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 [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) →

```
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 **decrase** 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;
```

x	y	S#1	S#2	S#3	S#4
2	0	✓	✓	✓	✓

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;
```

x	y	S#1	S#2	S#3	S#4
2	0	✓	✓	✓	✓

Decision Coverage

Every <u>decision</u> has taken all possible outcomes at least once.

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

x	y	Decision
2	1	TRUE
1	1	FALSE

Decision Coverage

The effect of the second condition is not tested!

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

x	y	Decision
2	1	TRUE
1	1	FALSE

Condition Coverage

Every condition has taken all possible outcomes at least once.

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

x	y	Cond#1	Cond#2
2	1	TRUE	FALSE
1	2	FALSE	TRUE

Condition Coverage

The decision is always TRUE!

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

x	y	Cond#1	Cond#2	Decision
2	1	TRUE	FALSE	TRUE
1	2	FALSE	TRUE	TRUE

Condition/Decision Coverage

Every <u>condition</u> and <u>decision</u> have taken all possible outcomes at least once.

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

x	y	Cond#1	Cond#2	Decision
2	2	TRUE	TRUE	TRUE
1	1	FALSE	FALSE	FALSE

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;
```

x	y	Cond#1	Cond#2	Decision
2	2	TRUE	TRUE	TRUE
1	1	FALSE	FALSE	FALSE

Modified Condition/Decision Coverage

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

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

x	y	Cond#1	Cond#2	Decision
2	2	TRUE	TRUE	TRUE
1	1	FALSE	FALSE	FALSE
2	1	TRUE	FALSE	FALSE
1	2	FALSE	TRUE	TRUE

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;
```

x	y	Cond#1	Cond#2	Decision
2	2	TRUE	TRUE	TRUE
1	1	FALSE	FALSE	FALSE
2	1	TRUE	FALSE	FALSE
1	2	FALSE	TRUE	TRUE

Modified Condition/Decision Coverage

The standard DO-178B[1] "Software Considerations in Airbone Systems and Equipment Certifications" requires:

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

1- http://www.verifysoft.com/en_do-178b.html

Automated Test Generation

How ?

Bounded Model Checking

Model Checking:

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

- if **M** ⊨ **P** (M models P), *P* holds in *M*, i. e. the system functions according to *P*.
- if **M** ⊭ **P** (*M* doens'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 $(M_k \wedge \neg P_k)$ is **unsatisfiable** then *P* holds in *M* of length *k*
- if $(M_k \wedge \neg P_k)$ is **satisfiable** then *P* doesn't hold in *M* of length *k*, and a counterexample is produced

Conjunctive Normal Form

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

↓

How to translate C programs into CNF ?

Conjunctive Normal Form

C programs into CNF:

1º - Unwinding loops

```
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;
}
```

k = 1

→

```
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 Assigment Form

```
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;
}
```

→

$$
\begin{aligned}
M := \; & r_0 = 0 \land \\
& i_0 = 0 \land \\
& a_1 = a_0 + 1 \land \\
& r_1 = max_0 + r_0 / a_1 \land \\
& i_1 = i_0 + 1 \land \\
& r_2 = (i_0 < max_0) \, ? \, r_1 : r_0 \land \\
& r_3 = r_2 * 2 \land
\end{aligned}
$$

$$P := \; a_1 \mathrel{!=} 0$$

Conjunctive Normal Form

$M_1 := (r_0 = 0) \wedge (i_0 = 0) \wedge (a_1 = a_0 + 1) \wedge (r_1 = max_0 + r_0 / a_1) \wedge$
$(i_1 = i_0 + 1) \wedge (r_2 = (i_0 < max_0)\ ?\ r_1 : r_0) \wedge (r_3 = r_2 * 2)$

$\neg P_1 := (a_1 = 0)$

$(M_k \wedge \neg P_k)$ ⟶ **SAT solver** ⟶ **SAT or UNSAT?**

CBMC

Bounded Model Checking for ANSI-C programs

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

CBMC calls an assertion generator (***goto-instrument***) to add assertions in the code to verify these properties

CBMC

How to use CBMC to Automated Test Generation?

CBMC [4]

1º - Assign nondeterminist values to the input variables (use the CBMC functions with prefix nondet_)

2º - add assertions

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

3º - run CBMC

```
$ cbmc file.c -D ASSERTION_1
```

CBMC

```
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
```

CBMC

```
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) || (y == 1)` as TRUE?

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;
```

```
Generic Property Instrumentation
Starting Bounded Model Checking
Unwinding loop c::func.0 iteration 1 file func.c line 5
function func
size of program expression: 38 assignments
simple slicing removed 11 assignments
Generated 1 VCC(s), 1 remaining after simplification
Passing problem to propositional reduction
Running propositional reduction
Solving with MiniSAT2 with simplifier
532 variables, 800 clauses
SAT checker: negated claim is SATISFIABLE, i.e., does
not hold
Runtime decision procedure: 0.003s
Building error trace
(...)
----------------------------------------
x=-1073741824 (11000000000000000000000000000000)
----------------------------------------
y=1 (00000000000000000000000000000001)
----------------------------------------
a=0 (00000000000000000000000000000000)
```

Test case

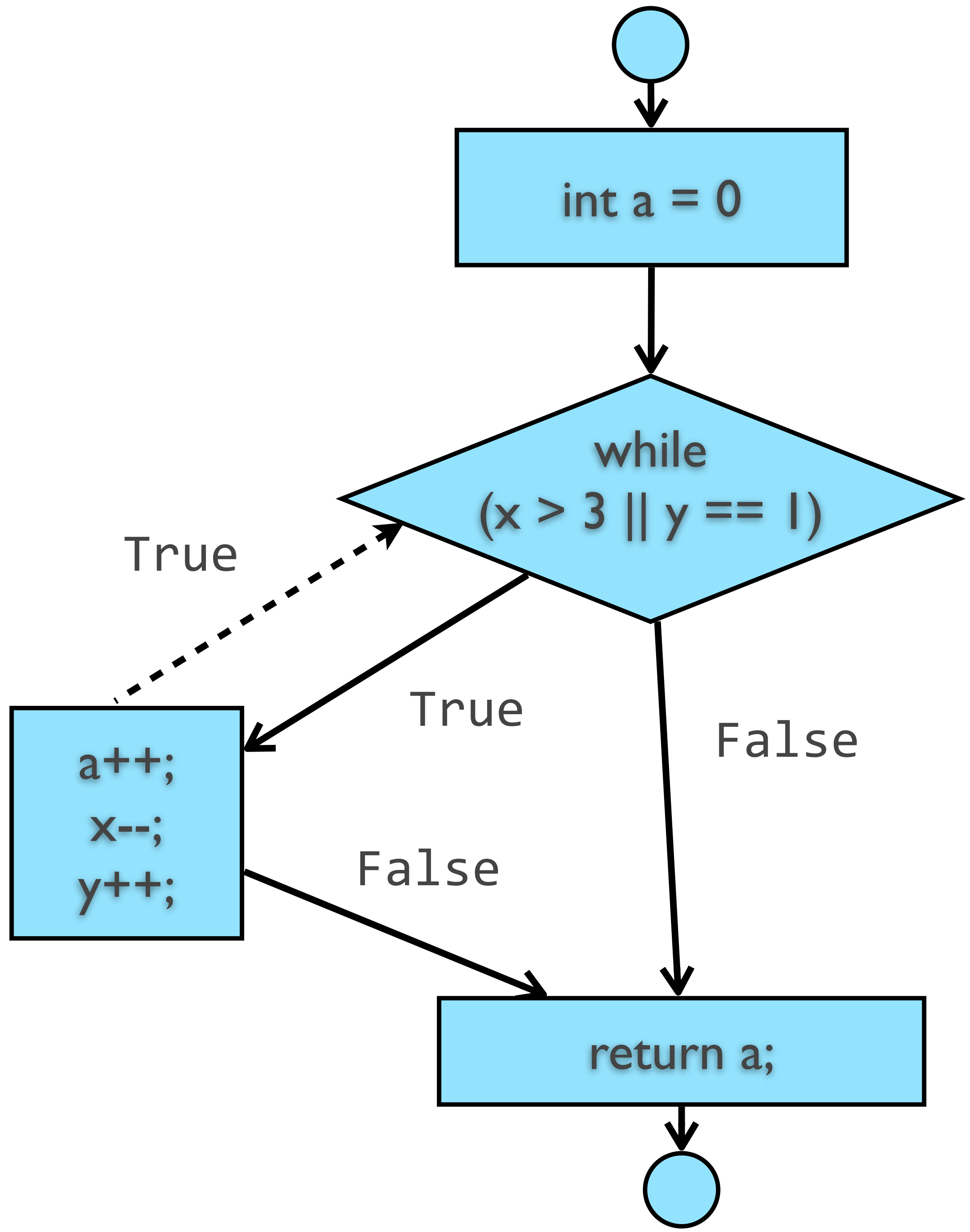
(x = -1073741824, y = 1)

CBMC and MC/DC

How to use CBMC to Automated Test Generation and achieve MC/DC?

Control Flow Graph

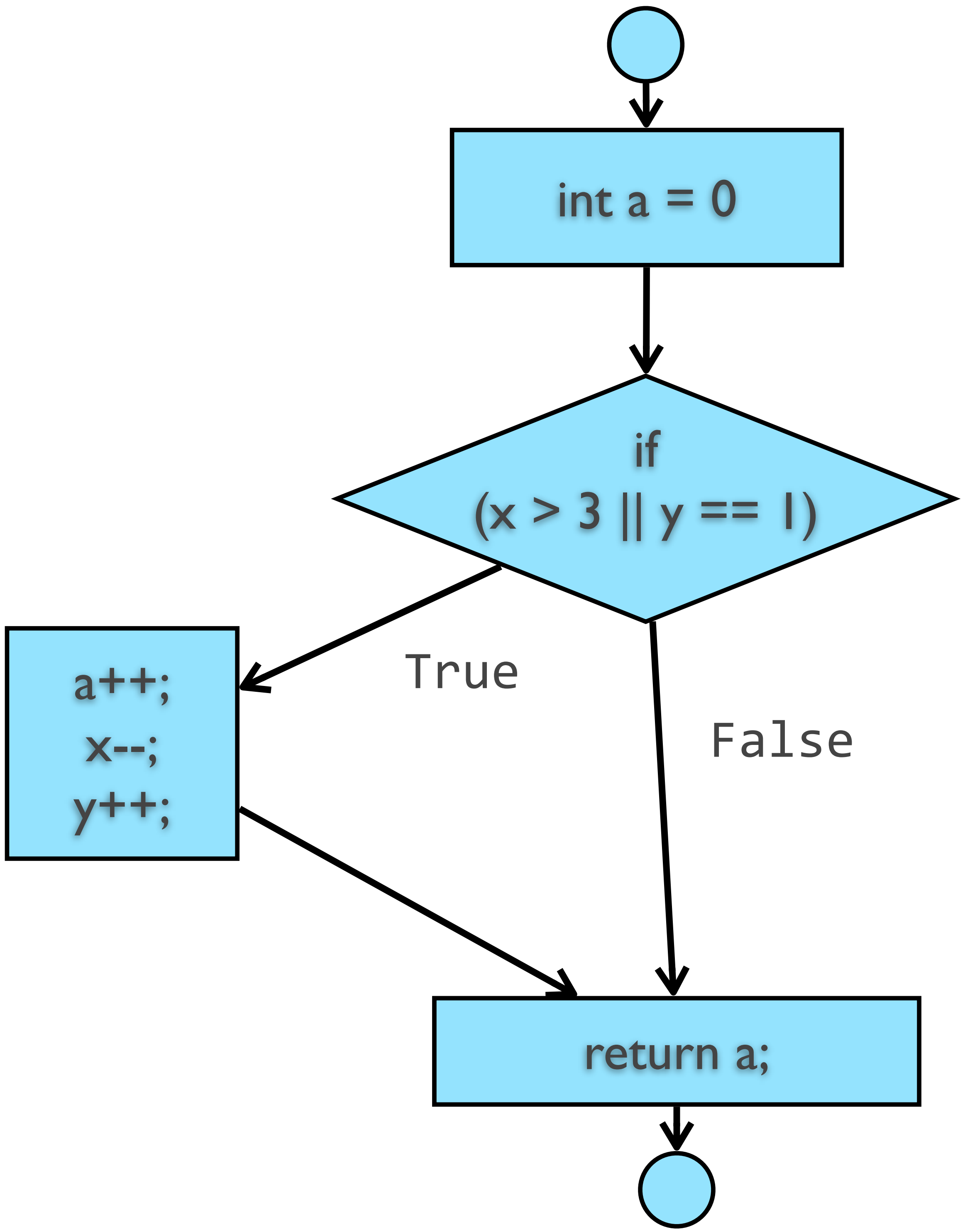
```
int func(int x, int y) {
  int a = 0;
  while (x > 3 || y == 1)
  {
    a++;
    x--;
    y++;
  }
  return a;
```



Control Flow Graph

unwind k = 1

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



CBMC and MC/DC

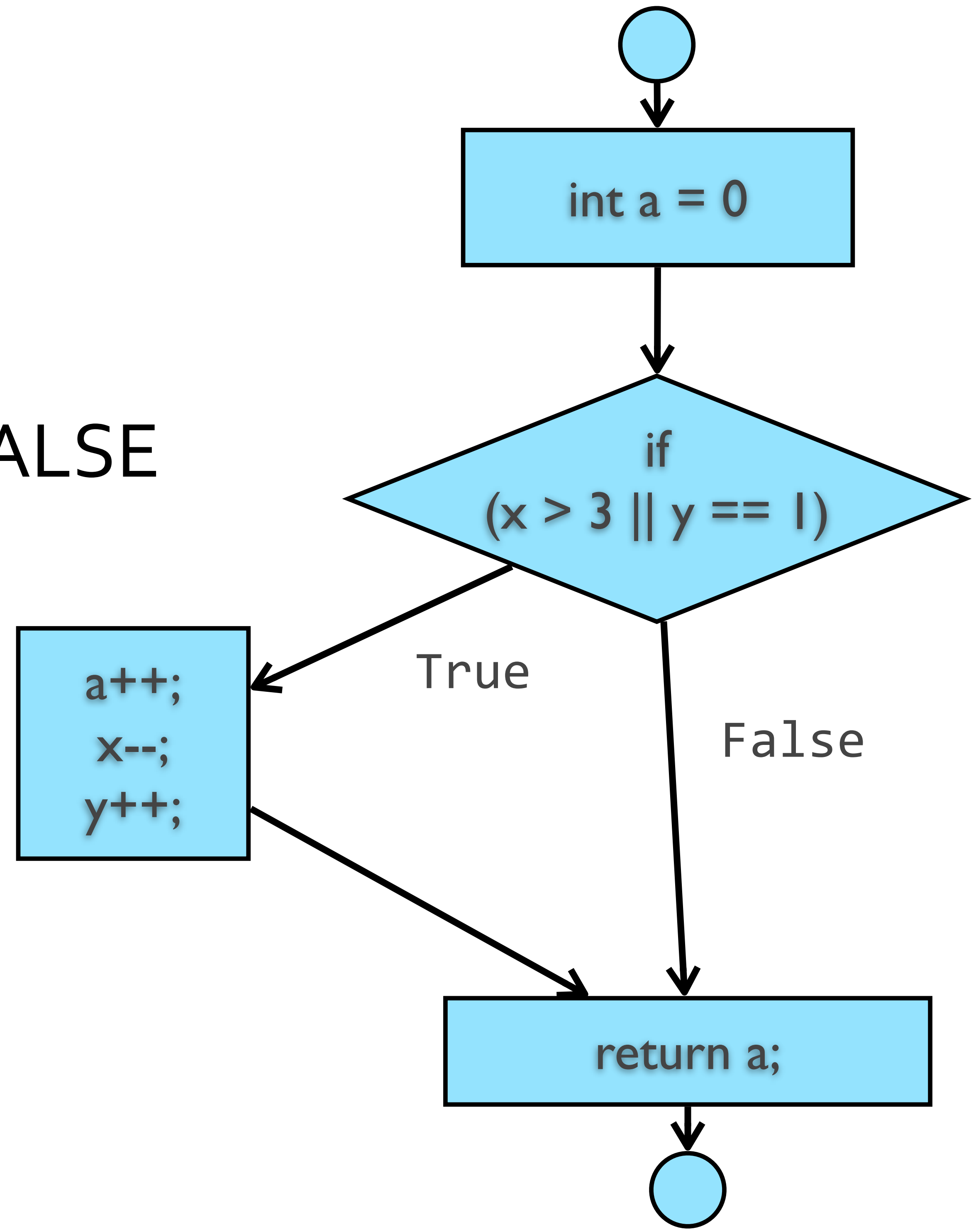
MC/DC requires:

`x > 3` : TRUE and FALSE

`y == 1` : TRUE and FALSE

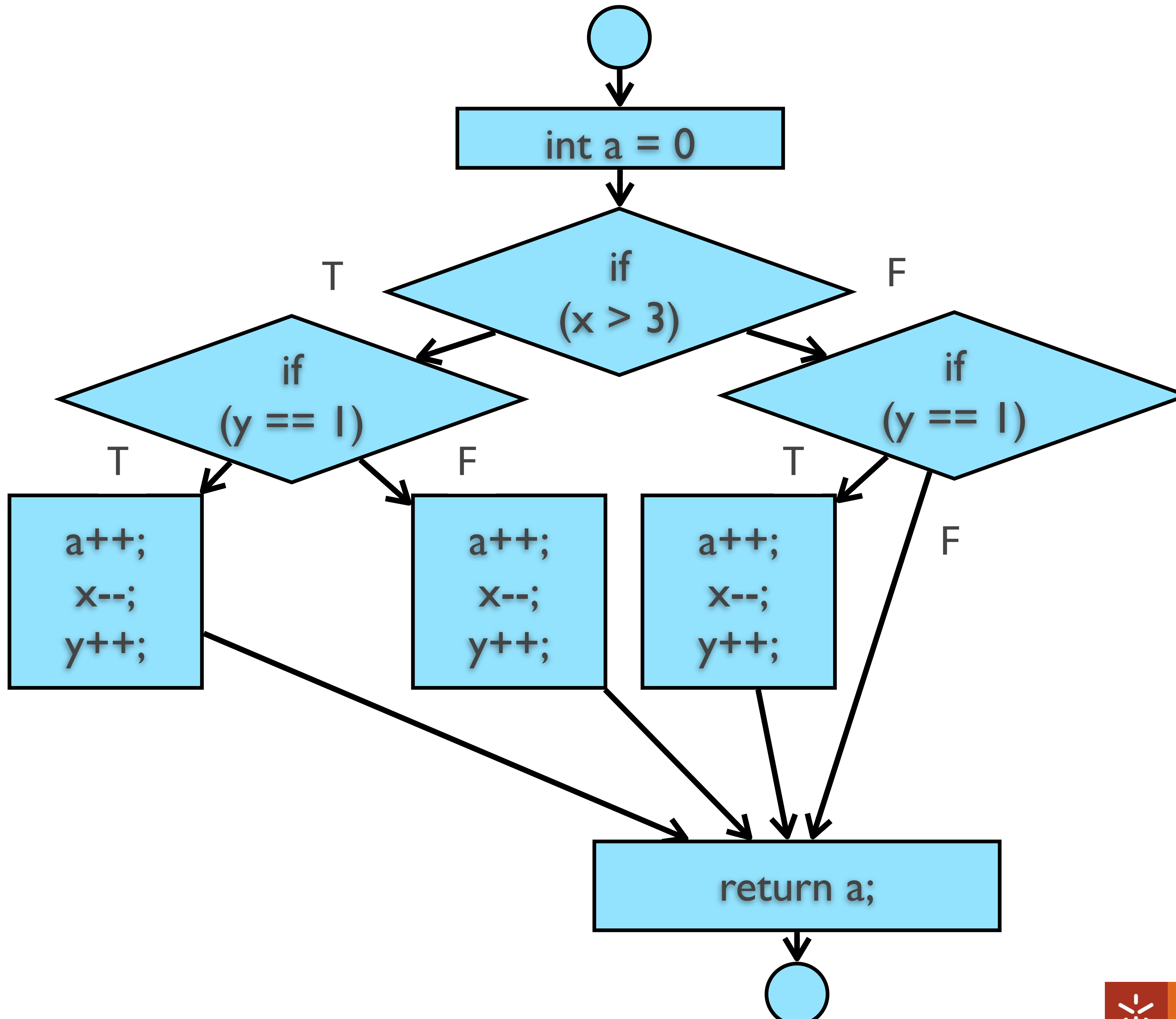
`x > 3 || y == 1`: TRUE and FALSE

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



CBMC and MC/DC

```
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

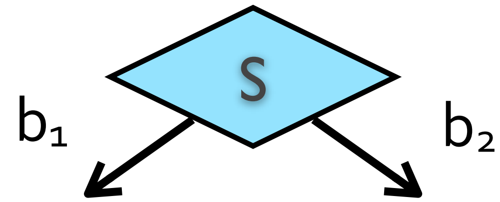
100% MC/DC

CBMC and MC/DC

How to use CBMC to Automated Test Generation and achieve MC/DC without redundant test cases?

CBMC effectively [5]

Consider the branches from **if** statements nodes



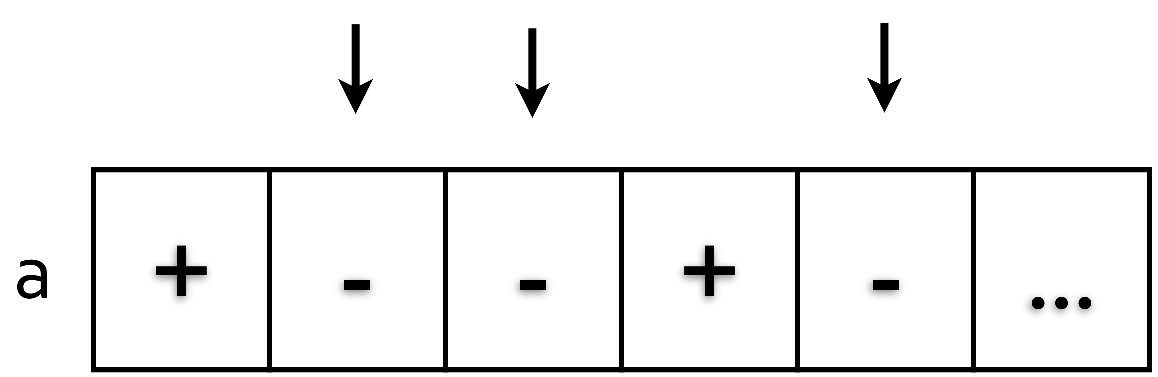
The algorithm builds paths that execute each branch only once.

CBMC effectively

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

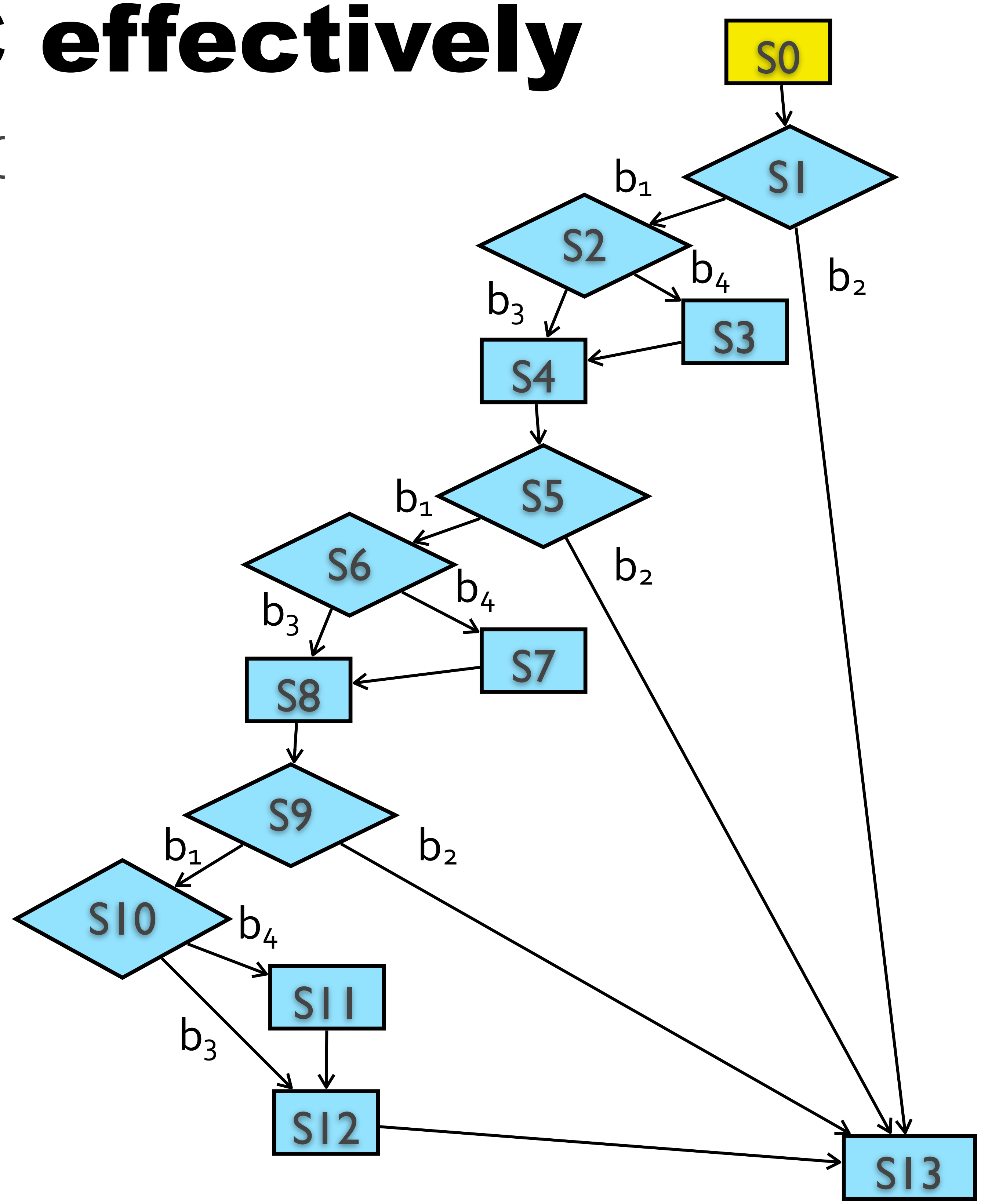
k = 3 →

```
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;
}
```



CBMC effectively

```
  int test(int a[], int size) {
S0:  int negatives = 0, i = 0;
S1:  if (i < size) { b1 b2
S2:    if (a[i] < 0) b3 b4
S3:      negatives++;
S4:     i++;
S5:     if (i < size) {  b1 b2
S6:       if (a[i] < 0)  b3 b4
S7:         negatives++;
S8:       i++;
S9:       if (i < size) {  b1 b2
S10:         if (a[i] < 0) b3 b4
S11:           negatives++;
S12:         i++;
         }
       }
     }
S13:  return negatives;
}
```



CBMC effectively

Path = {S0,S1}

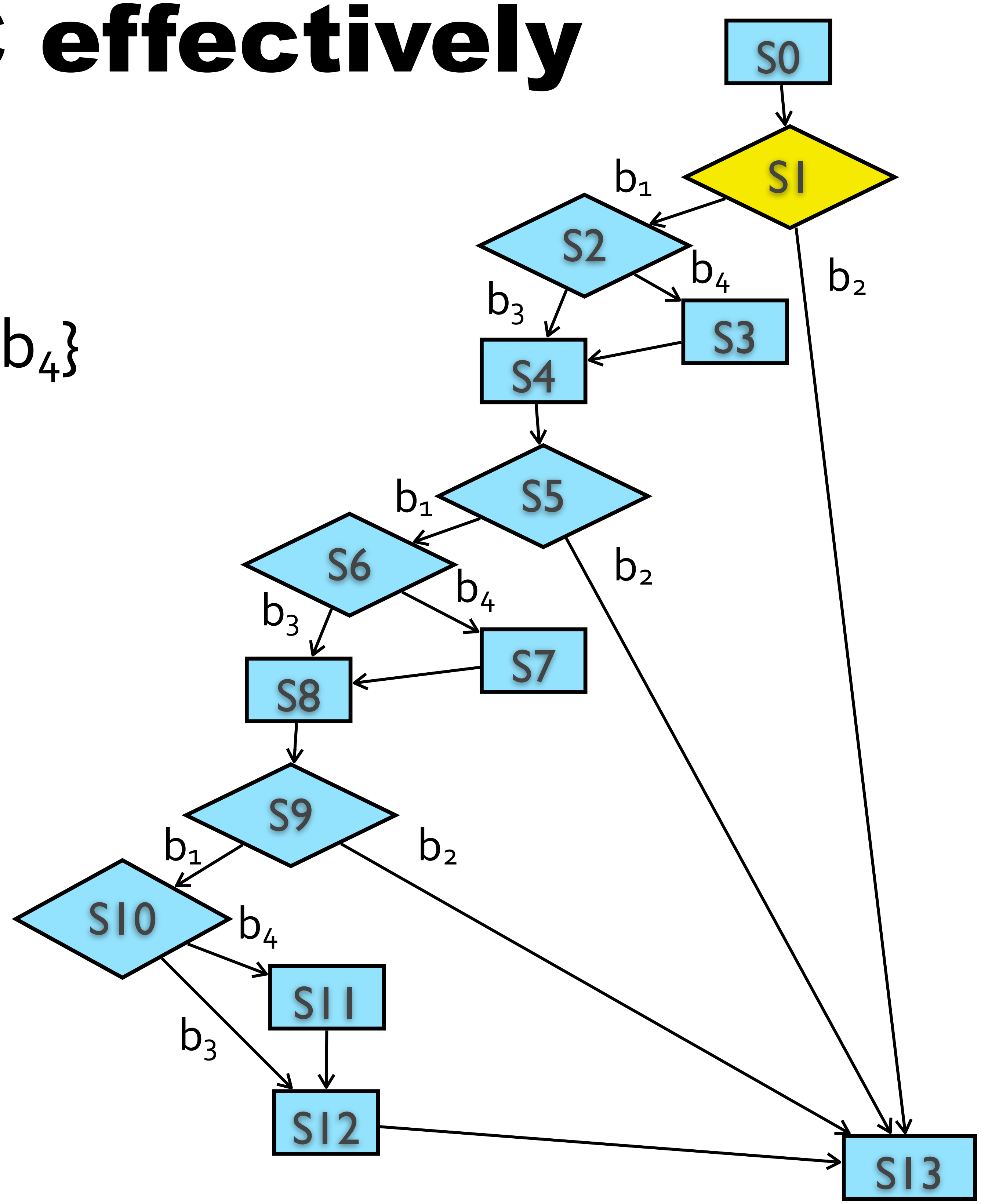
Branches to find = {b_1,b_2,b_3,b_4}

Succ of S1? S2 and S13

Which one to choose?

The one that has the higher number of branches to find

S2 -> {b_1,b_2,b_3,b_4}
S13 -> {}



CBMC effectively

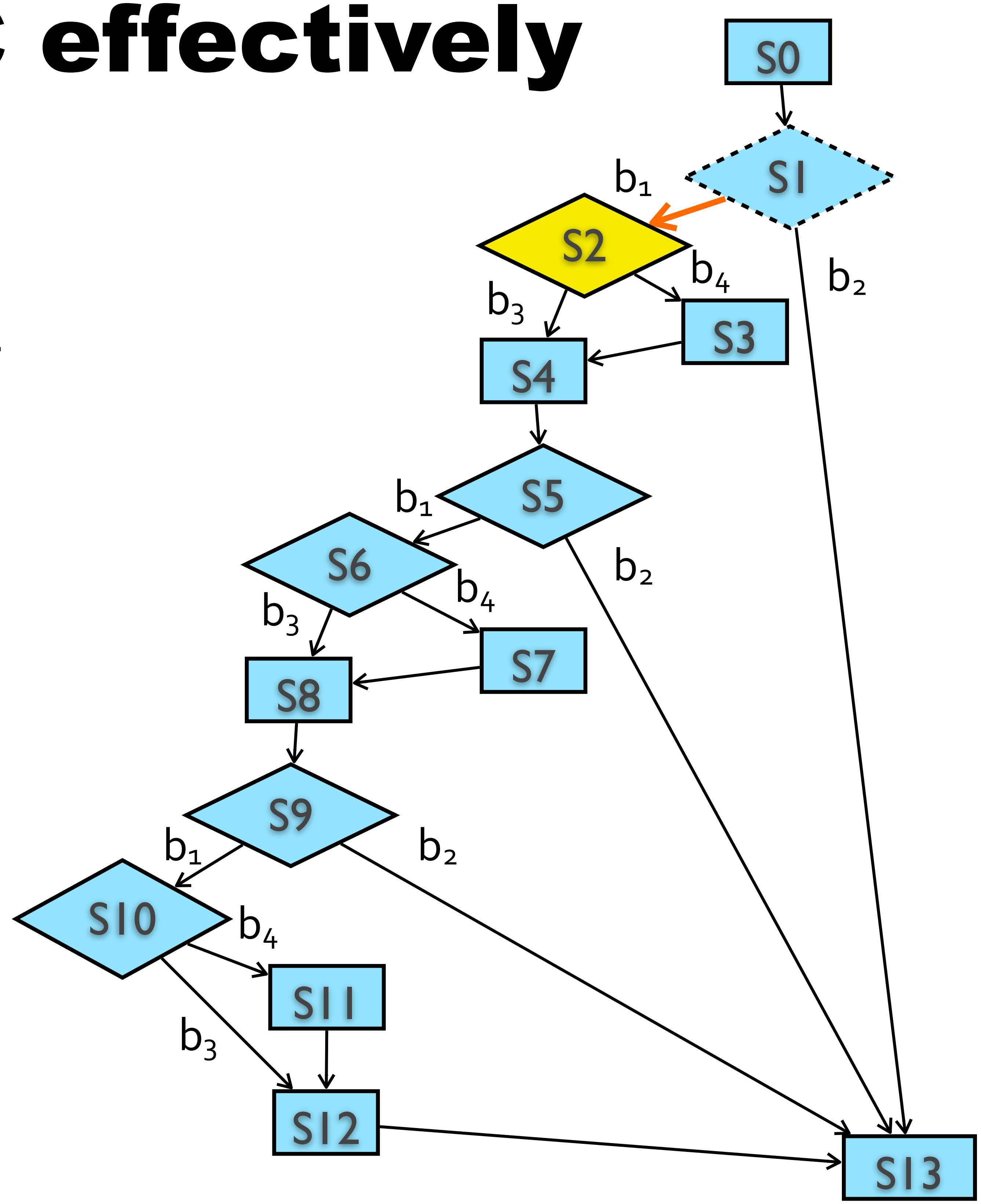
Path = {S0,S1,S2}

Branches to find = {b_2,b_3,b_4}

Succ of S2? S3 and S4

Which one to choose?

if the number of branches to find is the same, choose in lexicograph order



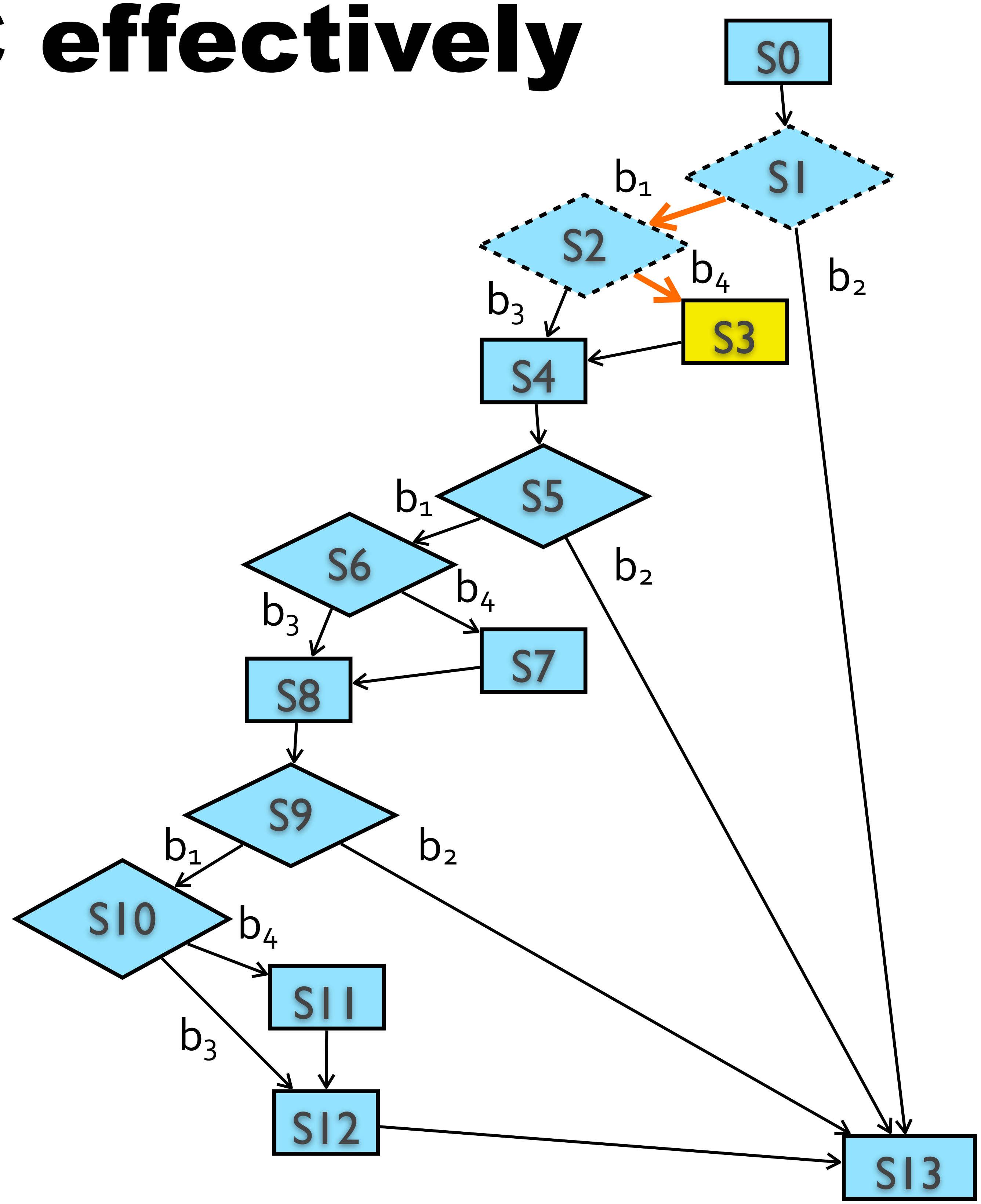
CBMC effectively

Path = {S0,S1,S2,S3}

Branches to find = {b_2,b_3}

Succ of S3? S4

Succ of S4? S5



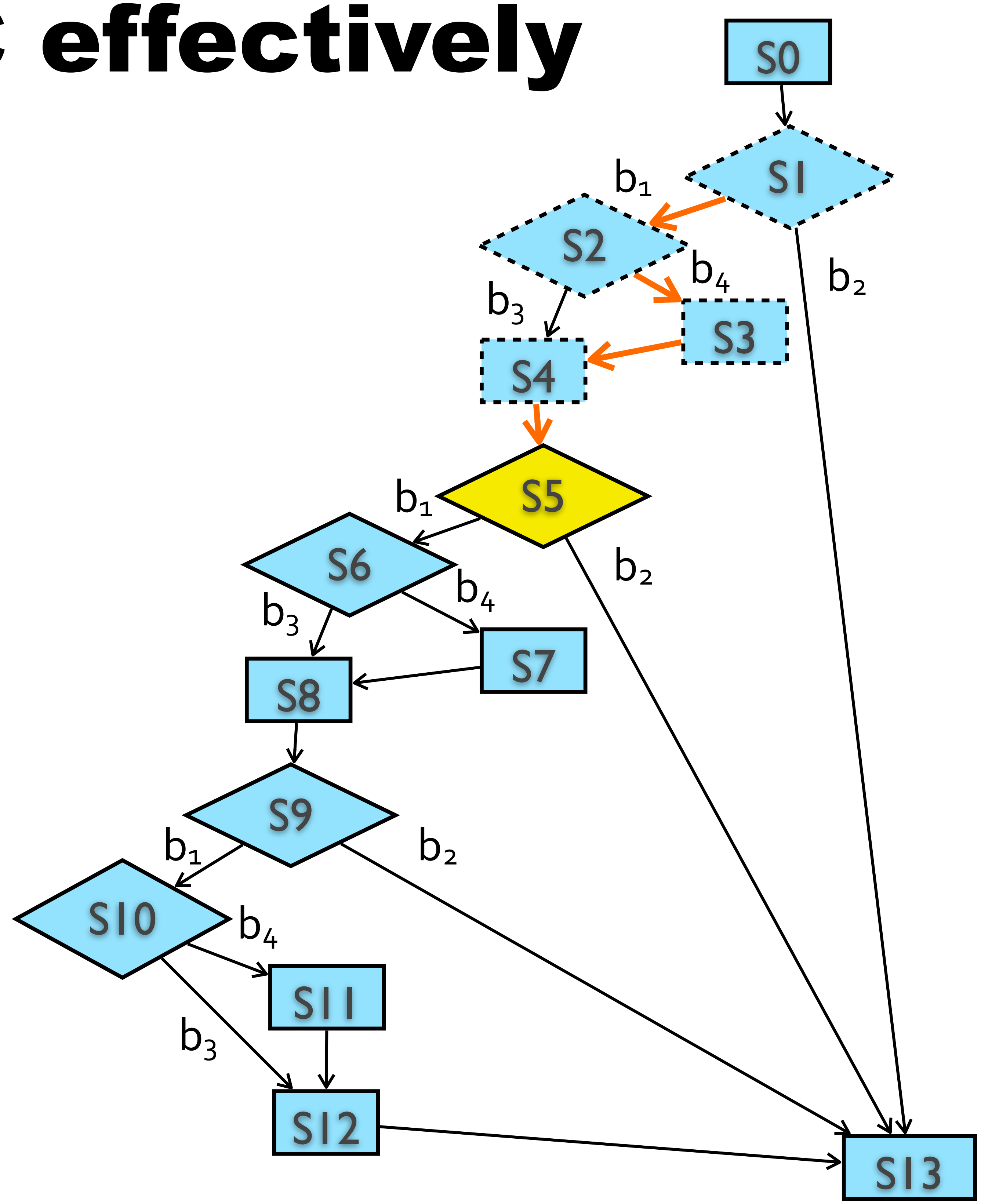
CBMC effectively

Path = {S0,S1,S2,S3,S4,S5}

Branches to find = {b_2,b_3}

Succ of S5? S6 and S13

S6 -> {b_2,b_3}
S13 -> {}



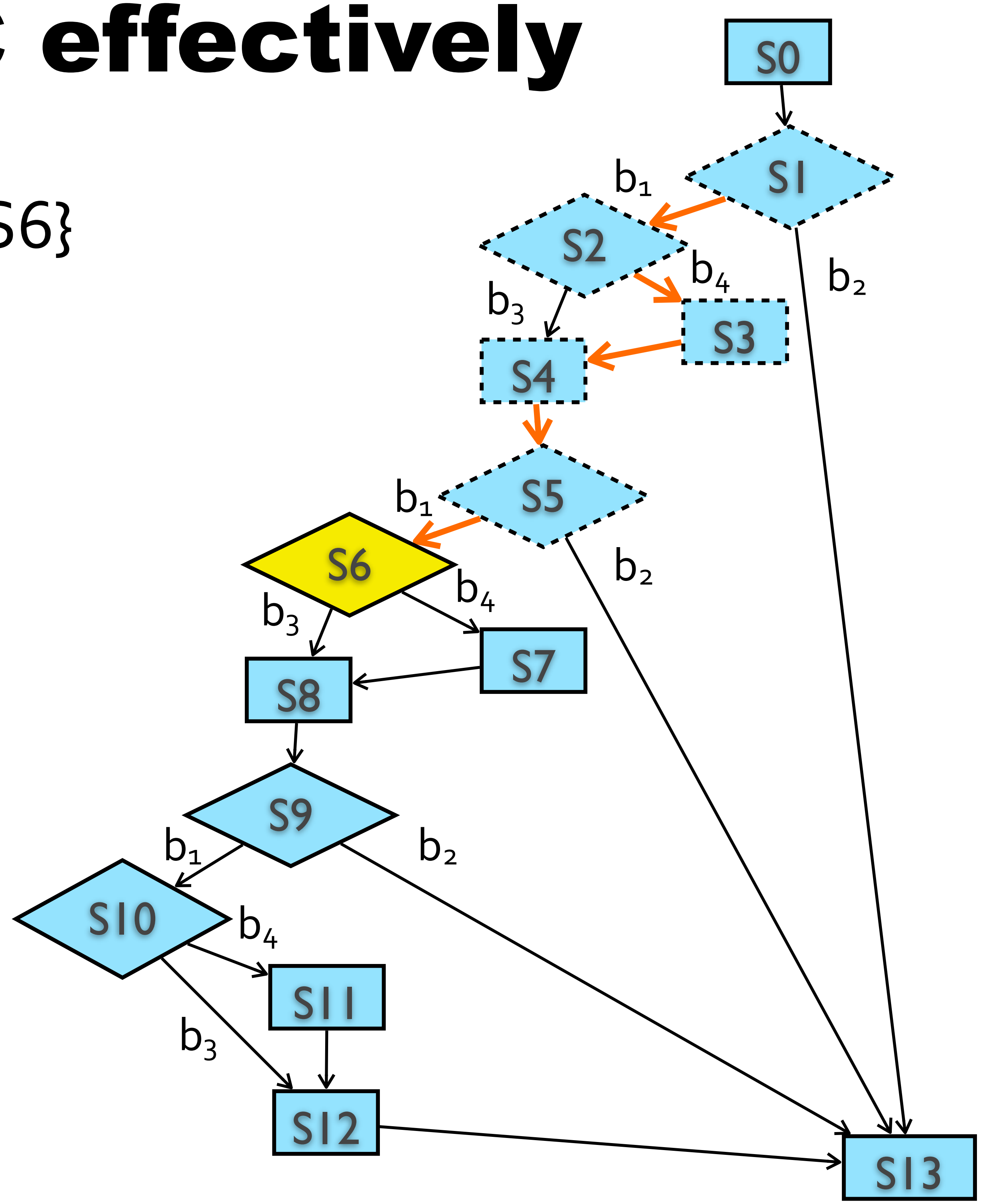
CBMC effectively

Path = {S0,S1,S2,S3,S4,S5,S6}

Branches to find = {b_2,b_3}

Succ of S6? S7 and S8

S7 -> {}
S8 -> {b_3}



CBMC effectively

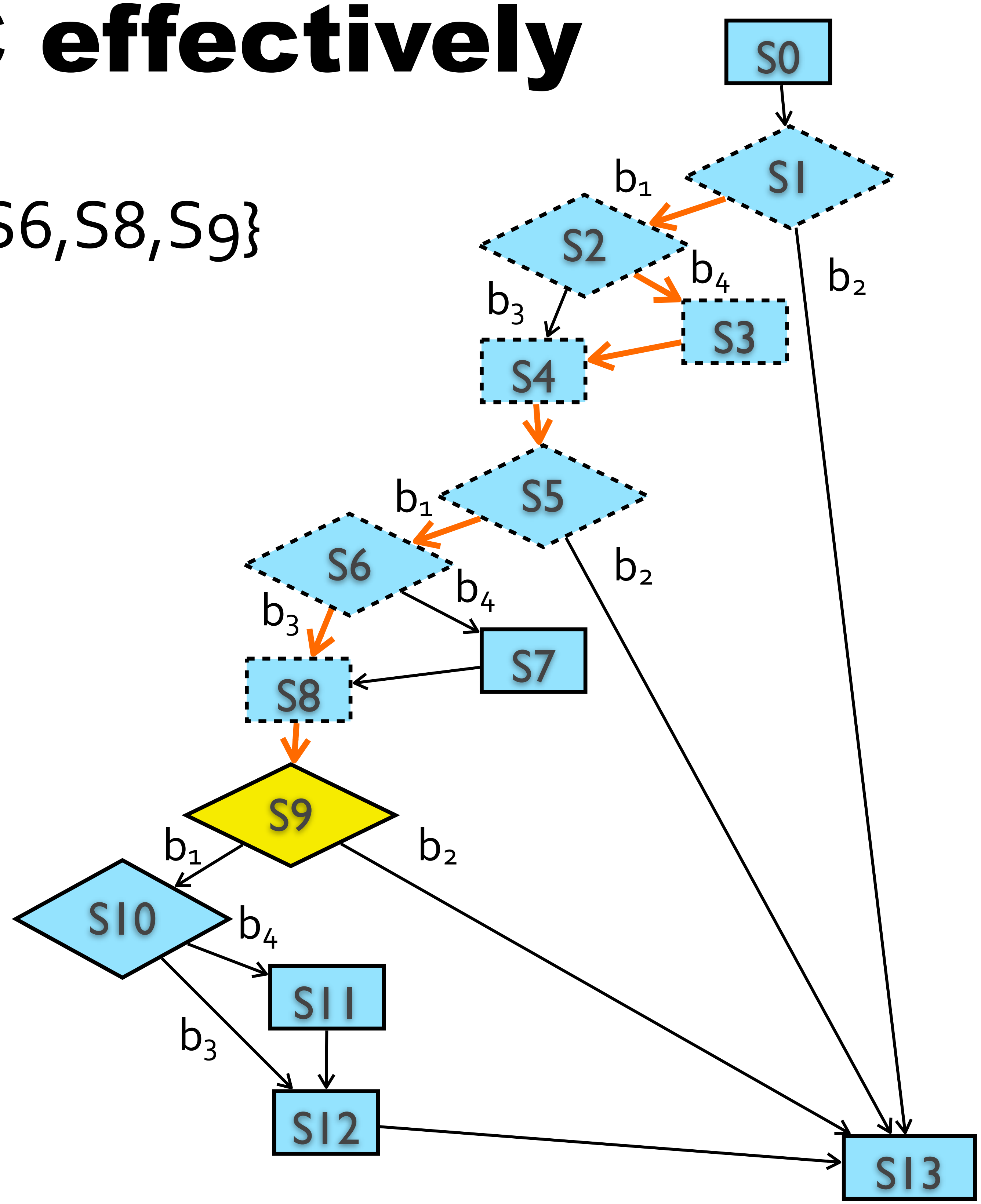
Path = {S0,S1,S2,S3,S4,S5,S6,S8,S9}

Branches to find = {b_2}

Succ of S9? S10 and S13

S10 -> {}
S13 -> {}

but b_1 was already found!!



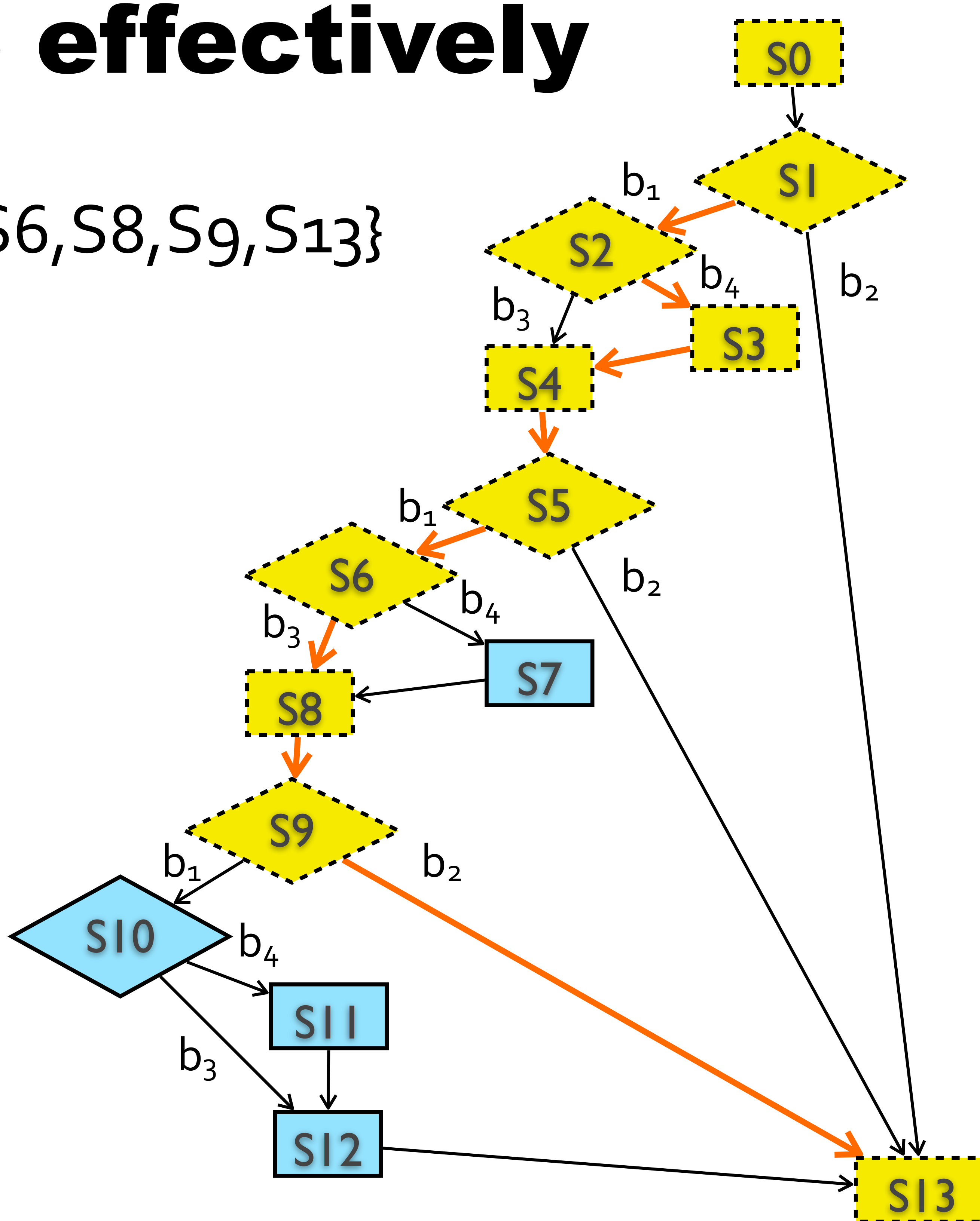
CBMC effectively

Path = {S0,S1,S2,S3,S4,S5,S6,S8,S9,S13}

Branches to find = {}

All branches found!
Algorithm is finished!

How may paths?
One was enough to cover all branches.



CBMC effectively

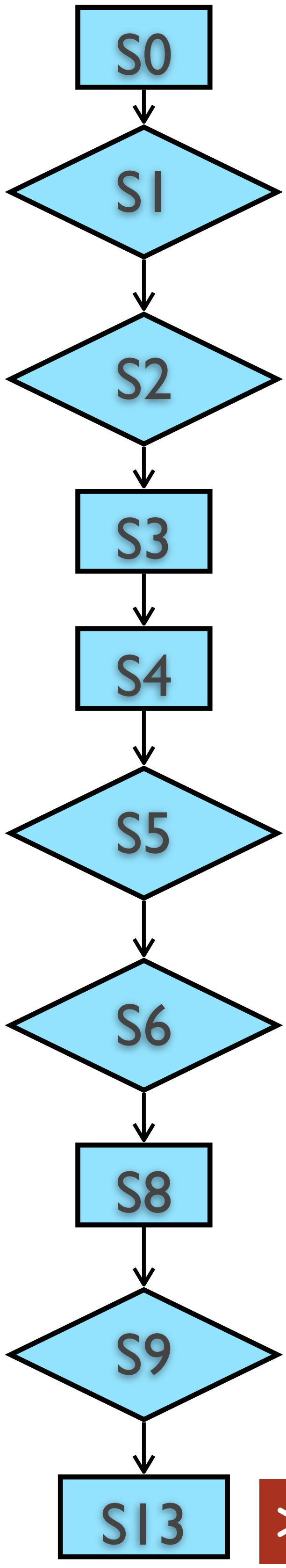
Intrument the code:

- No branch points;

- Force CBMC to go through that path.

- Insert __CPROVER_*assume*

```
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;
}
```



CBMC effectively

```
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:

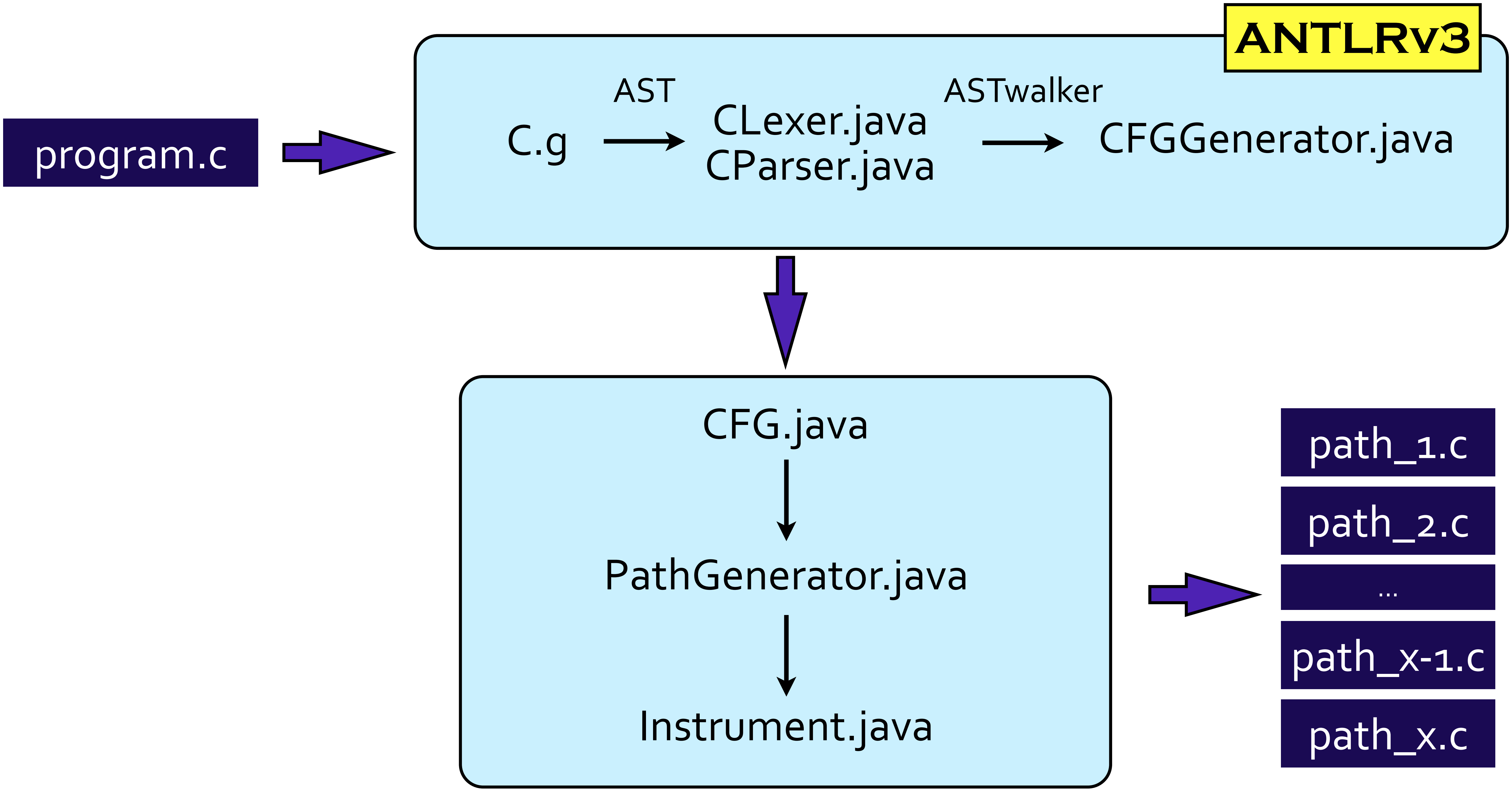
T = (`size=2,`
`a[0]=-2147483648,`
`a[1]=0`)

a		
	-2147483648	0

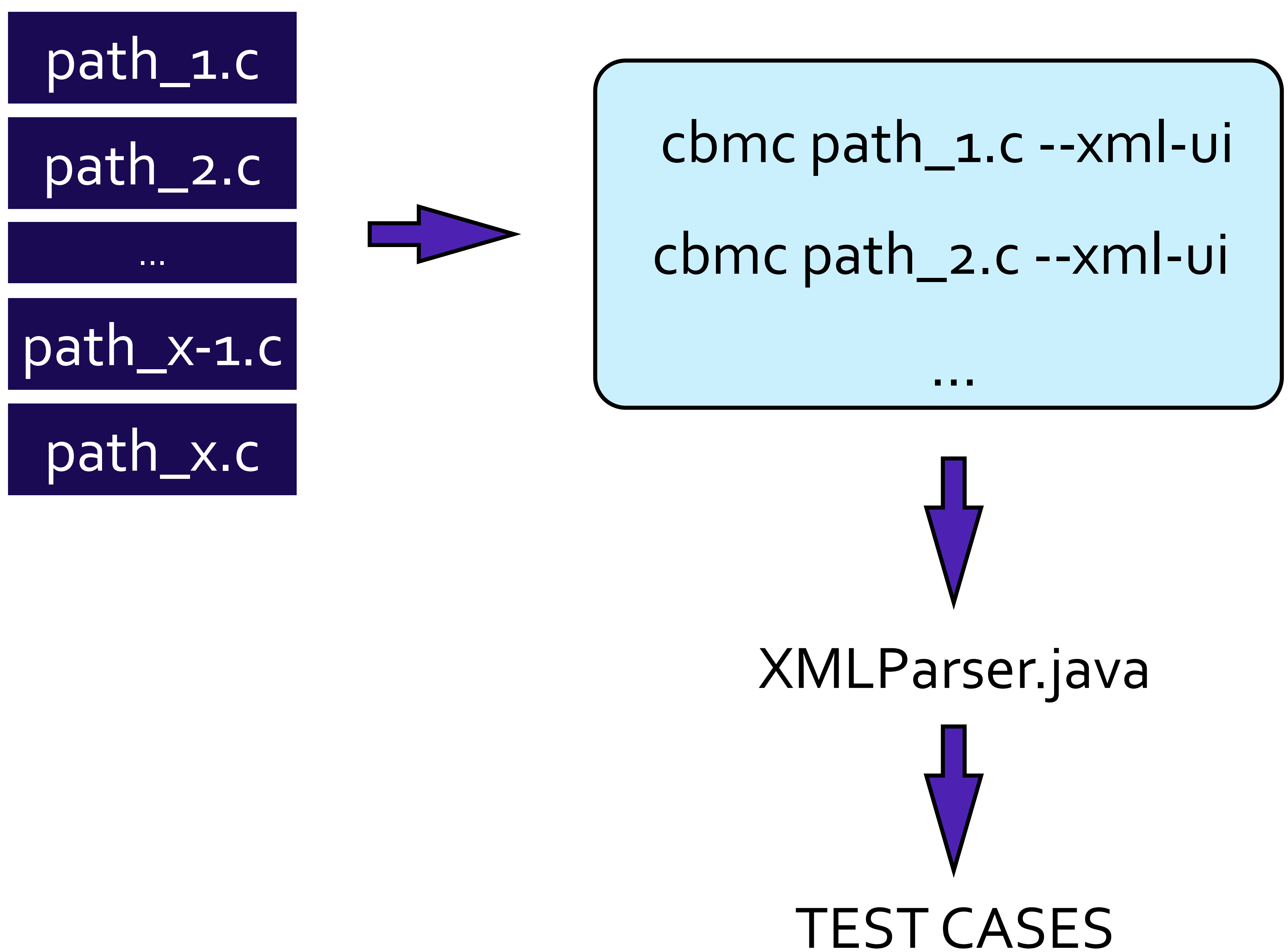
Goals

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

CBMCe


ANTLRv3
program.c
C.g
AST
CLexer.java
CParser.java
ASTwalker
CFGGenerator.java
CFG.java
PathGenerator.java
Instrument.java
path_1.c
path_2.c
...
path_x-1.c
path_x.c

CBMCe


path_1.c
path_2.c
...
path_x-1.c
path_x.c
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

[1] John Rushby. *Automated Test Generation and Verified Software,* Springer-Verlag 2008.

[2] Kelly J Hayhurst, Dan S Veerhusen, John J Chilenski, and Leanna K Rierson. *A practical tutorial on modified condition/decision coverage. Management*, NASA 2001.

[3] Concolic Testing: http://srl.cs.berkeley.edu/~ksen/ (Dec 2012)

[4] Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. *Using bounded model checking for coverage analysis of safety-critical software in an industrial setting*. J. Autom. Reason, December 2010.

[5] Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, Gabriele Palma, and Salvatore Sabina. *Improving the automatic test generation process for cover- age analysis using cbmc*. In Proceedings of the 16th International RCRA workshop, RCRA 2009, 2009.

Automated Test Generation using CBMC

Rui Gonçalo

CROSS Project
Computer Science Department
University of Minho

December 2012