

CROWN Tutorial

Moonzoo Kim

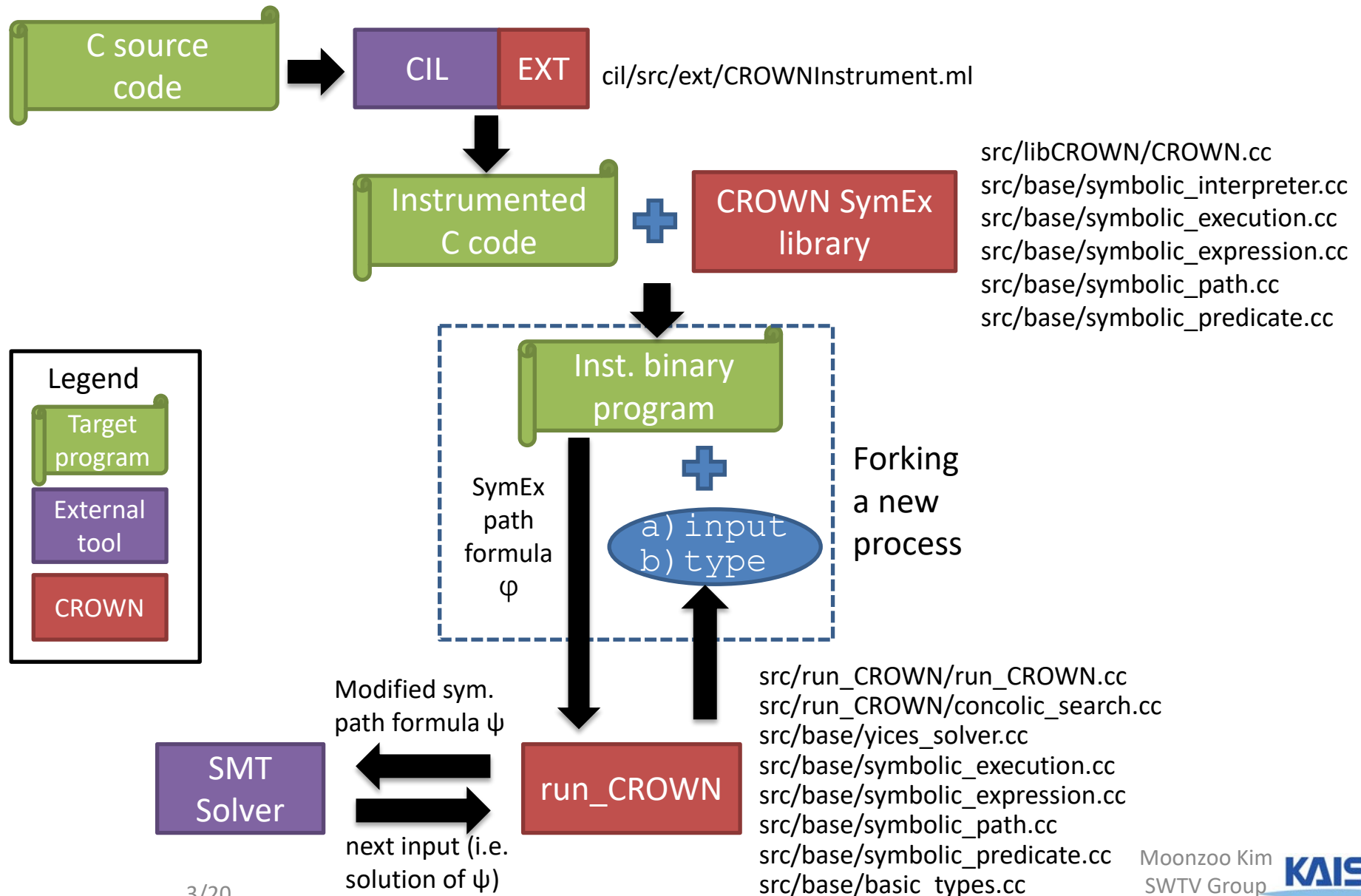
School of Computing

KAIST

CROWN

- CROWN is a concolic testing tool for C programs
 - Explore all possible execution paths of a target systematically
 - To extract symbolic path formulas from concrete executions, it inserts probes into target C code
 - CROWN's instrumentation is implemented as a module of CIL (C Intermediate Language) written in Ocaml
 - CIL provides convenient features to analyze/modify target C code
 - Generate test inputs automatically by solving symbolic path formulas by using a SMT solver
- CROWN is the core engine of the commercial automated testing tool CROWN 2.0
 - <https://www.vpluslab.kr/crown2>

Overview of CROWN code



Instrumented C Code

#line 10

```
{ /* Creates symbolic expression a==b */
__CrownLoad(36, (unsigned long )(& a), (long long )a);
__CrownLoad(35, (unsigned long )(& b), (long long )b);
__CrownApply2(34, 12, (long long )(a == b));
if (a == b) {

    __CrownBranch(37, 11, 1); //extern void __CrownBranch(int id , int bid , unsigned char b )
    __CrownLoad(41, (unsigned long )(& match), (long long )match);
    __CrownLoad(40, (unsigned long )0, (long long )1);
    __CrownApply2(39, 0, (long long )(match + 1));
    __CrownStore(42, (unsigned long )(& match));
    match ++;

} else {
    __CrownBranch(38, 12, 0);
} }
```

Control dependency v.s. Data dependency

- match has control dependency on a and b
- match does not have data dependency on a and b

CROWN Commands

- `crownc <filename>.c`
 - **Output**
 - `<filename>.cil.c` // instrumented C file
 - `branches` // lists of paired branches
 - `<filename>` // executable file
- `run_crown ./filename <n> -[dfs|cfg|random|random_input|hybrid]`
`[-TCDIR <tc_folder>] [-INIT_TC]`
 - **<n>: # of iterations/testings**
 - **Concolic search strategies**
 - `dfs`: depth first search
 - `rev-dfs` : reverse depth first search
 - `cfg`: uncovered branch first
 - `random`: negated branch is randomly selected
 - `random_input`: pure random input
 - `hybrid`: combination of `dfs` and `random`
 - **-INIT_TC: to use “input” file in a target directory as an initial test case**
 - if “input” file does not exist, `run_CROWN` terminates with an error message
 - **Output (updating at each iteration)**
 - `input`: containing concrete types and values of symbolic variables
 - `szd_execution`: symbolic execution path
 - `coverage`: coverage achieved so far
 - A test case file in `<tc_folder>` if `-TCDIR` option is given

4 Main Tasks of Human Engineers

1. Adding proper assert() statements
 - W/o assert(), only runtime crash can be detected
2. Selection of **symbolic variables** in a target program
 - Identify which parts of a target program are most important
3. Construction of **symbolic external environment**
 - To reduce false alarms and detect real bugs
4. Performance tuning and debugging
 - To obtain better concolic testing results

Supported Symbolic Data-types

- `#define SYM_unsigned_char(x) __CrownUChar(&x)`
- `#define SYM_unsigned_short(x) __CrownUShort(&x)`
- `#define SYM_unsigned_int(x) __CrownUInt(&x)`
- `#define SYM_char(x) __CrownChar(&x)`
- `#define SYM_short(x) __CrownShort(&x)`
- `#define SYM_int(x) __CrownInt(&x)`
- `#define SYM_float(x) __CrownFloat(&x)`
- `#define SYM_double(x) __CrownDouble(&x)`

Symbolic Variable w/ Initial Value

- `SYM_unsigned_char_init(x, 7)`
- `SYM_unsigned_short_init(x, 7)`
- `SYM_unsigned_int(x, 7)`
- `SYM_char_init(x, 7)`
- `SYM_short(x, 7)`
- `SYM_int(x, 7)`
- `SYM_float(x, 7.0)`
- `SYM_double(x, 7.0)`

```
#include<crow.h>
int main() {
    int x;
    SYM_int_init(x, 7);
    printf("x=%d\n", x);
    if ( x > 10)
        printf("x>10\n");
    else
        printf("x<=10\n");
}
```

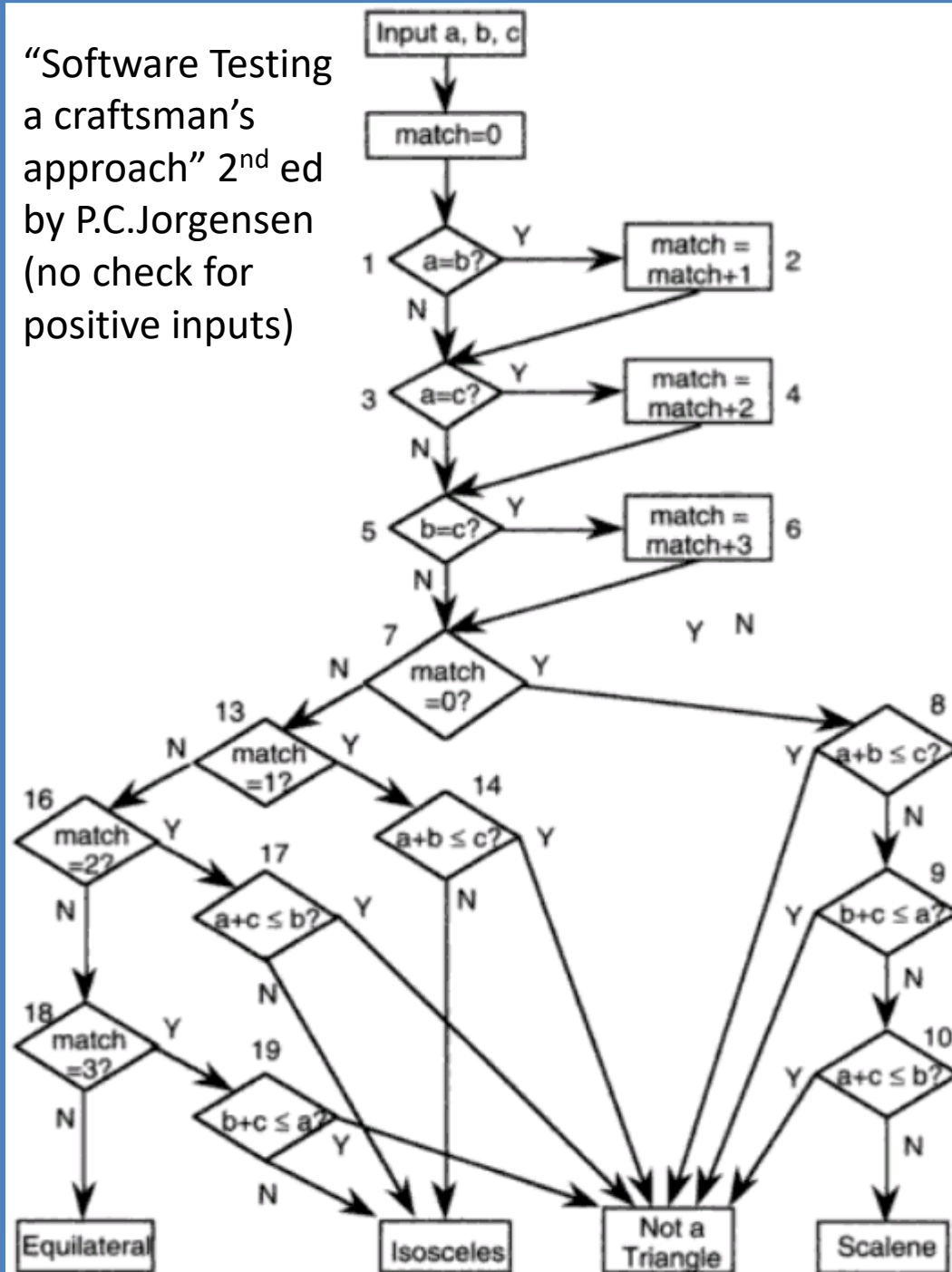

Symbolic Assumption

- You can describe symbolic assumption using `SYM_assume (exp)`
 - `exp` is **guaranteed to be true** right after `SYM_assume (exp)`
 - similar to `__CPROVER_assume (exp)` in CBMC
- Ex.

```
#include <crow.h>
#include <stdio.h>
#include <assert.h>

void main() {
    int x, y;
    SYM_int(x);
    SYM_int(y);
    SYM_assume( x + y > 10 );
    printf("x=%d, y=%d\n", x, y);
    assert( x + y > 10 );
}
```

“Software Testing
a craftsman’s
approach” 2nd ed
by P.C.Jorgensen
(no check for
positive inputs)



```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <crow.h>

#define CROWN

int triangle(int, int, int);

int main() {
    int a,b,c, result;

#ifdef CROWN
    SYM_int(a); SYM_int(b); SYM_int(c);

    //filtering out invalid inputs
    SYM_assume(a>0 && b>0 && c>0);
    printf("a,b,c = %d,%d,%d\n",a,b,c);
#else
    printf("Please type 3 integers:\n");
    scanf("%d,%d,%d",&a,&b,&c);
#endif

    result=triangle(a,b,c);

    char triangle_type[20];
    switch(result) {
        case 0: strcpy(triangle_type,
            "an equilateral"); break;
        case 1: strcpy(triangle_type,
            "an isoscele"); break;
        case 2: strcpy(triangle_type,
            "not a triangle"); break;
        case 3: strcpy(triangle_type,
            "a scalene"); break;
        defaults: break;
    }
    printf("Type: %s.\n",triangle_type);
}

```

```

// Return value:
// 0: Equilateral, 1:Isosceles,
// 2: Not a triangle, 3:Scalene
int triangle(int a, int b, int c) {
    int result=-1, match=0;

    if(a==b) match=match+1;
    if(a==c) match=match+2;
    if(b==c) match=match+3;

    if(match==0) {
        if( a+b <= c) result=2;
        else if( b+c <= a) result=2;
        else if(a+c <= b) result =2;
        else result=3;
    } else {
        if(match == 1) {
            if(a+b <= c) result =2;
            else result=1;
        } else {
            if(match ==2) {
                if(a+c <=b) result = 2;
                else result=1;
            } else {
                if(match==3) {
                    if(b+c <= a) result=2;
                    else result=1;
                } else result = 0;
            }
        }
    }

    return result;
}

```

Compile/Instrumentation Snapshot

```
verifier3:~/crown/triangle$ crownc triangle.c
```

```
>>> cilly command: /usr/bin/./cil/bin/cilly triangle.c -o t
triangle --save-temps --doCrownInstrument -I/usr/bin
../include -L/usr/bin/../lib -L/usr/bin/../lib -lcrown-f
p -lstdc++ -g -lrt -Wno-attributes -lpthread
gcc -D_GNUCC -E -I/usr/bin/../include -g -DCIL=1 triangle
.c -o ./triangle.i
/usr/cil/bin/cilly.native --out ./triangle.cil.c --doCrownInst
rument ./triangle.i
triangle.c:34: Warning: Body of function main falls-throu
gh. Adding a return statement
gcc -D_GNUCC -E -I/usr/bin/../include -g ./triangle.cil.c -o
./triangle.cil.i
gcc -D_GNUCC -c -I/usr/bin/../include -g -g -Wno-attribut
es -o ./triangle.o ./triangle.cil.i
gcc -D_GNUCC -o triangle -I/usr/bin/../include -g -g -Wno
-attributes ./triangle.o -L/usr/bin/../lib -L/usr/bin/../li
b -lcrown-fp -lstdc++ -g -lrt -lpthread
>>> Check cilly
>>> cilly - success!
>>> Find the file(triangle)
>>> start: 2022-11-17 12:02:26
>>> end: 2022-11-17 12:02:26
>>> gcc command(sanitizer none): gcc -o triangle_replay
triangle.c -I/usr/bin/../include -L/usr/bin/../lib -lcrow
n-replay --coverage -Wno-attributes -lpthread
>>> Check gcc
>>> gcc - success
>>> Find the file(triangle_replay)
>>> start: 2022-11-17 12:02:26
>>> end: 2022-11-17 12:02:26
```

Compiled with FP option
Read 42 branches.
Read 78 nodes.
Wrote 48 branch edges.

```
verifier3:~/crown/triangle$ l
```

```
triangle.c
triangle.i
triangle.cil.c
triangle.cil.i
```

```
triangle.gcno
triangle.o
```

```
triangle
triangle_replay
```

```
branches
cfg_branches
funcount
cfg
cfg_func_map
idcount
stmtcount
```

Execution Snapshot (1/2)

```
$ run_crown ./triangle 50 -dfs -TCDIR tcs
```

```
-----  
program command is triangle  
### SYM_assume(a>0 && b>0 && c>0) is violated  
at Line 17 (main in triangle.c) ###  
...
```

```
-----  
program command is triangle  
a,b,c = 1,1,1  
This triangle is an equilateral.  
Iteration 1 (0s, 0s, 0.018s): covered 11 branches  
[2 reach funs, 42 reach branches]
```

```
-----  
a,b,c = 1610612736,1610612736,536870912  
This triangle is not a triangle.  
Iteration 2 (1s, 1s, 0.043s): covered 18 branches  
[2 reach funs, 42 reach branches]
```

```
-----  
a,b,c = 2,2,1  
This triangle is an isoscele.  
Iteration 3 (1s, 0s, 0.050s): covered 20 branches  
[2 reach funs, 42 reach branches]
```

```
-----  
a,b,c = 1610612736,536870912,1  
This triangle is not a triangle.  
Iteration 4 (1s, 0s, 0.060s): covered 23 branches  
[2 reach funs, 42 reach branches]
```

```
-----  
a,b,c = 272629760,1346371584,809500672  
This triangle is not a triangle.  
Iteration 5 (1s, 0s, 0.067s): covered 25 branches  
[2 reach funs, 42 reach branches]
```

```
-----  
a,b,c = 1108719680,34977856,1108457536  
This triangle is not a triangle.  
Iteration 6 (1s, 0s, 0.074s): covered 27 branches  
[2 reach funs, 42 reach branches]
```

```
-----  
a,b,c = 427818799,427818767,377487598  
This triangle is a scalene.  
Iteration 7 (1s, 0s, 0.085s): covered 30 branches  
[2 reach funs, 42 reach branches]
```

```
-----  
a,b,c = 1610612736,536870912,536870912  
This triangle is not a triangle.  
Iteration 8 (1s, 0s, 0.091s): covered 32 branches  
[2 reach funs, 42 reach branches]
```

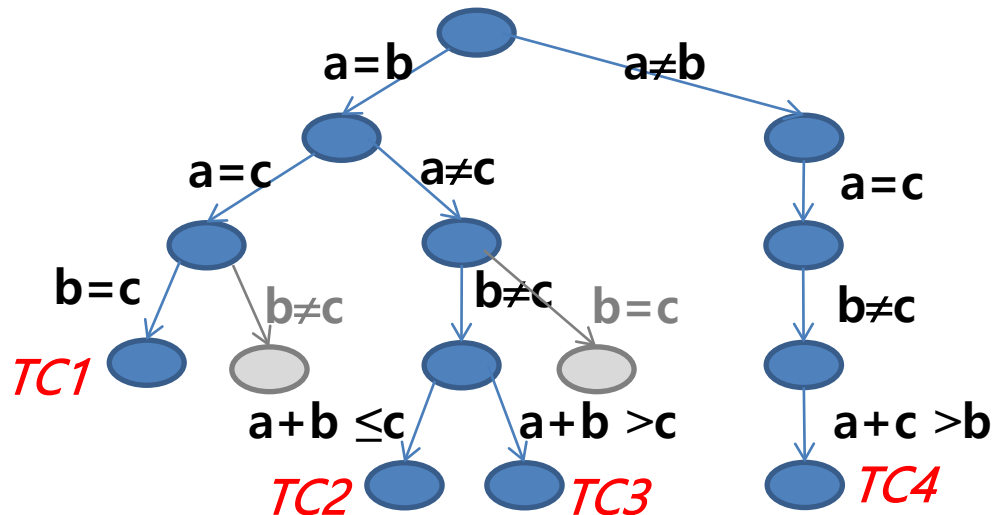
```
-----  
a,b,c = 1,2,2  
This triangle is an isoscele.  
Iteration 9 (1s, 0s, 0.098s): covered 33 branches  
[2 reach funs, 42 reach branches]
```

```
-----  
a,b,c = 1610612736,536870912,1610612736  
This triangle is not a triangle.  
Iteration 10 (1s, 0s, 0.104s): covered 35 branches  
[2 reach funs, 42 reach branches]
```

```
-----  
a,b,c = 2,1,2  
This triangle is an isoscele.  
Iteration 11 (1s, 0s, 0.111s): covered 36 branches  
[2 reach funs, 42 reach branches]
```

Concolic Testing the Triangle Program

Test case	Input (a,b,c)	Executed symbolic path formula (SPF) φ	Modified symbolic path formula φ'	Solution for the modified SPF
TC1	1,1,1	$a=b \wedge a=c \wedge b=c$	$a=b \wedge a=c \wedge b \neq c$	Unsat
			$a=b \wedge a \neq c$	1,1,2
TC2	1,1,2	$a=b \wedge a \neq c \wedge b \neq c \wedge a+b \leq c$	$a=b \wedge a \neq c \wedge b \neq c \wedge a+b > c$	2,2,3
TC3	2,2,3	$a=b \wedge a \neq c \wedge b \neq c \wedge a+b > c$	$a=b \wedge a \neq c \wedge b=c$	Unsat
			$a \neq b$	2,1,2
TC4	2,1,2	$a \neq b \wedge a=c \wedge b \neq c \wedge a+c > b$	$a \neq b \wedge a=c \wedge b \neq c \wedge a+c \leq b$	2,5,2



Execution Snapshot (2/2)

```
...$ cat branches
```

```
1 8 /* branch IDs */
```

```
6 11
```

```
7 10
```

```
8 9
```

```
13 14
```

```
21 22
```

```
24 25
```

```
27 28
```

```
30 31
```

```
2 13
```

```
48 49
```

```
51 52
```

```
54 55
```

```
57 64
```

```
58 59
```

```
60 61
```

```
62 63
```

```
65 68
```

```
66 67
```

```
69 72
```

```
70 71
```

```
73 76
```

```
74 75
```

```
...$ cat coverage
```

```
6 /*covered branch IDs*/
```

```
7
```

```
8
```

```
21
```

```
22
```

```
24
```

```
25
```

```
27
```

```
28
```

```
30
```

```
48
```

```
49
```

```
51
```

```
52
```

```
54
```

```
55
```

```
57
```

```
58
```

```
59
```

```
60
```

```
61
```

```
62
```

```
...
```

```
...$ | tcs
```

```
input.1 ... input.11 type.1 ... type.11
```

```
...$ print_execution
```

Symbolic variables & input values

(a_1 = 2) [Line: 14, File: triangle.c]

(b_1 = 1) [Line: 14, File: triangle.c]

(c_1 = 2) [Line: 14, File: triangle.c]

Symbolic path formula

(a_1 > 0) [Line: 17, File: triangle.c]

(b_1 > 0) [Line: 17, File: triangle.c]

(c_1 > 0) [Line: 17, File: triangle.c]

! (a_1 == b_1) [Line: 43, File: triangle.c]

(a_1 == c_1) [Line: 44, File: triangle.c]

! (b_1 == c_1) [Line: 45, File: triangle.c]

! ((a_1 + c_1) <= b_1) [Line: 58, File: triangle.c]

Sequence of reached branch ids

-1 [main enters]

6 [Line: 17, File: triangle.cil.c]

7 [Line: 17, File: triangle.cil.c]

8 [Line: 17, File: triangle.cil.c]

-1 [triangle enters]

49 [Line: 43, File: triangle.cil.c]

51 [Line: 44, File: triangle.cil.c]

55 [Line: 45, File: triangle.cil.c]

64 [Line: 47, File: triangle.cil.c]

68 [Line: 53, File: triangle.cil.c]

69 [Line: 57, File: triangle.cil.c]

71 [Line: 58, File: triangle.cil.c]

-2 [triangle exits]

22 [Line: 28, File: triangle.cil.c]

24 [Line: 29, File: triangle.cil.c]

-2 [main exits]

Symbolic Debugging [1/2]

1. Select `[TCDIR]/input.[n]` whose symbolic path formula you would like to know
2. Copy `[TCDIR]/input.[n]` to a target directory with a name “input”
 - Also copy `type.[n]`
3. Run an instrumented executable target program
 - Note that an instrumented executable target program reads “input” file w/ “type” as an initial test case
 - Ex. `./triable`
4. See symbolic information of `input.[n]` by using `print_execution`

Symbolic Debugging (2/2)

```
...$ run_crown ./triange...
```

```
...
```

```
-----
```

program command is triangle

`a,b,c = 1,1,1`

This triangle is an equilateral.

`Iteration 1` (0s, 0s, 0.018s):

covered 11 branches

[2 reach funs, 42 reach branches]

```
-----
```

```
...
```

```
...$ cp tcs/input.1 input
```

```
...$ cp tcs/type.1 type
```

```
...$ ./triangle
```

`a,b,c = 1,1,1`

This triangle is an equilateral.

```
...$ print_execution
```

Symbolic variables & input values

(a_1 = 1) [Line: 7, File: triangle-crown.c]

(b_1 = 1) [Line: 7, File: triangle-crown.c]

(c_1 = 1) [Line: 7, File: triangle-crown.c]

Symbolic path formula

(a_1 > 0) [Line: 15, File: triangle-crown.c]

(b_1 > 0) [Line: 15, File: triangle-crown.c]

(c_1 > 0) [Line: 15, File: triangle-crown.c]

(a_1 == b_1) [Line: 21, File: triangle-crown.c]

(a_1 == c_1) [Line: 22, File: triangle-crown.c]

(b_1 == c_1) [Line: 23, File: triangle-crown.c]

Sequence of reached branch ids

-1 [main enters]

6 [Line: 15, File: triangle-cro]

7 [Line: 15, File: triangle-cro]

8 [Line: 15, File: triangle-cro]

20 [Line: 21, File: triangle-cro]

23 [Line: 22, File: triangle-cro]

26 [Line: 23, File: triangle-cro]

36 [Line: 24, File: triangle-cro]

40 [Line: 30, File: triangle-cro]

44 [Line: 34, File: triangle-cro]

48 [Line: 38, File: triangle-cro]

51 [Line: 46, File: triangle-cro]

-2 [main exits]

Instrumented C Code

#line 10

```
{ /* Creates symbolic expression a==b */  
__CrownLoad(36, (unsigned long )(& a), (long long )a);  
__CrownLoad(35, (unsigned long )(& b), (long long )b);  
__CrownApply2(34, 12, (long long )(a == b));  
if (a == b) {  
  
    __CrownBranch(37, 11, 1); //extern void __CrownBranch(int id , int bid , unsigned char b )  
    __CrownLoad(41, (unsigned long )(& match), (long long )match);  
    __CrownLoad(40, (unsigned long )0, (long long )1);  
    __CrownApply2(39, 0, (long long )(match + 1));  
    __CrownStore(42, (unsigned long )(& match));  
    match ++;  
  
} else {  
    __CrownBranch(38, 12, 0);  
} }
```

Control dependency v.s. Data dependency

- match has control dependency on a and b
- match does not have data dependency on a and b

Decision/Condition Coverage Analysis by CROWN

```

1 int main(){
2     int A, B, C, D;
3     if (A && B || C && D){
4         printf("Yes\n");
5     }else{
6         printf("No\n");
7     }
8 }

```

- CROWN transforms a compound predicate into atomic ones with nested conditional statements
- CROWN consider all possible cases with short-circuit
- Thus, branch coverage reported by CROWN might be lower than actual branch coverage

```

1  if (A != 0) {
2      __CrownBranch(5, 2, 1); A == T
3      if (B != 0) {
4          __CrownBranch(10, 3, 1); A == T && B == T
5          printf("Yes\n");
6      } else {
7          __CrownBranch(11, 4, 0); A == T && B != T
8          goto _L;
9      }
10 } else {
11     __CrownBranch(6, 5, 0) A != TRUE
12     _L: /* CIL Label */
13     if (C != 0) {
14         __CrownBranch(16, 6, 1); (A != T || A == T && B != T) && C == T
15         if (D != 0) {
16             __CrownBranch(21, 7, 1); (A != T || A == T && B != T) && C == T && D == T
17             printf("Yes\n");
18         } else {
19             __CrownBranch(22, 8, 0); (A != T || A == T && B != T) && C == T && D != T
20             printf("No\n");
21         }
22     } else {
23         __CrownBranch(17, 9, 0); (A != T || A == T && B != T) && C != T
24         printf("No\n");
25     }
26 }

```

Measure Branch Coverage by Using gcov

```
...$ crown_replay
Usage : crown_replay <PROGRAM_CMD> [OPTION]...
Options:
  -d <DIR>  Use test inputs in the directory <DIR>.
             (Default : testdir)
  -s <ITER_START>
             Start the iteration from the <ITER_START>th test.
             (Default : 1)
  -e <ITER_END>
             End the iteration to the <ITER_END>th test.
             (Default : # of test inputs in <DIR>)
```

- `crownc` generates `<target>_replay` which is an original target program (i.e., conditional stmts *not* transformed, symbolic execution *not* extracted) that can read TCs generated by CROWN
 - Ex> `triangle_replay`
- `<target>_replay` reads “input” file in the same directory to replay
- `crown_replay` replays `<target>_replay` to measure branch coverage of the original target program with TCs in `<DIR>` generated by CROWN

```
...$ cp tcs/input.4 input .
...$ cp tcs/type.4 type .
...$ ./triangle_replay
a,b,c = 2,1,2
This triangle is an isoscele.
...$ gcov -b -f triangle
Function 'triangle'
Lines executed:61.90% of 21
Branches executed:61.54% of 26
Taken at least once:42.31% of 26
No calls
```

```
File 'triangle.c'
Lines executed:68.75% of 32
Branches executed:69.23% of 39
Taken at least once:41.03% of 39
Calls executed:54.55% of 11
```

```
...$ crown_replay ./triangle_replay -d tcs -s 5 -e 7
a,b,c = 1610612736,536870912,536870912
This triangle is not a triangle.
Iteration 5
-----
a,b,c = 1,2,2
This triangle is an isoscele.
Iteration 6
-----
a,b,c = 272629760,1346371584,809500672
This triangle is not a triangle.
Iteration 7
-----
```

```
... $ gcov -b triangle-crown
File 'triangle-crown.c'
Lines executed:76.67% of 30
Branches executed:84.62% of 39
Taken at least once:53.85% of 39
Calls executed:50.00% of 10
Creating 'triangle-crown.c.gcov'
```