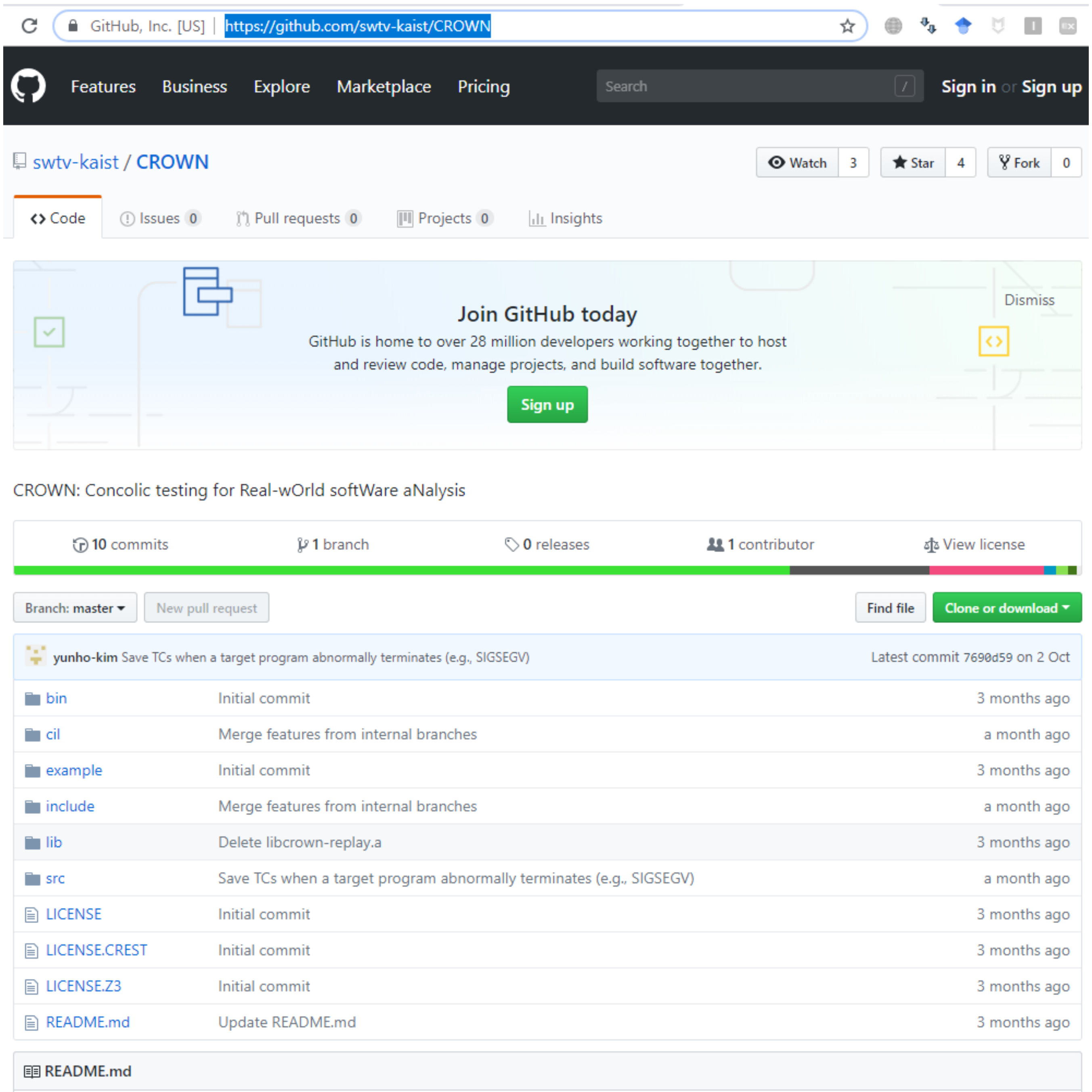
CROWN Tutorial

Moonzoo Kim

CS Dept. KAIST



Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

CROWN: Concolic testing for Real-wOrld softWare aNalysis

10 commits | 1 branch | 0 releases | 1 contributor | View license

yunho-kim Save TCs when a target program abnormally terminates (e.g., SIGSEGV) Latest commit 7690d59 on 2 Oct

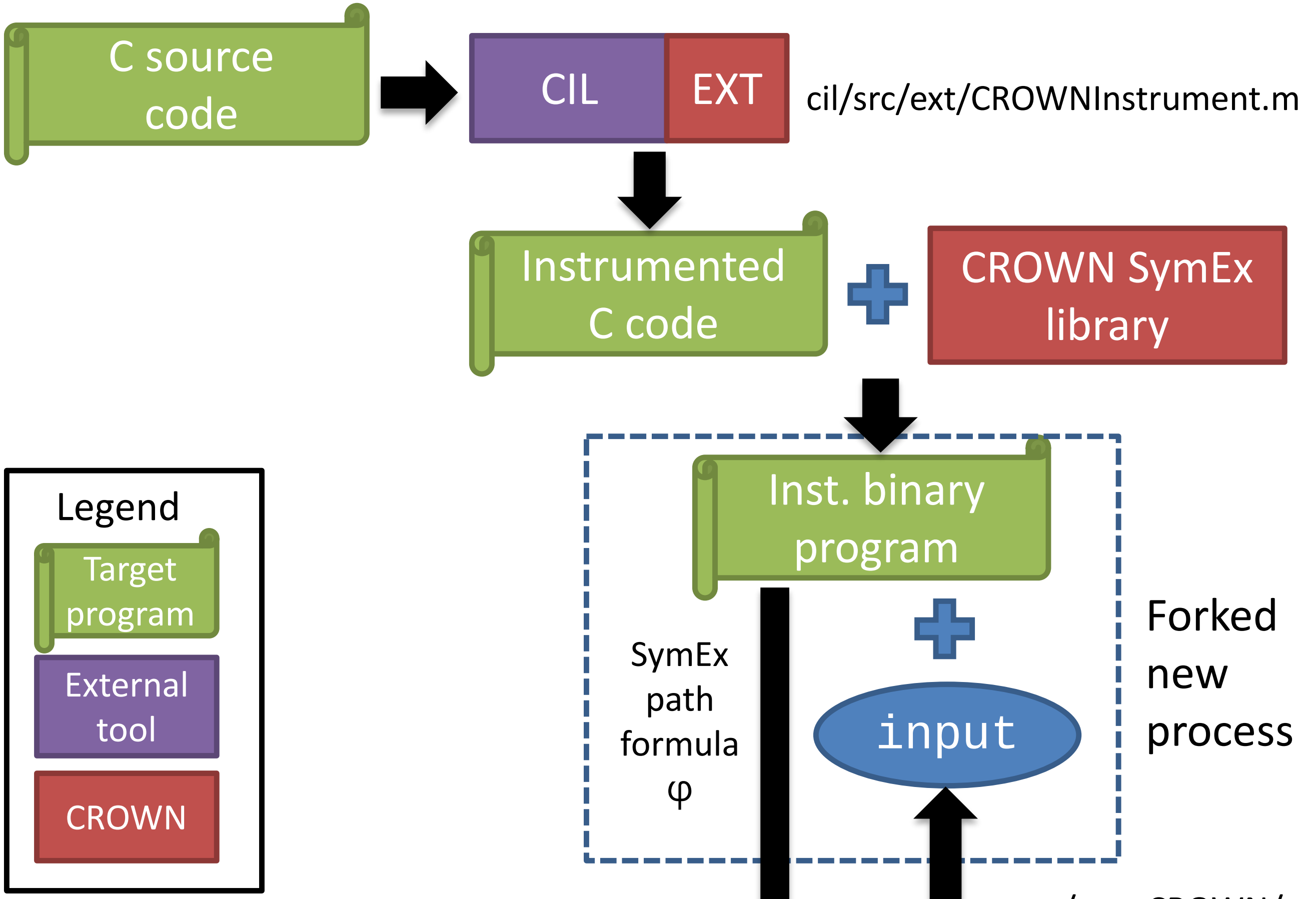
bin	Initial commit	3 months ago
cil	Merge features from internal branches	a month ago
example	Initial commit	3 months ago
include	Merge features from internal branches	a month ago
lib	Delete libcrown-replay.a	3 months ago
src	Save TCs when a target program abnormally terminates (e.g., SIGSEGV)	a month ago
LICENSE	Initial commit	3 months ago
LICENSE.CREST	Initial commit	3 months ago
LICENSE.Z3	Initial commit	3 months ago
README.md	Update README.md	3 months ago

README.md

CROWN

- CROWN is a concolic testing tool for C programs
  - Generate test inputs automatically
  - Execute target under test on generated test inputs
  - Explore all possible execution paths of a target systematically
- CROWN is a open-source extension of CREST
  - mainly written in C++
    - CROWN's instrumentation is implemented as a module of CIL (C Intermetiate Language) written in Ocaml
- https://github.com/swtv-kaist/CROWN/

Overview of CROWN code



C source code

CIL

EXT

cil/src/ext/CROWNInstrument.ml

Instrumented C code

CROWN SymEx library

src/libCROWN/CROWN.cc
src/base/symbolic_interpreter.cc
src/base/symbolic_execution.cc
src/base/symbolic_expression.cc
src/base/symbolic_path.cc
src/base/symbolic_predicate.cc

Legend

Target program

External tool

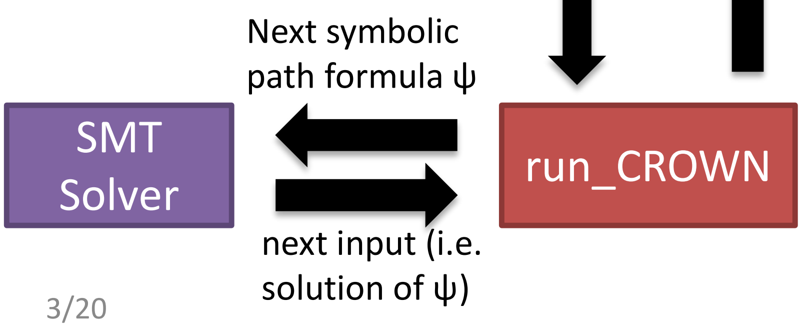
CROWN

Inst. binary program

SymEx path formula φ

input

Forked new process



Next symbolic path formula ψ

SMT Solver

run_CROWN

next input (i.e. solution of ψ)

src/run_CROWN/run_CROWN.cc
src/run_CROWN/concolic_search.cc
src/base/yices_solver.cc
src/base/symbolic_execution.cc
src/base/symbolic_expression.cc
src/base/symbolic_path.cc
src/base/symbolic_predicate.cc
src/base/basic_types.cc

4 Main Tasks of Human Engineers

1. Adding proper assert() statements
   – W/o assert(), no test results obtained
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 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<crown.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 <crown.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);
}
```

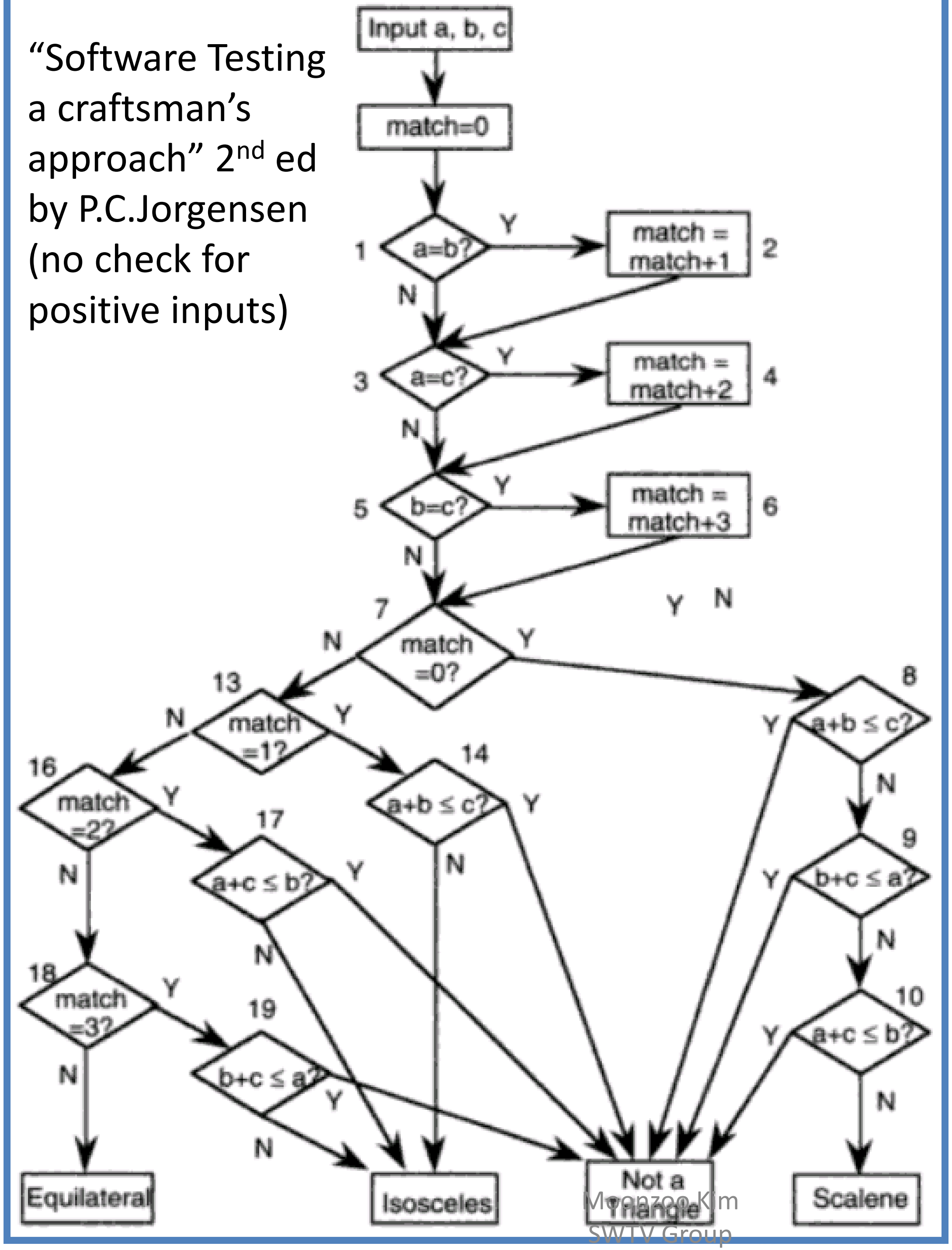
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <crown.h>
int main() {
   int a,b,c, match=0;
   SYM_int(a); SYM_int(b); SYM_int(c);

/*printf("Please type 3 integers:\n");
   scanf("%d", &a);
   scanf("%d", &b);
   scanf("%d", &c);  */

   //filtering out invalid inputs
   SYM_assume(a>0 && b>0 && c>0);
   printf("a,b,c = %d,%d,%d\n",a,b,c);

   //0: Equilateral, 1:Isosceles,
   // 2: Not a triangle, 3:Scalene
   int result=-1;
   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) {
```

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



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

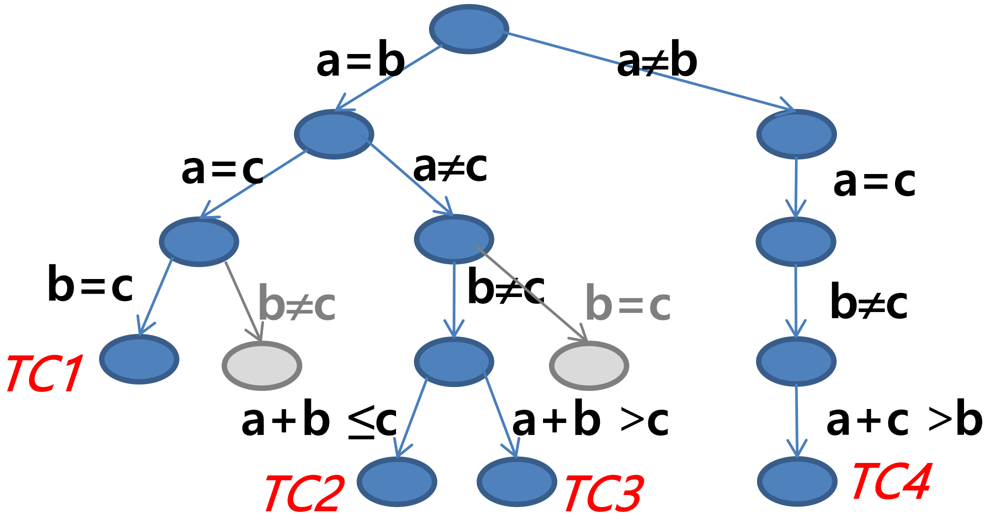
Execution Snapshot (1/2)

```
…$ run_crown ./triangle-crown  100  -rev-dfs -TCDIR test
       cases
-------------------------
### SYM_assume(a>0 && b>0 && c>0) is violated at Line
       15 (main in triangle-crown.c) ###
-------------------------
### SYM_assume(a>0 && b>0 && c>0) is violated at Line
       15 (main in triangle-crown.c) ###
-------------------------
### SYM_assume(a>0 && b>0 && c>0) is violated at Line
       15 (main in triangle-crown.c) ###
-------------------------
a,b,c = 1,1,1
This triangle is an equilateral.
Iteration 1 (0s, 0.42s): covered 11 branches [1 reach funs
       , 42 reach branches].(11, 0)
-------------------------
a,b,c = 1610612736,536870912,1
This triangle is not a triangle.
Iteration 2 (0s, 0.56s): covered 19 branches [1 reach funs
       , 42 reach branches].(19, 11)
-------------------------
a,b,c = 1610612736,536870912,1610612736
This triangle is not a triangle.
Iteration 3 (0s, 0.68s): covered 21 branches [1 reach funs
       , 42 reach branches].(21, 19)
-------------------------
a,b,c = 2,1,2
This triangle is an isoscele.
Iteration 4 (0s, 0.104s): covered 23 branches [1 reach fu
       ns, 42 reach branches].(23, 21)
-------------------------
a,b,c = 1610612736,536870912,536870912
This triangle is not a triangle.
Iteration 5 (0s, 0.118s): covered 25 branches [1 reach
       funs, 42 reach branches].(25, 23)
```

```
-------------------------
a,b,c = 1,2,2
This triangle is an isoscele.
Iteration 6 (0s, 0.133s): covered 26 branches [1 reach funs,
       42 reach branches].(26, 25)
-------------------------
a,b,c = 272629760,1346371584,809500672
This triangle is not a triangle.
Iteration 7 (0s, 0.149s): covered 28 branches [1 reach funs,
       42 reach branches].(28, 26)
-------------------------
a,b,c = 1108719680,34977856,1108457536
This triangle is not a triangle.
Iteration 8 (0s, 0.169s): covered 30 branches [1 reach funs,
       42 reach branches].(30, 28)
-------------------------
a,b,c = 427818799,427818767,377487598
This triangle is a scalene.
Iteration 9 (0s, 0.195s): covered 33 branches [1 reach funs,
       42 reach branches].(33, 30)
-------------------------
a,b,c = 1610612736,1610612736,536870912
This triangle is not a triangle.
Iteration 10 (0s, 0.209s): covered 35 branches [1 reach funs,
       42 reach branches].(35, 33)
-------------------------
a,b,c = 2,2,1
This triangle is an isoscele.
Iteration 11 (1s, 0.234s): covered 36 branches [1 reach funs,
       42 reach branches].(36, 35)
```

Concolic Testing the Triangle Program

Test case	Input (a,b,c)	Executed symbolic path formula w/ a test case φ	Next symbolic path formula ψ	Solution for the next sym. path formula
1	1,1,1	a=b Æa=c Æ b=c	a=b Æ a=c Æ b≠c	Unsat
			a=b Æ a≠c	1,1,2
2	1,1,2	a=b Æ a≠c Æ b≠c Æ a+b ≤c	a=b Æ a≠c Æ b≠c Æ a+b > c	2,2,3
3	2,2,3	a=b Æ a≠c Æ b≠c Æ a+b >c	a=b Æ a≠c Æ b=c	Unsat
			a≠b	2,1,2
4	2,1,2	a≠b Æ a=c Æ b≠c Æ a+c>b	a≠bÆ a=c Æ b≠c Æ a+c ≤b	2,5,2



Execution Snapshot (2/2)

```
…$ cat branches
1 21 /* branch IDs */
6 11
7 10
8 9
13 14
20 21
23 24
26 27
29 36
30 31
32 33
34 35
37 40
38 39
41 44
42 43
45 48
46 47
51 52
54 55
57 58
60 61
```

```
…$ cat coverage
6 /*covered branch IDs*/
7
8
20
21
23
24
26
27
29
30
31
32
33
34
35
36
37
38
39
40
…
```

```
…$ cat testcases/input.11
5 /* type of a symbolic input variable */
2 /* value of a symbolic input variable */
5
2
5
1

moonzoo@checker6:~/cs492d/CROWN/example$
print_execution

Symbolic variables & input values
(a_1 = 2)      [ Line: 7, File: triangle-crown.c ]
(b_1 = 2)      [ 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 ]
! ((a_1 + b_1) <= c_1)  [ Line: 31, 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 ]
24     [ Line: 22, File: triangle-cro ]
27     [ Line: 23, File: triangle-cro ]
36     [ Line: 24, File: triangle-cro ]
37     [ Line: 30, File: triangle-cro ]
39     [ Line: 31, File: triangle-cro ]
53     [ Line: 46, File: triangle-cro ]
```

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`"
3. Run an instrumented executable target program
   - Note that an instrumented executable target program reads "`input`" file as an initial test case
   - Ex. ./triable
4. See symbolic information of `input.[n]` by using `print_execution`

Symbolic Debugging (2/2)

```
…$ cat testcases/input.1
5 /* type of a symbolic input variable */
1 /* a: value of a symbolic input variable */
5
1
5
1

checker$ cp testcases/input.1 input

checker$ ./triangle-crown
a,b,c = 1,1,1:result=0
```

```
…$ 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

```
int main(){
    int A, B, C, D;
    if (A && B || C && D){
        printf("Yes\n");
    }else{
        printf("No\n");
    }
}
```

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

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

Measure Branch Coverage w/ TCs generated by crown_replay

```
…$ 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 testcases/input.4 input
…$ ./triangle-crown_replay
a,b,c = 2,1,2
This triangle is an isoscele.

…$ gcov -b triangle-crown
File 'triangle-crown.c'
Lines executed:56.67% of 30
Branches executed:64.10% of 39
Taken at least once:28.21% of 39
Calls executed:50.00% of 10
Creating 'triangle-crown.c.gcov'
…

…$ crown_replay ./triangle-crown_replay -d testcases -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
ile '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'
```