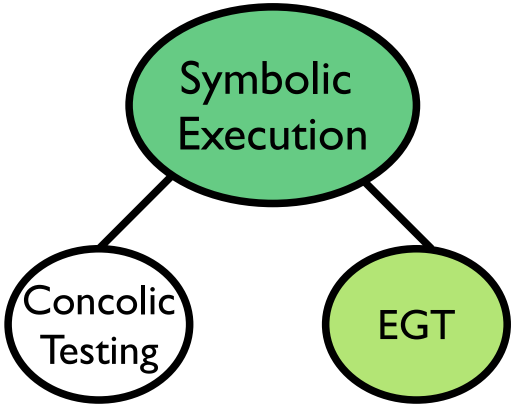
SWE3002-42: Introduction to Software Engineering

Lecture 9 – Software Testing (2)

Sooyoung Cha

Department of Computer Science and Engineering



Today's Lecture

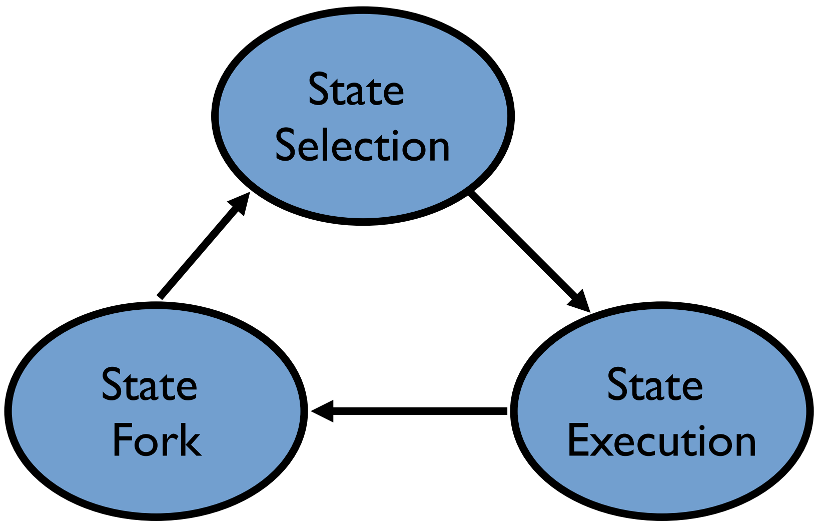
1. Execution-Generated Testing

2. Open Challenges in Symbolic Execution

3. A Key Technique for Symbolic Execution

Execution-Generated Testing

- Another major flavor of dynamic symbolic execution
  - Iteratively selects, executes, and forks a state while maintaining a set of states during its testing process.




```
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```

Reaching the Error location!

Execution-Generated Testing

- A state *S* consists a tuple (*instr, store, PC*)
  - *instr: The next instruction to be executed*
  - *store: A symbolic store which maps program variables into symbolic values*
  - *PC: A* conjunction of symbolic branch conditions

```c
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```

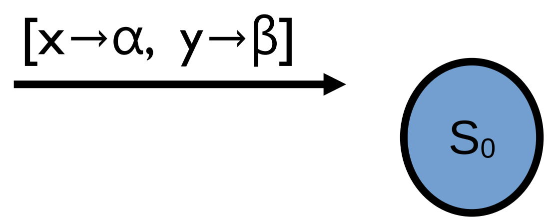
An initial state S_0 = (*$instr_0$, $store_0$, PC*)

- $instr_0$= *if(x==2y)*
- *$store_0$=[$x \rightarrow \alpha$, $y \rightarrow \beta$]*
- *PC = true*

Execution-Generated Testing

- How to generate the test-case reaching 'Error' ?

```c
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```



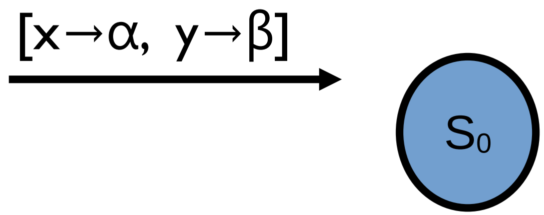
S_0 = (*if(x==2y), [x→α, y→β], true*)

- Candidate States

Execution-Generated Testing

- How to generate the test-case reaching 'Error' ?

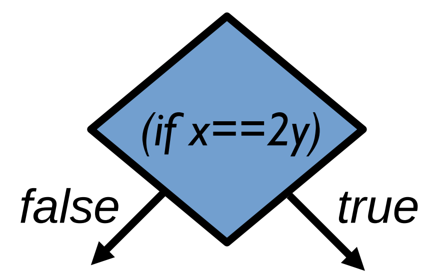
```
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```



S_0 = ((*if x==2y*), [*x*→*α*, *y*→*β*], *true*)

if(instr == if/else statement)

- Candidate States



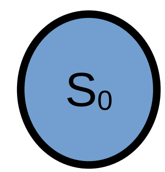
"Both true and false branch conditions are satisfiable?"

Execution-Generated Testing

- How to generate the test-case reaching 'Error' ?

```c
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```

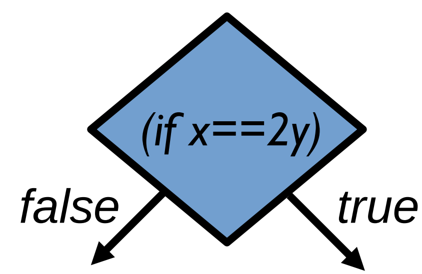
$[x \rightarrow \alpha,\ y \rightarrow \beta]$



S_0 = ((*if x==2y*), *[x→α, y→β]*, *true*)

if(instr == if/else statement)

- Candidate States



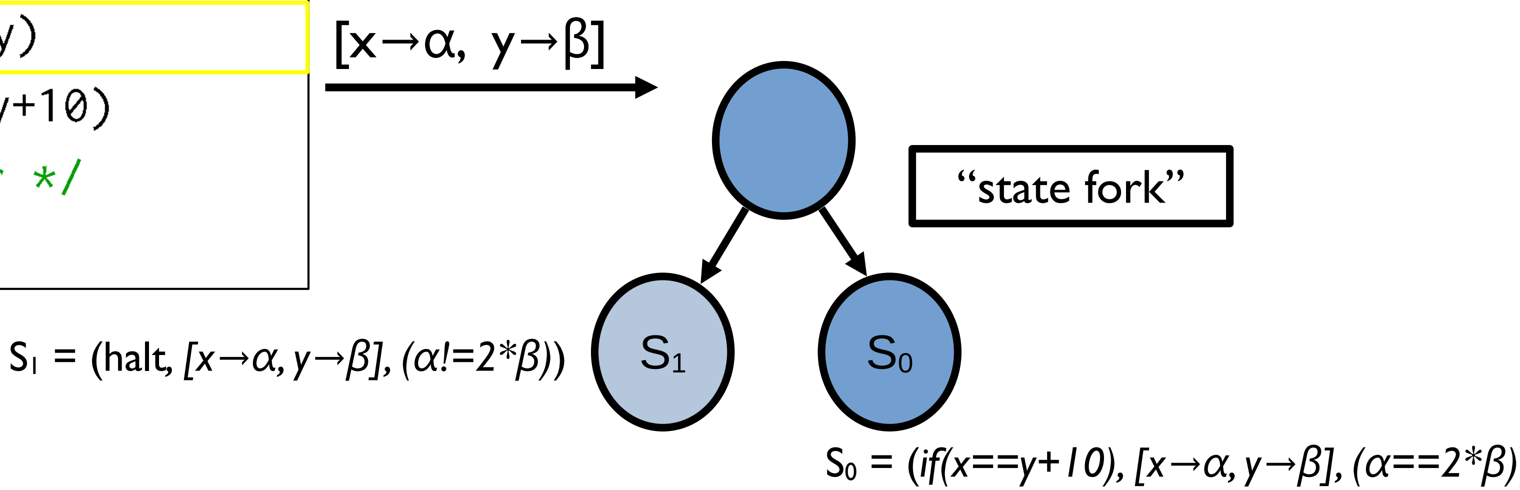
"Both true and false branch conditions are satisfiable?"

- Yes!
  (true when α=2 and β=1)
  (false when α=2 and β=0)

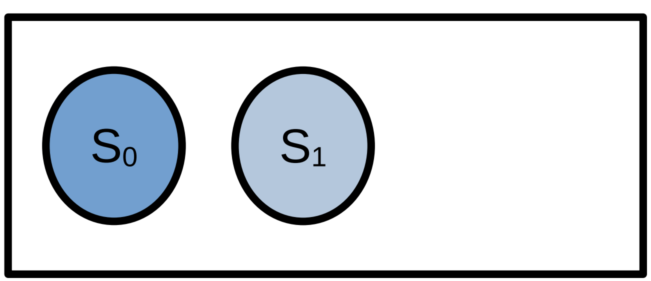
Execution-Generated Testing

- How to generate the test-case reaching 'Error' ?

```c
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```



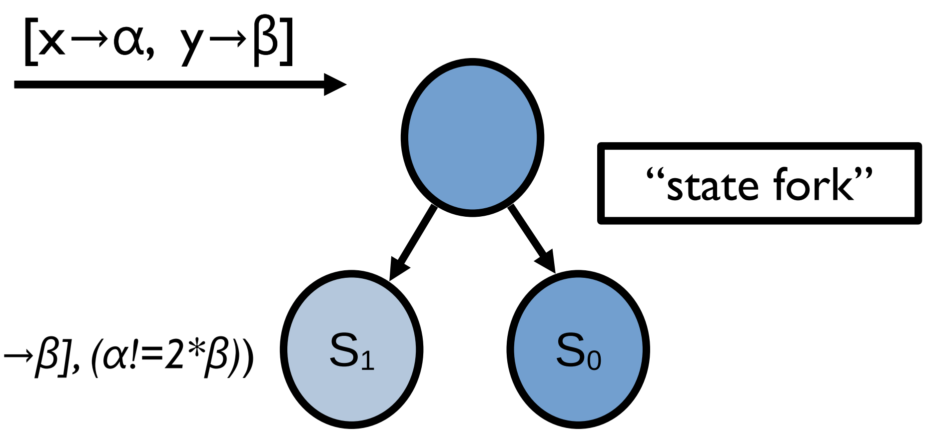
- Candidate States



Execution-Generated Testing

- How to generate the test-case reaching 'Error' ?

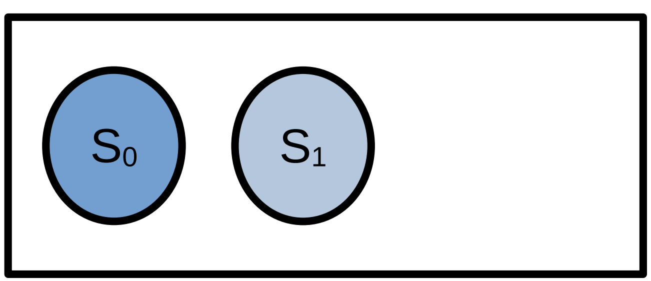
```c
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```



S_1 = (halt, *[x→α, y→β], (α!=2*β)*)

S_0 = (*if(x==y+10)*, *[x→α, y→β], (α==2*β)*)

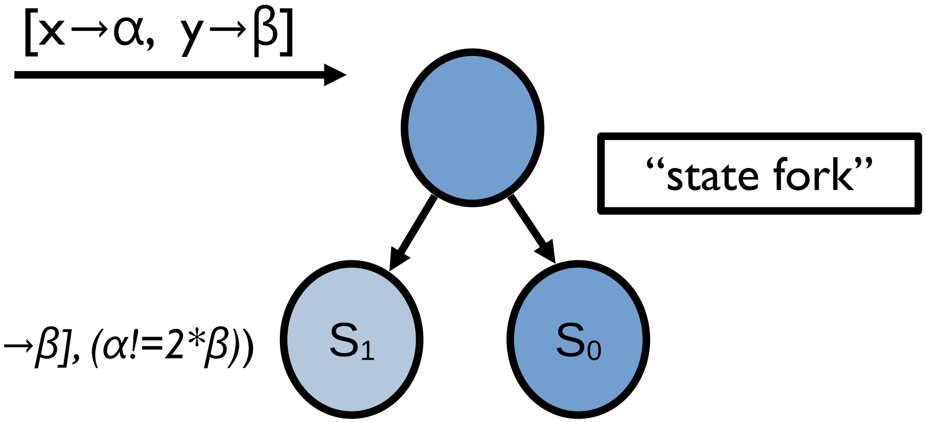
- Candidate States



Execution-Generated Testing

- How to generate the test-case reaching 'Error' ?

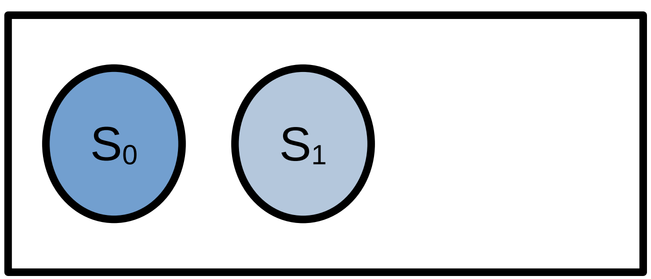
```
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```

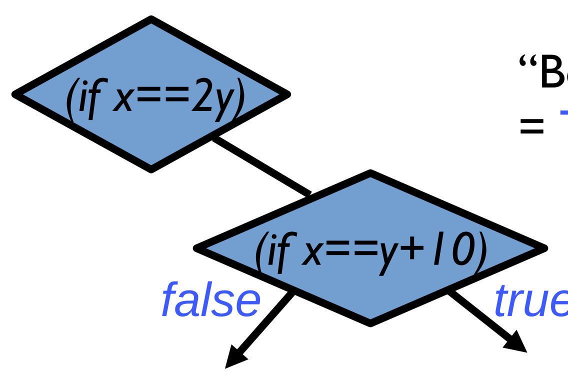


S_1 = (halt, $[x \rightarrow \alpha, y \rightarrow \beta]$, $(\alpha != 2 * \beta)$)

S_0 = ($if(x==y+10)$, $[x \rightarrow \alpha, y \rightarrow \beta]$, $(\alpha == 2 * \beta)$)

- Candidate States





"Both branch conditions are reachable?"
= Two path conditions are satisfiable?

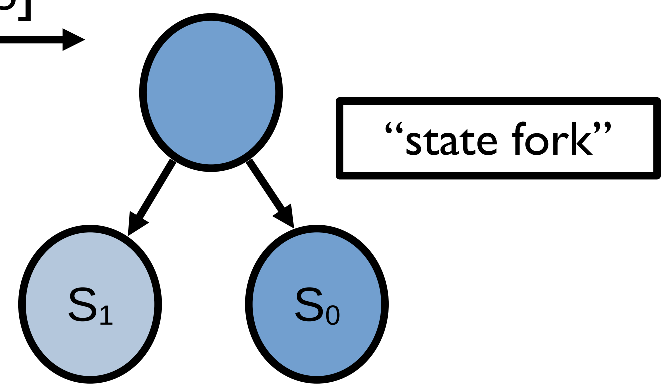
1. $(\alpha == 2\beta) \wedge (\alpha == \beta + 10)$
2. $(\alpha == 2\beta) \wedge (\alpha != \beta + 10)$

Execution-Generated Testing

- How to generate the test-case reaching 'Error' ?

```
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```

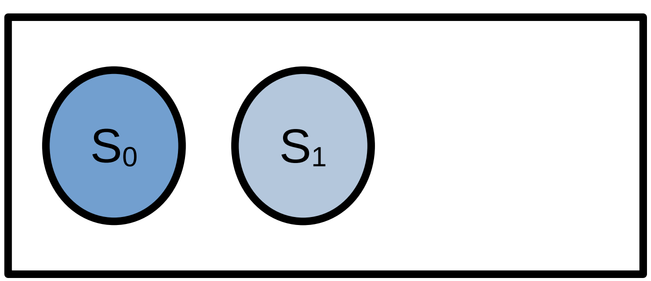
$[x \rightarrow \alpha,\ y \rightarrow \beta]$

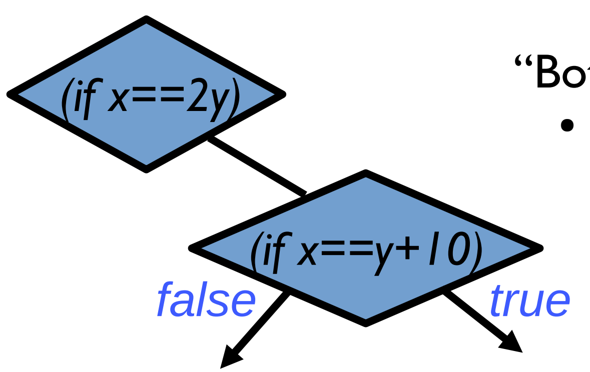


S_1 = (halt, $[x \rightarrow \alpha, y \rightarrow \beta]$, $(\alpha != 2*\beta)$)

S_0 = ($if(x==y+10)$, $[x \rightarrow \alpha, y \rightarrow \beta]$, $(\alpha == 2*\beta)$)

- Candidate States





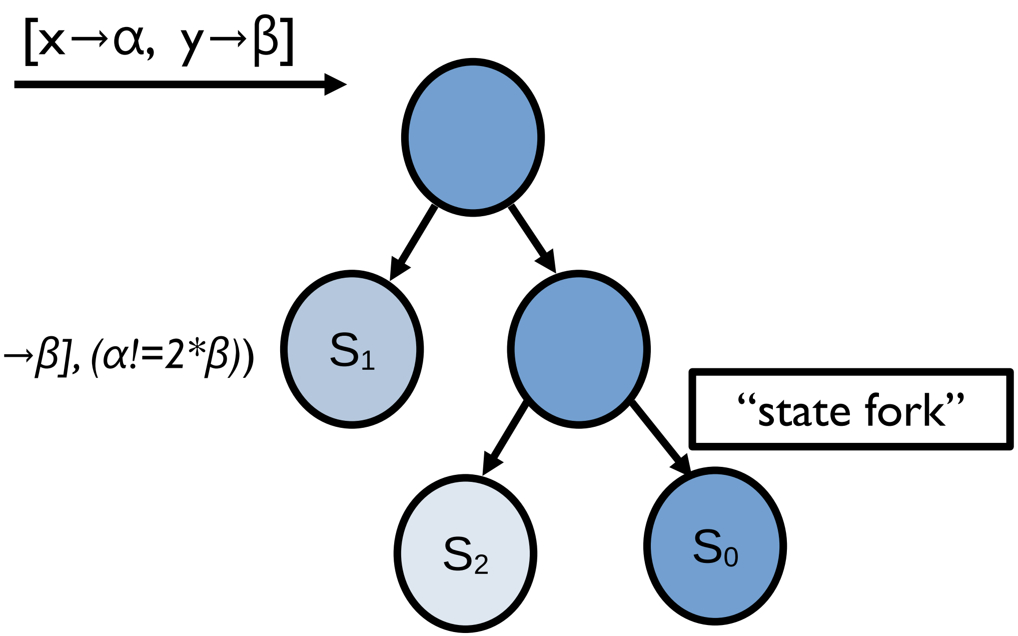
"Both branch conditions are reachable?"

- Yes!
  (true when x=20 and y=10)
  (false when x=10 and y=5)

Execution-Generated Testing

- How to generate the test-case reaching 'Error' ?

```c
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```

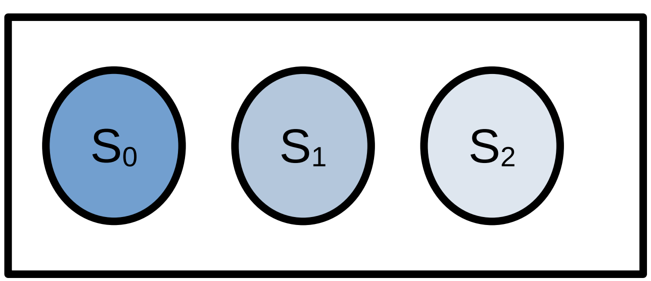


S_1 = (halt, $[x \to \alpha, y \to \beta]$, $(\alpha != 2*\beta)$)

S_2 = (halt, $[x \to \alpha, y \to \beta]$, $(\alpha == 2*\beta) \wedge (\alpha != \beta + 10)$)

S_0 = (Error, $[x \to \alpha, y \to \beta]$, $(\alpha == 2*\beta) \wedge (\alpha == \beta + 10)$)

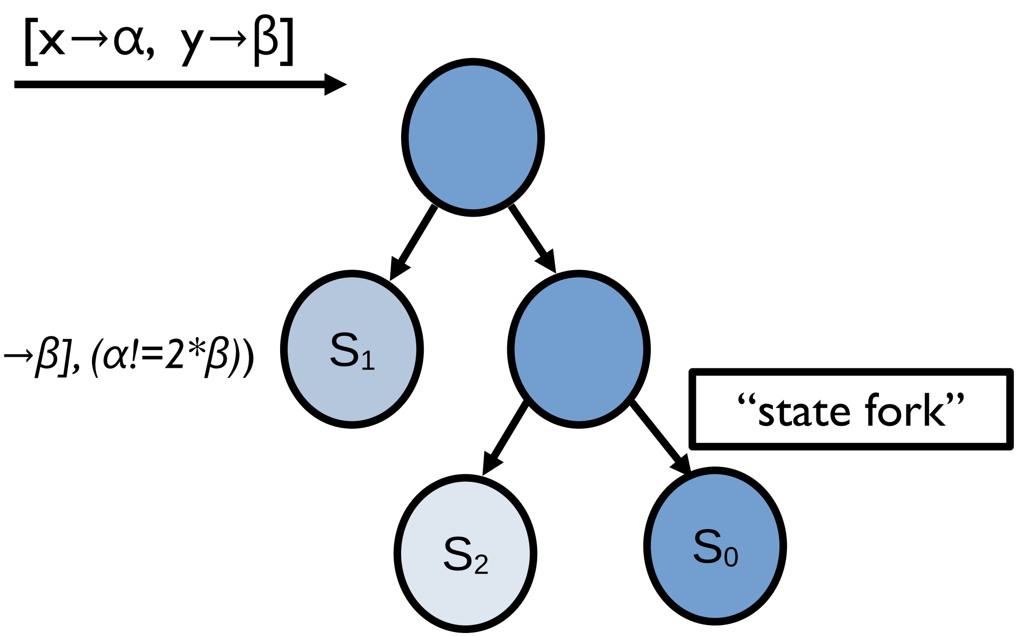
- Candidate States



Execution-Generated Testing

- How to generate the test-case reaching 'Error' ?

```c
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```

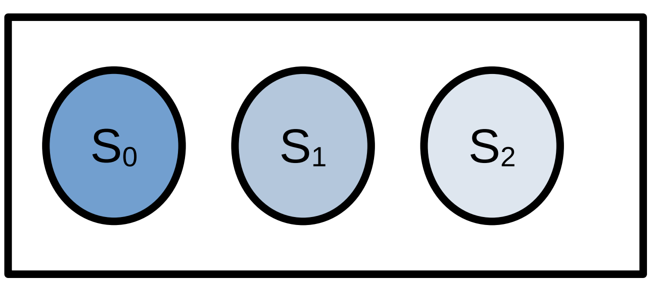


S_1 = (halt, $[x \rightarrow \alpha, y \rightarrow \beta]$, $(\alpha != 2*\beta)$)

S_2 = (halt, $[x \rightarrow \alpha, y \rightarrow \beta]$, $(\alpha == 2*\beta) \wedge (\alpha != \beta + 10)$)

S_0 = (Error, $[x \rightarrow \alpha, y \rightarrow \beta]$, $(\alpha == 2*\beta) \wedge (\alpha == \beta + 10)$)

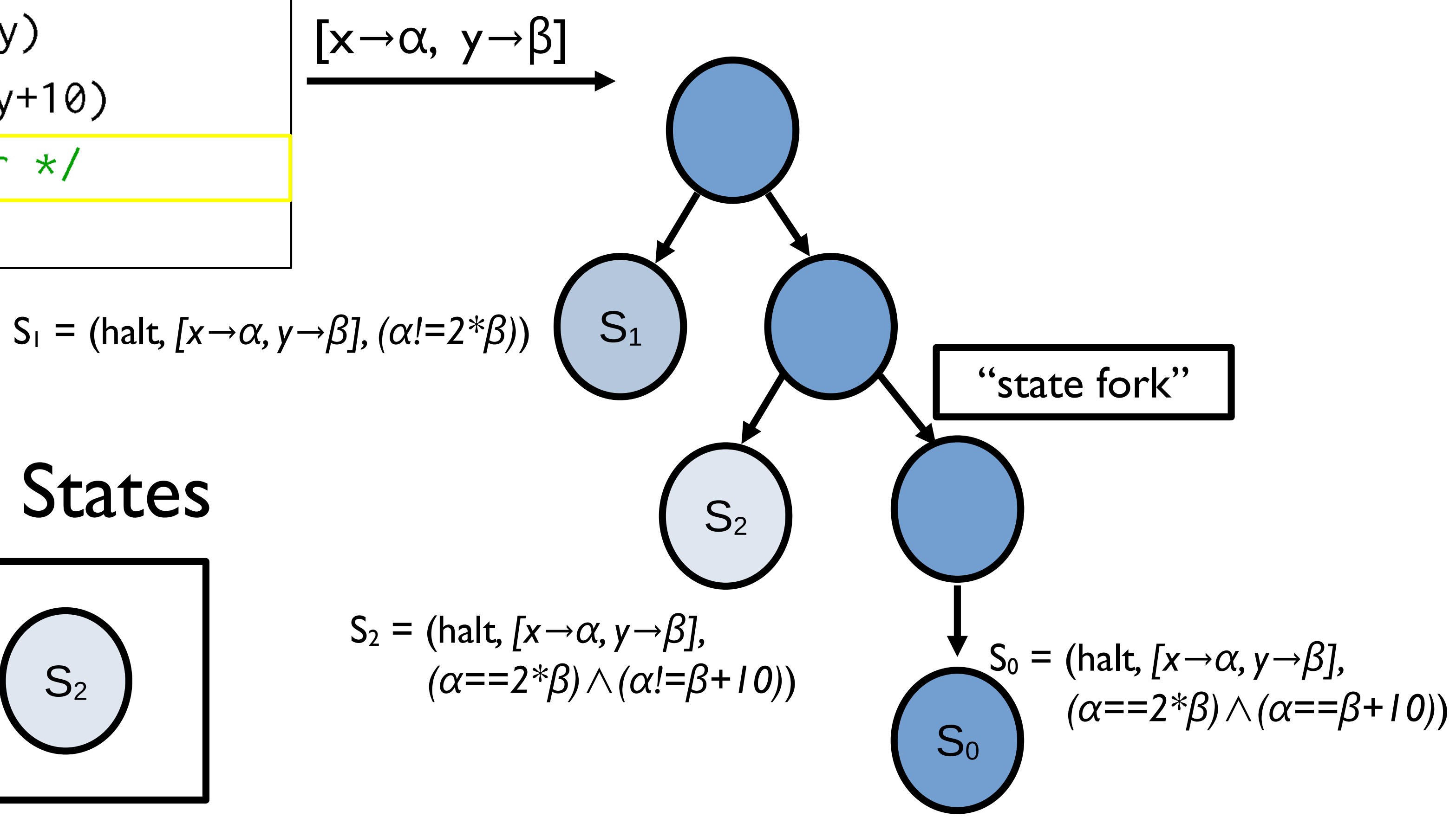
- Candidate States



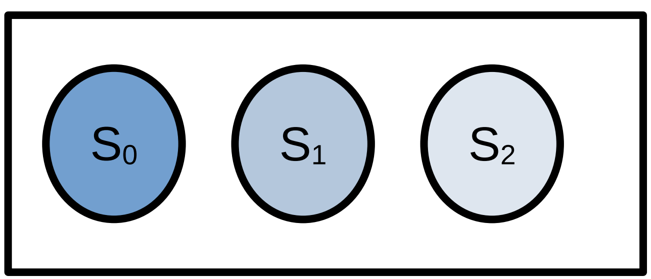
Execution-Generated Testing

- How to generate the test-case reaching 'Error' ?

```c
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```

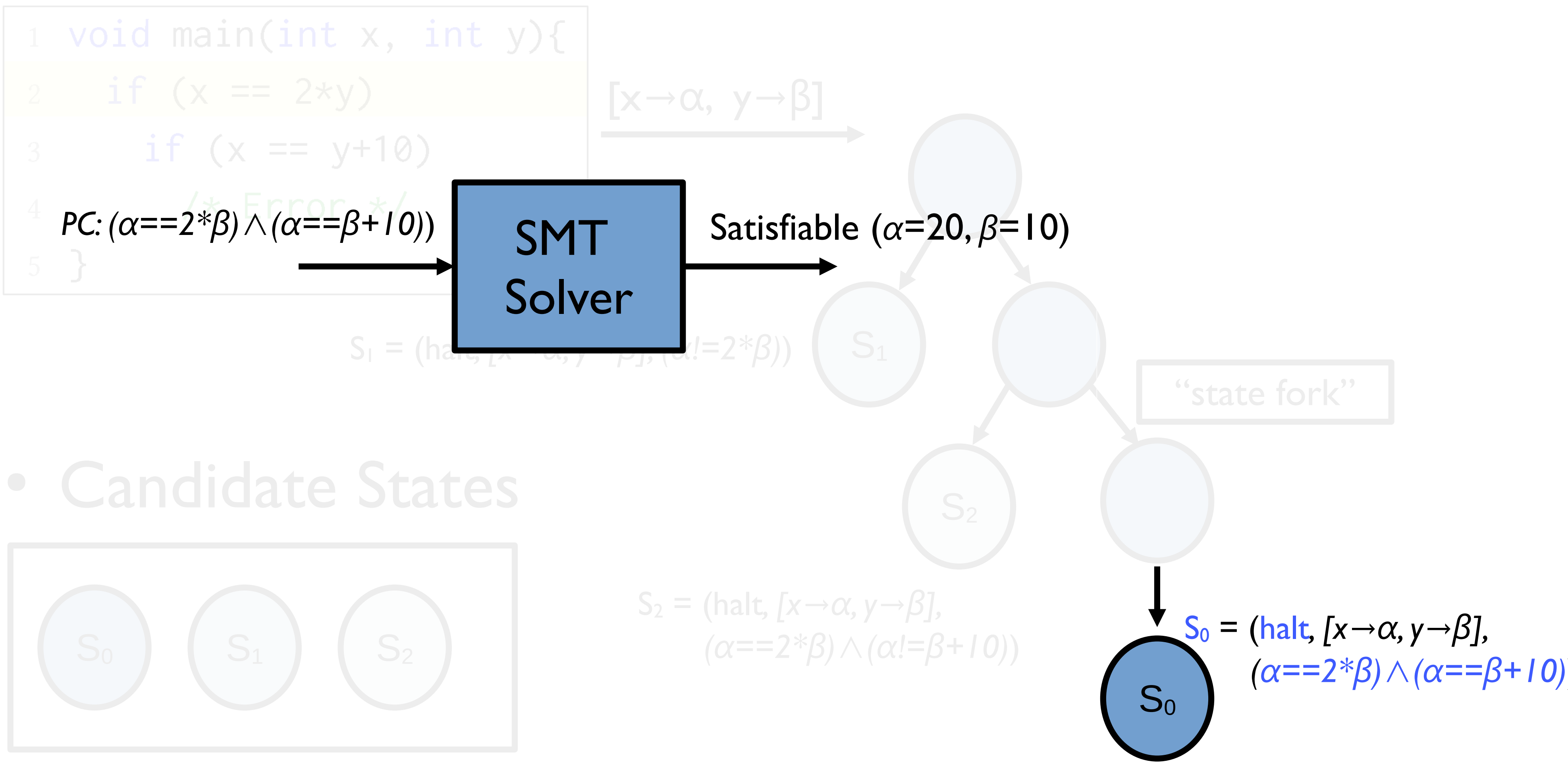


- Candidate States



Execution-Generated Testing

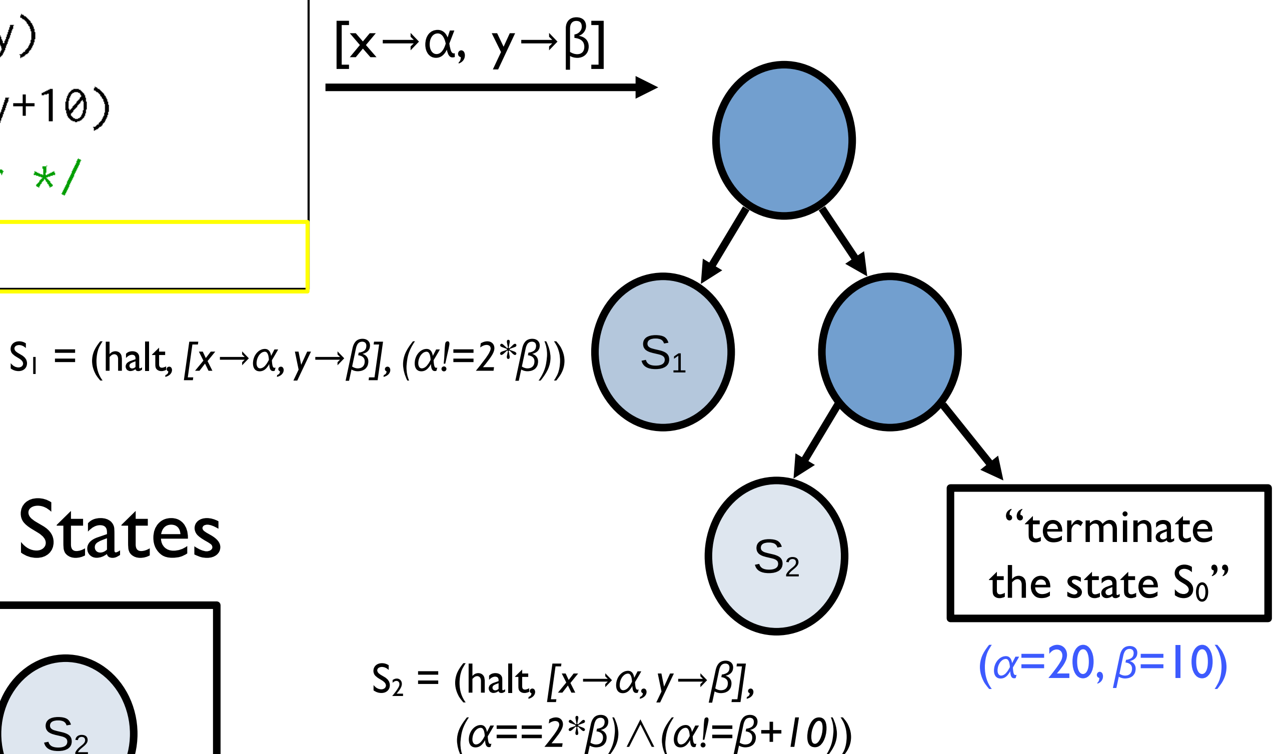
- How to generate the test-case reaching ‘Error’ ?



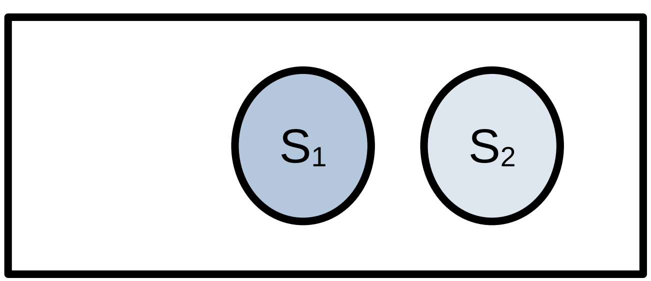
Execution-Generated Testing

- How to generate the test-case reaching 'Error' ?

```
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```



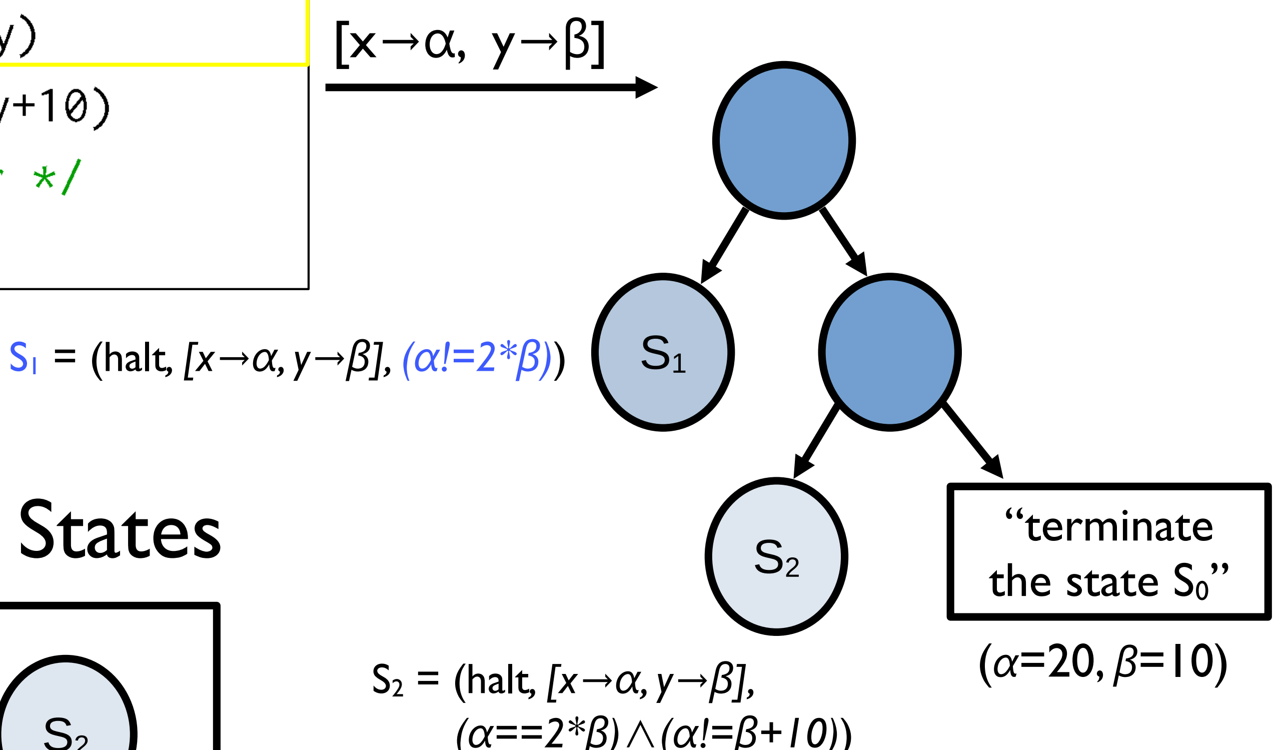
- Candidate States



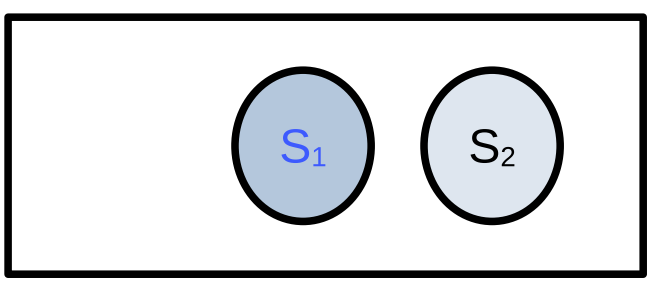
Execution-Generated Testing

- How to generate the test-case reaching 'Error' ?

```
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```

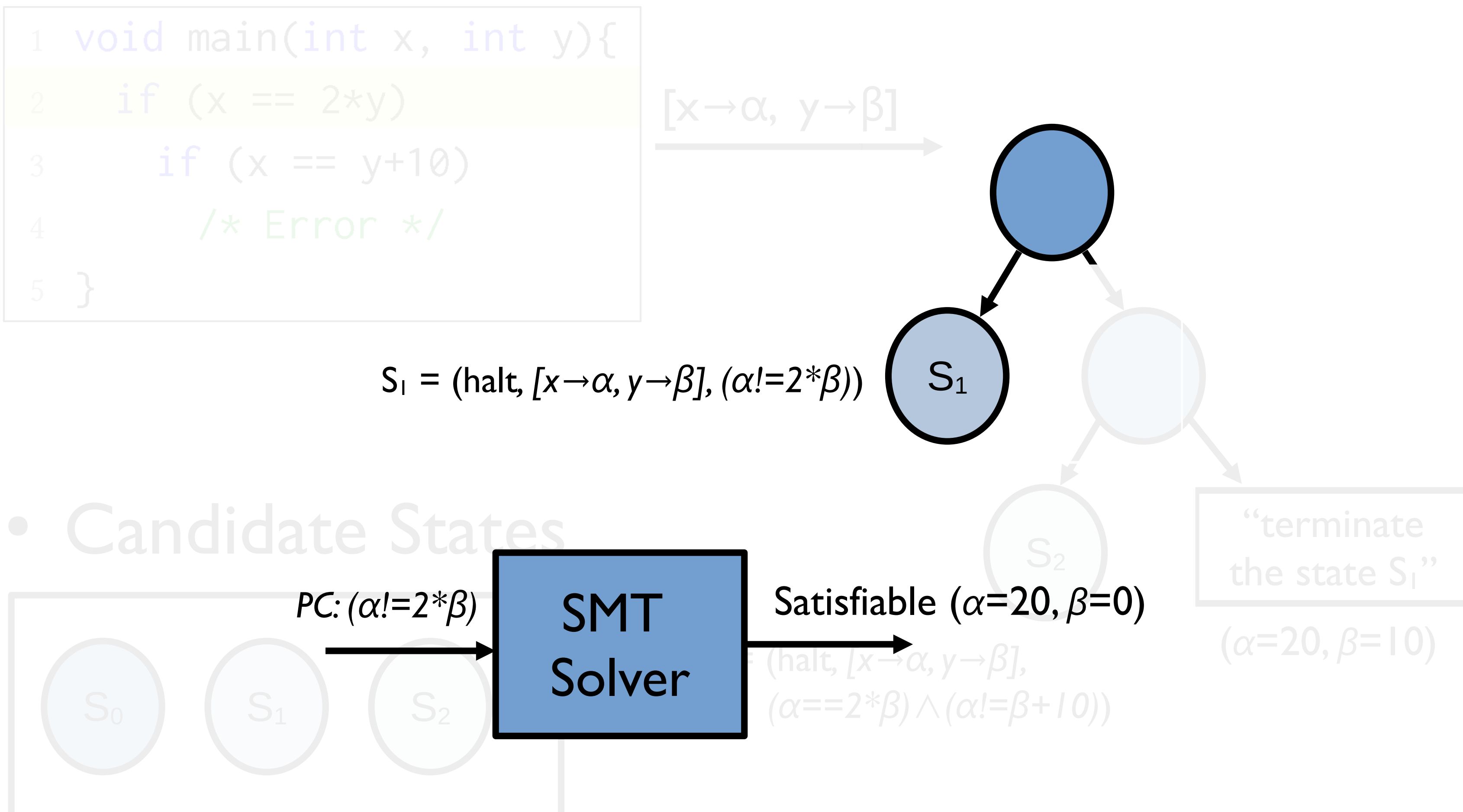


- Candidate States



Execution-Generated Testing

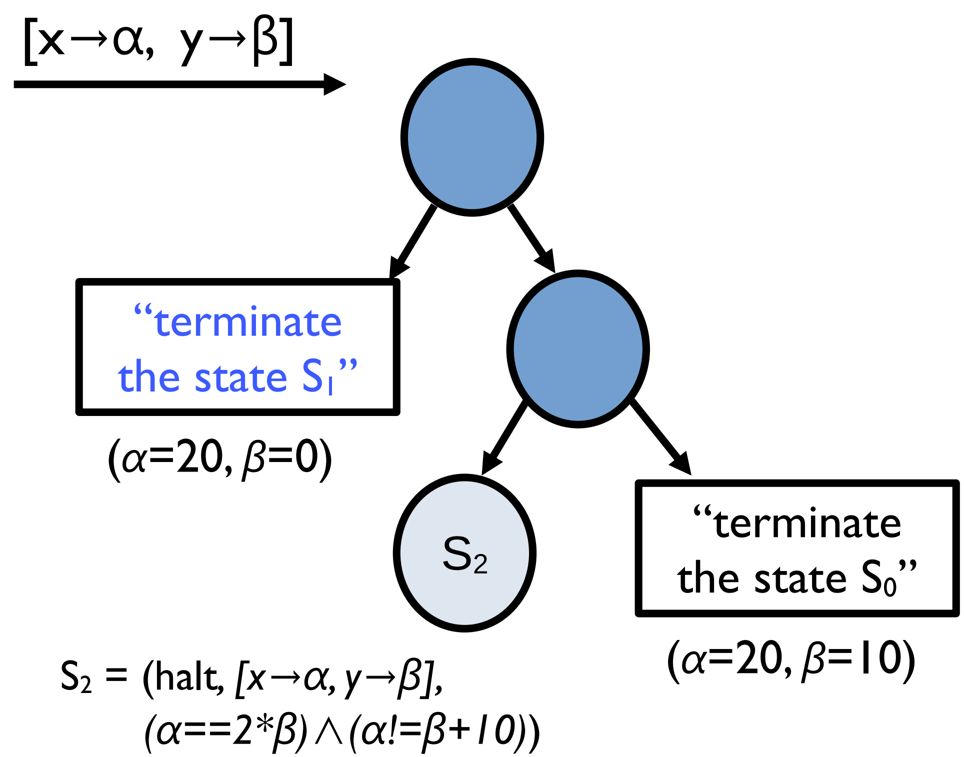
- How to generate the test-case reaching 'Error' ?



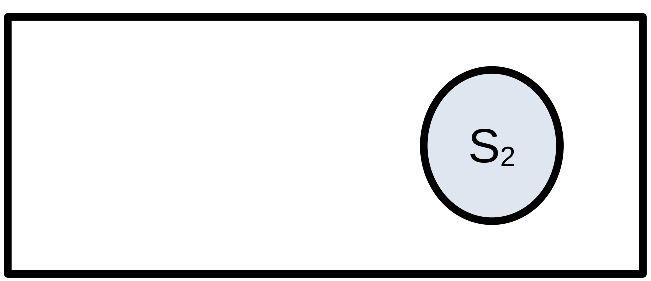
Execution-Generated Testing

- How to generate the test-case reaching ‘Error’ ?

```
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```



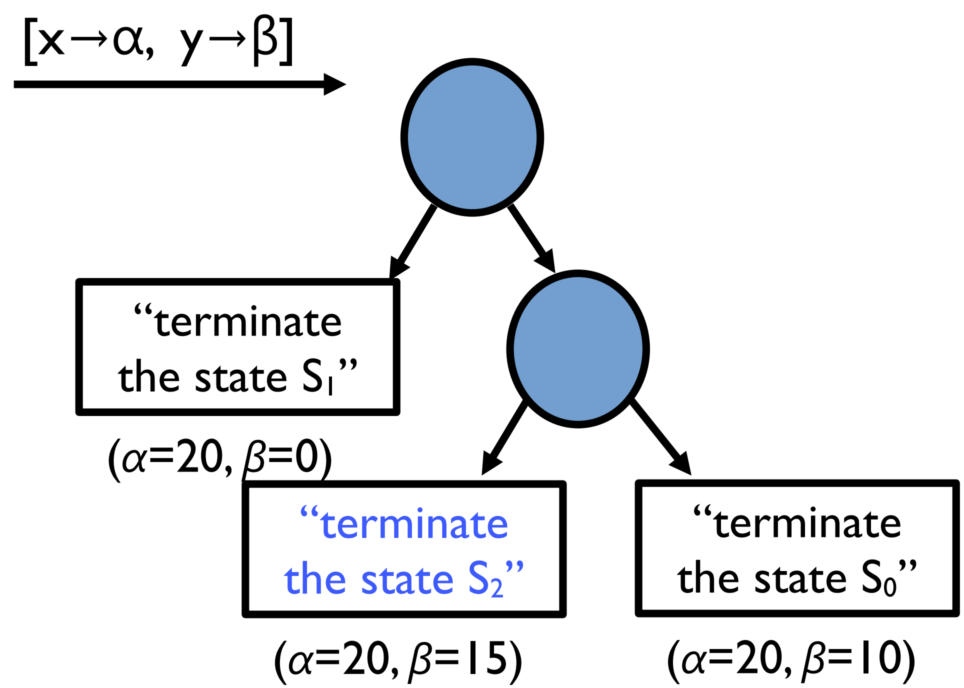
- Candidate States



Execution-Generated Testing

- How to generate the test-case reaching ‘Error’ ?

```
void main(int x, int y){
  if (x == 2*y)
    if (x == y+10)
      /* Error */
}
```



- Candidate States

Execution-Generated Testing

- How does EGT work?

Algorithm 2 Execution-Generated Testing

Input: Program (P) and time budget (N).

$S \leftarrow \{(instr_0, store_0, true)\}$ ▹ initial states

repeat

  $(instr, store, \Phi) \leftarrow \text{Select}(S)$ ▹ choose a state

  $S \leftarrow S \setminus \{(instr, store, \Phi)\}$

  $(instr', store', \Phi) \leftarrow \text{Execute}(instr, store, \Phi)$

  if $instr' = (\text{if } (\phi) \text{ then } instr_1 \text{ else } instr_2)$ **then**

    if $\text{SAT}(\Phi \wedge \phi)$ **then** $S \leftarrow S \cup \{(instr_1, store', \Phi \wedge \phi)\}$

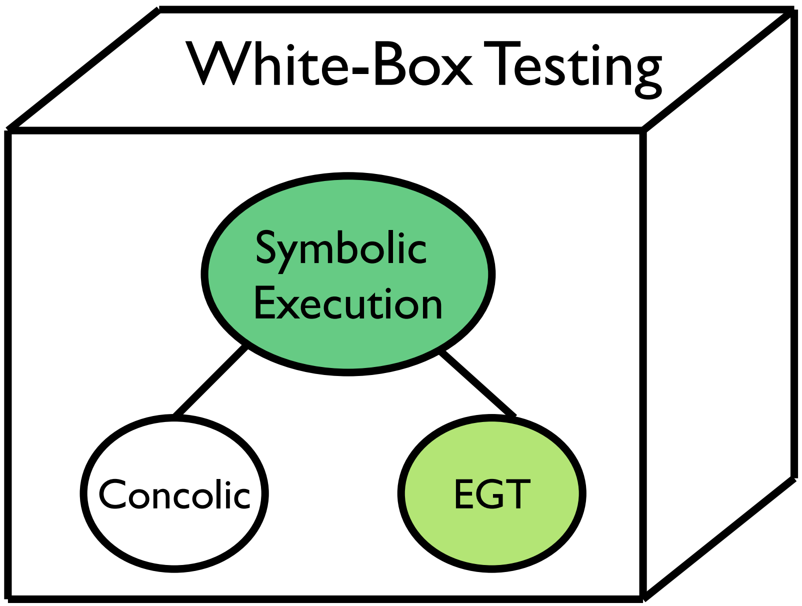
    if $\text{SAT}(\Phi \wedge \neg\phi)$ **then** $S \leftarrow S \cup \{(instr_2, store', \Phi \wedge \neg\phi)\}$

  else if $instr' = \text{halt}$ **then**

    $v \leftarrow \text{generate}(\Phi)$ ▹ generate test cases

until budget N expires (or $S = \emptyset$)

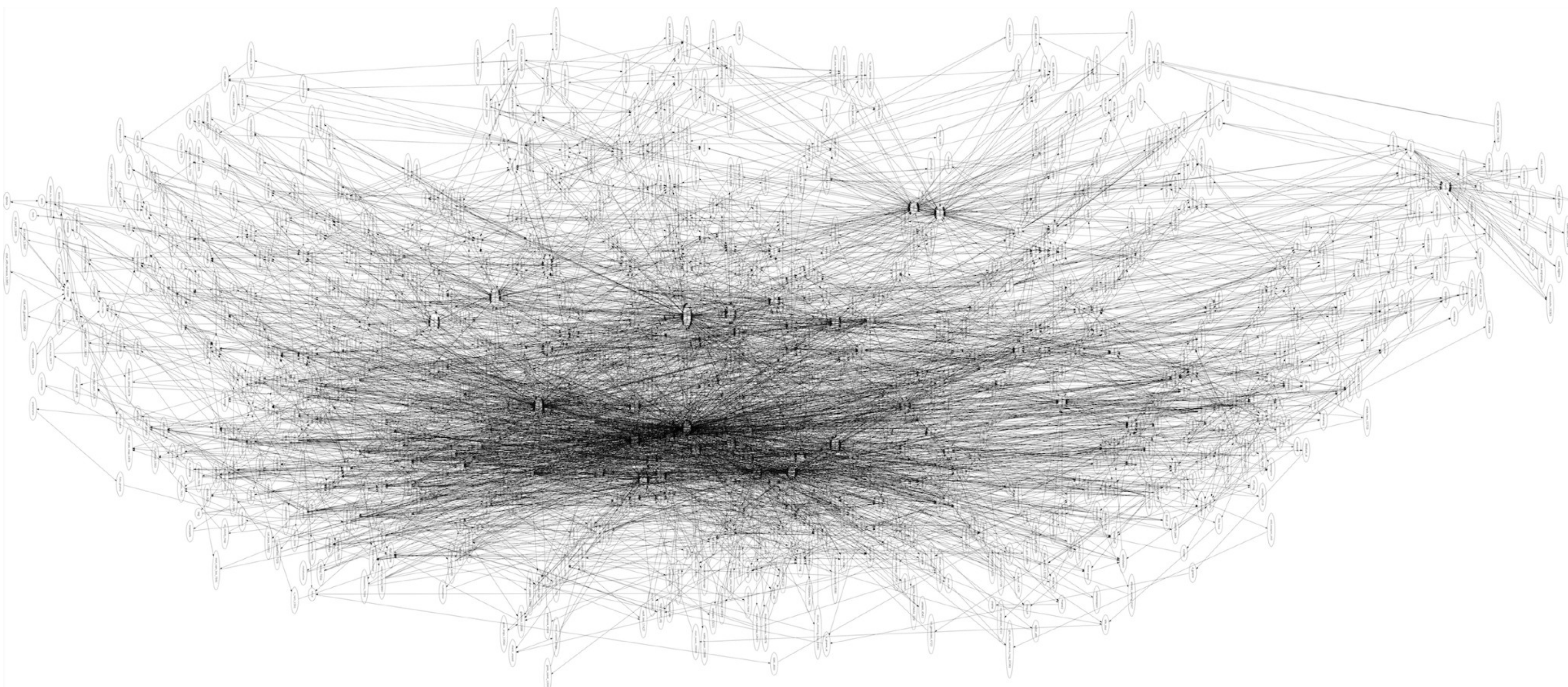
Open Challenge



1. Path-explosion problem
2. Constraint solving cost

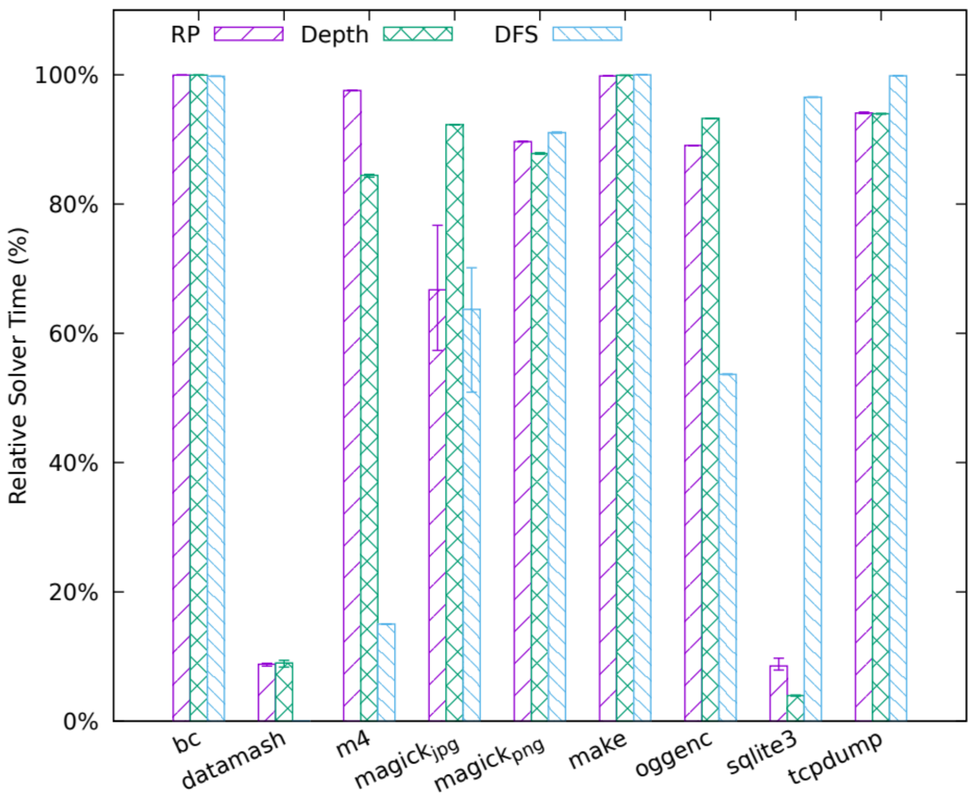
Open Challenge

- Path Explosion (or State Explosion)
  - # of execution paths: $2^{\text{\# of if/while statements}}$
    - ex) grep-2.2(3,836) : $2^{3,386}$ paths (worst case)
  - Exploring all paths is impossible in our lifetime :)



Open Challenge

- Constraint Solving Cost
  - More than 80% of the total testing time is spent on solving the path-conditions.[1]



<SMT Solver>

Solve: $(2*\beta^2 = \alpha) \land (\alpha < \beta+10)$

$\downarrow$

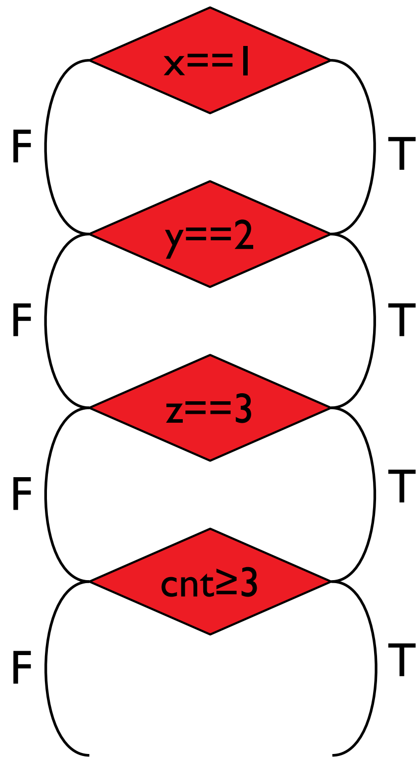
Solution: $\alpha=8, \beta=2$

1. Pending Constraints in Symbolic Execution for Better Exploration and Seeding (ASE 20)

Open Challenge

- Path Explosion (or State Explosion)
  - # of execution paths: $2^{\text{\# of if/while statements}}$

```
void main(int x, int y, int z) {
   int cnt=0;
   if (x == 1) cnt++;
   if (y == 2) cnt++;
   if (z == 3) cnt++;
   if (cnt >= 3) /* error */
}
```

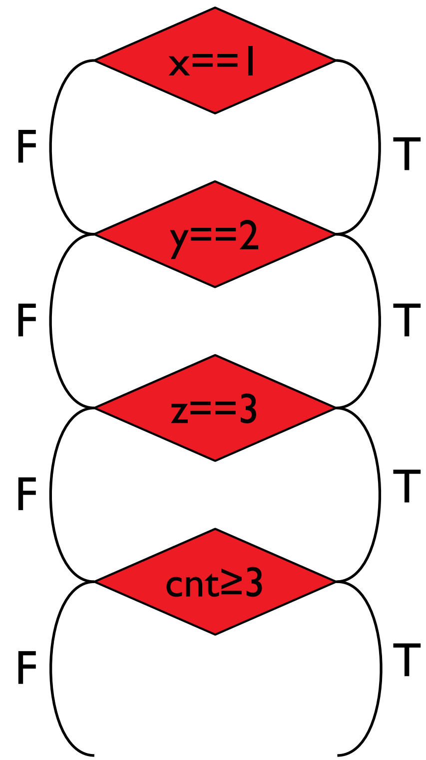


# of total paths = ?

Open Challenge

- Path Explosion (or State Explosion)
  - # of execution paths: $2^{\text{\# of if/while statements}}$

```
void main(int x, int y, int z) {
   int cnt=0;
   if (x == 1) cnt++;
   if (y == 2) cnt++;
   if (z == 3) cnt++;
   if (cnt >= 3) /* error */
}
```



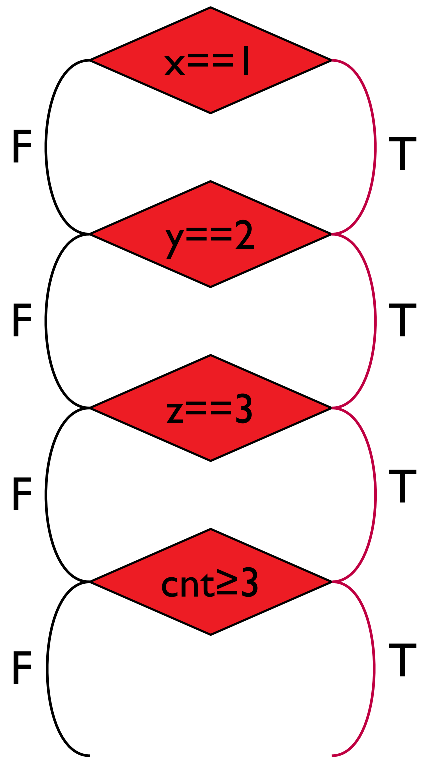
# of total paths = **16 (2^4)**

# of paths for reaching the error = **?**

Open Challenge

- Path Explosion (or State Explosion)
  - # of execution paths: $2^{\text{\# of if/while statements}}$

```
void main(int x, int y, int z) {
   int cnt=0;
   if (x == 1) cnt++;
   if (y == 2) cnt++;
   if (z == 3) cnt++;
   if (cnt >= 3) /* error */
}
```



# of total paths = **16 (2^4)**

# of paths for reaching the error = **1**

Probability for reaching the error = $\frac{1}{16}$

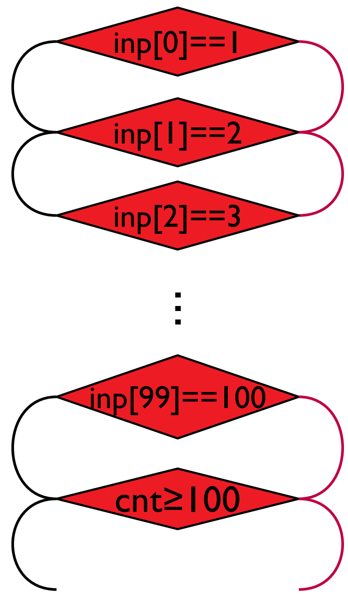
Open Challenge

- Path Explosion (or State Explosion)
  - # of execution paths: $2^{\text{\# of if/while statements}}$

```
void main(int inp[100]) {
   int cnt=0;
   if (inp[0] == 1) cnt++;
   if (inp[1] == 2) cnt++;
   if (inp[2] == 3) cnt++;
   if (inp[3] == 4) cnt++;
         ...
   if (inp[99] == 100) cnt++;

   if (cnt >=100) /* error */
}
```

# of branches =100



# of total paths = ?

# of paths for reaching the error = ?

Probability for reaching the error = ?

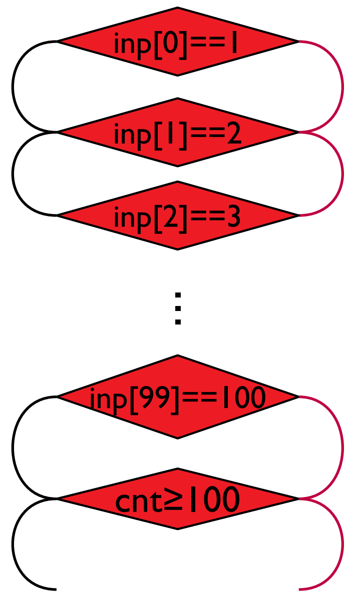
Open Challenge

- Path Explosion (or State Explosion)
  - # of execution paths: $2^{\#\text{ of if/while statements}}$

```
void main(int inp[100]) {
   int cnt=0;
   if (inp[0] == 1) cnt++;
   if (inp[1] == 2) cnt++;
   if (inp[2] == 3) cnt++;
   if (inp[3] == 4) cnt++;
         ...
   if (inp[99] == 100) cnt++;

   if (cnt >=100) /* error */
}
```

# of branches =100



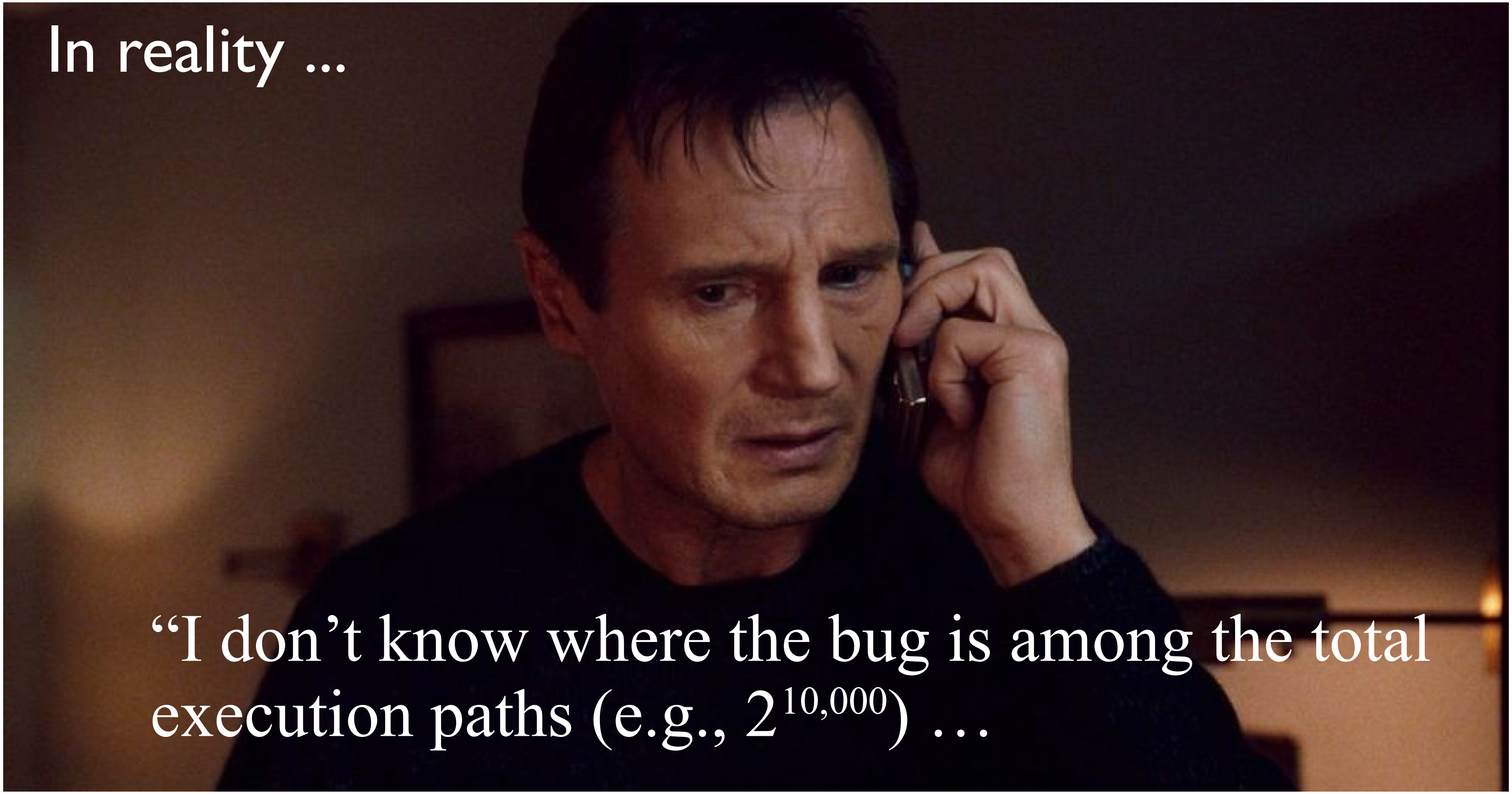
# of total paths = 2^{101}

# of paths for reaching the error = 1

Probability for reaching the error = $\frac{1}{2^{101}}$

Open Challenge

- Path Explosion (or State Explosion)



Diverse Solutions

- Solutions for mitigating the path-explosion problem
  - Which execution paths should we explore first?
    - Search heuristic (Search strategy)
  - Which execution paths are redundant?
    - State-pruning heuristic (Path-pruning heuristic)
    - State-merging heuristic (Path-merging heuristic)

…

Diverse Solutions

- Solutions for mitigating the path-explosion problem
  - Which execution paths should we explore first?
    - Search heuristic (Search strategy)

Today's topic

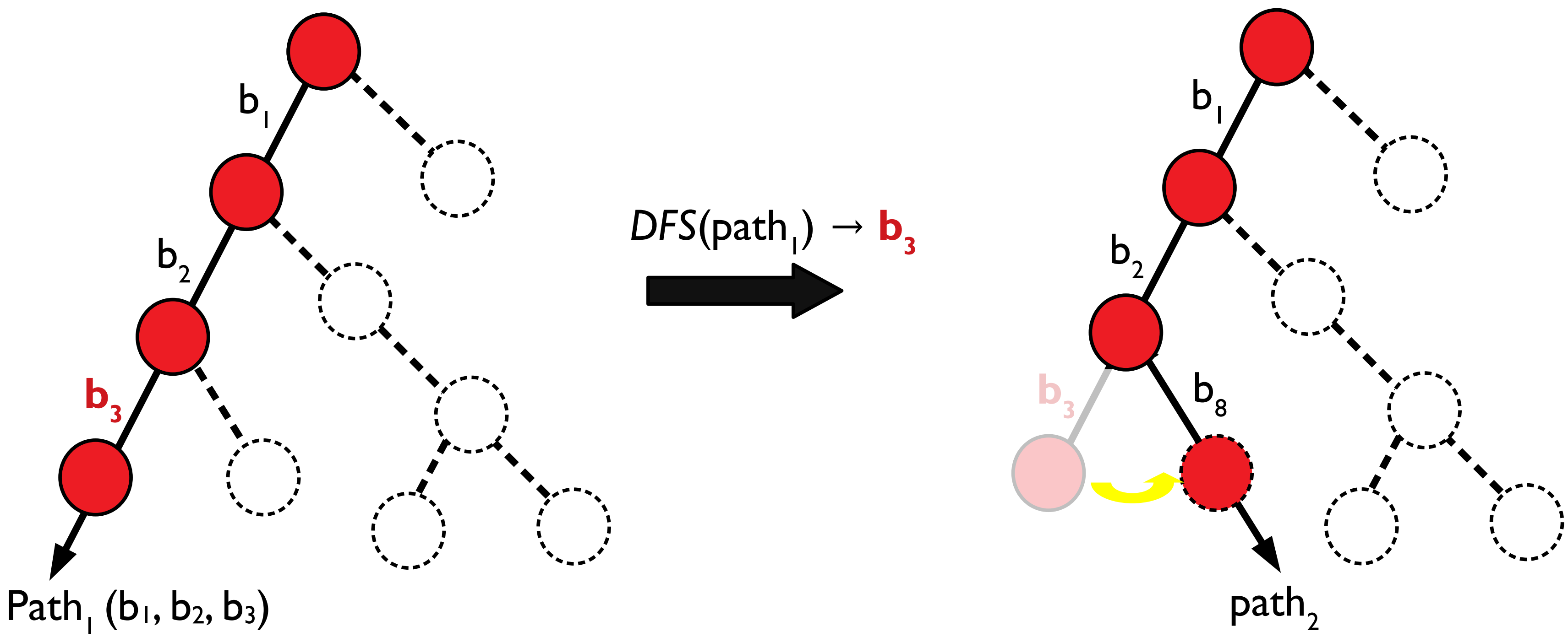
  - Which execution paths are redundant?
    - State-pruning heuristic (Path-pruning heuristic)
    - State-merging heuristic
    - Function and loop summarization

…

Concolic Testing

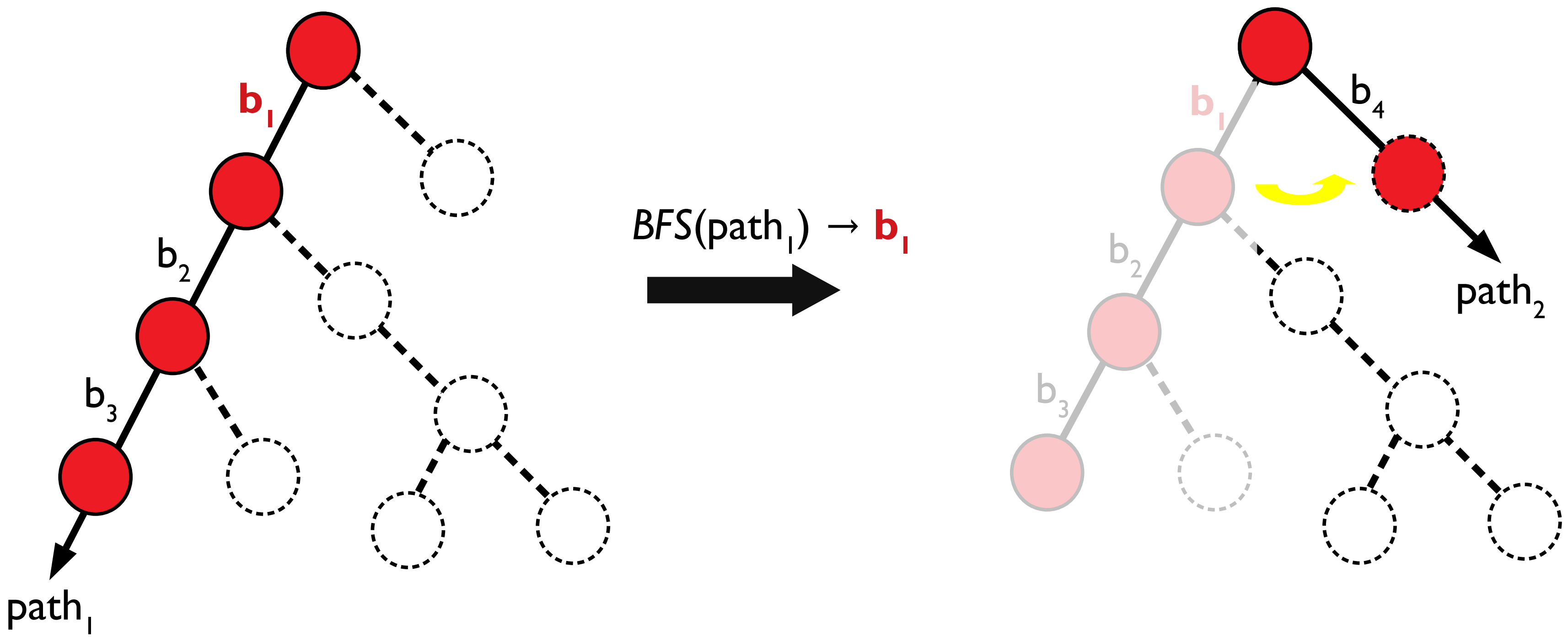
Search Heuristic

- Selecting branches first that maximize coverage or find bugs.
- Having its own branch-selection criteria.



Search Heuristic

- Selecting branches first that maximize coverage or find bugs.
- Having its own branch-selection criteria.



Search Heuristic

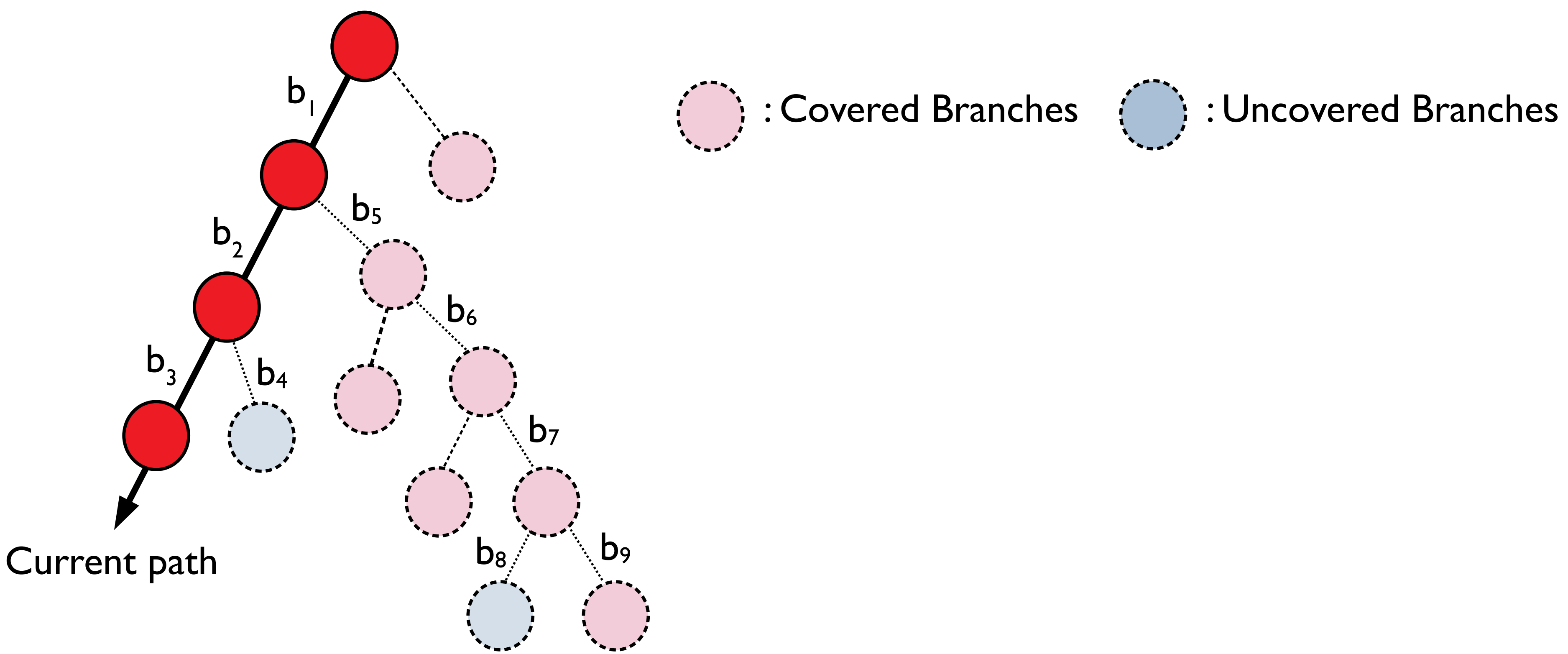
- Introduce the branch-selection criteria of effective search heuristics in concolic testing.
  - ASE'08: CFDS (Control-Flow Directed Search)
  - NDSS'08: Generational Search
  - FSE'14: CGS (Context-Guided Search)
  - ICSE'18: Parametric Search (my paper :)

ASE'08

Search Heuristic

- CFDS (Control-Flow Directed Search)
  - Uncovered branches near the current path will be easier to reach.

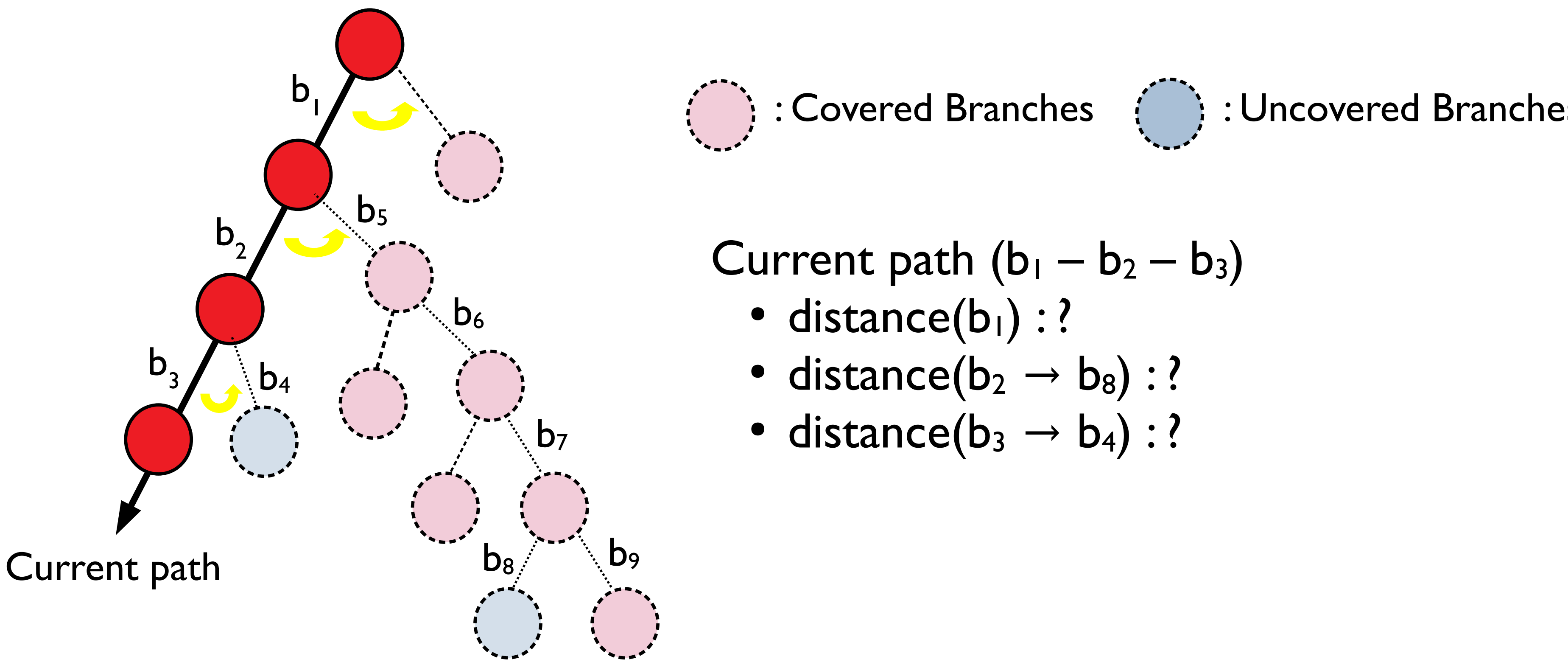
  (Which branch, b_4 or b_8, is easier to cover in the current path?):



ASE'08

Search Heuristic

- CFDS (Control-Flow Directed Search)
  - Selecting a branch first that is close to uncovered branches.
  - Calculating the minimum distance, the number of branches from the selected branch to the uncovered branch.



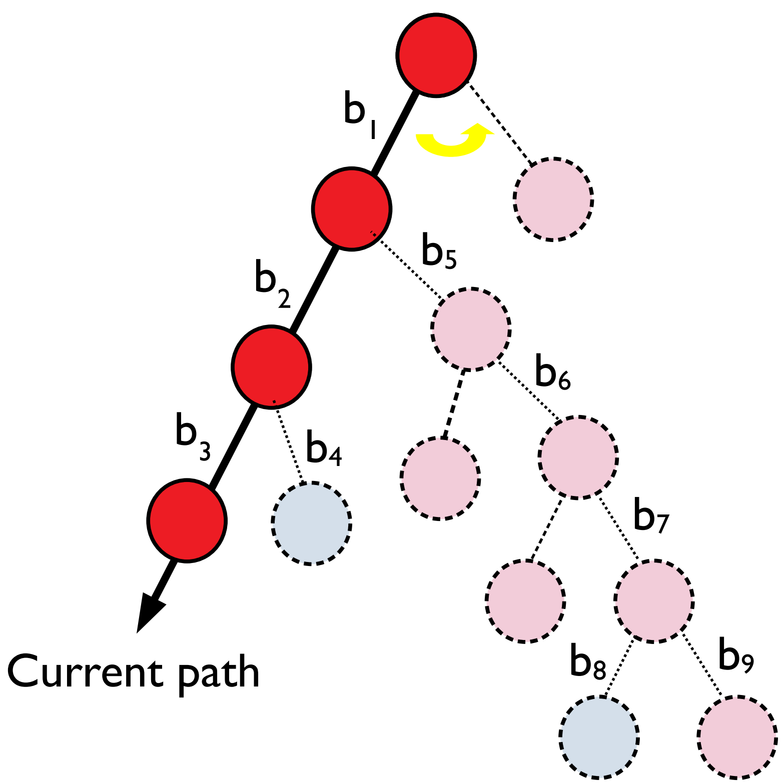
Current path ($b_1 - b_2 - b_3$)

- distance(b_1) : ?
- distance($b_2 \rightarrow b_8$) : ?
- distance($b_3 \rightarrow b_4$) : ?

ASE'08

Search Heuristic

- CFDS (Control-Flow Directed Search)
  - Selecting a branch first that is close to uncovered branches.
  - Calculating the minimum distance, the number of branches from the selected branch to the uncovered branch.



: Covered Branches : Uncovered Branches

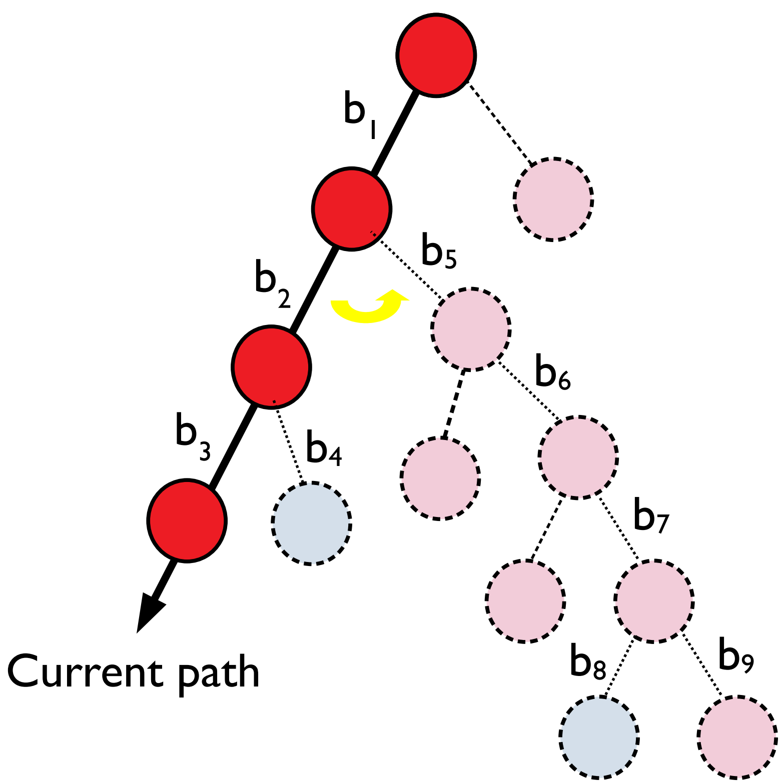
Current path ($b_1 - b_2 - b_3$)

- distance(b_1) : ∞ (unreachable)
- distance($b_2 \rightarrow b_8$) : ?
- distance($b_3 \rightarrow b_4$) : ?

ASE'08

Search Heuristic

- CFDS (Control-Flow Directed Search)
  - Selecting a branch first that is close to uncovered branches.
  - Calculating the minimum distance, the number of branches from the selected branch to the uncovered branch.



: Covered Branches : Uncovered Branches

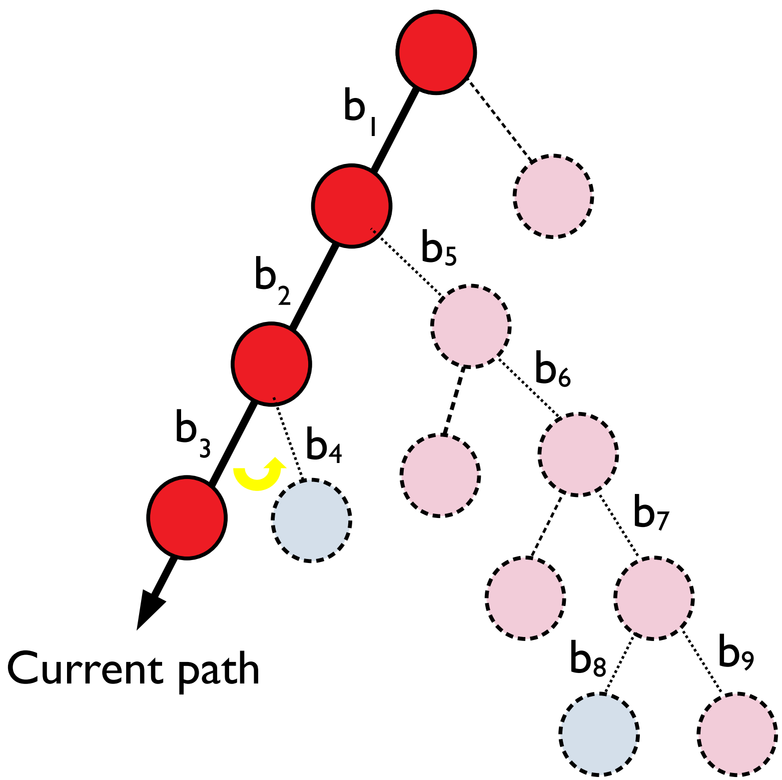
Current path ($b_1 - b_2 - b_3$)

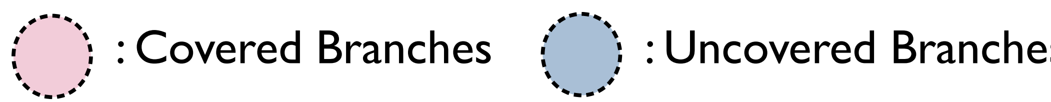
- distance(b_1) : ∞ (unreachable)
- distance($b_2 \rightarrow b_8$) : 4 ($b_2 - b_5 - b_6 - b_7 - b_8$)
- distance($b_3 \rightarrow b_4$) : ?

ASE'08

Search Heuristic

- CFDS (Control-Flow Directed Search)
  - Selecting a branch first that is close to uncovered branches.
  - Calculating the minimum distance, the number of branches from the selected branch to the uncovered branch.





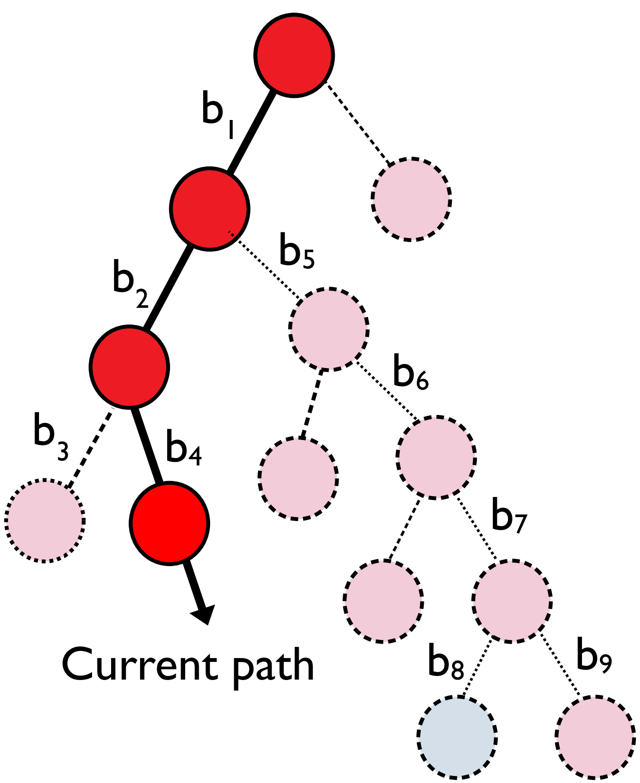
Current path ($b_1 - b_2 - b_3$)

- distance(b_1) : ∞ (unreachable)
- distance($b_2 \rightarrow b_8$) : 4 ($b_2 - b_5 - b_6 - b_7 - b_8$)
- distance($b_3 \rightarrow b_4$) : 1 ($b_3 - b_4$)

ASE'08

Search Heuristic

- CFDS (Control-Flow Directed Search)
  - Selecting a branch first that is close to uncovered branches.
  - Calculating the minimum distance, the number of branches from the selected branch to the uncovered branch.



: Covered Branches   : Uncovered Branches

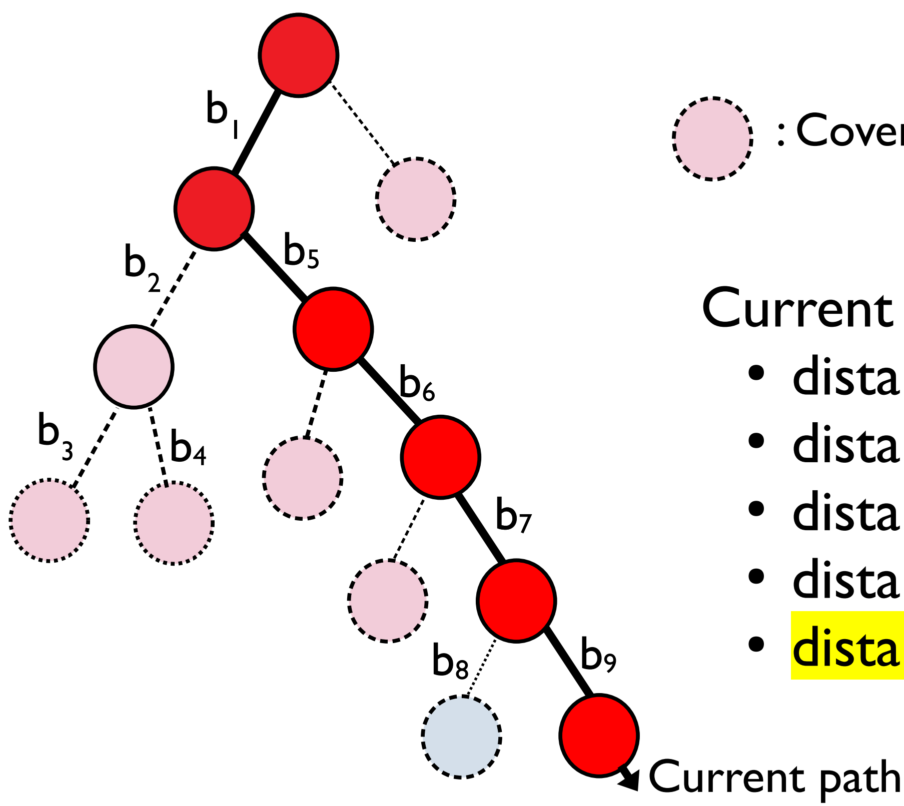
Current path ($b_1 - b_2 - b_4$)

- distance(b_1) : ∞ (unreachable)
- distance($b_2 \rightarrow b_8$) : 4 ($b_2 - b_5 - b_6 - b_7 - b_8$)
- distance(b_4) : ∞ (unreachable)

ASE'08

Search Heuristic

- CFDS (Control-Flow Directed Search)
  - Selecting a branch first that is close to uncovered branches.
  - Calculating the minimum distance, the number of branches from the selected branch to the uncovered branch.



: Covered Branches　: Uncovered Branches

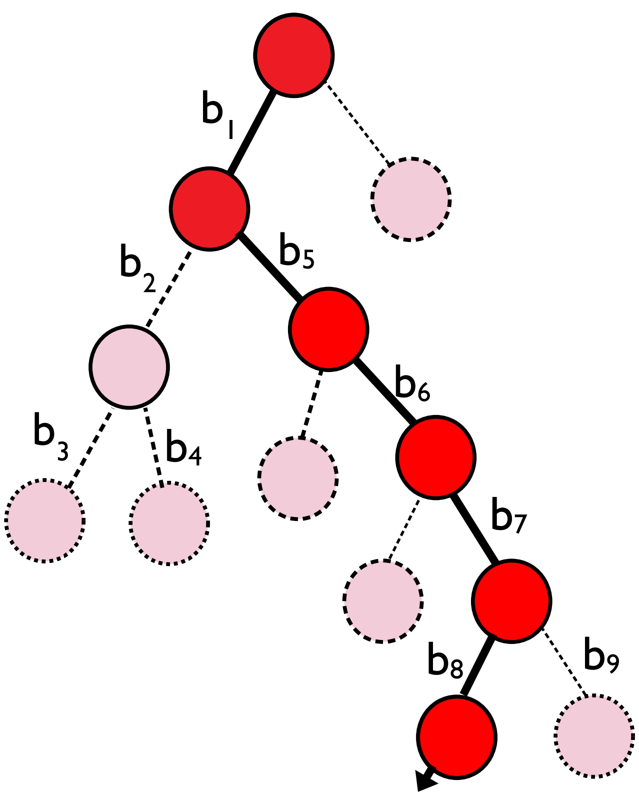
Current path ($b_1 - b_5 - b_6 - b_7 - b_9$)

- distance(b_1) : ∞ (unreachable)
- distance(b_5) : ∞ (unreachable)
- distance(b_6) : ∞ (unreachable)
- distance(b_7) : ∞ (unreachable)
- distance(b_9) : 1 ($b_9 - b_8$)

ASE'08

Search Heuristic

- CFDS (Control-Flow Directed Search)
  - Selecting a branch first that is close to uncovered branches.
  - Calculating the minimum distance, the number of branches from the selected branch to the uncovered branch.



: Covered Branches : Uncovered Branches

Current path ($b_1 - b_5 - b_6 - b_7 - b_8$)

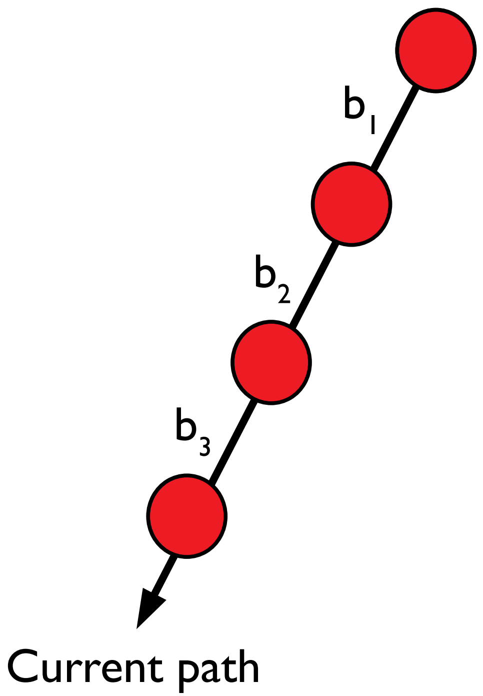
- distance(b_1) : ∞ (unreachable)
- distance(b_5) : ∞ (unreachable)
- distance(b_6) : ∞ (unreachable)
- distance(b_7) : ∞ (unreachable)
- distance(b_8) : ∞ (unreachable)

NDSS'08

Search Heuristic

- Generational Search
  - Which branch has the highest coverage gain in the current path?

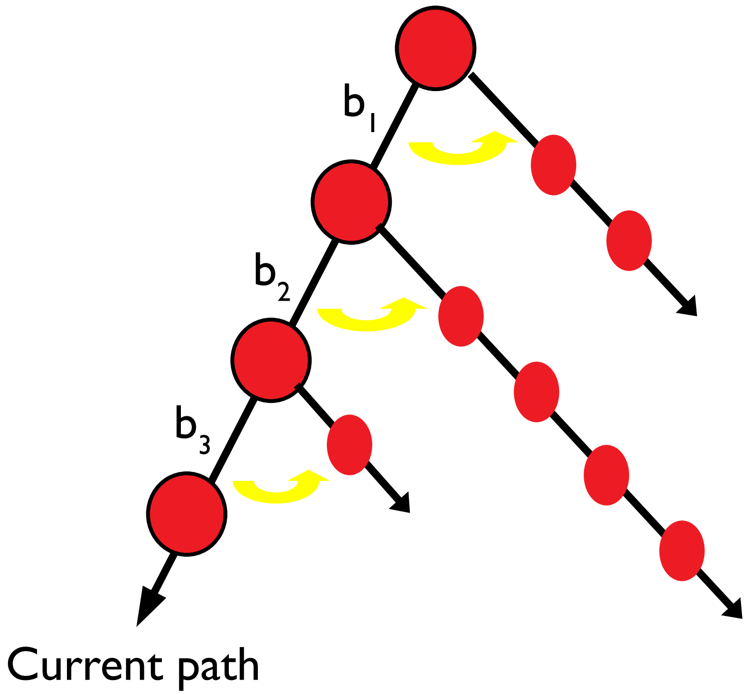
    (b_1? b_2? b_3?)



NDSS'08

Search Heuristic

- Generational Search
  - Step 1: Negating and Executing each branch in the path
  - Step 2: Calculating the coverage gain of each selected branch
  - Step 3: Selecting the branch first with the highest coverage gain



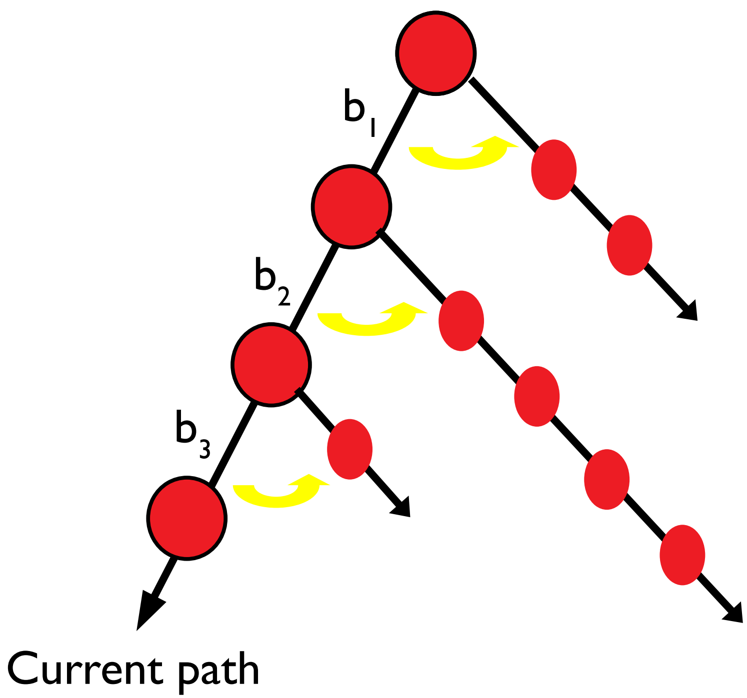
Current path ($b_1 - b_2 - b_3$)

- gain(b_1): 2
- gain(b_2): 4
- gain(b_3): 1

NDSS'08

Search Heuristic

- Generational Search
  - Step 1: Negating and Executing each branch in the path
  - Step 2: Calculating the coverage gain of each selected branch
  - Step 3: Selecting the branch first with the highest coverage gain



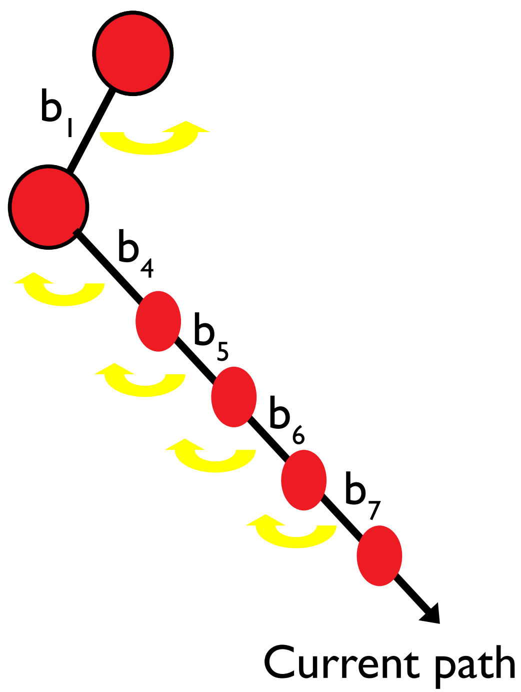
Current path (b_1 – b_2 – b_3)

- gain(b_1): 2
- gain(b_2): 4
- gain(b_3): 1

NDSS'08

Search Heuristic

- Generational Search
  - Step 1: Negating and Executing each branch in the path
  - Step 2: Calculating the coverage gain of each selected branch
  - Step 3: Selecting the branch first with the highest coverage gain



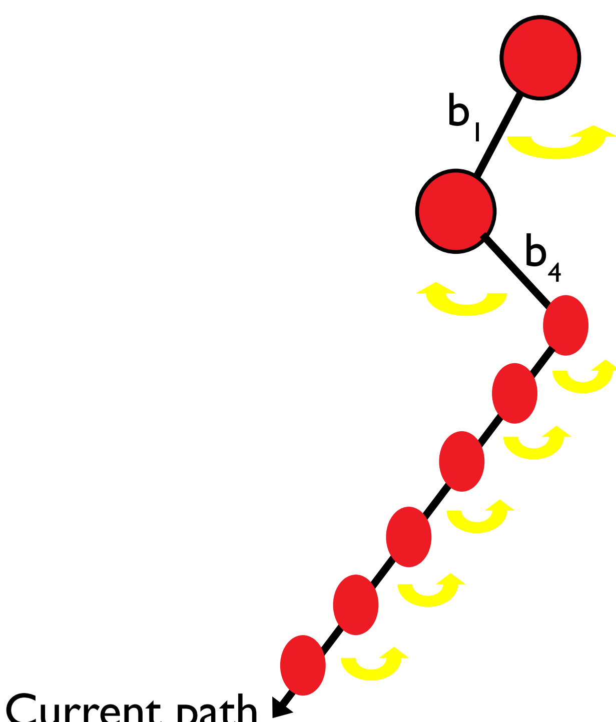
Current path (b_1 – b_4 – b_5 – b_6 – b_7)

- gain(b_1): 2
- gain(b_4): 1
- gain(b_5): 5
- gain(b_6): 3
- gain(b_7): 2

NDSS'08

Search Heuristic

- Generational Search
  - Step 1: Negating and Executing each branch in the path
  - Step 2: Calculating the coverage gain of each selected branch
  - Step 3: Selecting the branch first with the highest coverage gain

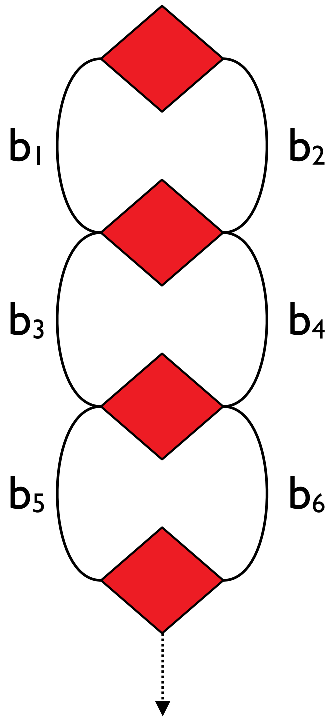


Drives concolic testing towards the highest incremental coverage gain!

FSE' 14

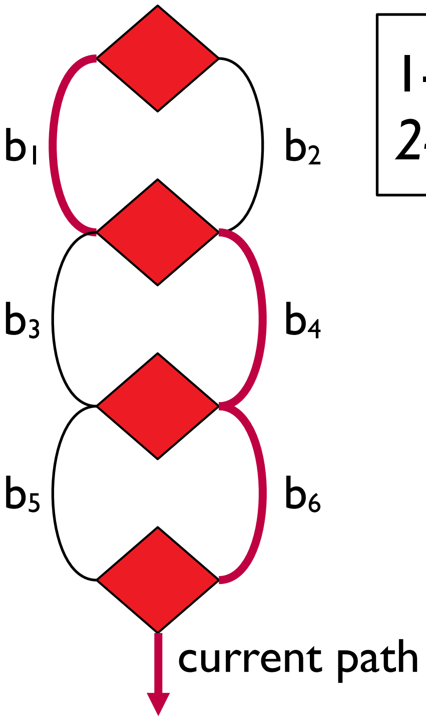
Search Heuristic

- CGS (Context-Guided Search)
  - Select a branch having new *context* first.
  - *context*: a sequence of preceding branches in the current path



Search Heuristic

- CGS (Context-Guided Search)
  - Select a branch having new *context* first.
  - *k-context*: a sequence of *k* preceding branches in the current path



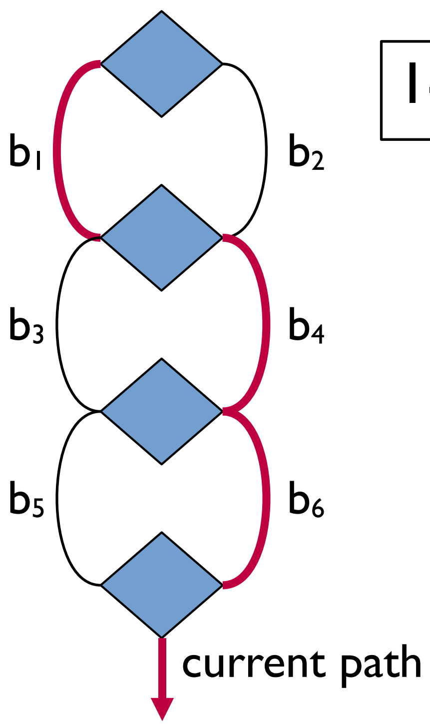
1-*context* of b_6 : b_4
2-*context* of b_6 : $b_4 - b_1$

FSE'14

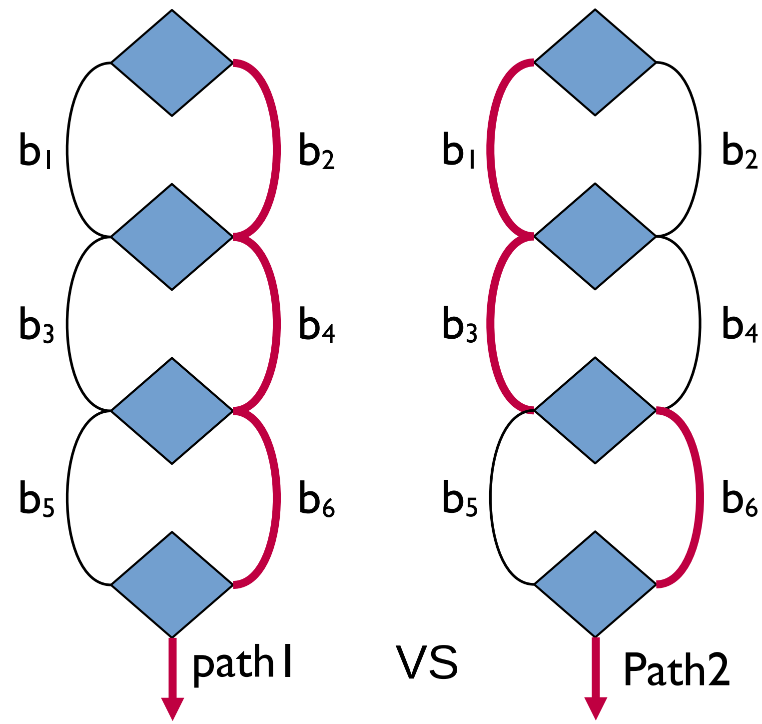
Search Heuristic

- CGS (Context-Guided Search)
  - Select a branch having new *context* first.
  - *context*: a sequence of preceding branches in the current path

  Which path has *new context* of b_6 among path1 and 2?



1-*context* of b_6: b_4

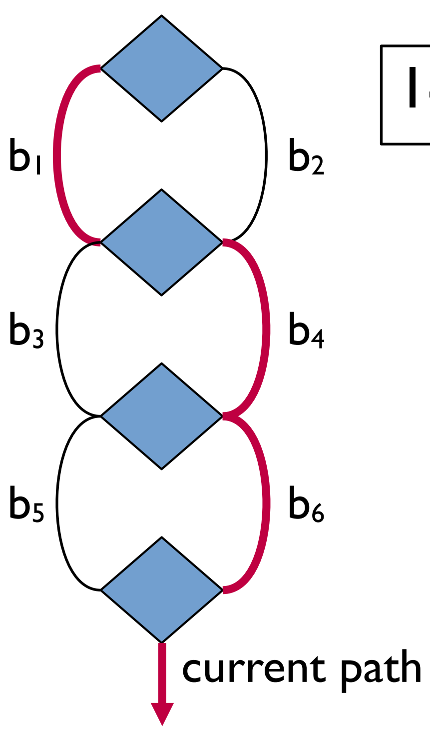


FSE'14

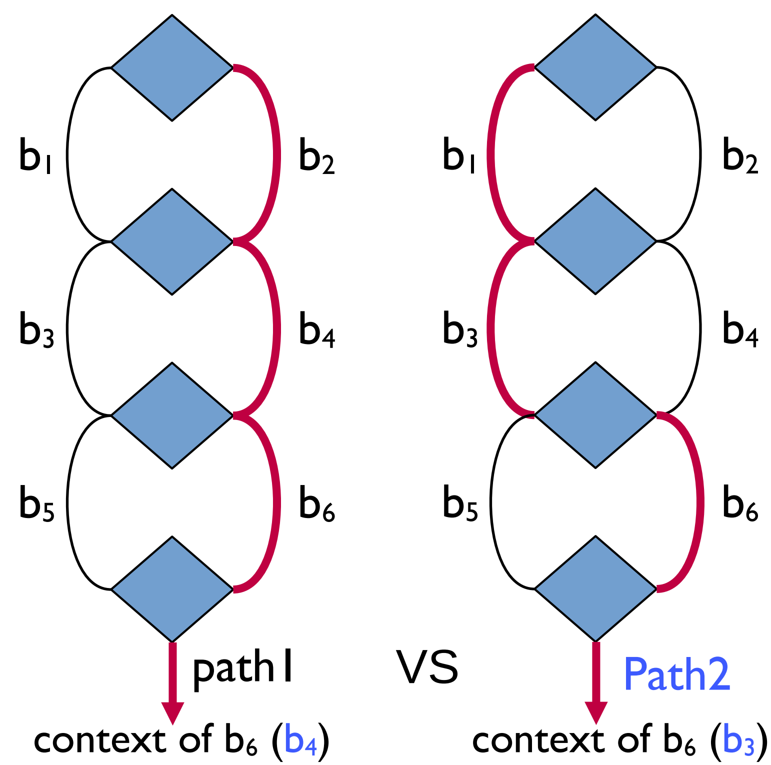
Search Heuristic

- CGS (Context-Guided Search)
  - Select a branch having new *context* first.
  - *context*: a sequence of preceding branches in the current path

Which path has *new context* of b_6 among path1 and 2? : path2



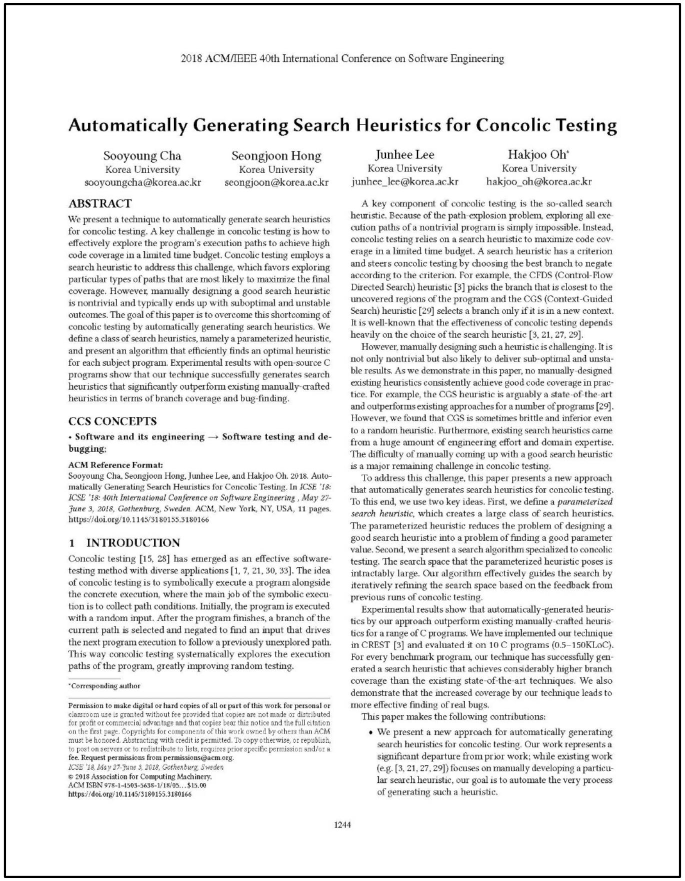
1-*context* of b_6: b_4, b_3



ICSE' 18

Search Heuristic

- Parametric Search (State-of-the art search heuristic)



2018 ACM/IEEE 40th International Conference on Software Engineering

Automatically Generating Search Heuristics for Concolic Testing

Sooyoung Cha
Korea University
sooyoungcha@korea.ac.kr

Seongjoon Hong
Korea University
seongjoon@korea.ac.kr

Junhee Lee
Korea University
junhee_lee@korea.ac.kr

Hakjoo Oh*
Korea University
hakjoo_oh@korea.ac.kr

ABSTRACT

We present a technique to automatically generate search heuristics for concolic testing. A key challenge in concolic testing is how to effectively explore the program's execution paths to achieve high code coverage in a limited time budget. Concolic testing employs a search heuristic to address this challenge, which favors exploring particular types of paths that are most likely to maximize the final coverage. However, manually designing a good search heuristic is nontrivial and typically ends up with suboptimal and unstable outcomes. The goal of this paper is to overcome this shortcoming of concolic testing by automatically generating search heuristics. We define a class of search heuristics, namely a parameterized heuristic, and present an algorithm that efficiently finds an optimal heuristic for each subject program. Experimental results with open-source C programs show that our technique successfully generates search heuristics that significantly outperform existing manually-crafted heuristics in terms of branch coverage and bug-finding.

CCS CONCEPTS

- **Software and its engineering → Software testing and debugging;**

ACM Reference Format:
Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. 2018. Automatically Generating Search Heuristics for Concolic Testing. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180166

1 INTRODUCTION

Concolic testing [15, 28] has emerged as an effective software-testing method with diverse applications [1, 7, 21, 30, 33]. The idea of concolic testing is to symbolically execute a program alongside the concrete execution, where the main job of the symbolic execution is to collect path conditions. Initially, the program is executed with a random input. After the program finishes, a branch of the current path is selected and negated to find an input that drives the next program execution to follow a previously unexplored path. This way concolic testing systematically explores the execution paths of the program, greatly improving random testing.

A key component of concolic testing is the so-called search heuristic. Because of the path-explosion problem, exploring all execution paths of a nontrivial program is simply impossible. Instead, concolic testing relies on a search heuristic to maximize code coverage in a limited time budget. A search heuristic has a criterion and steers concolic testing by choosing the best branch to negate according to the criterion. For example, the CFDS (Control-Flow Directed Search) heuristic [3] picks the branch that is closest to the uncovered regions of the program and the CGS (Context-Guided Search) heuristic [29] selects a branch only if it is in a new context. It is well-known that the effectiveness of concolic testing depends heavily on the choice of the search heuristic [3, 21, 27, 29].

However, manually designing such a heuristic is challenging. It is not only nontrivial but also likely to deliver sub-optimal and unstable results. As we demonstrate in this paper, no manually-designed existing heuristics consistently achieve good code coverage in practice. For example, the CGS heuristic is arguably a state-of-the-art and outperforms existing approaches for a number of programs [29]. However, we found that CGS is sometimes brittle and inferior even to a random heuristic. Furthermore, existing search heuristics came from a huge amount of engineering effort and domain expertise. The difficulty of manually coming up with a good search heuristic is a major remaining challenge in concolic testing.

To address this challenge, this paper presents a new approach that automatically generates search heuristics for concolic testing. To this end, we use two key ideas. First, we define a *parameterized search heuristic*, which creates a large class of search heuristics. The parameterized heuristic reduces the problem of designing a good search heuristic into a problem of finding a good parameter value. Second, we present a search algorithm specialized to concolic testing. The search space that the parameterized heuristic poses is intractably large. Our algorithm effectively guides the search by iteratively refining the search space based on the feedback from previous runs of concolic testing.

Experimental results show that automatically-generated heuristics by our approach outperform existing manually-crafted heuristics for a range of C programs. We have implemented our technique in CREST [3] and evaluated it on 10 C programs (0.5–150KLoC). For every benchmark program, our technique has successfully generated a search heuristic that achieves considerably higher branch coverage than the existing state-of-the-art techniques. We also demonstrate that the increased coverage by our technique leads to more effective finding of real bugs.

This paper makes the following contributions:

- We present a new approach for automatically generating search heuristics for concolic testing. Our work represents a significant departure from prior work; while existing work (e.g. [3, 21, 27, 29]) focuses on manually developing a particular search heuristic, our goal is to automate the very process of generating such a heuristic.

*Corresponding author

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05...$15.00
https://doi.org/10.1145/3180155.3180166

ICSE'18

Motivation

- Parametric Search (State-of-the art search heuristic)

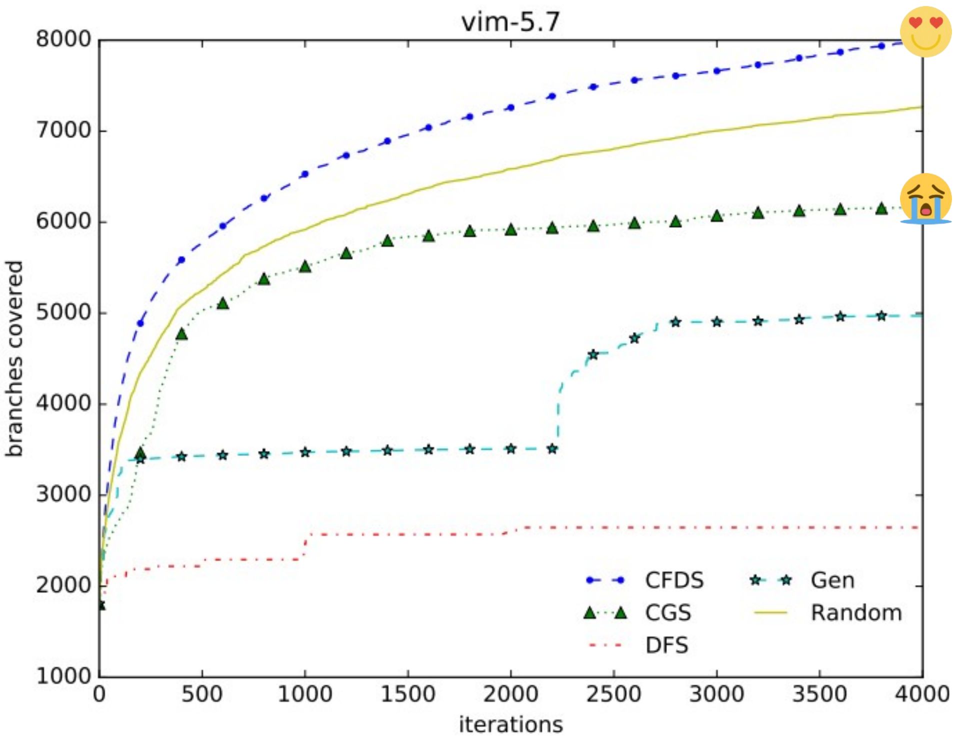
Q) Which existing search heuristic is most effective for increasing code coverage and finding bugs?

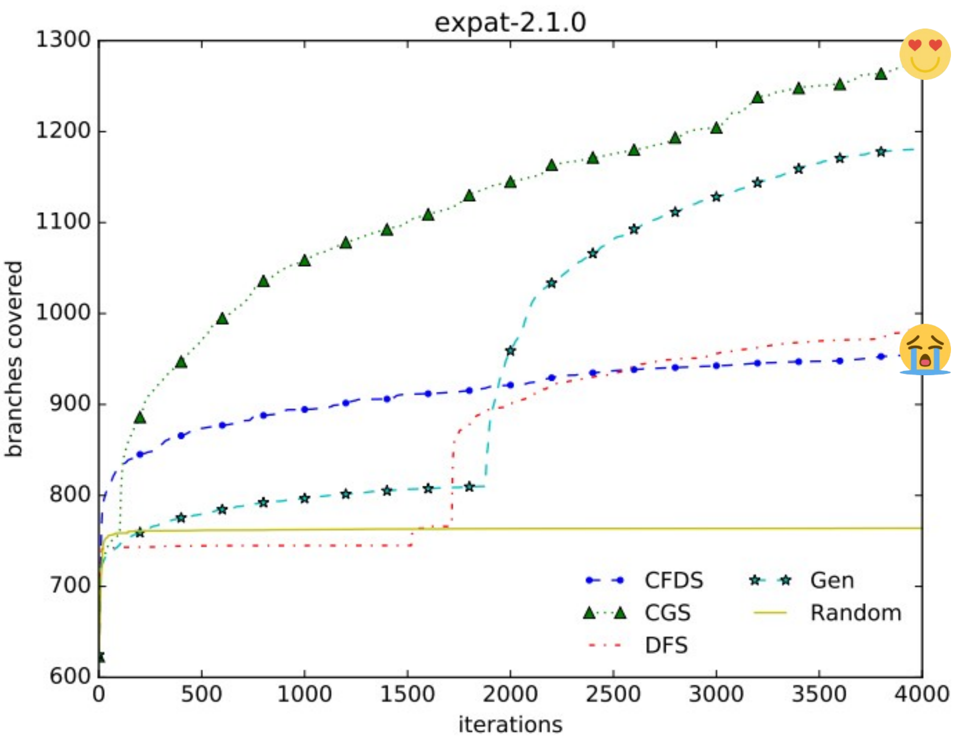
- CFDS (Control Flow Directed Search)
- Generational Search
- CGS (Context-Guided Search)
- Random Branch Selection Search
- DFS, BFS, ...

ICSE'18

Motivation

- No existing search heuristics consistently achieve high coverage.
- Designing new heuristic is highly nontrivial.
  - Search Heuristic → (ICSE, FSE, ASE, NDSS, ...)

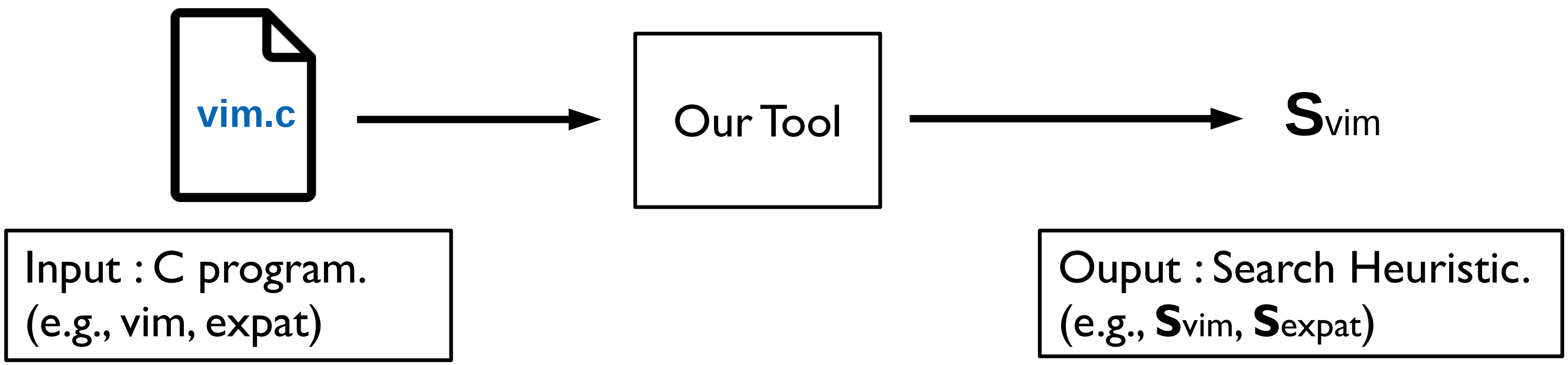




ICSE'18

Goal

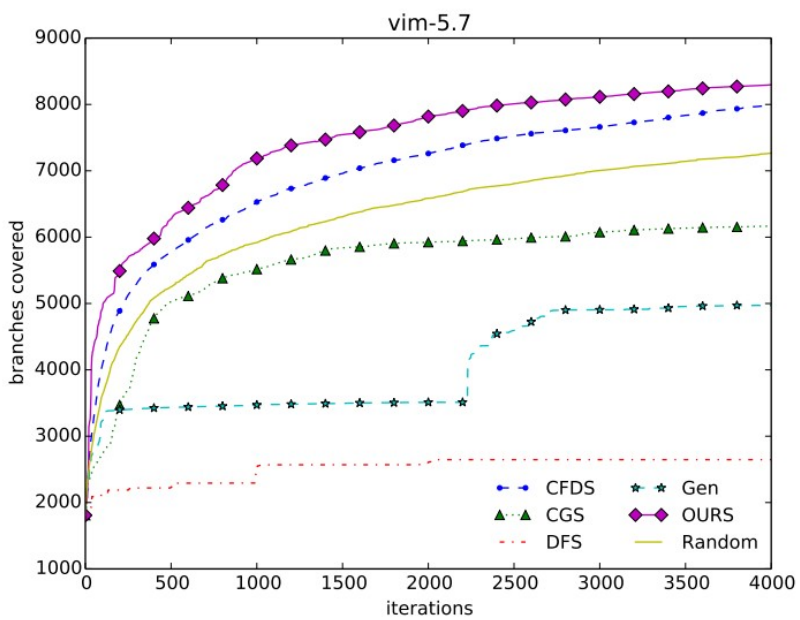
- Automatically Generating Search Heuristics

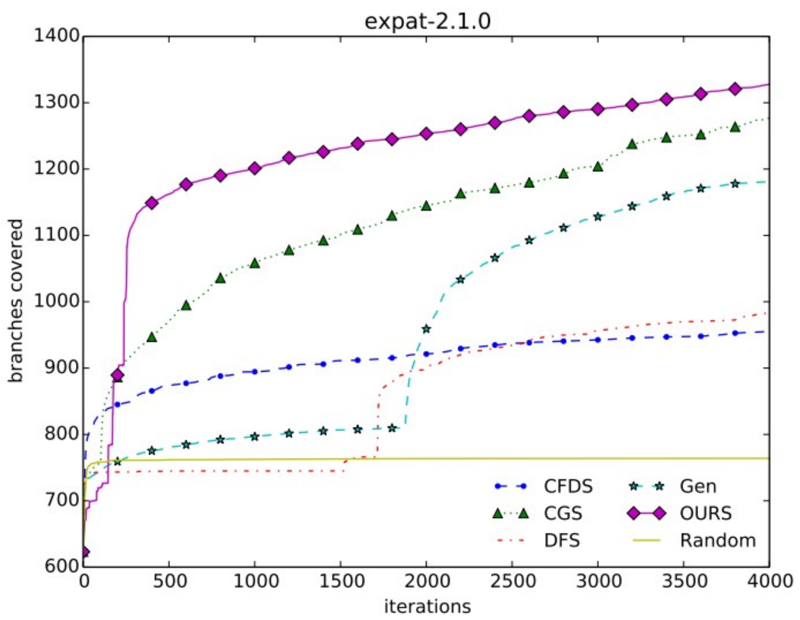


- Key ideas
  - Parameterized Search Heuristic.
  - Effective Parameter Search Algorithm.

Effectiveness

- Considerable increase in branch coverage.



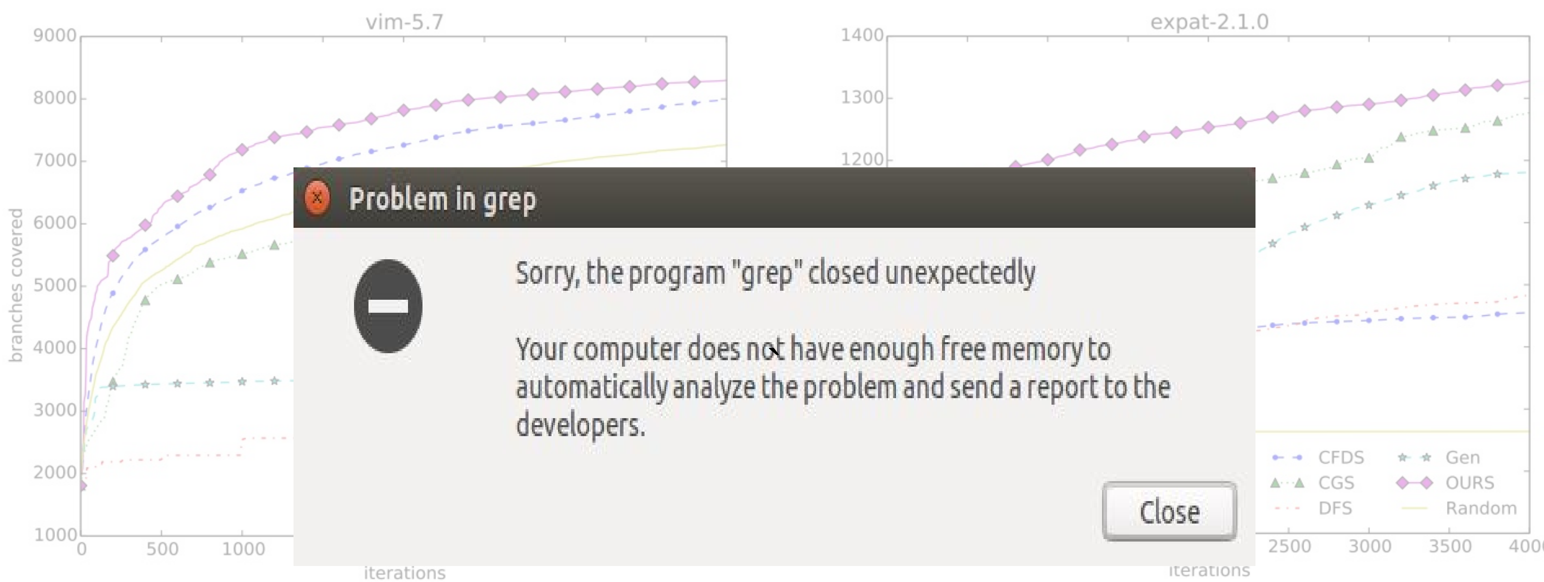


- Found real-world performance bugs.
  - gawk-3.0.3: ./gawk 'V0\\n^000000000000070000000' file
  - grep-2.2: ./grep '\(\)\1\+**' file
    - Trigger the error in grep-3.1

ICSE'18

Effectiveness

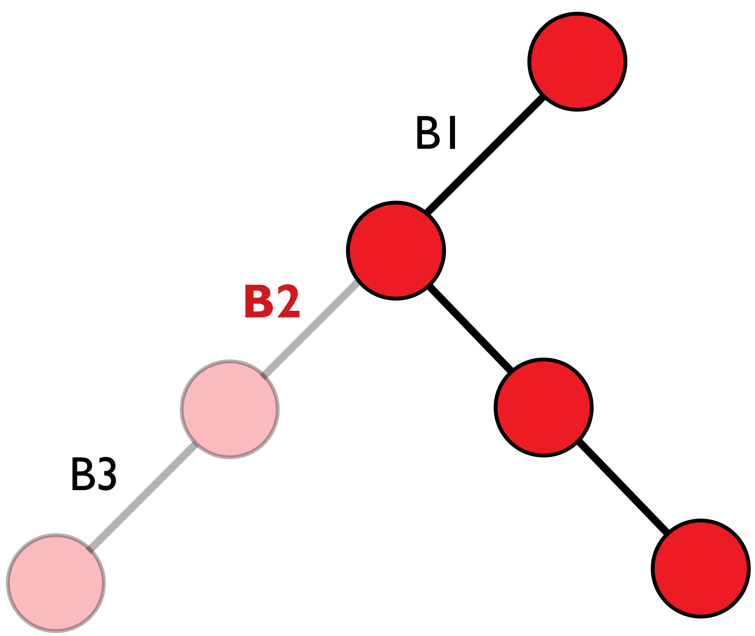
- Considerable increase in branch coverage.



- Found real-world performance bugs.
  - gawk-3.0.3: ./gawk 'V0\\n^0000000000000007000000' file
  - grep-2.2: ./grep '\(\)\1\+**' file
    - Trigger the error in grep-3.1 (the latest version)

Parameterized Search Heuristic

- Search Heuristic$_\theta$: Path → Branch
  - Generating a "good search heuristic" → Finding a "good parameter θ"



score$_\theta$(B1) = 0.1

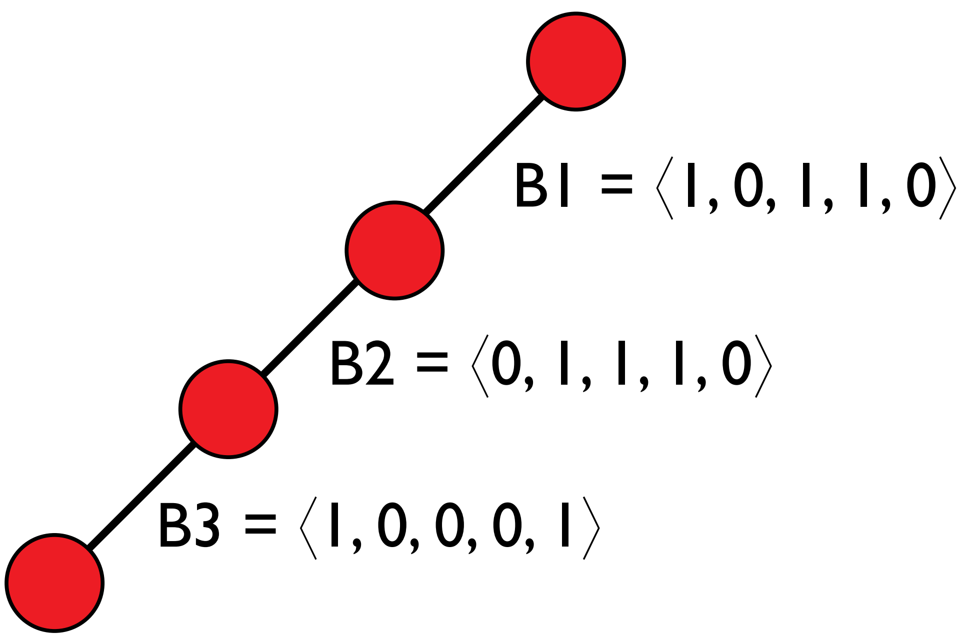
score$_\theta$(B2) = 0.7

score$_\theta$(B3) = -0.5

Parameterized Search Heuristic

(1). Represent branches as feature vectors

- A feature : a boolean predicate on branches.
    - ex1) the branch in main function ?
    - ex2) true branch of a case statement ?



Parameterized Search Heuristic

- Design 40 features.
  - 12 static features
    - extracted without executing a program.

      (e.g., true branch of a loop)
  - 28 dynamic features
    - extracted by the program execution.

      (e.g., branch newly covered in the previous execution)

#	Description
1	branch in the main function
2	true branch of a loop
3	false branch of a loop
4	nested branch
5	branch containing external function calls
6	branch containing integer expressions
7	branch containing constant strings
8	branch containing pointer expressions
9	branch containing local variables
10	branch inside a loop body
11	true branch of a case statement
12	false branch of a case statement
13	first 10% branches of a path
14	last 10% branches of a path
15	branch appearing most frequently in a path
16	branch appearing least frequently in a path
17	branch newly covered in the previous execution
18	branch located right after the just-negated branch
19	branch whose context ($k = 1$) is already visited
20	branch whose context ($k = 2$) is already visited
21	branch whose context ($k = 3$) is already visited
22	branch whose context ($k = 4$) is already visited
23	branch whose context ($k = 5$) is already visited
24	branch negated more than 10 times
25	branch negated more than 20 times
26	branch negated more than 30 times
27	branch near the just-negated branch
28	branch failed to be negated more than 10 times
29	the opposite branch failed to be negated more than 10 times
30	the opposite branch is uncovered (depth 0)
31	the opposite branch is uncovered (depth 1)
32	branch negated in the last 10 executions
33	branch negated in the last 20 executions
34	branch negated in the last 30 executions
35	branch in the function that has the largest number of uncovered branches
36	the opposite branch belongs to unreached functions (top 10% of the largest func.)
37	the opposite branch belongs to unreached functions (top 20% of the largest func.)
38	the opposite branch belongs to unreached functions (top 30% of the largest func.)
39	the opposite branch belongs to unreached functions (# of branches > 10)
40	branch inside the most recently reached function

Parameterized Search Heuristic

(2). Scoring

- The parameter : a k-dimension vector.

$$\theta = \langle -0.5, 0.1, 0.4, 0.2, 0 \rangle$$

- Linear combination of feature vector and parameter
  - $\mathrm{Score}_\theta(B1) = \langle 1, 0, 1, 1, 0 \rangle \cdot \langle -0.5, 0{,}1, 0.4, 0.2, 0 \rangle = 0.1$
  - $\mathrm{Score}_\theta(B2) = \langle 0, 1, 1, 1, 0 \rangle \cdot \langle -0.5, 0.1, 0.4, 0.2, 0 \rangle = 0.7$
  - $\mathrm{Score}_\theta(B3) = \langle 1, 0, 0, 0, 1 \rangle \cdot \langle -0.5, 0.1, 0.4, 0.2, 0 \rangle = -0.5$

(3). Choosing the branch with the highest score

- B2

Parameter Search Algorithm

- Finding good parameters is crucial.
- Naive algorithm based on random sampling.

$\theta_1 = \langle -0.5, 0.1, 0.4, 0.2, 0 \rangle \rightarrow$ Coverage(519)

$\theta_2 = \langle -0.9, 0.5, 0.9, -0.2, 1.0 \rangle \rightarrow$ Coverage(423) $\xrightarrow[\text{Timeout}]{}$ θ_{22} (best)

...

$\theta_n = \langle 0.7, -0.2, -0.9, -0.9, 0.3 \rangle \rightarrow$ Coverage(782)

  – Failed to find good parameters.
    - Search space is intractably large.
    - Performance variation in concolic testing.

Parameter Search Algorithm

- Our Algorithm
  - Iteratively refine the sample search space via feedback.
  - Repeat the three steps. (Find, Check, Refine)

0. The 40 sample spaces are Initialized: [-1, 1]

   - $\theta = \langle -0.5, 0.1, 0.4, 0.2, \ldots, 0 \rangle$

     $$\underset{[-1,\,1]}{\uparrow} \quad \underset{[-1,\,1]}{\uparrow} \quad \ldots \quad \underset{[-1,\,1]}{\uparrow}$$

1. Find

   - Find good candidate parameters quickly.

grep-2.2	+	$\theta_1(1{,}230),_2(1{,}100), \theta_3(1{,}321), \ldots, \theta_{1,000}(872)$	→	Top 10 θ'

Parameter Search Algorithm

- Our Algorithm

2. Check : rule out unreliable parameters.

grep-2.2	+	*Avg* $\theta'_1(1,310)$, *Avg* $\theta'_2(1,457)$, *Avg* $\theta'_3(1,436)$, …, *Avg* $\theta_{10}(1,500)$.

$\downarrow$

Top 2 ($\theta_{t1,}$ θ_{t2})

3. Refine

- $\theta_{t1} = \langle\ +0.3,\ -0.6,\ +0.6,\ \ldots,\ +0.8\ \rangle$
- $\theta_{t2} = \langle\ +0.7,\ -0.2,\ -0.7,\ \ldots,\ +0.8\ \rangle$

Parameter Search Algorithm

- Our Algorithm

2. Check

grep-2.2 + *Avg* θ'_1(1,310), *Avg* θ'_2(1,457), *Avg* θ'_3(1,436), …, *Avg* θ_{10}(1,500).

↓

Top 2 (θ_{t1}, θ_{t2})

3. Refine

- $\theta_{t1} = \langle +0.3, -0.6, +0.6, ..., +0.8 \rangle$
- $\theta_{t2} = \langle +0.7, -0.2, -0.7, ..., +0.8 \rangle$

1[st] Sample Space: [-1, 1] → [*min*(0.3, 0.7), 1] → [0.3, 1]

Parameter Search Algorithm

- Our Algorithm

2. Check

grep-2.2 + *Avg* $\theta'_1(1{,}310)$, *Avg* $\theta'_2(1{,}457)$, *Avg* $\theta'_3(1{,}436)$, …, *Avg* $\theta_{10}(1{,}500)$.

↓

Top 2 (θ_{t1}, θ_{t2})

3. Refine

- $\theta_{t1} = \langle +0.3,\ -0.6,\ +0.6,\ \ldots,\ +0.8 \rangle$
- $\theta_{t2} = \langle +0.7,\ -0.2,\ -0.7,\ \ldots,\ +0.8 \rangle$

2nd Sample Space: [-1, 1] → [-1, *max*(-0.6, -0.2)] → [-1, -0.2]

Parameter Search Algorithm

- Our Algorithm

2. Check

grep-2.2 + *Avg* θ'_1(1,310), *Avg* θ'_2(1,457), *Avg* θ'_3(1,436), …, *Avg* θ_{10}(1,500).

Top 2 (θ_{t1}, θ_{t2})

3. Refine

  - $\theta_{t1} = \langle +0.3, -0.6, +0.6, ..., +0.8 \rangle$
  - $\theta_{t2} = \langle +0.7, -0.2, -0.7, ..., +0.8 \rangle$

  3rd Sample Space: [-1, 1] → [-1,1]

Parameter Search Algorithm

- Our Algorithm
  - The 40 sample spaces are refined !
    - $\theta = \langle 0.5, -0.4, 0.4, 0.2, \ldots, 0 \rangle$

      $$\begin{array}{cccc c} \uparrow & \uparrow & \uparrow & & \uparrow \\ [0.3, 1] & [-1, -0.2] & [-1, 1] & \ldots & [-1, 1] \end{array}$$

  - 'Find' stage again !
    - Randomly sample the parameters in refined sample space !

Experiments

- Implemented in CREST
- Compared with five existing heuristics
  - CGS, CFDS, Random, Generational, DFS
- Used 10 open-source C programs

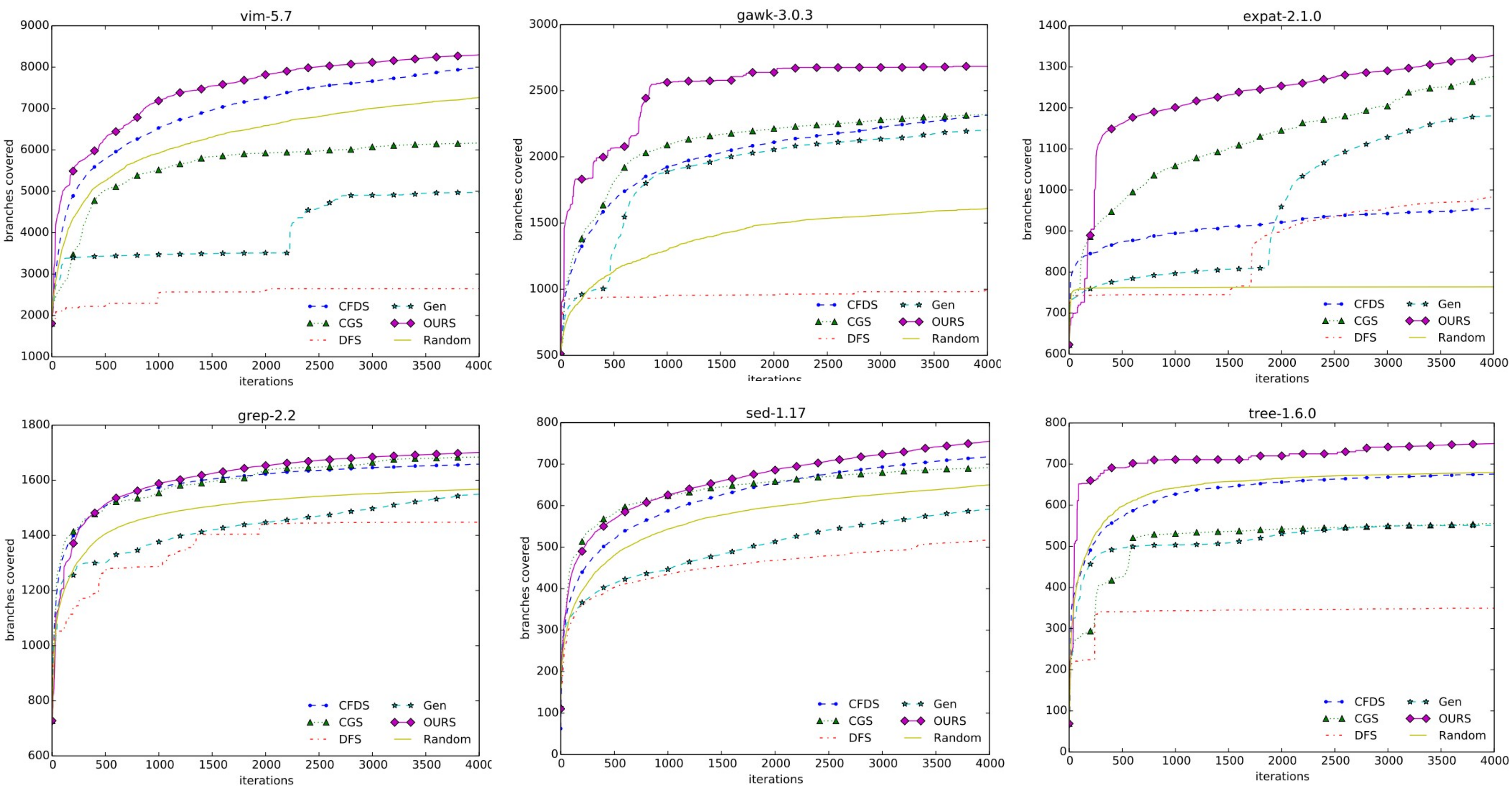
Program	# Total branches	LOC
vim-5.7	35,464	165K
gawk-3.0.3	8,038	30K
expat-2.1.0	8,500	49K
grep-2.2	3,836	15K
sed-1.17	2,656	9K
tree-1.6.0	1,438	4K
cdaudio	358	3K
floppy	268	2K
kbfiltr	204	1K
replace	196	0.5K

Evaluation Setting

- The same initial inputs
- The same testing budget (4,000 executions)
- Average branch coverage for 100 trials (50 for vim)
  - 1 trial = 4,000 executions

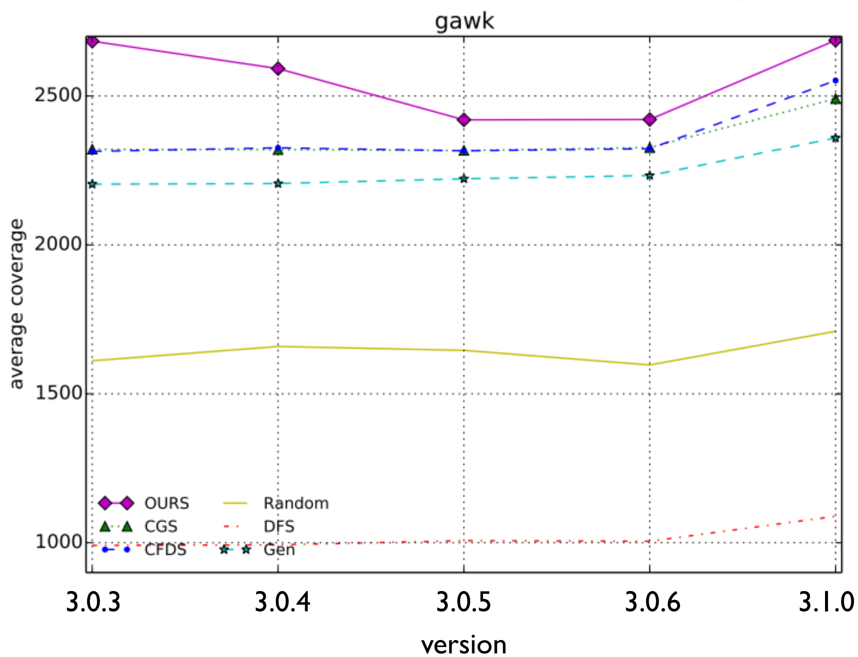
Effectiveness

- Average branch coverage (6 Smiles)

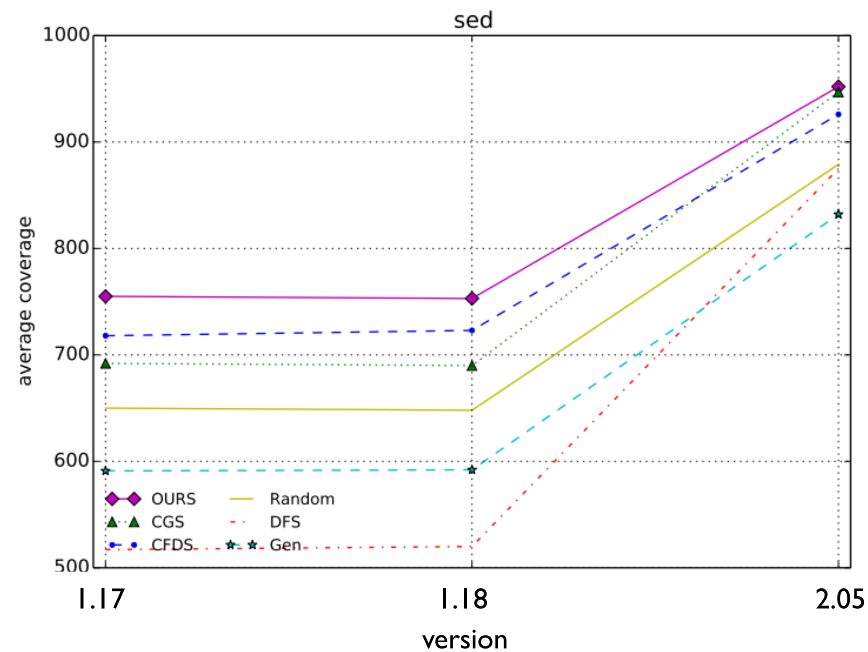


Effectiveness

- Reusable over multiple subsequent programs



(4 Years)



(A Year)

- Time for obtaining the heuristics (with 20 cores)
  - vim-5.7(24 h), expat-2.1.0(10h), grep-2.2(5h), tree-1.6.0(3h)

Tool

- Make our tool publicly available.
  - Parametric Dynamic Symbolic Execution

Tool: ParaDySE

- Make our tool publicly available.
  - Parametric Dynamic Symbolic Execution

https://github.com/kupl/ParaDySE

Summary

- There are two major approaches of dynamic symbolic execution: concolic testing and execution-generated testing.
- Symbolic execution still suffers from two open challenges: path-explosion problem and constraint solving cost.
- Search heuristic is a key technique for mitigating the path-explosion problem.
  - CFDS, CGS, Generational, Param

Thank You