome.

```plaintext
if (x == 2 || y > 1) a = x + y;
```

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>Cond#1</th>
<th>Cond#2</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
Modified Condition/Decision Coverage

The number of test cases must be at least \( n + 1 \), where \( n \) is the number of variables in the decision

if (x == 2 || y > 1)  
a = x + y;

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Cond#1</th>
<th>Cond#2</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
Modified Condition/Decision Coverage

The standard DO-178B\textsuperscript{1} “Software Considerations in Airborne Systems and Equipment Certifications” requires:

Level A  MC/DC
Level B  Decision Coverage
Level C  Statement Coverage

Automated Test Generation

How?
Bounded Model Checking

Model Checking:

Given a model $M$ of a system and a property $P$:

- if $M \models P$ (M models P), $P$ holds in $M$, i. e. the system functions according to $P$.
- if $M \not\models P$ (M doesn’t model $P$), $P$ doesn’t hold in $M$, and a counterexample is produced, i. e. an execution of the system that does not satisfy $P$.
Bounded Model Checking

Bounded Model Checking:

Given a model $M$ of a system, a property $P$ and a bound $k$ (>0):

- Encode all executions of $M$ of length $k$ into a formula $M_k$
- Encode all executions of $M$ of length $k$ that violate $P$ into $\neg P_k$

- if $(M_k \land \neg P_k)$ is unsatisfiable then $P$ holds in $M$ of length $k$
- if $(M_k \land \neg P_k)$ is satisfiable then $P$ doesn’t hold in $M$ of length $k$, and a counterexample is produced
Conjunctive Normal Form

The formula \((M_k \land \neg P_k)\) is passed to a SAT solver in Conjunctive Normal Form (CNF).

\[ \downarrow \]

How to translate C programs into CNF?
Conjunctive Normal Form

C programs into CNF:

1º - Unwinding loops

```c
int func(int a) {
    int r = 0, i = 0;
    while (i < max) {
        a++;
        assert(a != 0);
        r = max + (r / a);
        i++;
    }
    r = r * 2;
    return r;
}
```

```c
int func(int a) {
    int r = 0, i = 0;
    if (i < max) {
        a++;
        assert(a != 0);
        r = max + (r / a);
        i++;
    }
    r = r * 2;
    return r;
}
```
**Conjunctive Normal Form**

C programs into CNF:

2º - Static Single Assignment Form

```c
int func(int a) {
    int r = 0, i = 0;
    if (i < max) {
        a++;
        assert(a != 0);
        r = max + (r / a);
        i++;
    }
    r = r * 2;
    return r;
}
```

\[ M := r_0 = 0 \land \\
    i_0 = 0 \land \\
    a_1 = a_0 + 1 \land \\
    r_1 = max_0 + r_0 / a_1 \land \\
    i_1 = i_0 + 1 \land \\
    r_2 = (i_0 < max_0) \? r_1 : r_0 \land \\
    r_3 = r_2 * 2 \land \\
    P := a_1 != 0 \]

Tuesday, December 18, 12
Conjunctive Normal Form

\[ M_1 := (r_0 = 0) \land (i_0 = 0) \land (a_1 = a_0 + 1) \land (r_1 = \max_0 + r_0 / a_1) \land (i_1 = i_0 + 1) \land (r_2 = (i_0 < \max_0) ? r_1 : r_0) \land (r_3 = r_2 \times 2) \]

\[ \neg P_1 := (a_1 = 0) \]

\[ (M_k \land \neg P_k) \rightarrow \text{SAT solver} \rightarrow \text{SAT or UNSAT?} \]
CBMC

Bounded Model Checking for ANSI-C programs

Checks safety properties:
- buffer overflows
- pointer safety
- division by zero
- not-a-number
- uninitialized variable
- data race

CBMC calls an assertion generator \textit{(goto-instrument)} to add assertions in the code to verify these properties
How to use CBMC to Automated Test Generation?
1º - Assign nondeterminist values to the input variables (use the CBMC functions with prefix nondet_)

2º - add assertions

```c
#ifdef ASSERTION_1
assert(0);
#endif
```

3º - run CBMC

```
$ cbmc file.c -D ASSERTION_1
```
int func(int x, int y) {
    int a = 0;
    while (x > 3 || y == 1) {
        #ifdef ASSERTION_1
        assert(0);
        #endif
        a++;
        x--; y++;
    }
    return a;
}

int main() {
    int x = nondet_int();
    int y = nondet_int();
    return func(x, y);
}

$ cbmc file.c -D ASSERTION_1 --unwind 1 --no-unwinding-assertions
int func(int x, int y) {
    int a = 0;
    while (x > 3 || y == 1) {
        #ifdef ASSERTION_1
        assert(0);
        #endif
        a++; x--; y++;
    }
    return a;
}

When CBMC reaches an assert(0) stops the execution and give us the variables values that lead the program to this point

Which test case returns the decision

\((x > 3) \lor (y == 1)\) as TRUE?
Automated Test Generation using CBMC

Rui Gonçalo

CBMC

$ cbmc file.c -D ASSERTION_1 --unwind 1 --no-unwinding-assertions

int func(int x, int y) {
  int a = 0;
  while (x > 3 || y == 1) {
    #ifdef ASSERTION_1
    assert(0);
    #endif
    a++; x--; y++;
  }
  return a;
}

Test case
(x = -1073741824, y = 1)
CBMC and MC/DC

How to use CBMC to Automated Test Generation and achieve MC/DC?
int func(int x, int y) {
    int a = 0;
    while (x > 3 || y == 1) {
        a++;
        x--;  
        y++;
    }
    return a;
}
Control Flow Graph

unwind $k = 1$

```c
int func(int x, int y) {
    int a = 0;
    if (x > 3 || y == 1) {
        a++;
        x--;
        y++;
    }
    return a;
}
```

True

False
CBMC and MC/DC

MC/DC requires:
- \( x > 3 \) : TRUE and FALSE
- \( y == 1 \) : TRUE and FALSE
- \( x > 3 \ || \ y == 1 \) : TRUE and FALSE

```c
int func(int x, int y) {
  int a = 0;
  if (x > 3 || y == 1) {
    a++;
    x--;
    y++;
  }
  return a;
}
```
if (x > 3) {
  if (y == 1) {
    ASSERTION_1
    a++; x--; y++;
  } else {
    ASSERTION_2
    a++; x--; y++;
  }
} else {
  if (y == 1) {
    ASSERTION_3
    a++; x--; y++;
  } else {
    ASSERTION_4
  }
}
### CBMC and MC/DC

| x         | y         | C#1: x > 3 | C#2: y == 1 | C#1 || C#2 |
|-----------|-----------|------------|-------------|--------|
| 1073741824 | 1         | TRUE       | TRUE        | TRUE   |
| 1073741824 | -2096361621 | TRUE       | FALSE       | TRUE   |
| -2130706432 | 1         | FALSE      | TRUE        | TRUE   |
| -2147483584 | -2122265085 | FALSE      | FALSE       | FALSE  |
# CBMC and MC/DC

| x          | y          | C#1: x > 3 | C#2: y == 1 | C#1 || C#2 |
|------------|------------|------------|-------------|-------|
| 1073741824 | 1          | TRUE       | TRUE        | TRUE  |
| 1073741824 | -2096361621| TRUE       | FALSE       | TRUE  |
| -2130706432| 1          | FALSE      | TRUE        | TRUE  |
| -2147483584| -2122265085| FALSE      | FALSE       | FALSE |

100% MC/DC
CBMC and MC/DC

How to use CBMC to Automated Test Generation and achieve MC/DC without redundant test cases?
Consider the branches from if statements nodes $S_{b_1}^b_{b_2}$.

The algorithm builds paths that execute each branch only once.
CBMC effectively

int test(int a[], int size) {
    int negatives = 0, i = 0;
    while(i < size) {
        if (a[i] < 0) negatives++;
        i++;
    }
    return negatives;
}

a

+  -  -  +  -  ...

int test(int a[], int size) {
    int negatives = 0, i = 0;
    if (i < size) {
        if (a[i] < 0) negatives++;
        i++;
        if (i < size) {
            if (a[i] < 0) negatives++;
            i++;
            if (i < size) {
                if (a[i] < 0) negatives++;
                i++;
            }
        }
    }
    return negatives;
}
int test(int a[], int size) {
    int negatives = 0, i = 0;
    if (i < size) {
        if (a[i] < 0) negatives++;
        i++;
    }
    if (i < size) {
        if (a[i] < 0) negatives++;
        i++;
    }
    if (i < size) {
        if (a[i] < 0) negatives++;
        i++;
    }
    return negatives;
}
Automated Test Generation using CBMC

CBMC effectively

Path = \{S_0, S_1\}

Branches to find = \{b_1, b_2, b_3, b_4\}

Succ of S_1? \ S_2 \text{ and } S_{13}

Which one to choose?

The one that has the higher number of branches to find

S_2 \rightarrow \{b_1, b_2, b_3, b_4\}
S_{13} \rightarrow \{\}

\[\text{Branches to find} = \{b_1, b_2, b_3, b_4\}\]
Path = \{S_0, S_1, S_2\}

Branches to find = \{b_2, b_3, b_4\}

Succ of S_2?  \(S_3 \text{ and } S_4\)

Which one to choose?

If the number of branches to find is the same, choose in lexicograph order
CBMC effectively

Path = \{S_0, S_1, S_2, S_3\}
Branches to find = \{b_2, b_3\}

Succ of S_3? \quad S_4
Succ of S_4? \quad S_5
CBMC effectively

Path = \{S_0, S_1, S_2, S_3, S_4, S_5\}
Branches to find = \{b_2, b_3\}

Succ of S_5? S_6 and S_{13}

S_6 \rightarrow \{b_2, b_3\}
S_{13} \rightarrow \{\}

Tuesday, December 18, 12
CBMC effectively

Path = \{S_0, S_1, S_2, S_3, S_4, S_5, S_6\}
Branches to find = \{b_2, b_3\}

Succ of S_6?  S_7 and S_8

S_7 → \{\}\nS_8 → \{b_3\}
Automated Test Generation using CBMC

CBMC effectively

Path = \{S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_8, S_9\}

Branches to find = \{b_2\}

Succ of S_9? S_10 and S_13

S_10 -> \{
S_13 -> \{

but b_1 was already found!!
CBMC effectively

Path = \{S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_8, S_9, S_{13}\}

Branches to find = \{\}

All branches found!
Algorithm is finished!

How many paths?
One was enough to cover all branches.
CBMC effectively

Intrument the code:

- No branch points;
- Force CBMC to go through that path.
- Insert __CPROVER_assume

```c
int test(int a[], int size) {
  int b = 0, c = 0;
  __CPROVER_assume(c < size);
  __CPROVER_assume(a[c] < 0);
  b++;
  c++;
  __CPROVER_assume(c < size);
  __CPROVER_assume(!(a[c] < 0));
  c++;
  __CPROVER_assume(!(c < size));
  assert(0);
  return b;
}
```
int test(int a[], int size) {
    int negatives = 0, i = 0;
    if (i < size) {
        if (a[i] < 0) negatives++;
        i++;
    }
    if (i < size) {
        if (a[i] < 0) negatives++;
        i++;
        if (i < size) {
            if (a[i] < 0) negatives++;
            i++;
        }
    }
    return negatives;
}

Test case: 
T = (size=2, 
    a[0]=-2147483648, 
    a[1]=0)

\[ \begin{array}{cccc}
-2147483648 & & 0 \\
\end{array} \]
Goals

- Automated Test Generation survey
- Apply CBMC in Automated Test Generation
- How to achieve MC/DC?
- Implement CBMCe
- Experimental results
Automated Test Generation using CBMC

**CBM Ce**

- program.c → C.g → CLexer.java → CParser.java → CFGGenerator.java
- AST → ASTwalker
- CFG.java
- PathGenerator.java
- Instrument.java
- path_1.c
- path_2.c
- path_x-1.c
- path_x.c

ANTLRv3
Automated Test Generation using CBMC

CBM Ce

cbmc path_1.c --xml-ui

cbmc path_2.c --xml-ui

... XMLParser.java

TEST CASES
Conclusion

- Bounded model checking is useful for test generation

- CBMC achieved good results when applied to critical software

- CBMC effective method was proved to generate less number of test cases to the same MC/DC percentage (100%) than manual methods, in much less time (~4h to +100h)
References


Automated Test Generation using CBMC

Rui Gonçalo
CROSS Project
Computer Science Department
University of Minho

December 2012