ROYAL
HOLLOWAY
UNIVERSITY
OF LONDON

Efficient Symbolic Execution for Software Testing

Johannes Kinder

Royal Holloway, University of London

Joint work with:

Stefan Bucur, George Candea, Volodymyr Kuznetsov @ EPFL

Symbolic Execution

- Automatically explore program paths
  - *Execute program on “symbolic” input values*
  - *“Fork” execution at each branch*
  - *Record branching conditions*
- Constraint solver
  - *Decides path feasibility*
  - *Generates test cases for paths and bugs*

Symbolic Execution

- (Very brief) history
  - *Test generation by SE in 70s* [King '75] [Boyer et al. '75]
  - *SAT / SMT solvers lead to boom in 2000s* [Godefroid et al. '05][Cadar et al. '06]
- Many successful tools
  - *KLEE, SAGE, PEX, SPF, CREST, Cloud9, S2E, …*
- Specific advantages
  - *No false positives, useful partial results*
  - *Reduces need for modeling*

Outline

- Symbolic Execution for Testing
- State Merging – Fighting Path Explosion
- Interpreted High-Level Code

Outline

- Symbolic Execution for Testing
- State Merging – Fighting Path Explosion
- Interpreted High-Level Code

$pc = \text{true}$

$\mathtt{x} = X$

$\mathtt{r} = 0$

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x – 2
12   }
13
14   return r;
15 }
```

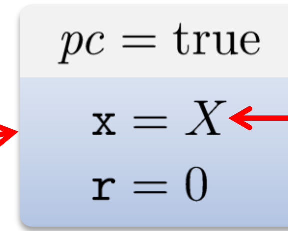
Symbolic program state

$pc = \text{true}$

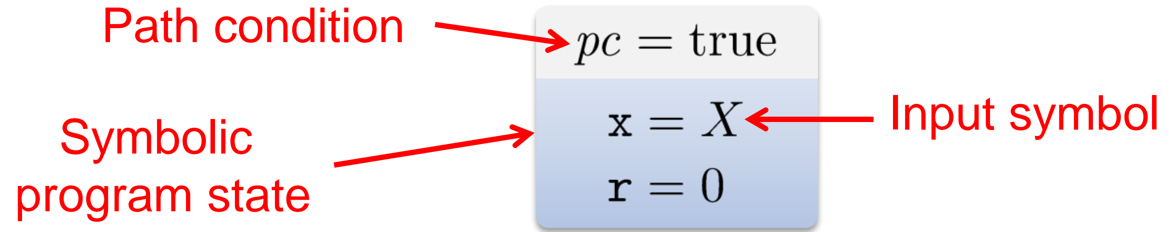
$\texttt{x} = X$

$\texttt{r} = 0$

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x – 2
12   }
13
14   return r;
15 }
```

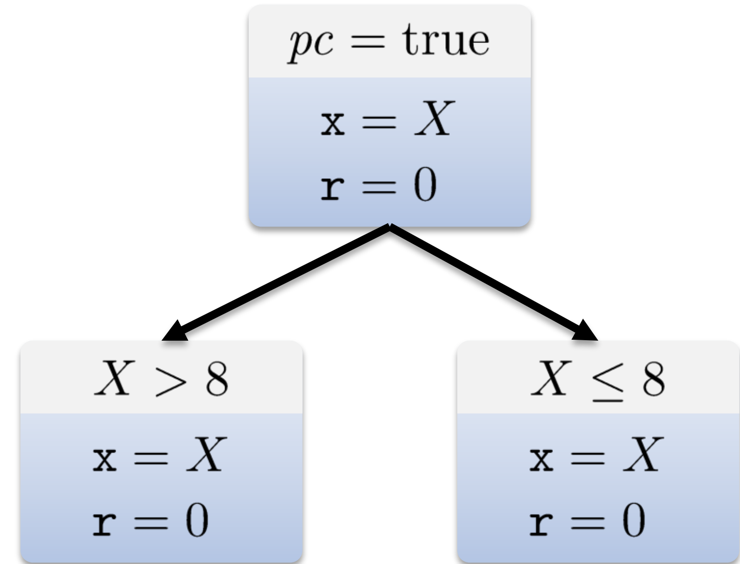



```
1   int proc(int x) {
2
3     int r = 0
5
6     if (x > 8) {
7       r = x - 7
8     }
9
10    if (x < 5) {
11      r = x – 2
12    }
13
14    return r;
15 }
```

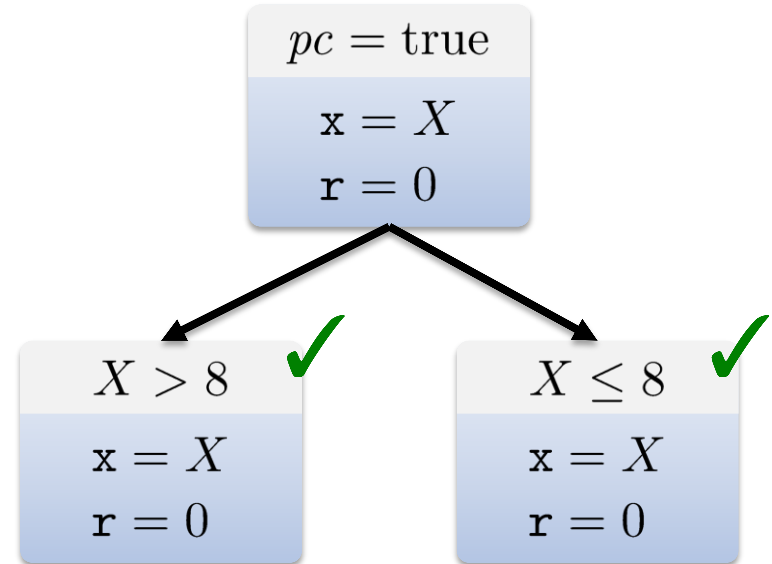



```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x – 2
12   }
13
14   return r;
15 }
```

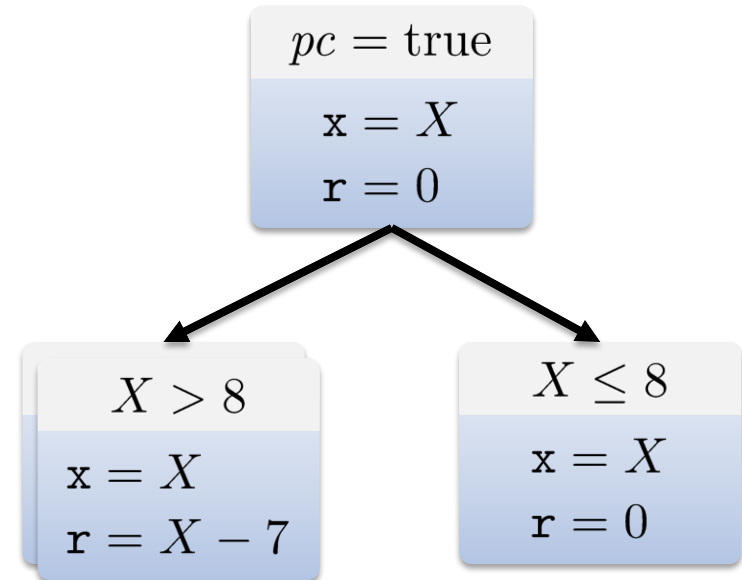
```
1  int proc(int x) {
2
3   int r = 0
5
6   if (x > 8) {
7     r = x - 7
8   }
9
10  if (x < 5) {
11    r = x – 2
12  }
13
14  return r;
15 }
```



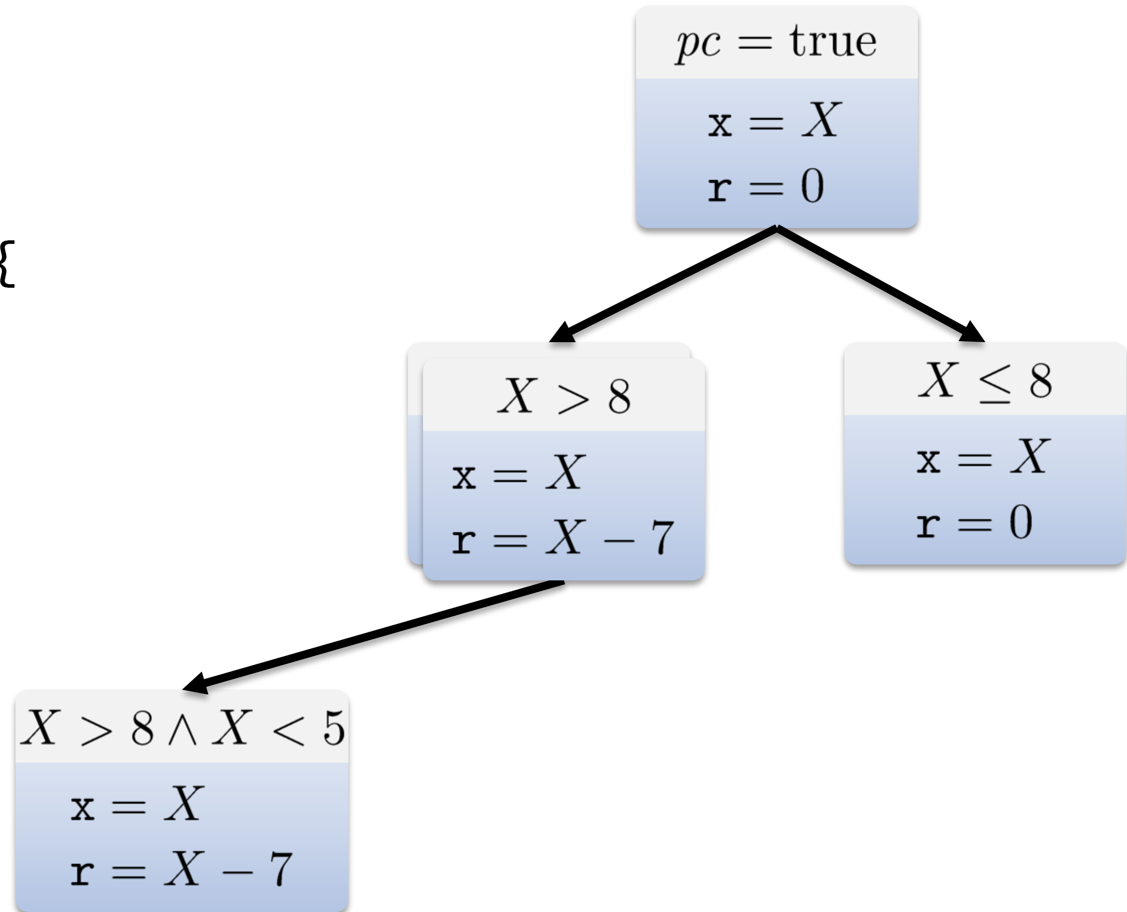
```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x – 2
12   }
13
14   return r;
15 }
```



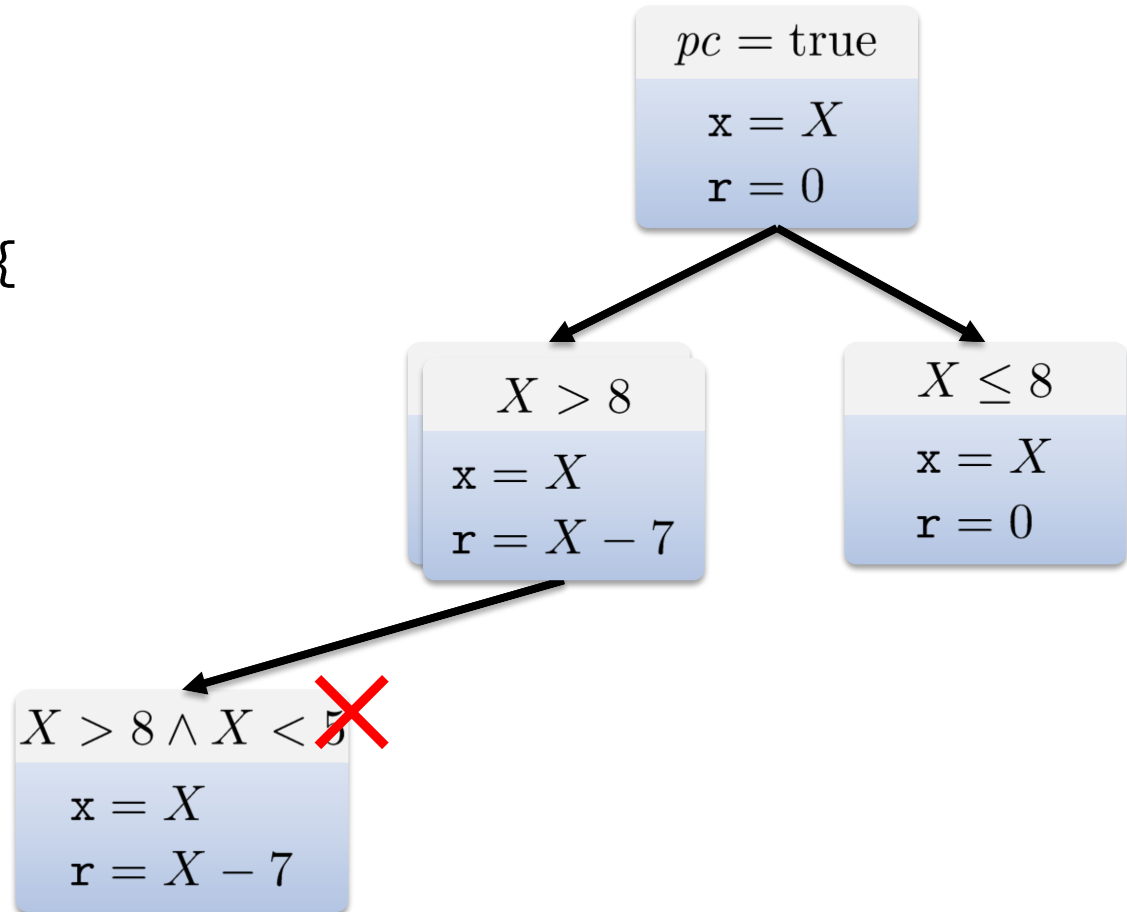
```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x – 2
12   }
13
14   return r;
15 }
```



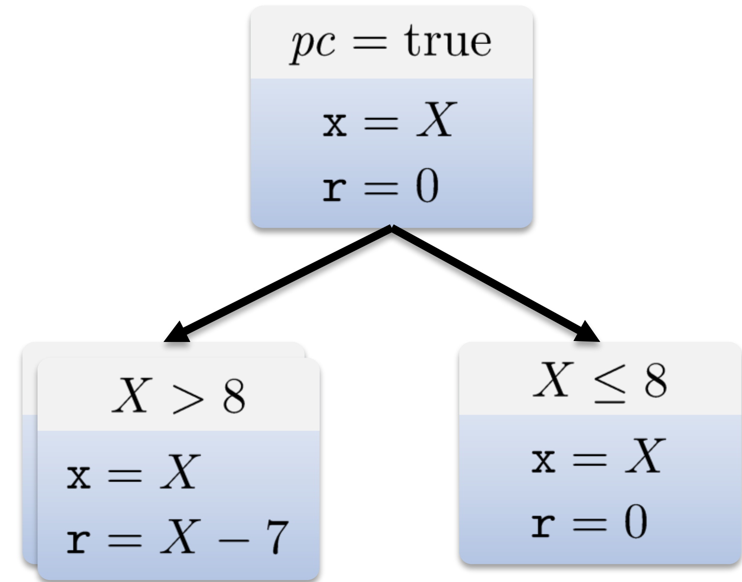
```
1  int proc(int x) {
2
3   int r = 0
5
6   if (x > 8) {
7     r = x - 7
8   }
9
10  if (x < 5) {
11    r = x - 2
12  }
13
14  return r;
15 }
```



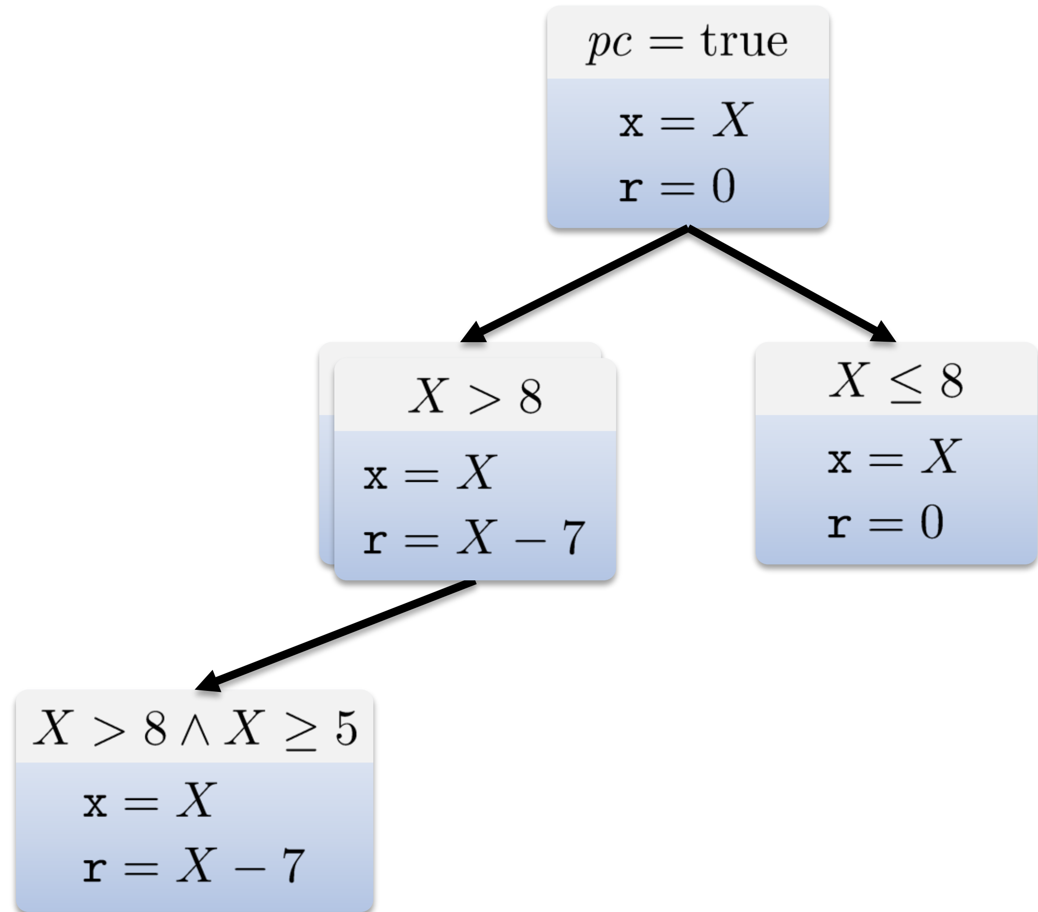
```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x – 2
12   }
13
14   return r;
15 }
```



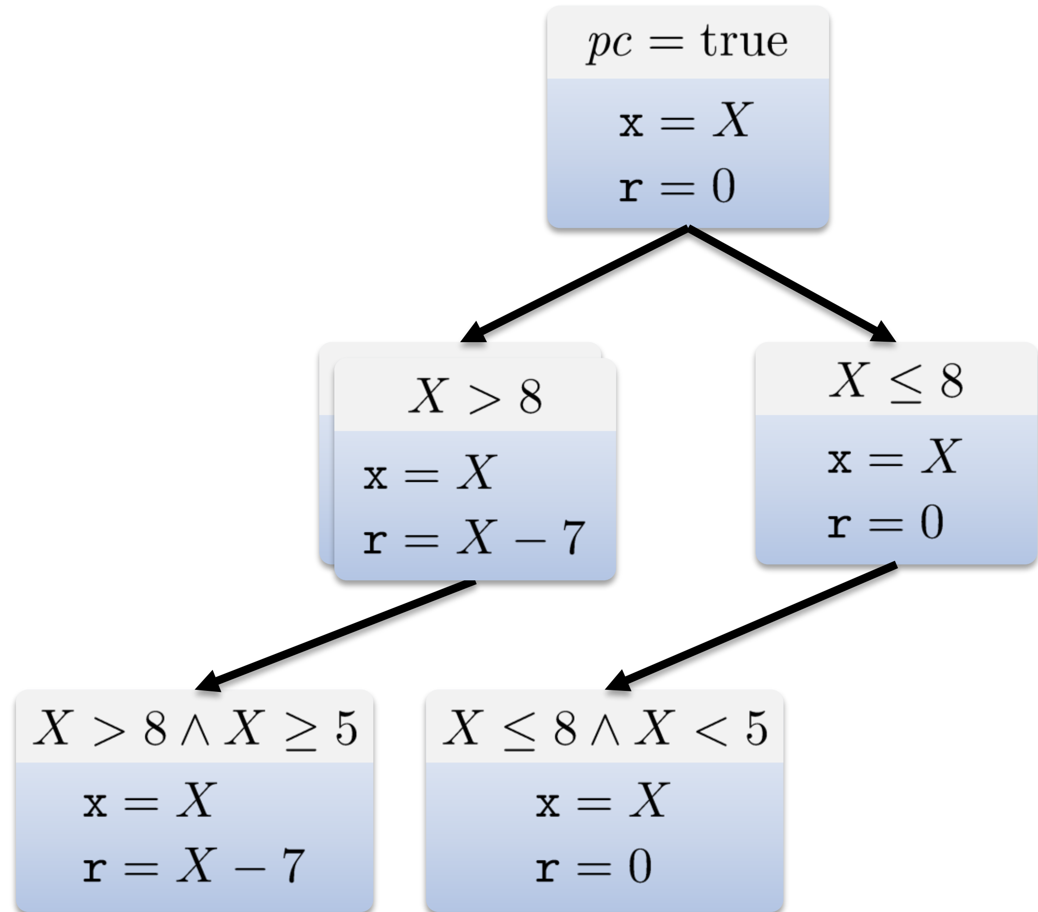
```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



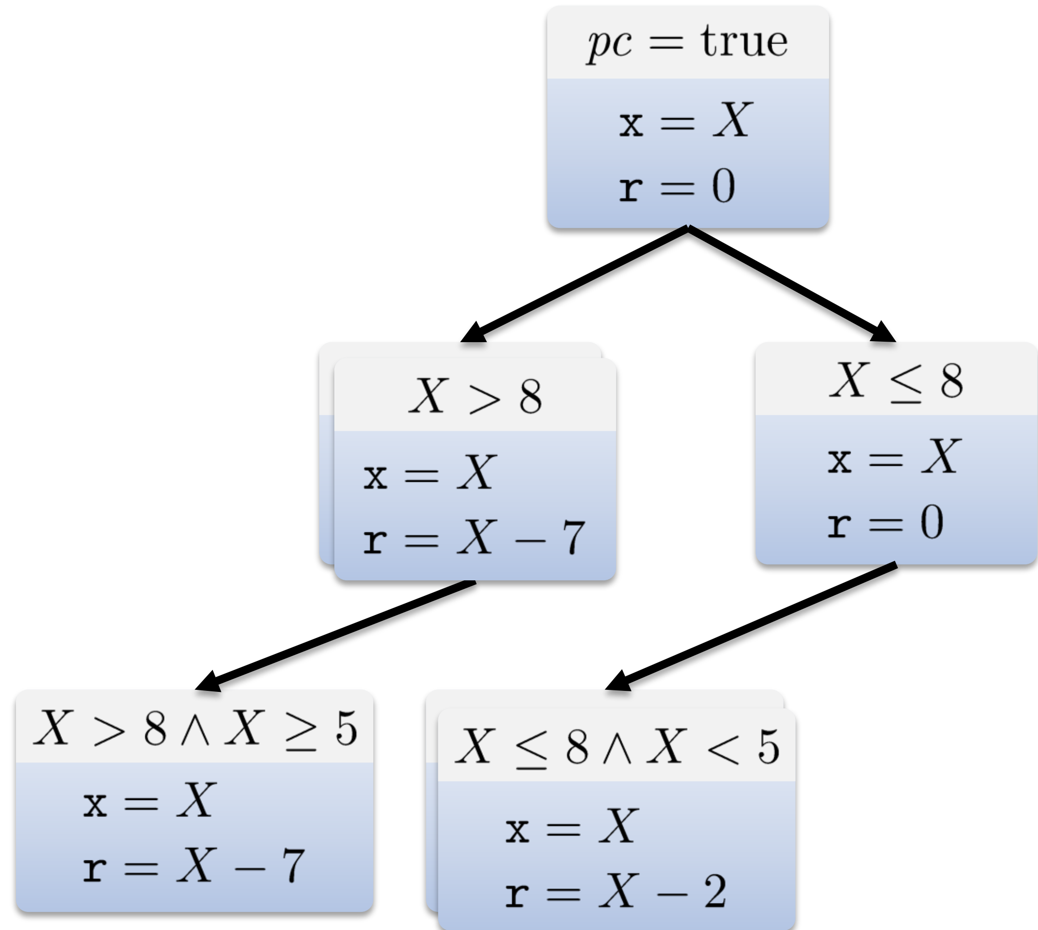
```
1   int proc(int x) {
2
3     int r = 0
5
6     if (x > 8) {
7       r = x - 7
8     }
9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15  }
```



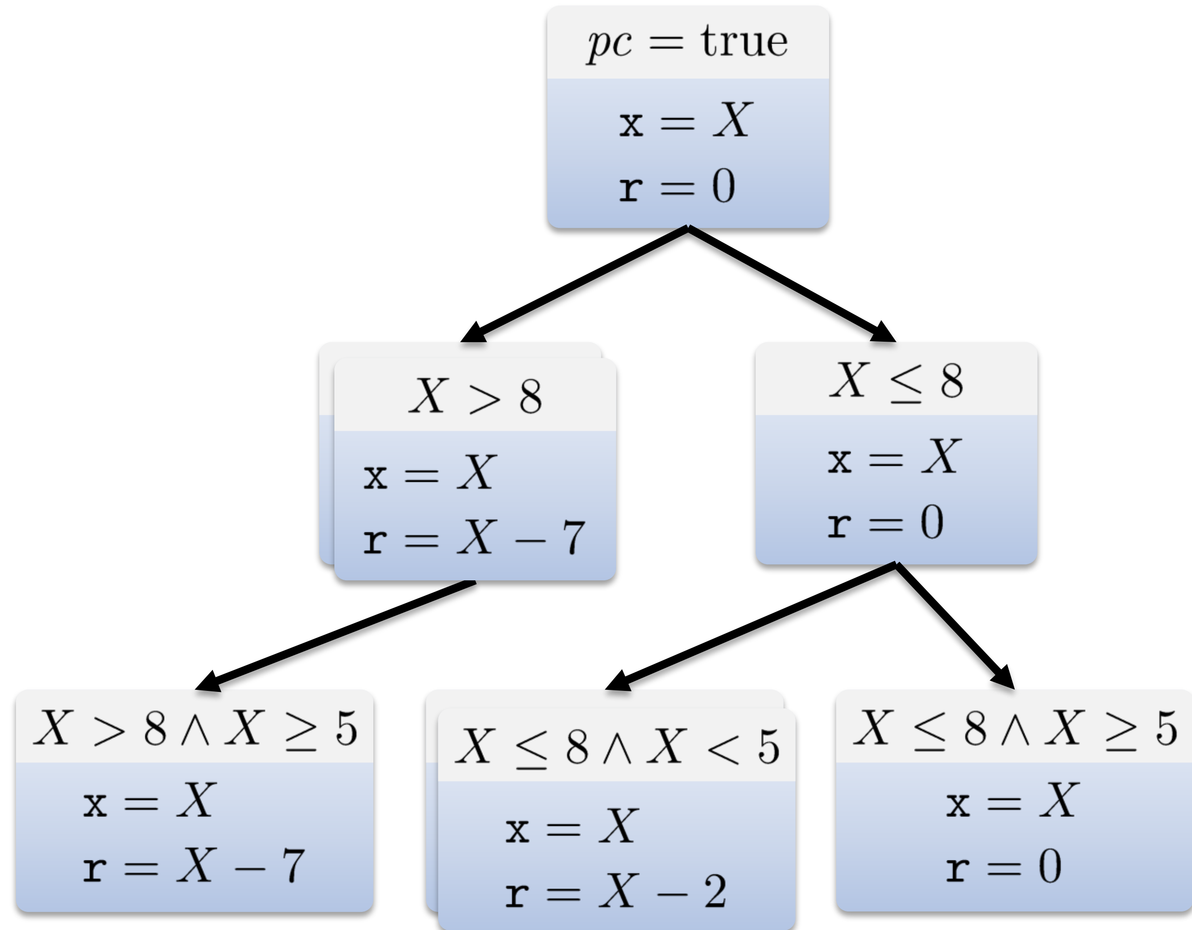
```
1  int proc(int x) {
2
3   int r = 0
5
6   if (x > 8) {
7     r = x - 7
8   }
9
10  if (x < 5) {
11    r = x - 2
12  }
13
14  return r;
15 }
```



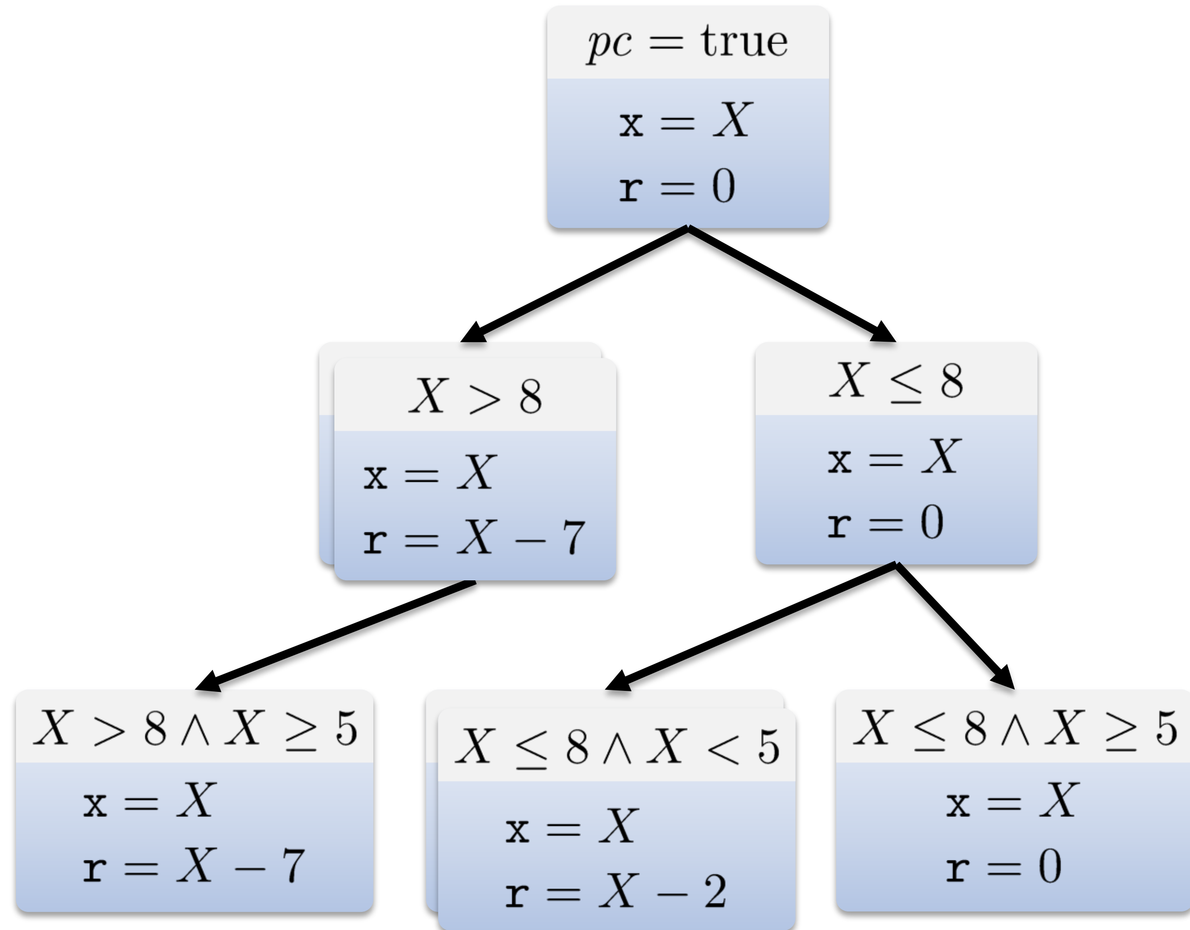
```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



Satisfying assignments:

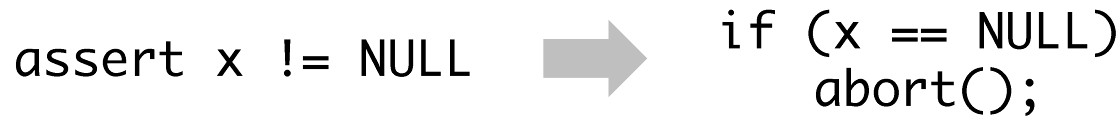
`X = 9` `X = 4` `X = 7`

Test cases:

`proc(9)` `proc(4)` `proc(7)`

Finding Bugs

- Symbolic execution enumerates paths
  - *Runs into bugs that trigger whenever path executes*
  - *Assertions, buffer overflows, division by zero, etc., require specific conditions*
- Error conditions
  - *Treat assertions as conditions*

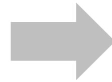


  - *Creates explicit error paths*

Finding Bugs

- Instrument program with properties
  - *Translate any safety property to reachability*
- Division by zero

```
y = 100 / x
```

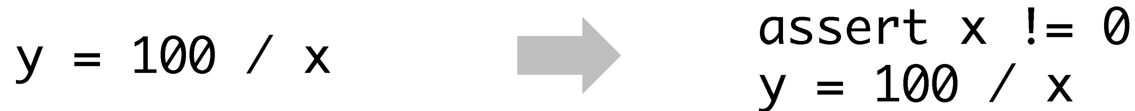


```
assert x != 0
y = 100 / x
```

Finding Bugs

- Instrument program with properties
  - *Translate any safety property to reachability*

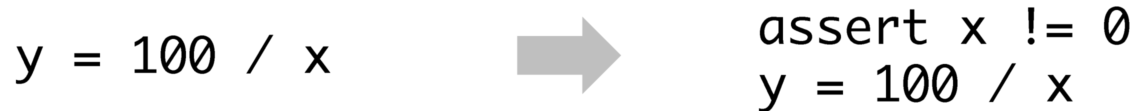
- Division by zero



- Buffer overflows

Finding Bugs

- Instrument program with properties
  - *Translate any safety property to reachability*
- Division by zero



- Buffer overflows

- Implementation is usually implicit

Symbolic Execution Algorithms

- Static symbolic execution
  - *Simulate execution on program source code*
  - *Computes strongest postconditions from entry point*
- Dynamic symbolic execution (DSE)
  - *Run / interpret the program with concrete state*
  - *Symbolic state computed in parallel ("concolic")*
  - *Solver generates new concrete state*
- DSE-Flavors
  - *EXE-style [Cadar et al. '06] vs. DART [Godefroid et al. '05]*

EXE

$$pc = \text{true}$$

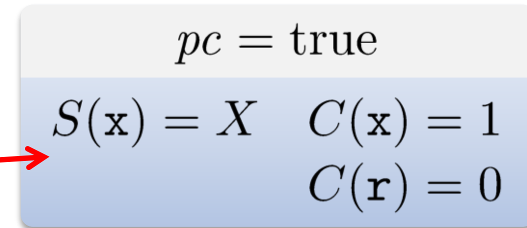
$$S(\mathtt{x}) = X \quad C(\mathtt{x}) = 1$$

$$C(\mathtt{r}) = 0$$

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x – 2
12   }
13
14   return r;
15 }
```

EXE

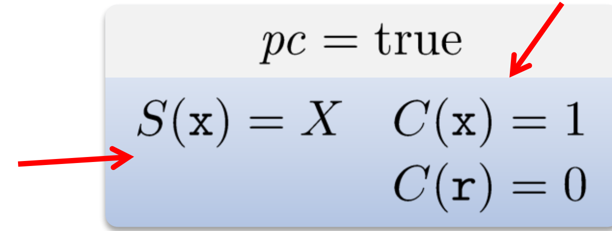
Symbolic program state




```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x – 2
12   }
13
14   return r;
15 }
```

EXE

Symbolic program state




```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x – 2
12   }
13
14   return r;
15 }
```

EXE

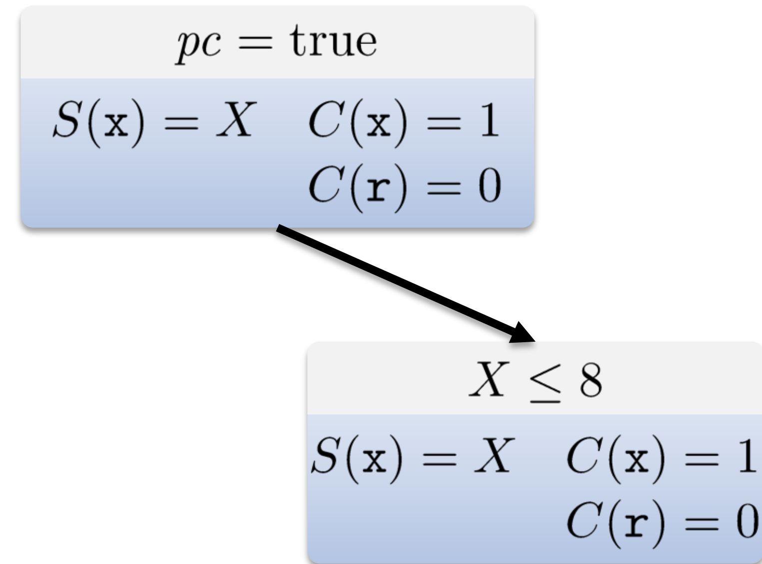
$$pc = \text{true}$$

$$S(\mathbf{x}) = X \quad C(\mathbf{x}) = 1$$
$$C(\mathbf{r}) = 0$$

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x – 2
12   }
13
14   return r;
15 }
```

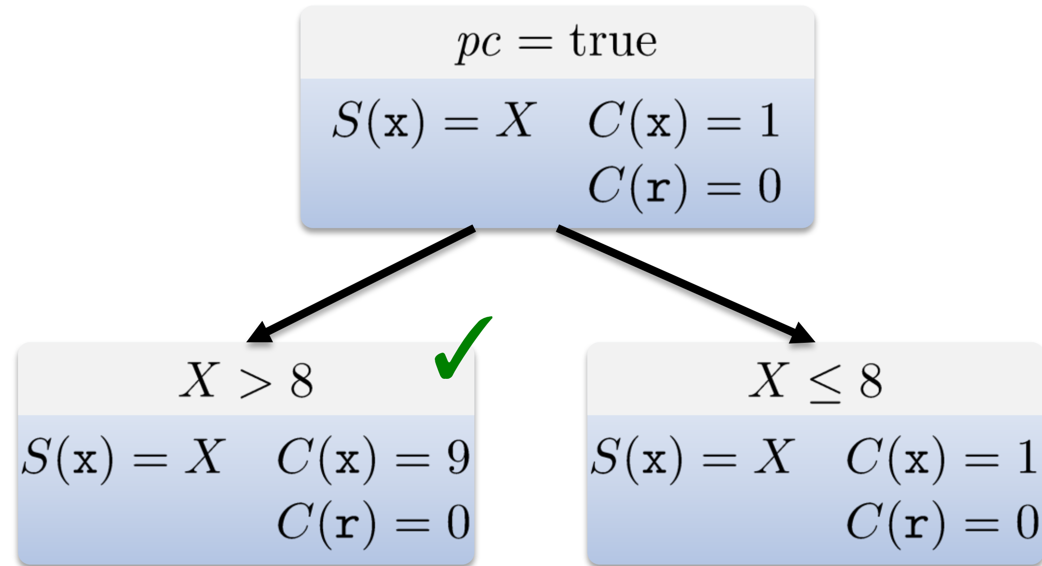
EXE

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x – 2
12   }
13
14   return r;
15 }
```



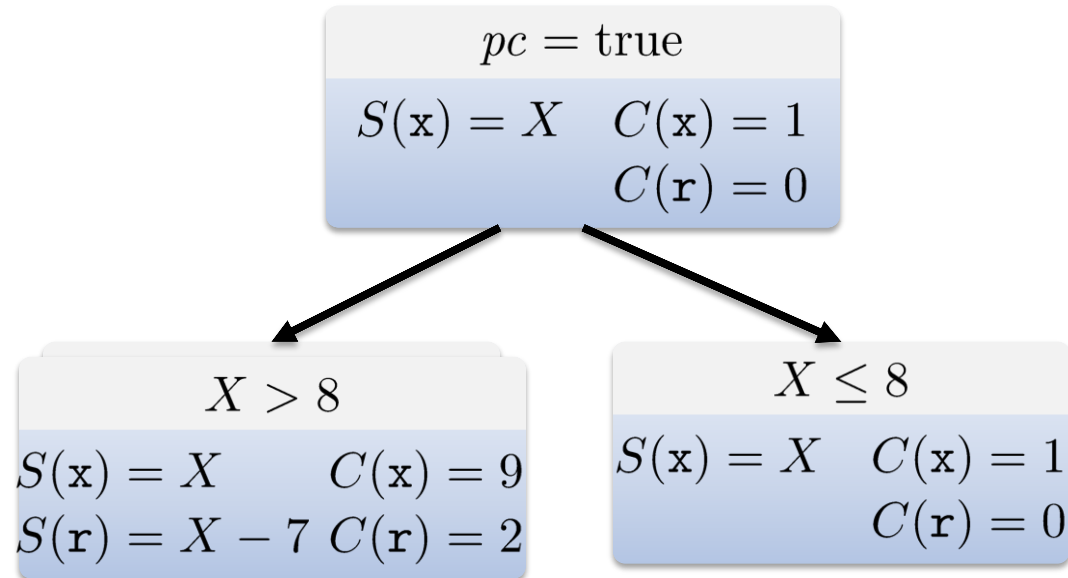
EXE

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x – 2
12   }
13
14   return r;
15 }
```



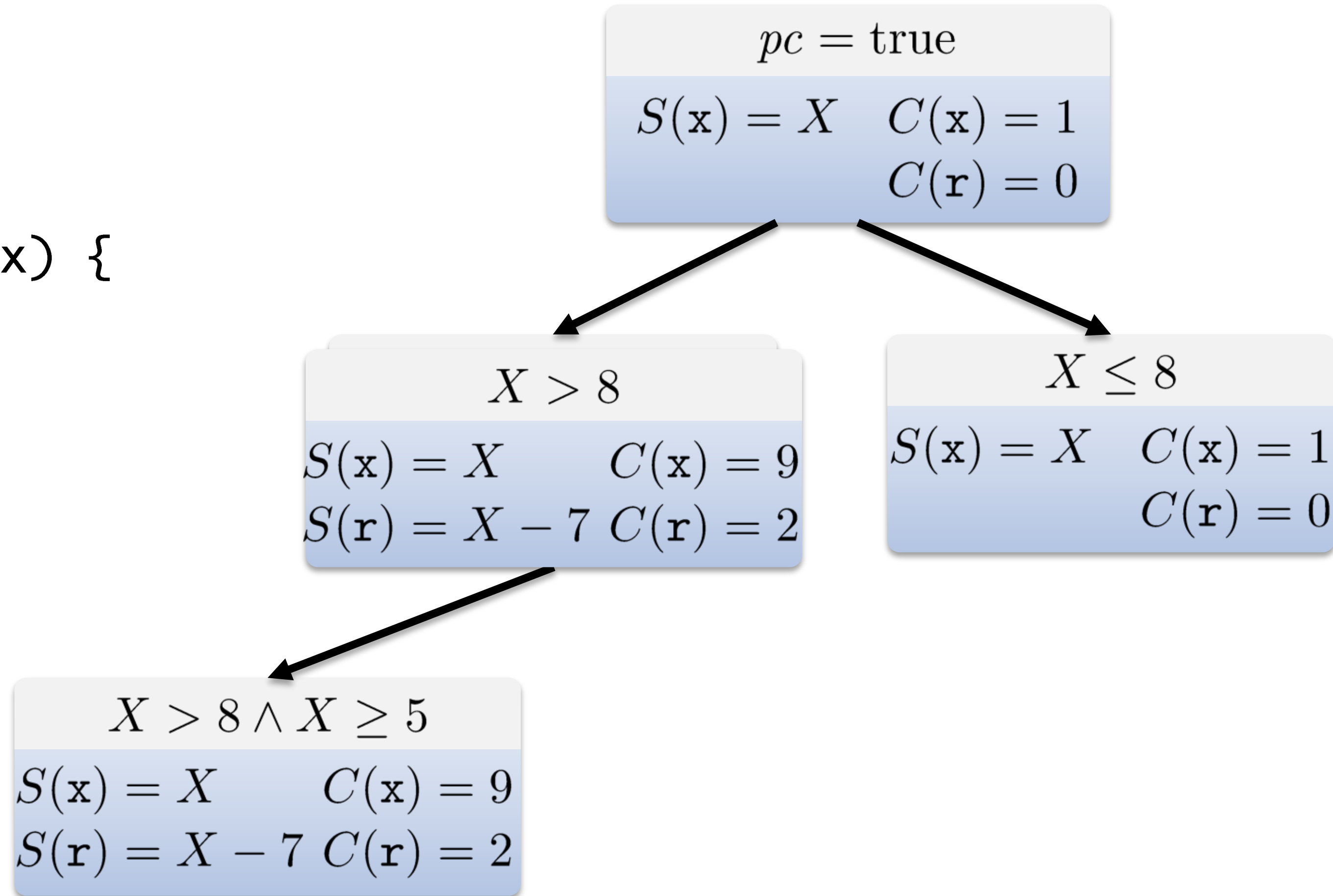
EXE

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x – 2
12   }
13
14   return r;
15 }
```



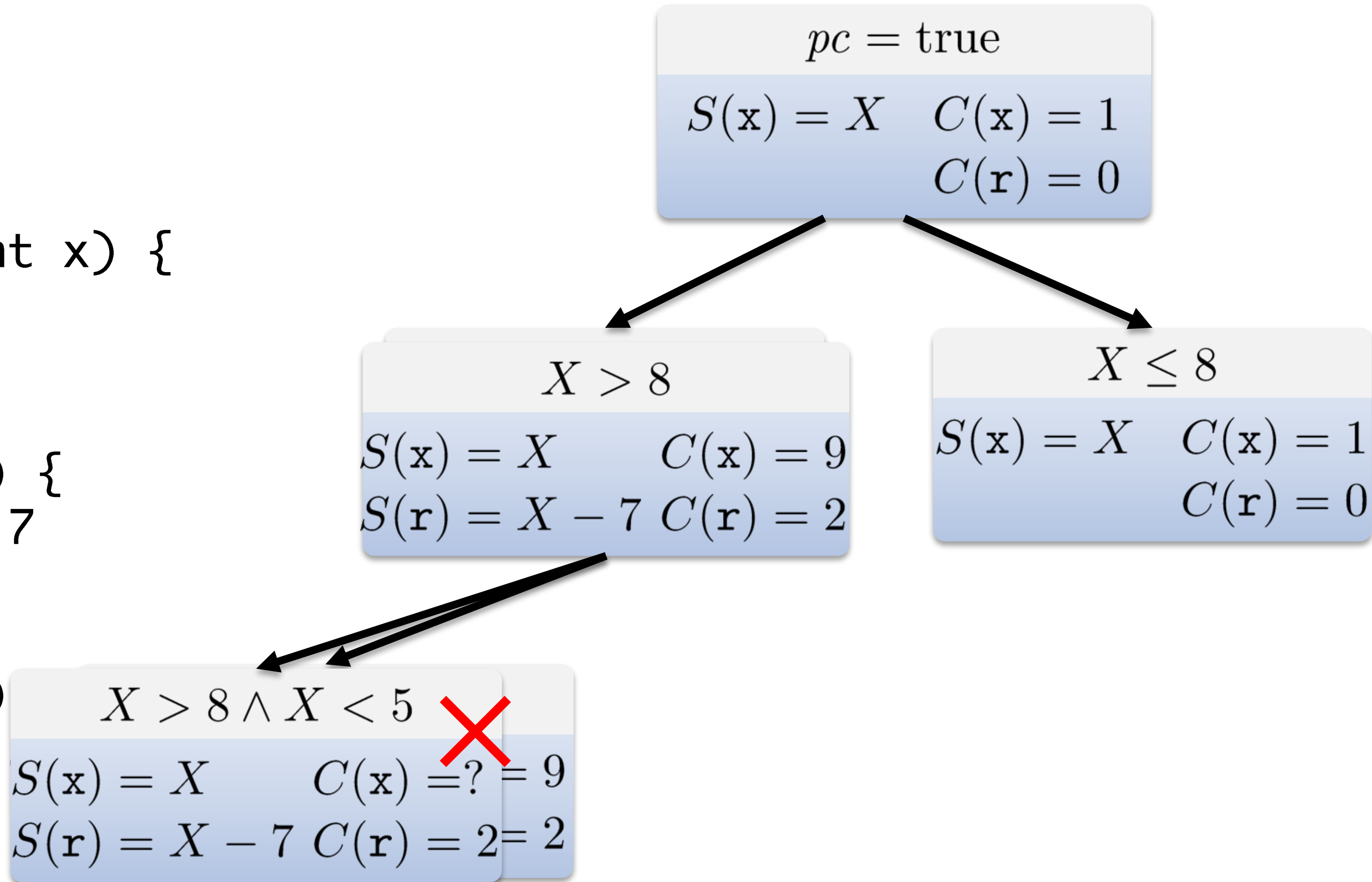
EXE

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



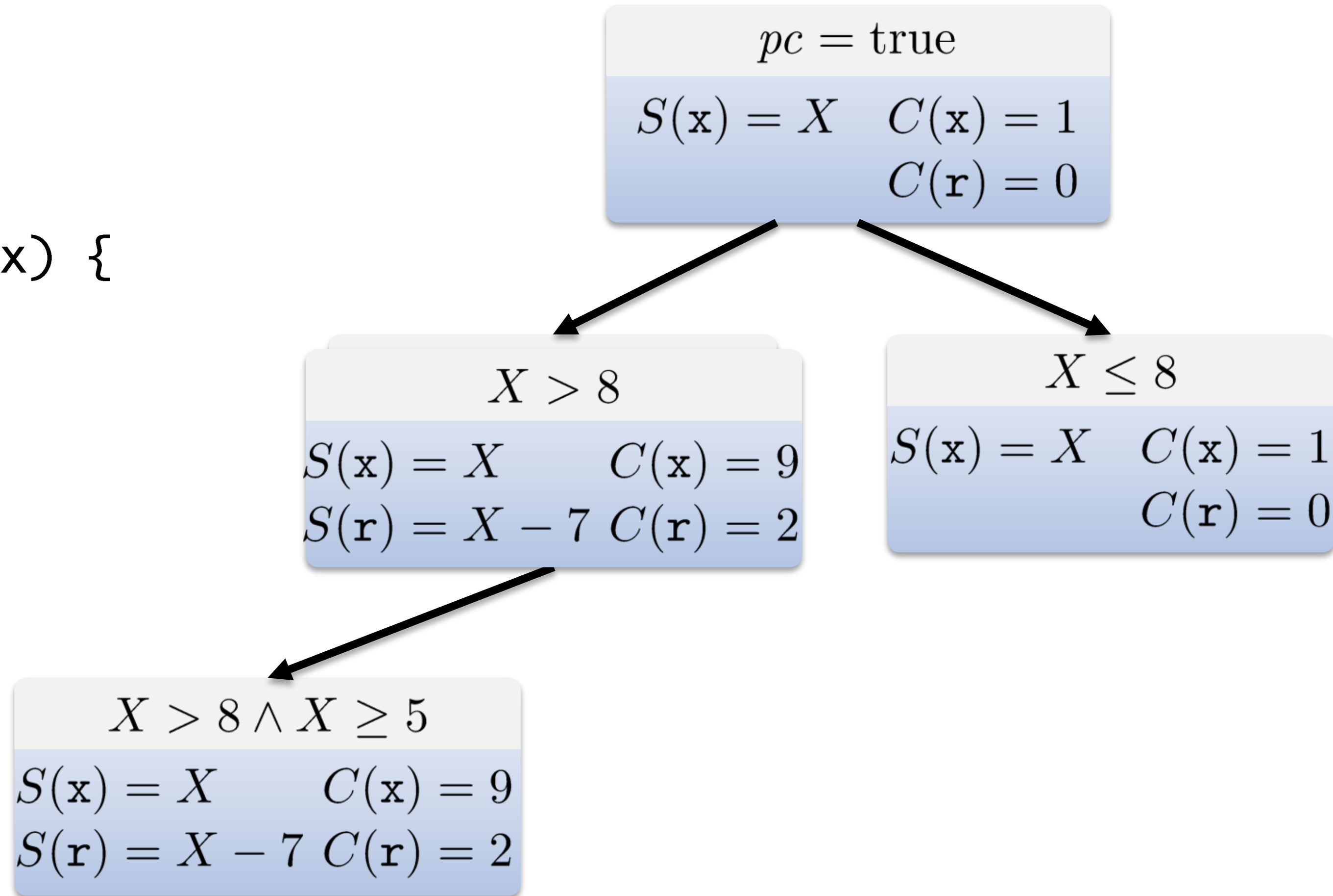
EXE

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5)
11     r = x –
12   }
13
14   return r;
15 }
```



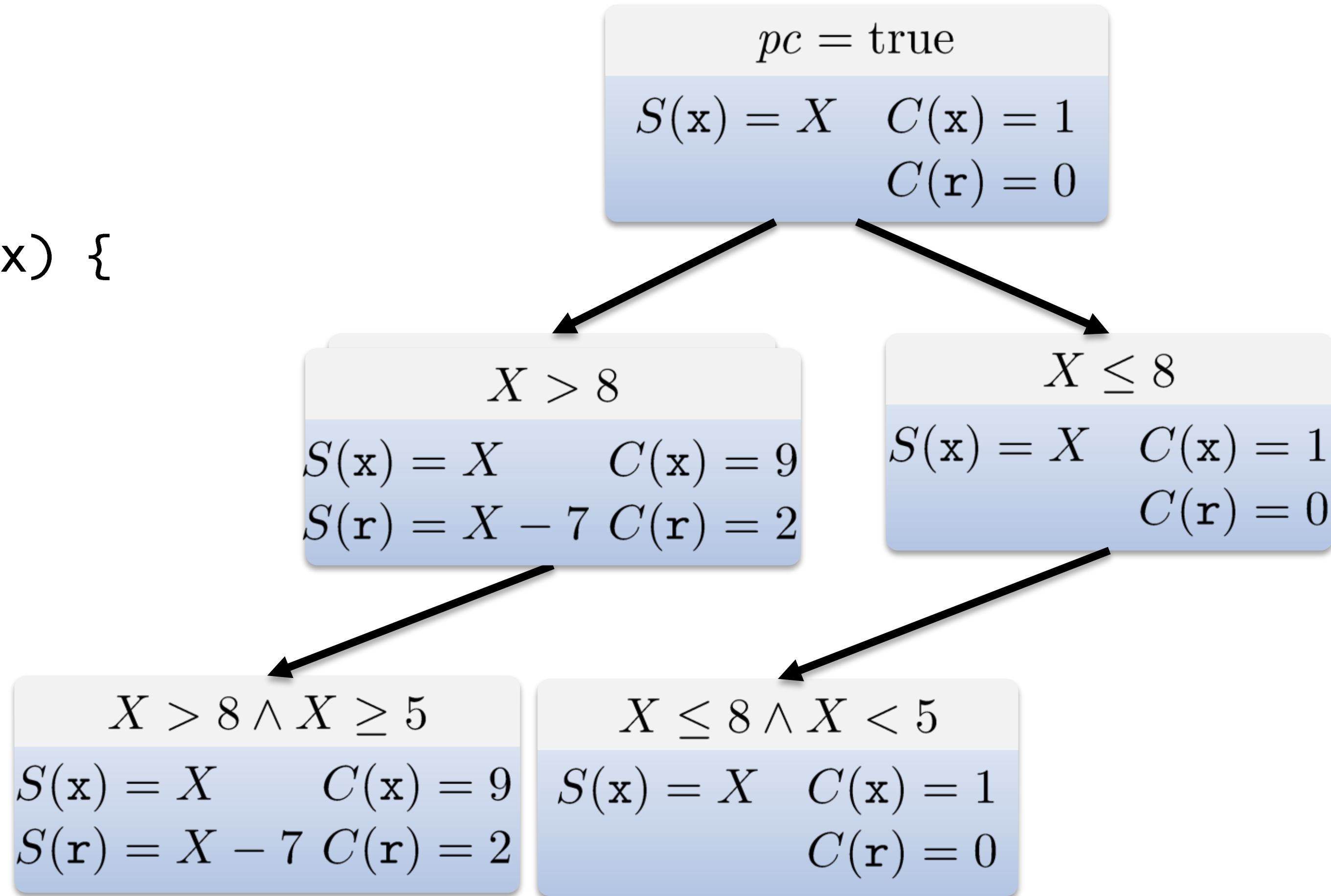
EXE

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



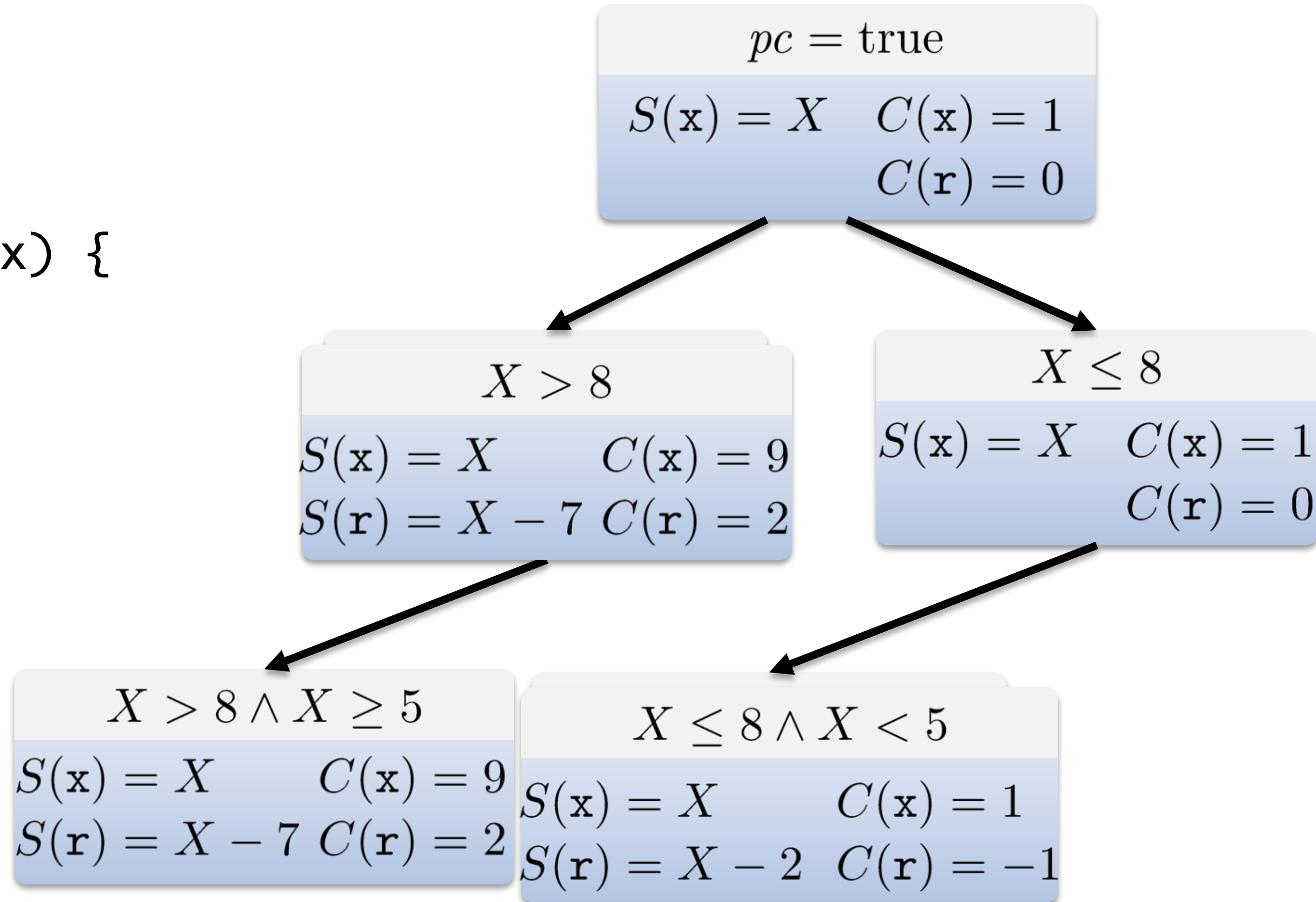
EXE

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



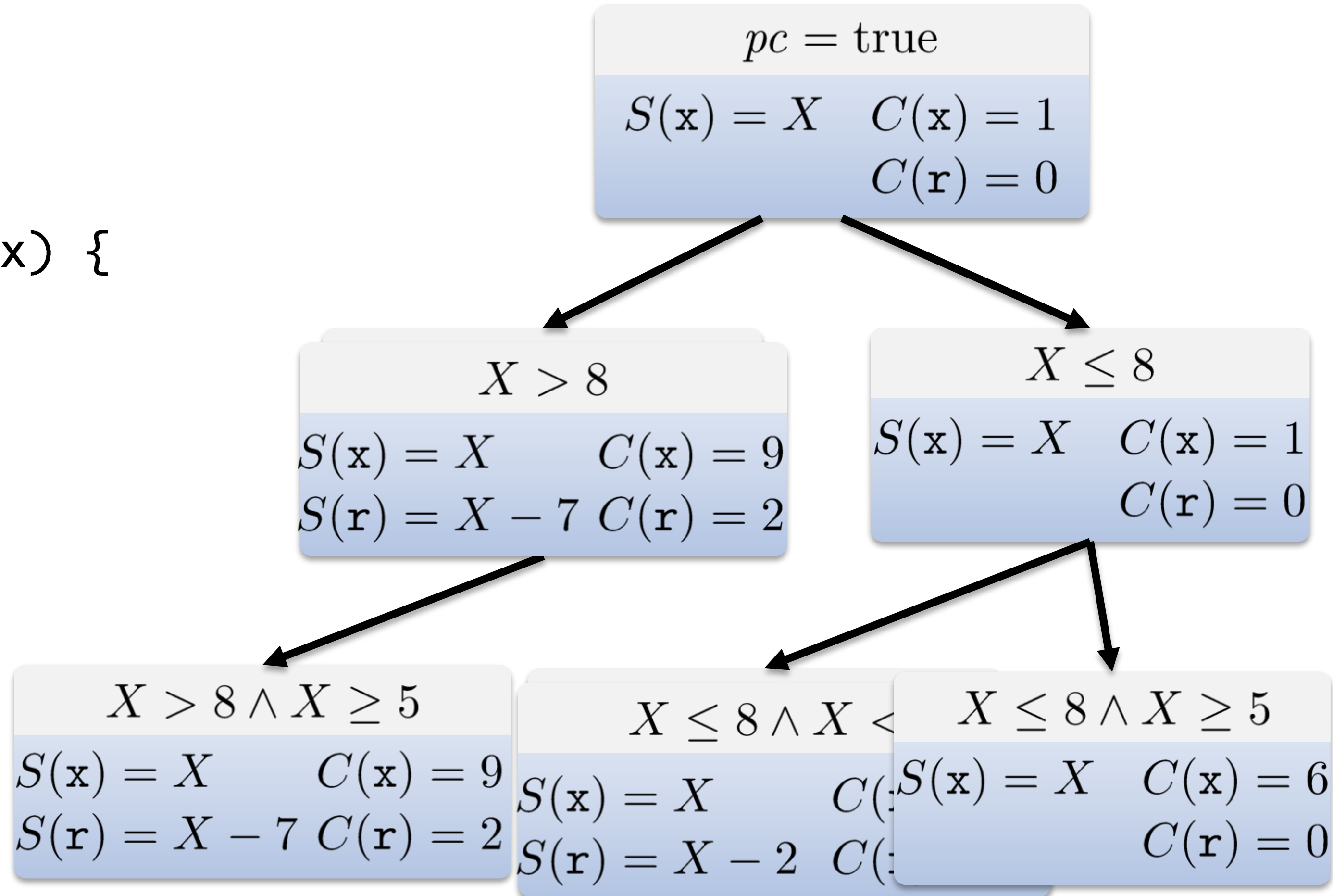
EXE

```
1   int proc(int x) {
2
3     int r = 0
5
6     if (x > 8) {
7       r = x - 7
8     }
9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15  }
```



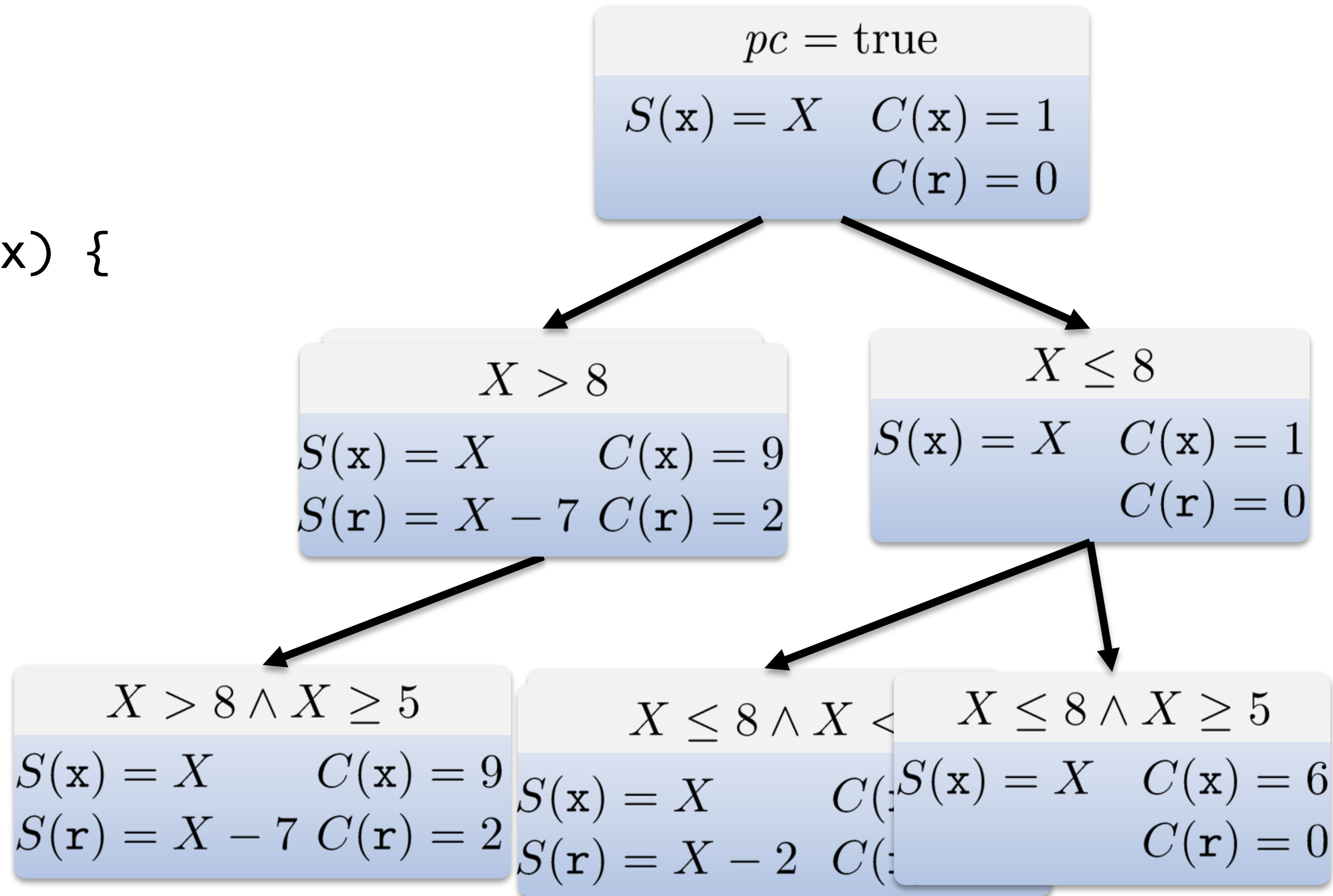
EXE

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



EXE

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



Satisfying assignments:

`X = 9`   `X = 1`   `X = 6`

Test cases:

`proc(9)`   `proc(1)`   `proc(6)`

DART

$$pc = \text{true}$$

$$S(\mathtt{x}) = X \quad C(\mathtt{x}) = 1$$
$$C(\mathtt{r}) = 0$$

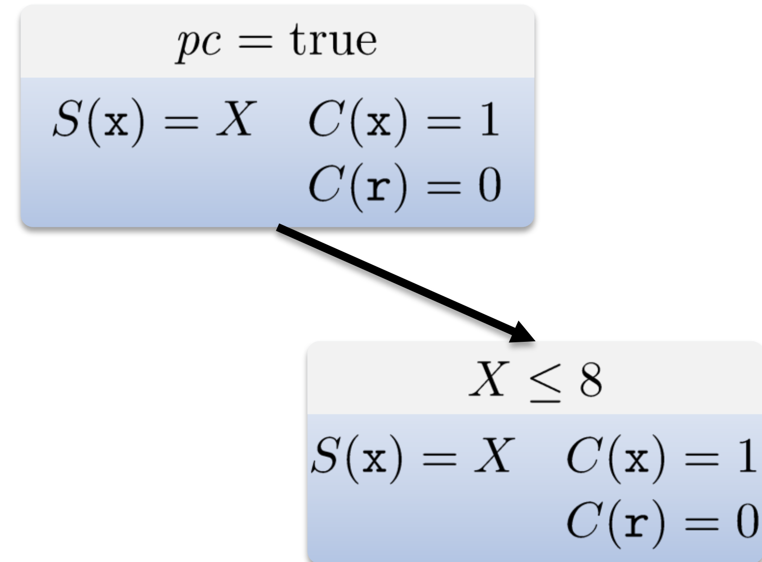
```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```

Path condition:

Test cases:

```
proc(1)
```

DART




```
1  int proc(int x) {
2
3   int r = 0
5
6   if (x > 8) {
7     r = x - 7
8   }
9
10  if (x < 5) {
11    r = x – 2
12  }
13
14  return r;
15 }
```

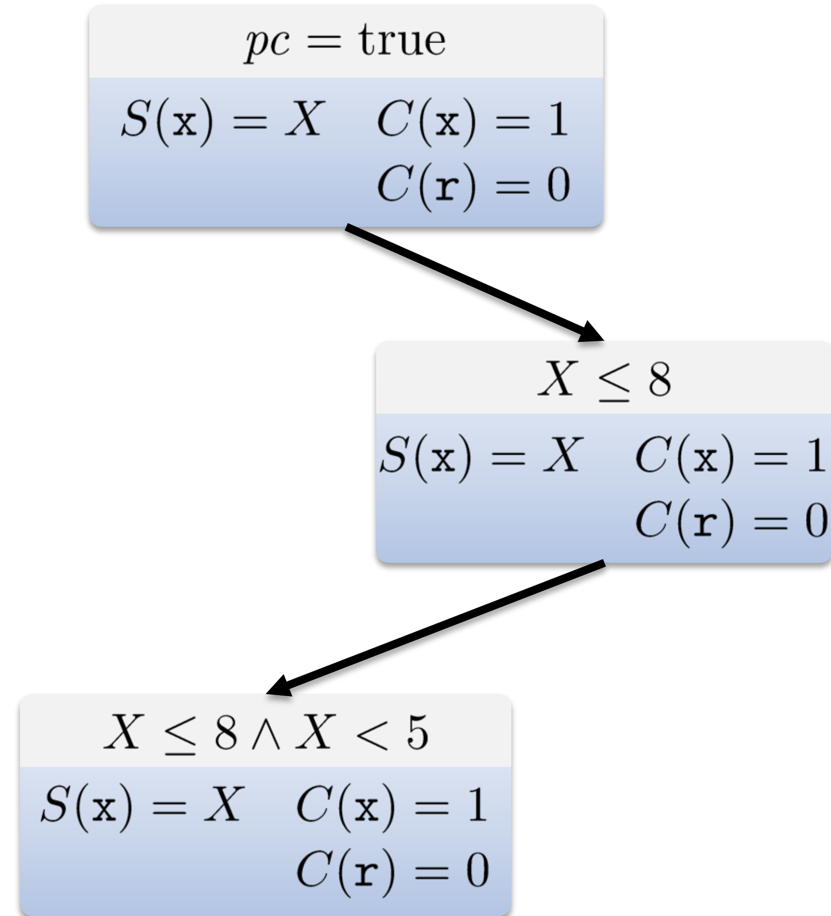
Path condition:

Test cases:

```
proc(1)
```

DART

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



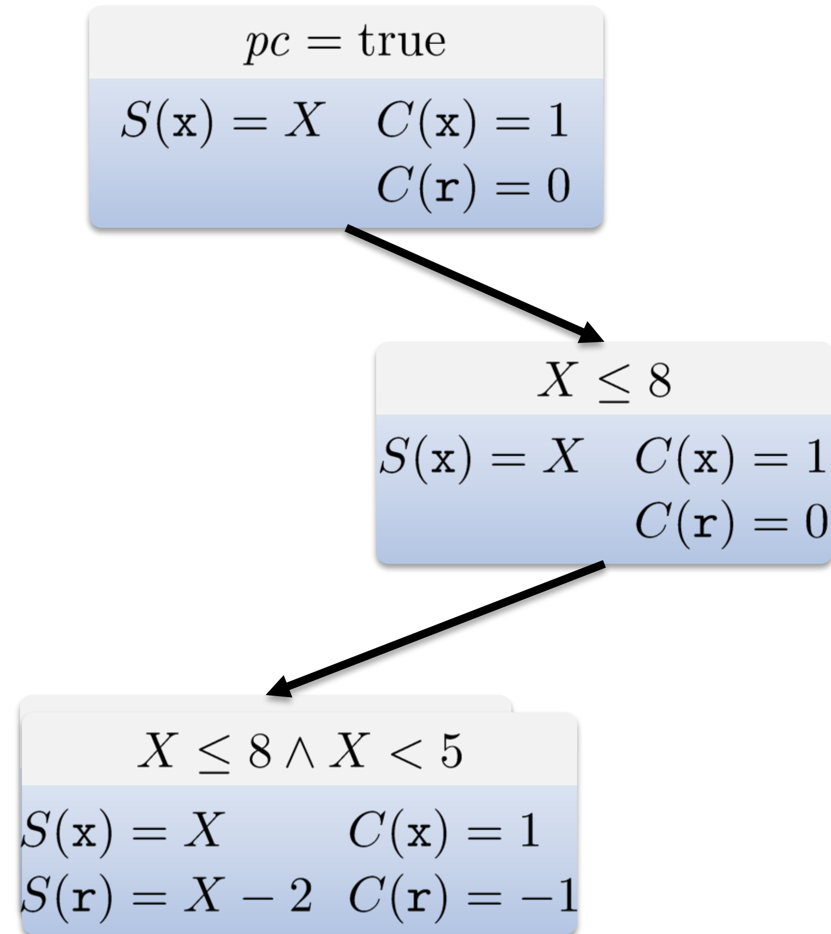
Path condition:

Test cases:

`proc(1)`

DART

```
1  int proc(int x) {
2
3   int r = 0
5
6   if (x > 8) {
7     r = x - 7
8   }
9
10  if (x < 5) {
11    r = x - 2
12  }
13
14  return r;
15 }
```



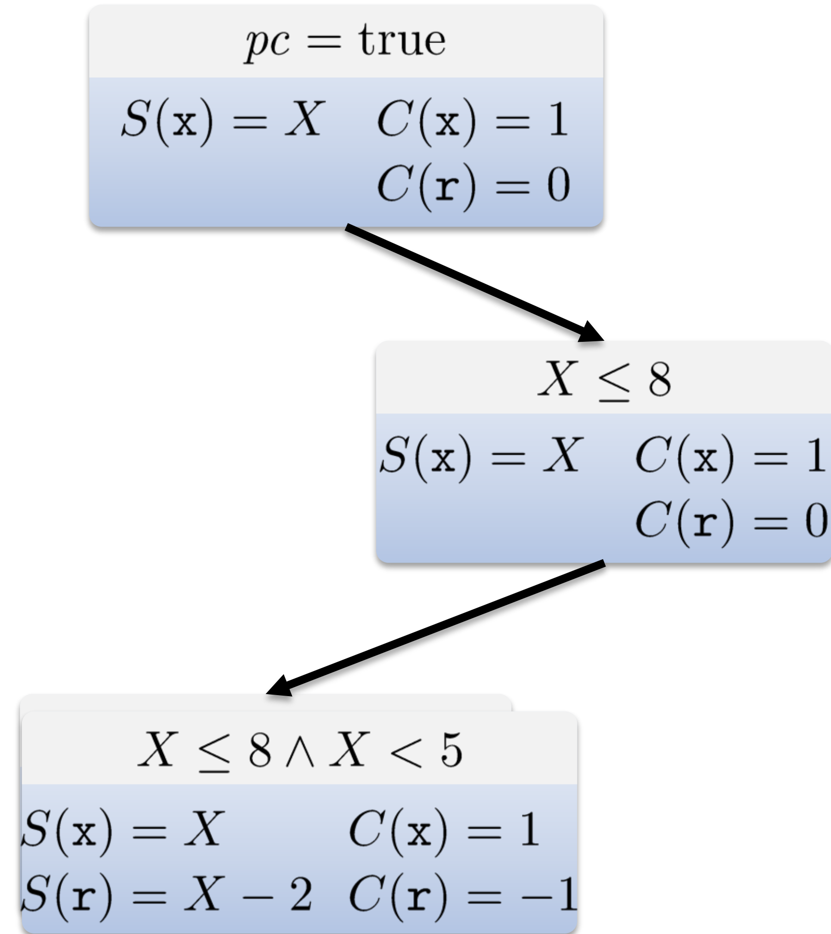
Path condition:

Test cases:

`proc(1)`

DART

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



Path condition:

$$X \leq 8 \wedge \neg(X < 5)$$

Test cases:

`proc(1)`

DART

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```

$pc = \text{true}$

$S(\mathtt{x}) = X \quad C(\mathtt{x}) = 1$

$C(\mathtt{r}) = 0$

$X \leq 8$

$S(\mathtt{x}) = X \quad C(\mathtt{x}) = 1$

$C(\mathtt{r}) = 0$

$X \leq 8 \wedge X < 5$

$S(\mathtt{x}) = X \quad C(\mathtt{x}) = 1$

$S(\mathtt{r}) = X - 2 \quad C(\mathtt{r}) = -1$

Path condition:

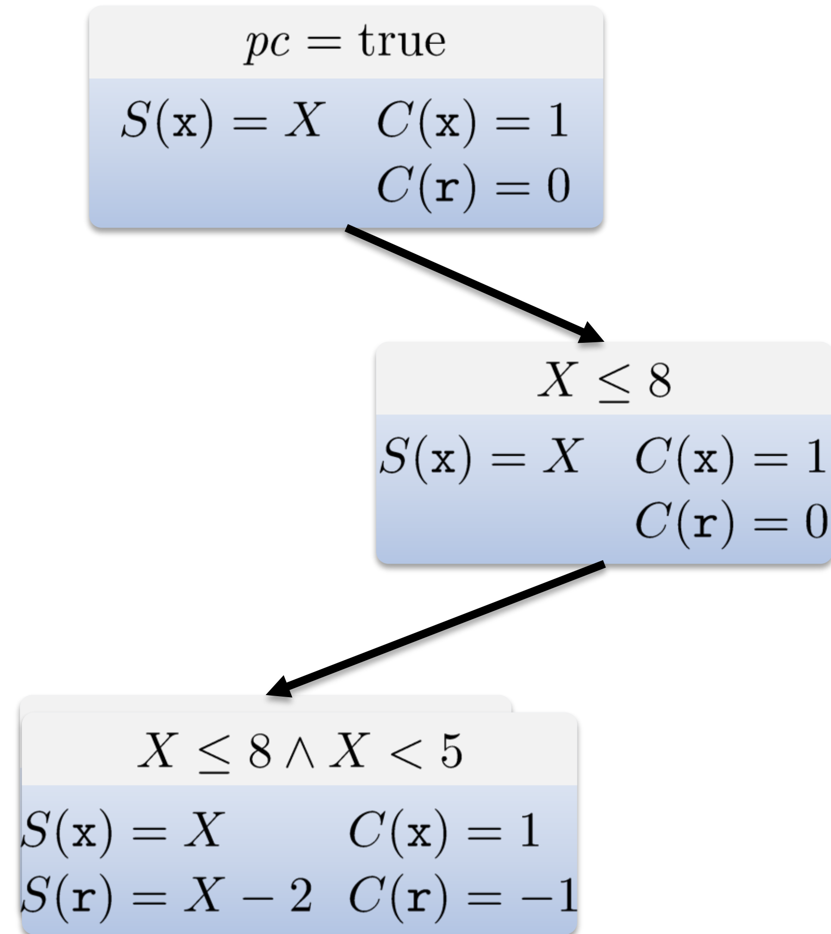
$$X \leq 8 \wedge \neg(X < 5)$$ ✓

Test cases:

`proc(1)`

DART

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



Path condition:

$$X \leq 8 \wedge \neg(X < 5)$$

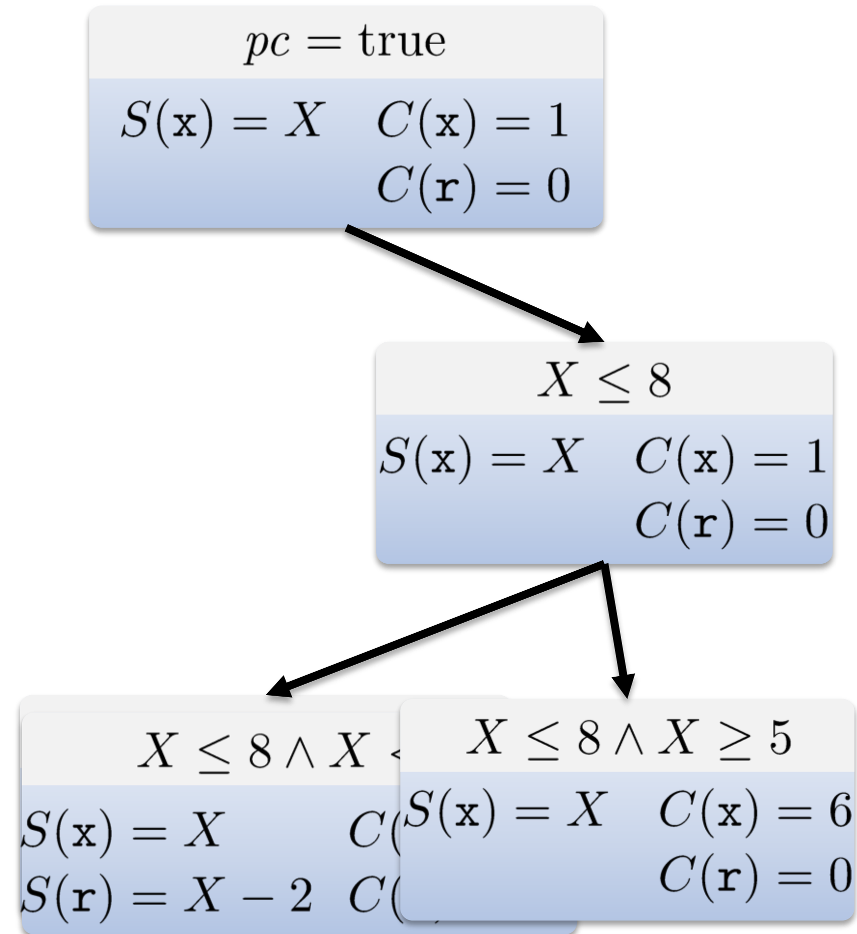
✓

Test cases:

proc(1) proc(6)

DART

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



Path condition:

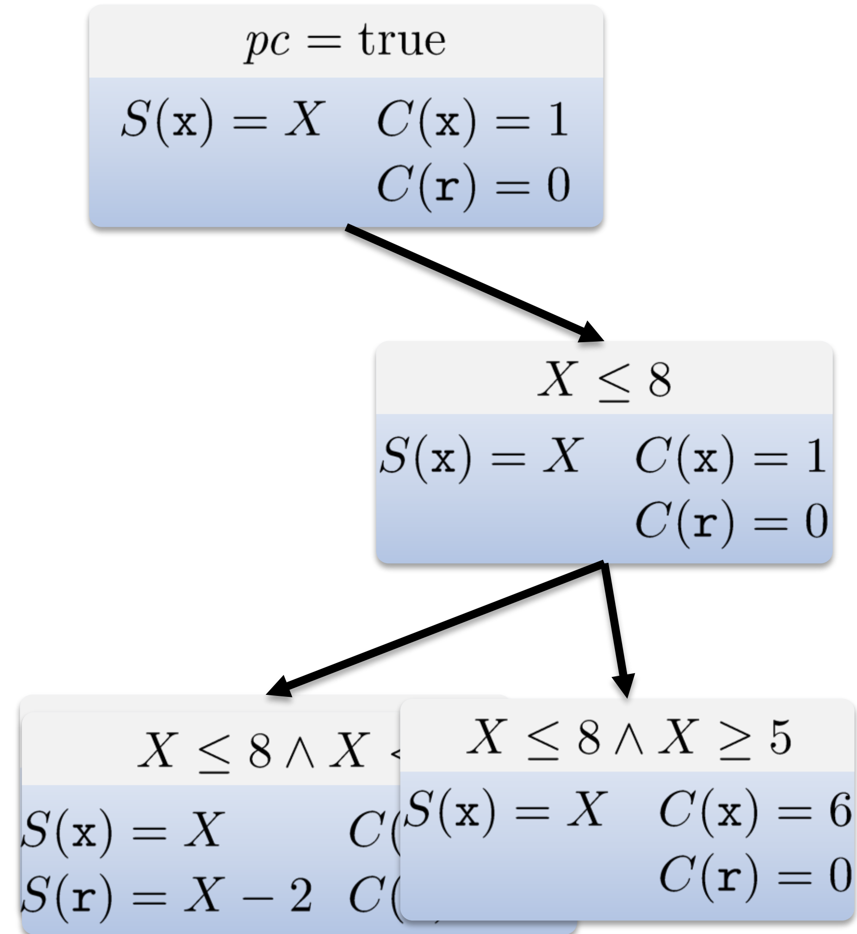
$$X \leq 8 \wedge \neg(X < 5)$$

Test cases:

`proc(1)` `proc(6)`

DART

```
1   int proc(int x) {
2
3     int r = 0
5
6     if (x > 8) {
7       r = x - 7
8     }
9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



Path condition:

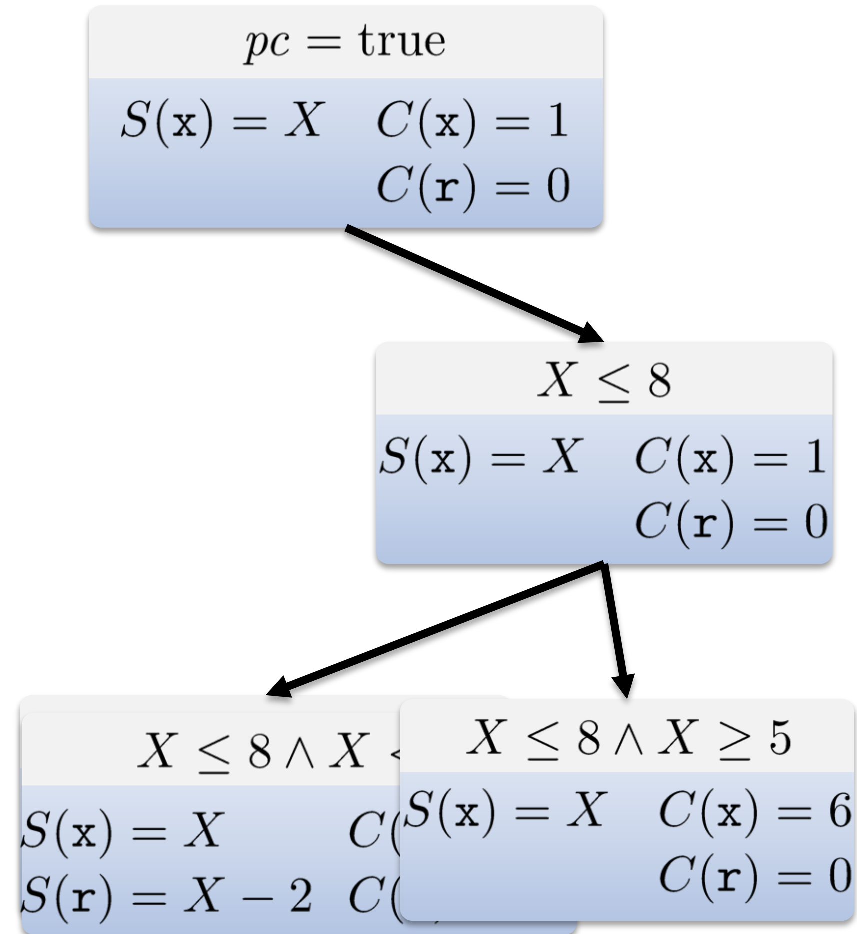
$$\neg(X \leq 8)$$

Test cases:

`proc(1)` `proc(6)`

DART

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



Path condition:

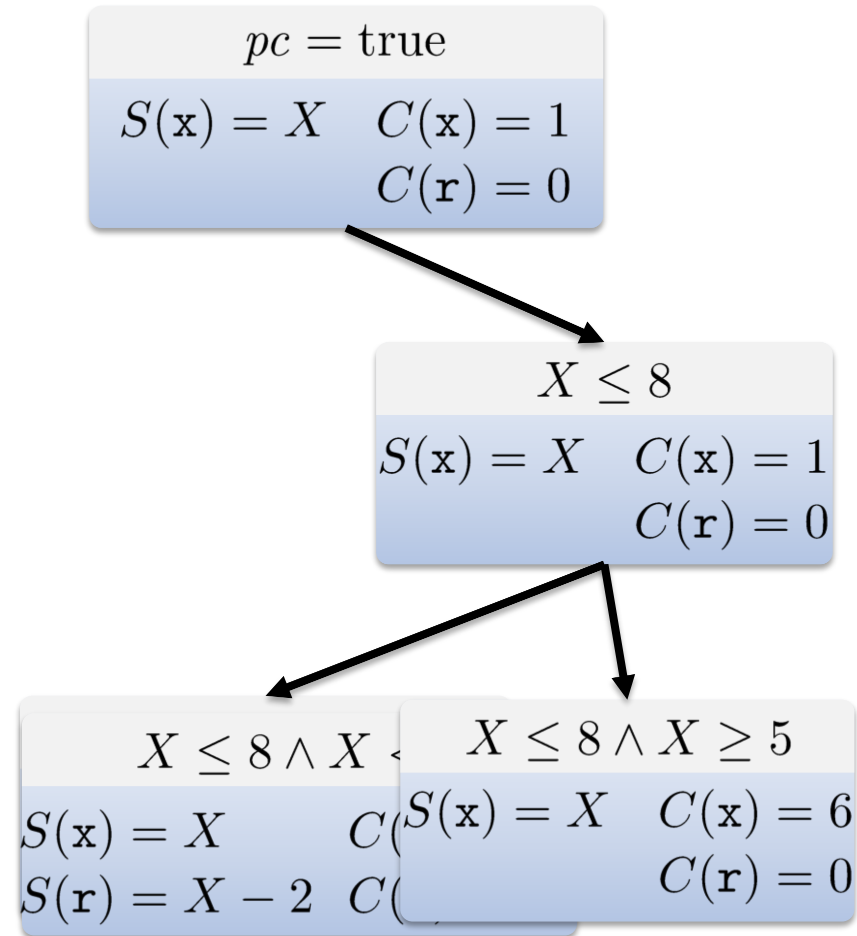
$\neg(X \leq 8)$ ✓

Test cases:

`proc(1)` `proc(6)`

DART

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



Path condition:

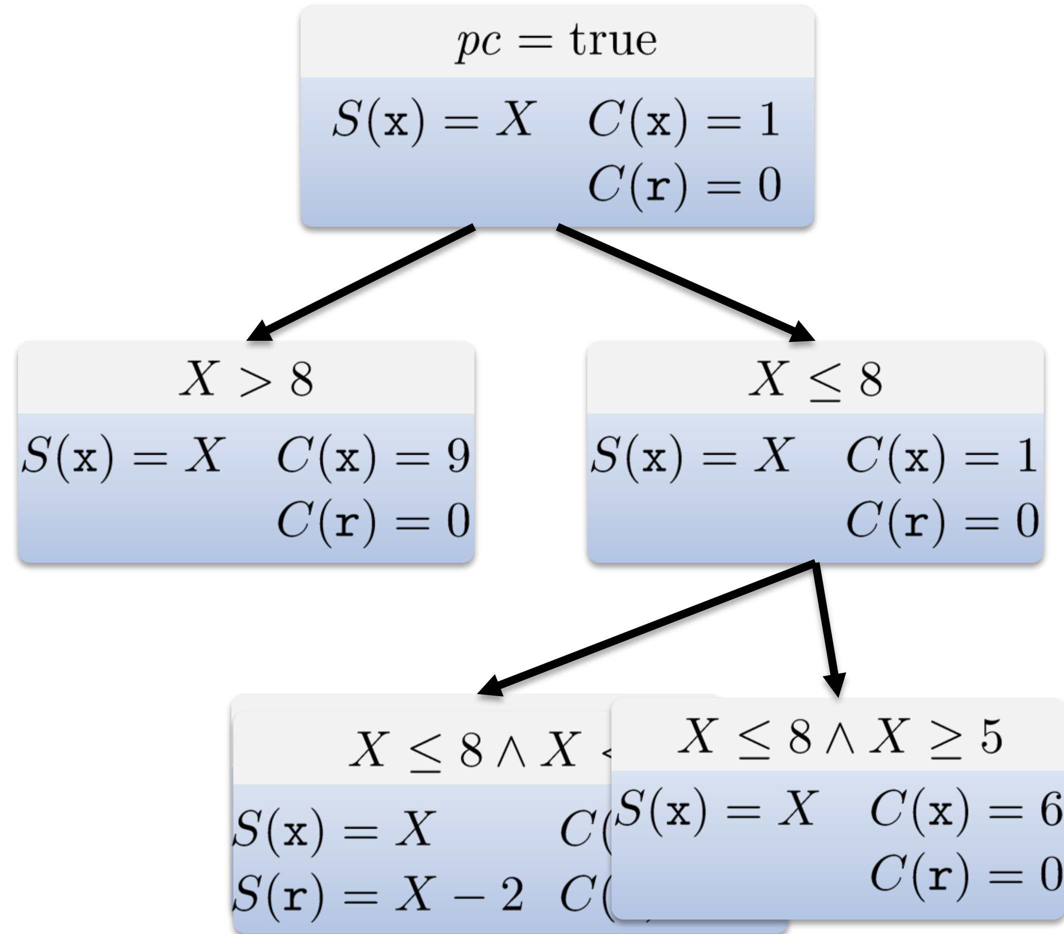
$\neg(X \leq 8)$ ✓

Test cases:

`proc(9)` `proc(1)` `proc(6)`

DART

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



Path condition:

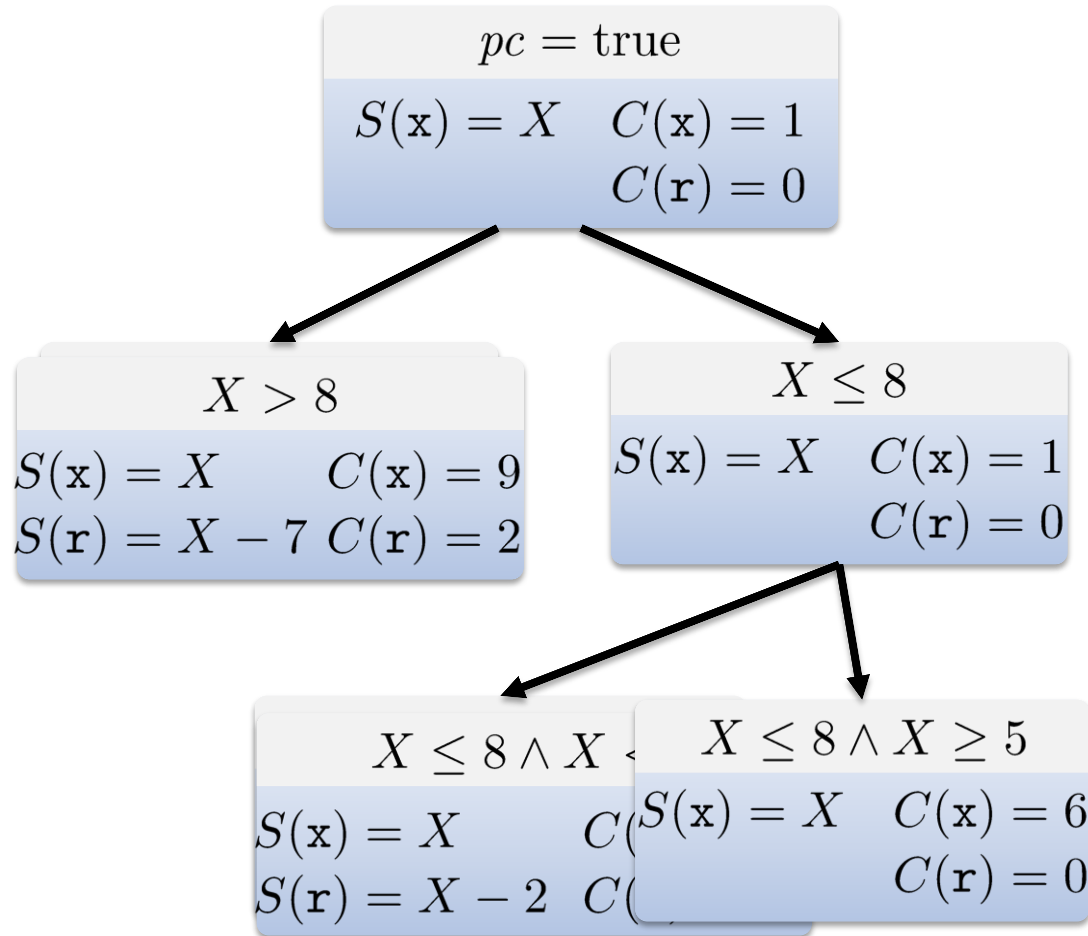
$$\neg(X \leq 8)$$

Test cases:

`proc(9)` `proc(1)` `proc(6)`

DART

```
1   int proc(int x) {
2
3     int r = 0
5
6     if (x > 8) {
7       r = x - 7
8     }
9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15  }
```



Path condition:

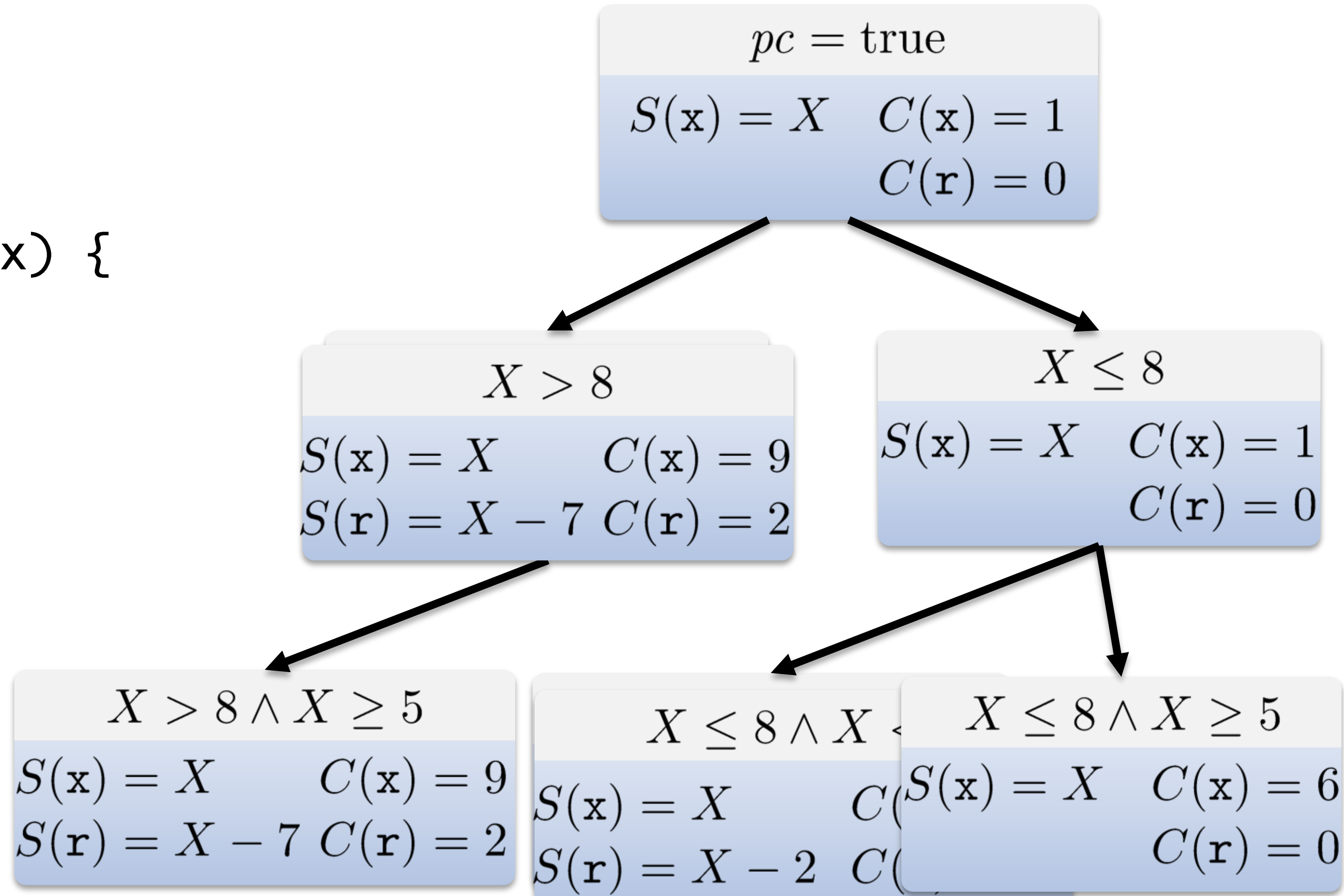
$$\neg(X \leq 8)$$

Test cases:

`proc(9)` `proc(1)` `proc(6)`

DART

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



Path condition:

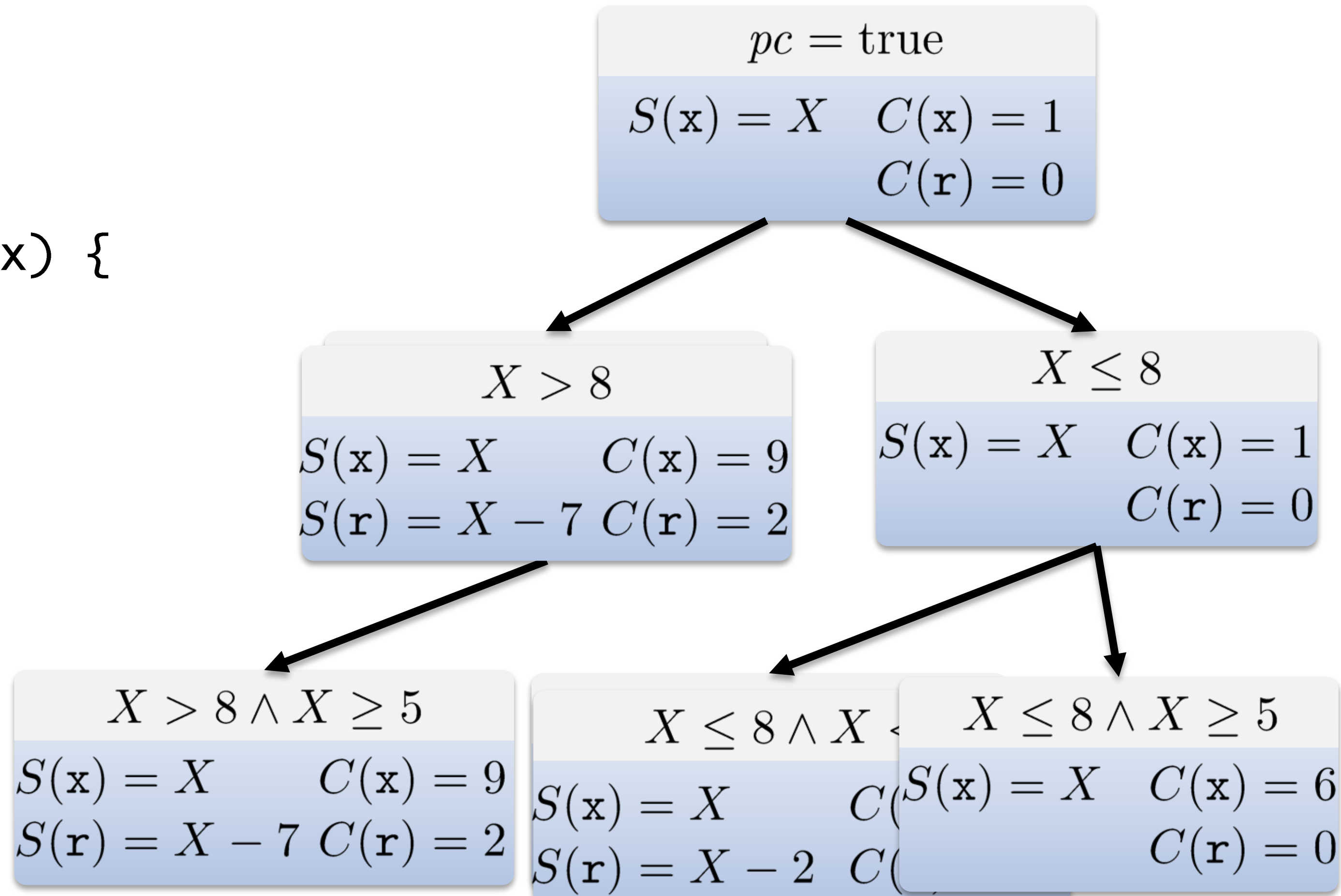
$$\neg(X \leq 8)$$

Test cases:

`proc(9)` `proc(1)` `proc(6)`

DART

```
1  int proc(int x) {
2
3   int r = 0
5
6   if (x > 8) {
7     r = x - 7
8   }
9
10  if (x < 5) {
11    r = x - 2
12  }
13
14  return r;
15 }
```



Path condition:

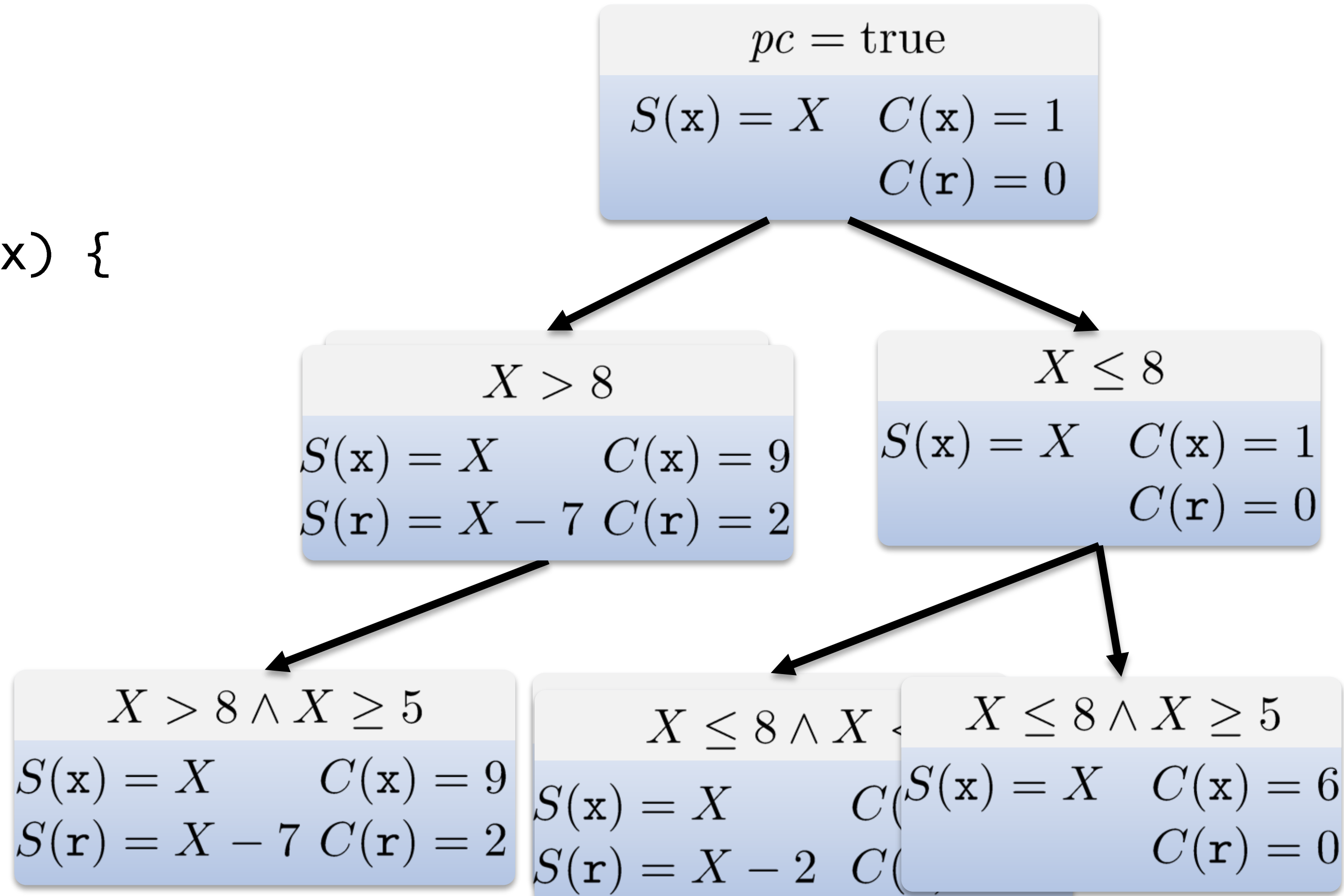
$$X > 8 \wedge \neg(X \geq 5)$$

Test cases:

`proc(9)` `proc(1)` `proc(6)`

DART

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



Path condition:

$$X > 8 \wedge \neg(X \geq 5)$$

Test cases:

`proc(9)` `proc(1)` `proc(6)`

DART vs. EXE

- Complete execution from 1st step
- Deep exploration
  - *One query per run*
- Offline SE possible
  - *Follow recorded trace*

- Fine-grained control of execution
- Shallow exploration
  - *Many queries early on*
- Online SE
  - *SE and interpretation in lockstep*

Concretization

- Parts of input space can be kept concrete
  - *Reduces complexity*
  - *Focuses search*
- Expressions can be concretized at runtime
  - *Avoid expressions outside of SMT solver theories (non-linear etc.)*
- Sound but incomplete

Concretization (Example)

$true$	
	$C(X) = 5$
$S(\texttt{m}) = X + 2$	$C(\texttt{m}) = 7$
$S(\texttt{size}) = Y$	$C(\texttt{size}) = 256$

```
if (m*m > size) {
   …
```

Concretization (Example)

$true$	
	$C(X) = 5$
$S(\texttt{m}) = X + 2$	$C(\texttt{m}) = 7$
$S(\texttt{size}) = Y$	$C(\texttt{size}) = 256$

```
if (m*m > size) {
    …
```

$$(X + 2)(X + 2) > Y$$

Concretization (Example)

$$true$$

$$C(X) = 5$$

$$S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) = 7$$

$$S(\texttt{size}) = Y \qquad C(\texttt{size}) = 256$$

```
if (m*m > size) {
    …
```

$$(X+2)(X+2) > Y$$

$$(5+2)(5+2) > Y$$

Concretization (Example)

$true$	
	$C(X) = 5$
$S(\texttt{m}) = X + 2$	$C(\texttt{m}) = 7$
$S(\texttt{size}) = Y$	$C(\texttt{size}) = 256$

```
if (m*m > size) {
    …
```

$$(X + 2)(X + 2) > Y$$

$$49 > Y$$

Concretization (Example)

$$true$$

$$C(X) = 5$$

$$S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) = 7$$

$$S(\texttt{size}) = Y \qquad C(\texttt{size}) = 256$$

```
if (m*m > size) {
    ...
```

$$(X+2)(X+2) > Y$$

$$49 > Y$$

$$49 > Y$$

$$C(X) = 5$$

$$S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) = 7$$

$$S(\texttt{size}) = Y \qquad C(\texttt{size}) = 48$$

Concretization (Example)

true	
	$C(X) = 5$
$S(\texttt{m}) = X + 2$	$C(\texttt{m}) = 7$
$S(\texttt{size}) = Y$	$C(\texttt{size}) = 256$

$49 > Y$	
	$C(X) = 5$
$S(\texttt{m}) = X + 2$	$C(\texttt{m}) = 7$
$S(\texttt{size}) = Y$	$C(\texttt{size}) = 48$

```
if (m*m > size) {
  …
  if (m < 5) {
    …
```

Concretization (Example)

$true$

$C(X) = 5$
$S(\mathtt{m}) = X + 2$ $C(\mathtt{m}) = 7$
$S(\mathtt{size}) = Y$ $C(\mathtt{size}) = 256$

```
if (m*m > size) {
  …
  if (m < 5) {
    …
```

$$X + 2 < 5$$

$49 > Y$

$C(X) = 5$
$S(\mathtt{m}) = X + 2$ $C(\mathtt{m}) = 7$
$S(\mathtt{size}) = Y$ $C(\mathtt{size}) = 48$

Concretization (Example)

$$true$$

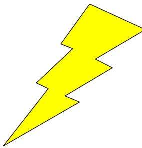
$$C(X) = 5$$
$$S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) = 7$$
$$S(\texttt{size}) = Y \qquad C(\texttt{size}) = 256$$

$$49 > Y$$

$$C(X) = 5$$
$$S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) = 7$$
$$S(\texttt{size}) = Y \qquad C(\texttt{size}) = 48$$

```
if (m*m > size) {
…
  if (m < 5) {
  …
```

$$X + 2 < 5$$



Solution diverges from expected path! (e.g., X = 2)

Concretization (Example)

$$true$$

$$S(\mathtt{m}) = X + 2 \qquad C(X) = 5$$
$$S(\mathtt{size}) = Y \qquad C(\mathtt{m}) = 7$$
$$C(\mathtt{size}) = 256$$

```
if (m*m > size) {
  …
  if (m < 5) {
    …
```

Concretization constraint

$$49 > Y \wedge X = 5$$

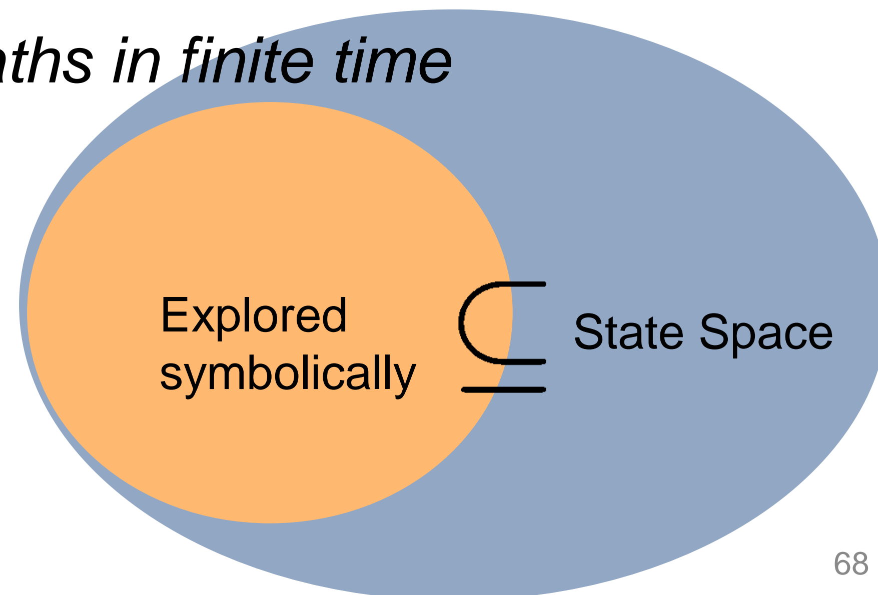
$$S(\mathtt{m}) = X + 2 \qquad C(X) = 5$$
$$S(\mathtt{size}) = Y \qquad C(\mathtt{m}) = 7$$
$$C(\mathtt{size}) = 48$$

Soundness & Completeness

- Conceptually, each path is exact
  - *Strongest postcondition in predicate transformer semantics*
  - *No over-approximation, no under-approximation*
- Globally, SE under-approximates
  - *Explores only subset of paths in finite time*
  - *“Eventual” completeness*

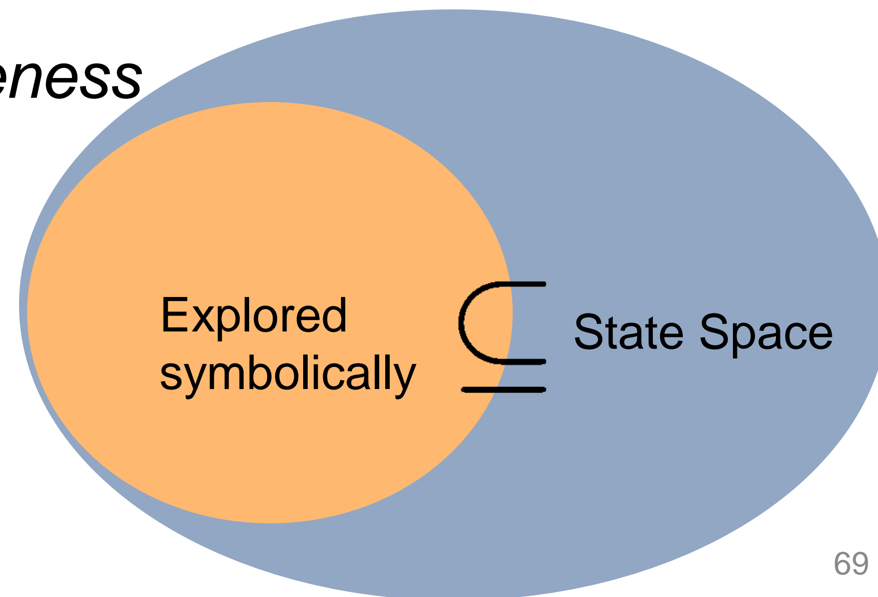
Soundness & Completeness

- Conceptually, each path is exact
  - *Strongest postcondition in predicate transformer semantics*
  - *No over-approximation, no under-approximation*
- Globally, SE under-approximates
  - *Explores only subset of paths in finite time*
  - *“Eventual” completeness*



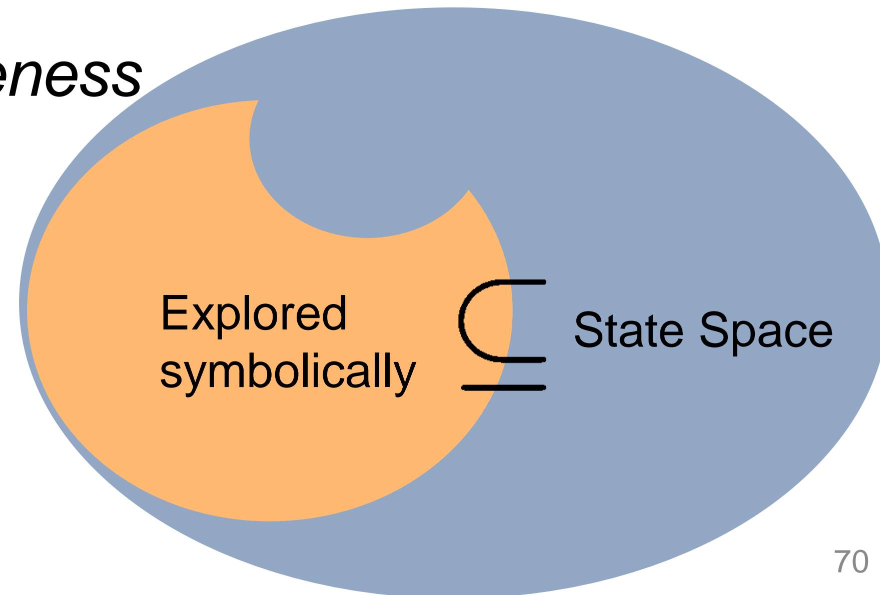
Soundness & Completeness

- Symbolic Execution = Underapproximates
  - *Soundness = does not include infeasible behavior*
  - *Completeness = explores all behavior*
- Concretization restricts state covered by path
  - *Remains sound*
  - *Loses (eventual) completeness*



Soundness & Completeness

- Symbolic Execution = Underapproximates
  - *Soundness = does not include infeasible behavior*
  - *Completeness = explores all behavior*
- Concretization restricts state covered by path
  - *Remains sound*
  - *Loses (eventual) completeness*



Concretization

- Key strength of dynamic symbolic execution
- Enables external calls
  - *Concretize call arguments*
  - *Callee executes concretely*
- Concretization constraints can be omitted
  - *Sacrifices soundness (original DART)*
  - *Deal with divergences by random restarts*

Outline

- Symbolic Execution for Testing
- State Merging – Fighting Path Explosion
- Interpreted High-Level Code

“echo” Coreutil

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

“echo” Coreutil

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

“echo” Coreutil

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

“echo” Coreutil

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

Symbolic Execution

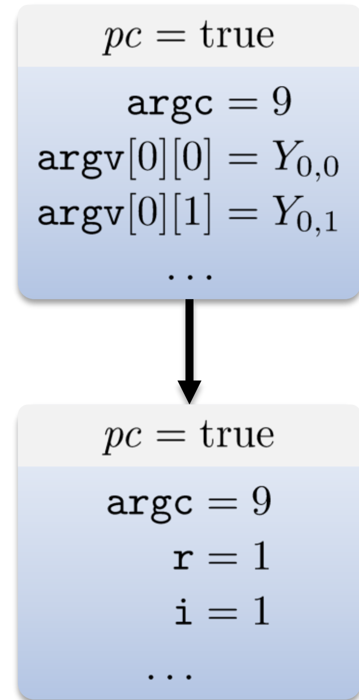
$pc = \text{true}$

$\texttt{argc} = 9$
$\texttt{argv}[0][0] = Y_{0,0}$
$\texttt{argv}[0][1] = Y_{0,1}$
$\ldots$

```c
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

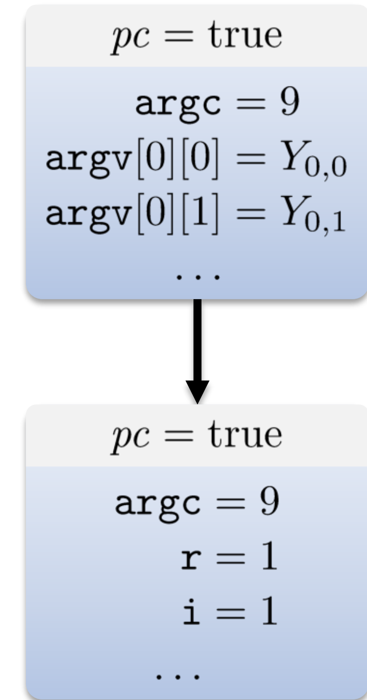
Symbolic Execution

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```



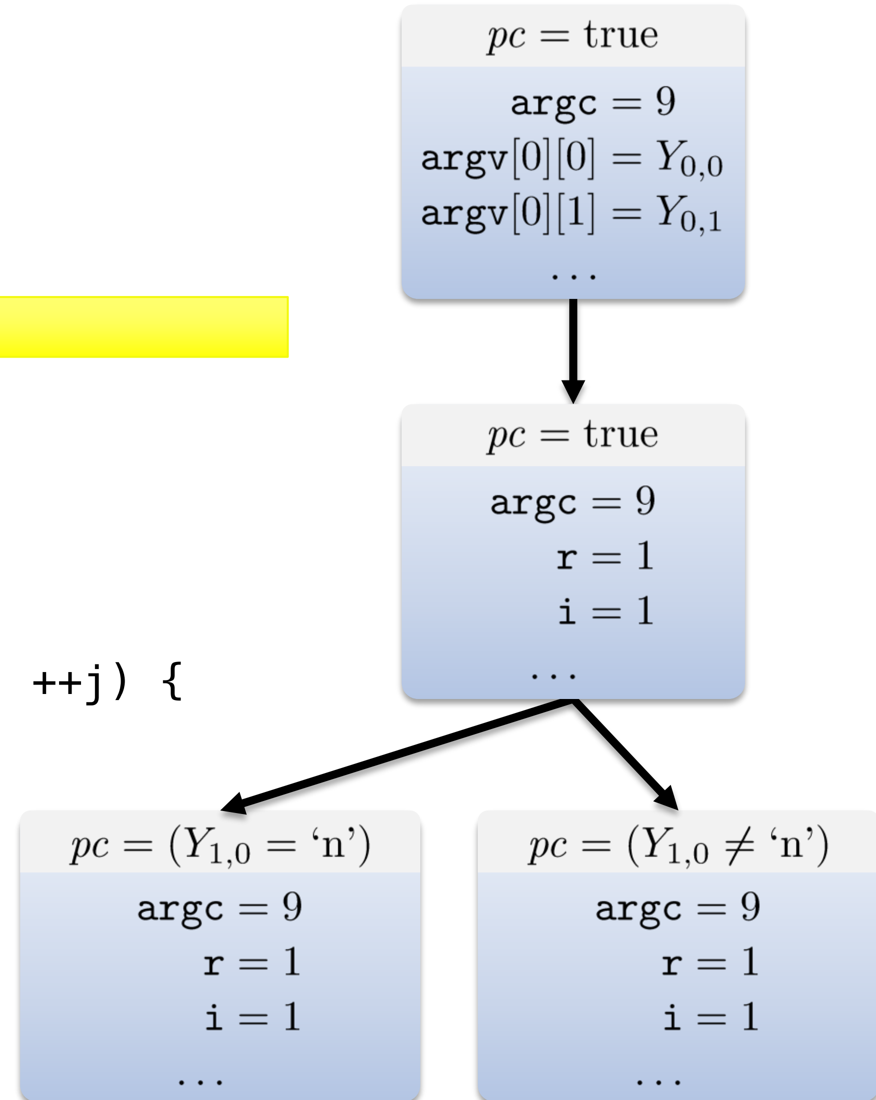
Symbolic Execution

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```



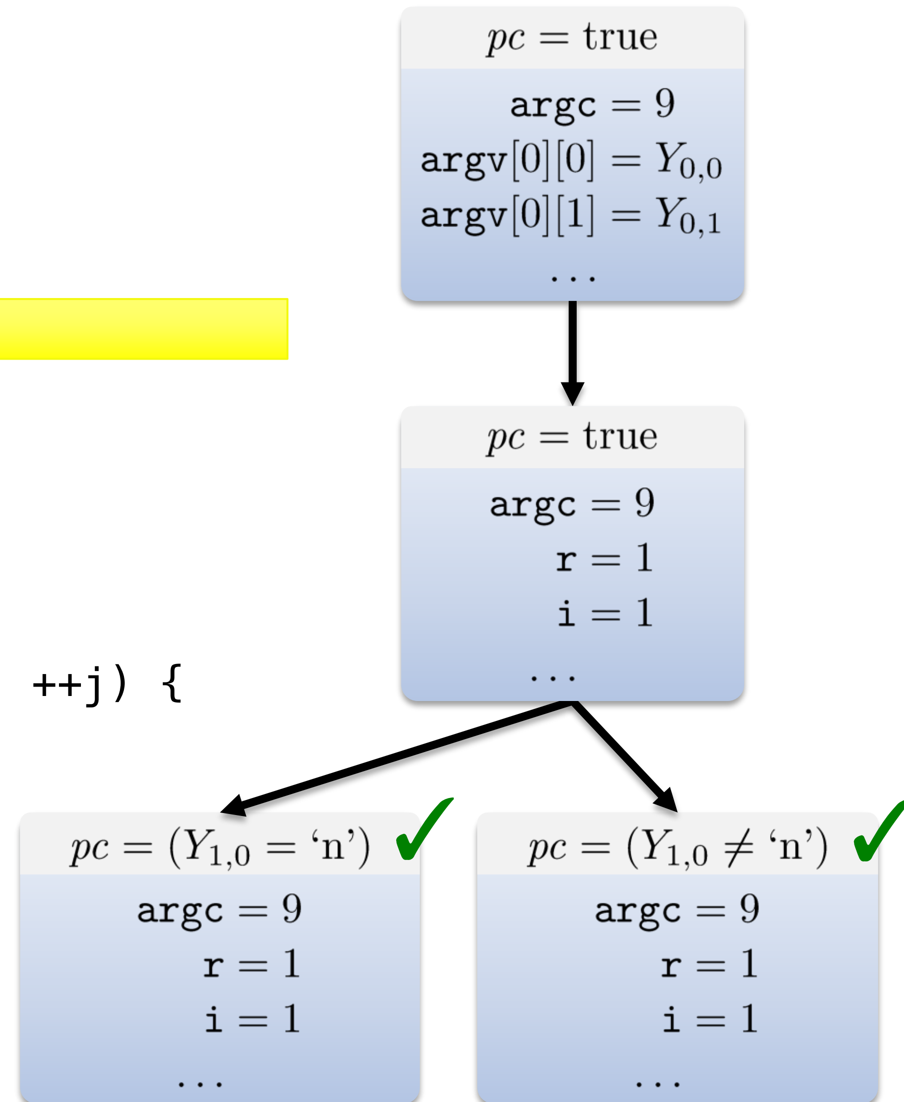
Symbolic Execution

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

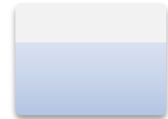


Symbolic Execution

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```



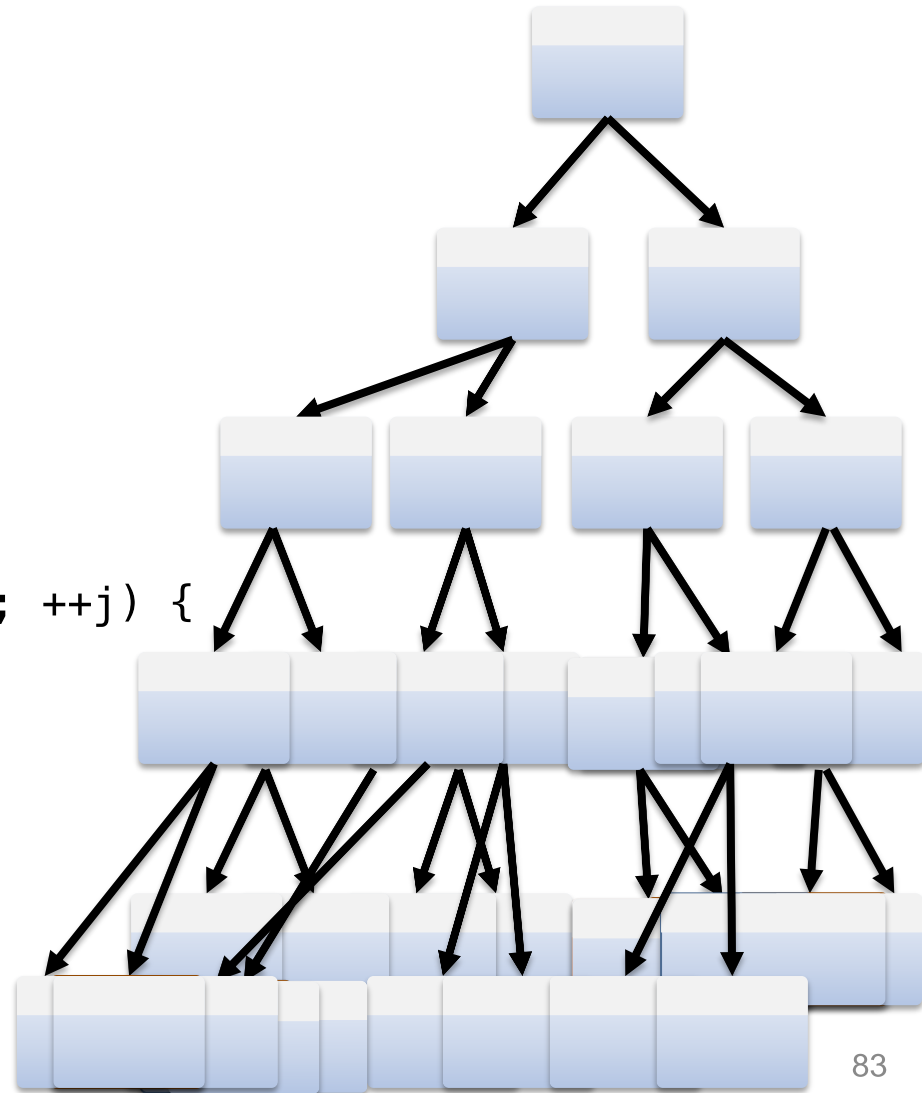
Problem: Path Explosion



```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

Problem: Path Explosion

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```



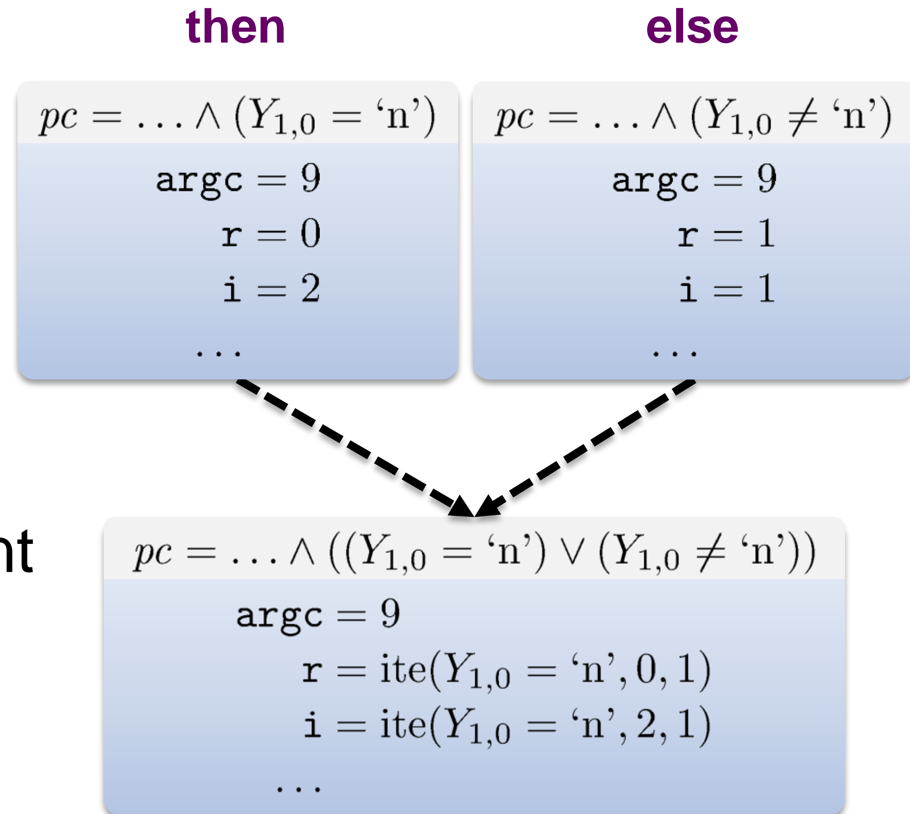
Solution (?): State Merging

```
if (argv[i][0] == 'n') {
  r = 0;
  ++i;
}
```

then	else
$pc = \ldots \wedge (Y_{1,0} = \text{'n'})$	$pc = \ldots \wedge (Y_{1,0} \neq \text{'n'})$
$\texttt{argc} = 9$	$\texttt{argc} = 9$
$\texttt{r} = 0$	$\texttt{r} = 1$
$\texttt{i} = 2$	$\texttt{i} = 1$
$\ldots$	$\ldots$

Solution (?): State Merging

```
if (argv[i][0] == 'n') {
  r = 0;
  ++i;
}
```

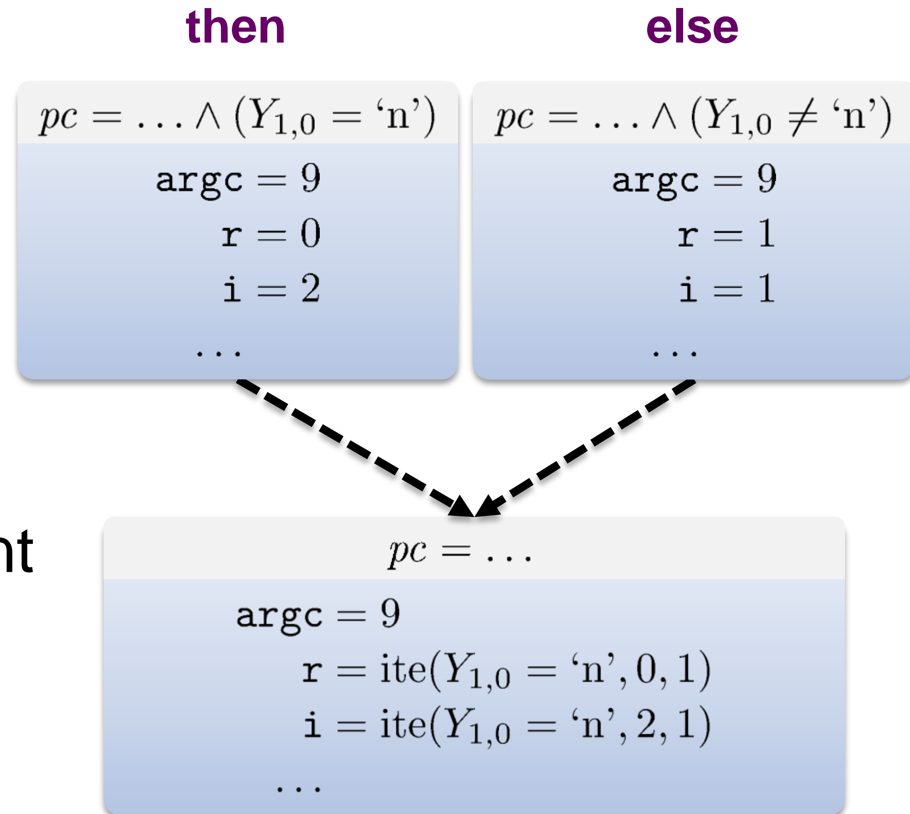


- Use disjunctions to represent state at join points
  - ite(x, y, z) : if x then y else z

Solution (?): State Merging

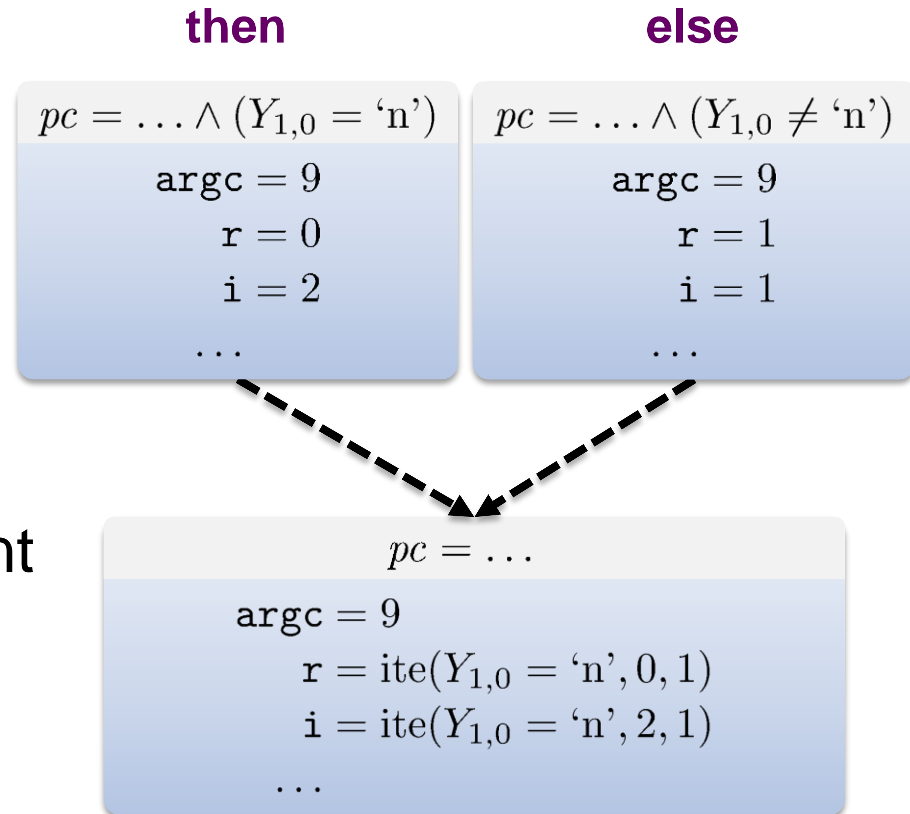
```
if (argv[i][0] == 'n') {
  r = 0;
  ++i;
}
```

- Use disjunctions to represent state at join points
  - ite(x, y, z) : if x then y else z



Solution (?): State Merging

```
if (argv[i][0] == 'n') {
  r = 0;
  ++i;
}
```



- Use disjunctions to represent state at join points
  - $\text{ite}(x, y, z)$: if x then y else z
- SE tree becomes a DAG
  - *Whole program can be turned into one verification condition (BMC)*

Symbolic Execution vs. BMC

- Complexity does not disappear
  - *Work moved from the SE engine to the solver*
  - *SE: set of conjunctive queries, BMC: 1 query with nested disjunctions*
- Complete merging sacrifices advantages of SE
  - *No dynamic mode*
  - *No continuous progress*
  - *No quick reaching of coverage goals*
- Try to get the best of both worlds

Symbolic Execution

Verification Condition Generation

EXE (KLEE)
[Cadar et al., CCS'06]

DART (SAGE)
[Godefroid, PLDI'05]

F-Soft
[Ivancic et al., CAV'05]

CBMC
[Clarke et al., TACAS'04]

1 formula / path

1 formula / CFG

Symbolic Execution

Verification Condition Generation

Boogie
[Barnett et al., FMCO'05]

EXE (KLEE)
[Cadar et al., CCS'06]

F-Soft
[Ivancic et al., CAV'05]

DART (SAGE)
[Godefroid, PLDI'05]

CBMC
[Clarke et al., TACAS'04]

1 formula / path

1 formula / CFG

Symbolic Execution

Verification Condition Generation

Compositional SE / Summaries
[Godefroid, POPL'07]

EXE (KLEE)
[Cadar et al., CCS'06]

DART (SAGE)
[Godefroid, PLDI'05]

Boogie
[Barnett et al., FMCO'05]

F-Soft
[Ivancic et al., CAV'05]

CBMC
[Clarke et al., TACAS'04]

1 formula / path

1 formula / CFG

Symbolic Execution

Verification Condition Generation

BMC slicing
[Ganai&Gupta, DAC'08]

Compositional SE / Summaries
[Godefroid, POPL'07]

Boogie
[Barnett et al., FMCO'05]

EXE (KLEE)
[Cadar et al., CCS'06]

F-Soft
[Ivancic et al., CAV'05]

DART (SAGE)
[Godefroid, PLDI'05]

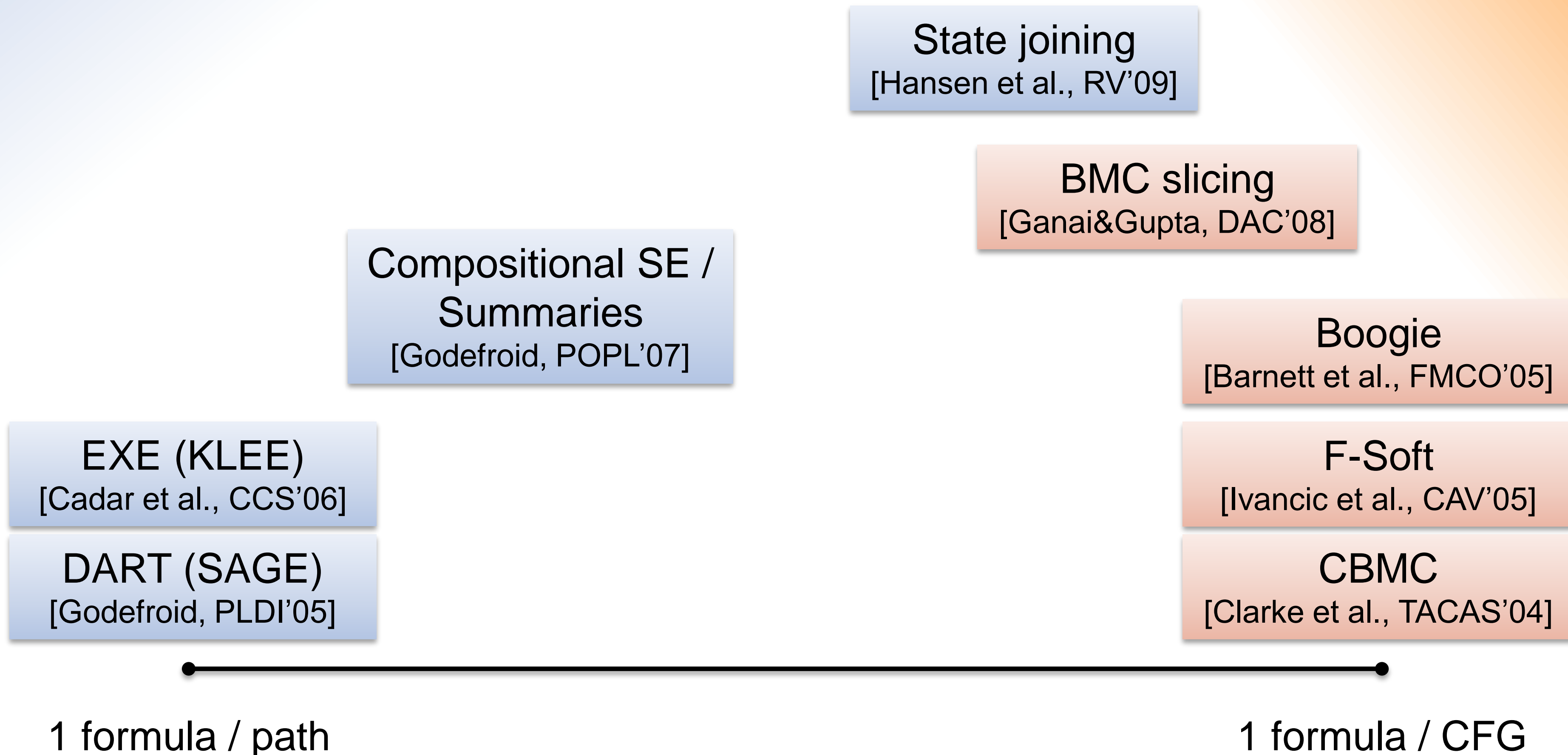
CBMC
[Clarke et al., TACAS'04]

1 formula / path

1 formula / CFG

Symbolic Execution

Verification Condition Generation



Symbolic Execution

Verification Condition Generation

Dynamic State Merging

State joining
[Hansen et al., RV'09]

BMC slicing
[Ganai&Gupta, DAC'08]

Compositional SE / Summaries
[Godefroid, POPL'07]

Boogie
[Barnett et al., FMCO'05]

EXE (KLEE)
[Cadar et al., CCS'06]

F-Soft
[Ivancic et al., CAV'05]

DART (SAGE)
[Godefroid, PLDI'05]

CBMC
[Clarke et al., TACAS'04]

1 formula / path

1 formula / CFG

Symbolic Execution

Verification Condition Generation

Query Count Estimation

[KKBC PLDI'12]

State joining
[Hansen et al., RV'09]

BMC slicing
[Ganai&Gupta, DAC'08]

Compositional SE / Summaries
[Godefroid, POPL'07]

Boogie
[Barnett et al., FMCO'05]

EXE (KLEE)
[Cadar et al., CCS'06]

F-Soft
[Ivancic et al., CAV'05]

DART (SAGE)
[Godefroid, PLDI'05]

CBMC
[Clarke et al., TACAS'04]

1 formula / path

1 formula / CFG

Merging Increases Solving Cost

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

$pc = \ldots$

$\texttt{argc} = 9$

$\texttt{i} = \text{ite}(Y_{1,0} = \text{`n'}, 2, 1)$

$\ldots$

Merging Increases Solving Cost

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

$pc = \ldots$

argc $= 9$

i $= \text{ite}(Y_{1,0} = \text{'n'}, 2, 1)$

$\ldots$

Merging Increases Solving Cost

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

$pc = \ldots$

$\texttt{argc} = 9$

$\texttt{i} = \text{ite}(Y_{1,0} = \text{`n'}, 2, 1)$

$\ldots$

$\ldots \wedge \text{ite}(Y_{1,0} = \text{`n'}, 2, 1) \geq 9$

$\texttt{argc} = 9$

$\texttt{i} = \text{ite}(Y_{1,0} = \text{`n'}, 2, 1)$

$\ldots$

$\ldots \wedge \text{ite}(Y_{1,0} = \text{`n'}, 2, 1) < 9$

$\texttt{argc} = 9$

$\texttt{i} = \text{ite}(Y_{1,0} = \text{`n'}, 2, 1)$

$\ldots$

Condition becomes symbolic, extra check required.

Merging Increases Solving Cost

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

$pc = \ldots$

$\texttt{argc} = 9$

$\texttt{i} = \mathrm{ite}(Y_{1,0} = \text{‘n’}, 2, 1)$

$\ldots$

$\ldots \land \mathrm{ite}(Y_{1,0} = \text{‘n’}, 2, 1) \geq 9$ ✗

$\texttt{argc} = 9$

$\texttt{i} = \mathrm{ite}(Y_{1,0} = \text{‘n’}, 2, 1)$

$\ldots$

$\ldots \land \mathrm{ite}(Y_{1,0} = \text{‘n’}, 2, 1) < 9$ ✓

$\texttt{argc} = 9$

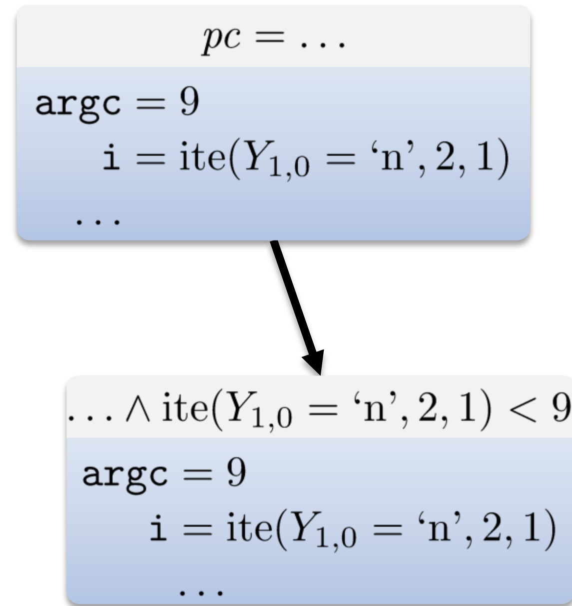
$\texttt{i} = \mathrm{ite}(Y_{1,0} = \text{‘n’}, 2, 1)$

$\ldots$

Condition becomes symbolic, extra check required.

Merging Increases Solving Cost

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```



Merging Increases Solving Cost

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

$pc = \ldots$

argc $= 9$

i $= \mathrm{ite}(Y_{1,0} = \text{‘n’}, 2, 1)$

$\ldots$

$\ldots \wedge \mathrm{ite}(Y_{1,0} = \text{‘n’}, 2, 1) < 9$

argc $= 9$

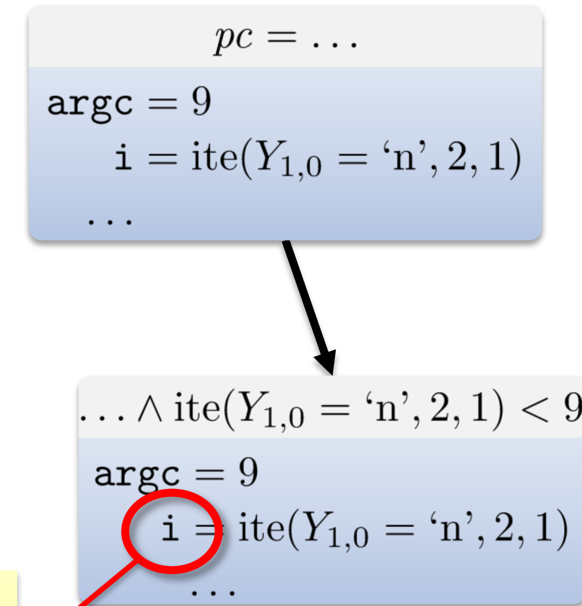
i $= \mathrm{ite}(Y_{1,0} = \text{‘n’}, 2, 1)$

$\ldots$

Query becomes more complex.

Merging Increases Solving Cost

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i <
    for (int
      putchar
    }
  }

  if (r) {
    putchar('\n');
  }
}
```



Should not merge after checking 1st argument!

nes more complex.

Query Count Estimation (QCE)

Intuition

- Estimate the extra burden on the solver
- Merge only when merging amortizes extra cost

Cost ≈ *number of solver queries*

Applying QCE

1. Estimate query counts from each program location
   - *Total number of queries* $Q_t(\ell)$ *on all paths*
   - *For each variable, number of dependent queries* $Q_{add}(\ell, v)$

Applying QCE

Static

1. Estimate query counts from each program location
   - *Total number of queries* $Q_t(\ell)$ *on all paths*
   - *For each variable, number of dependent queries* $Q_{add}(\ell, v)$

Applying QCE

Static

1. Estimate query counts from each program location
   - *Total number of queries* $Q_t(\ell)$ *on all paths*
   - *For each variable, number of dependent queries* $Q_{add}(\ell, v)$
2. Determine "hot" variables

$$H(\ell) = \{v \in V \mid Q_{add}(\ell, v) > \alpha \cdot Q_t(\ell)\}$$

Applying QCE

Static

1. Estimate query counts from each program location
   - *Total number of queries* $Q_t(\ell)$ *on all paths*
   - *For each variable, number of dependent queries* $Q_{add}(\ell, v)$
2. Determine "hot" variables

$$H(\ell) = \{v \in V \mid Q_{add}(\ell, v) > \alpha \cdot Q_t(\ell)\}$$

Dynamic

3. Symbolic Execution
   - *Do not merge two candidate states if they differ in hot variables*
   - *Avoids creating ite expressions*

Merging not Beneficial

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

$pc = \ldots \wedge (Y_{1,0} = \text{`n'})$

$\texttt{argc} = 9$

$\texttt{r} = 0$

$\texttt{i} = 2$

$\ldots$

$pc = \ldots \wedge (Y_{1,0} \neq \text{`n'})$

$\texttt{argc} = 9$

$\texttt{r} = 1$

$\texttt{i} = 1$

$\ldots$

$$H(\ell) = \{\texttt{i}\}$$

i is "hot", leads to many extra queries

Merging not Beneficial

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

$pc = \ldots \wedge (Y_{1,0} = \text{`n'})$

$\texttt{argc} = 9$

$\texttt{r} = 0$

$\texttt{i} = 2$

$\ldots$

$pc = \ldots \wedge (Y_{1,0} \neq \text{`n'})$

$\texttt{argc} = 9$

$\texttt{r} = 1$

$\texttt{i} = 1$

$\ldots$

$$H(\ell) = \{\texttt{i}\}$$

i is "hot", leads to many extra queries

Merging not Beneficial

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

$pc = \ldots \wedge (Y_{1,0} = \text{'n'})$

$\texttt{argc} = 9$

$\texttt{r} = 0$

$\texttt{i} = 2$

$\ldots$

$pc = \ldots \wedge (Y_{1,0} \neq \text{'n'})$

$\texttt{argc} = 9$

$\texttt{r} = 1$

$\texttt{i} = 1$

$\ldots$

$$H(\ell) = \{\texttt{i}\}$$

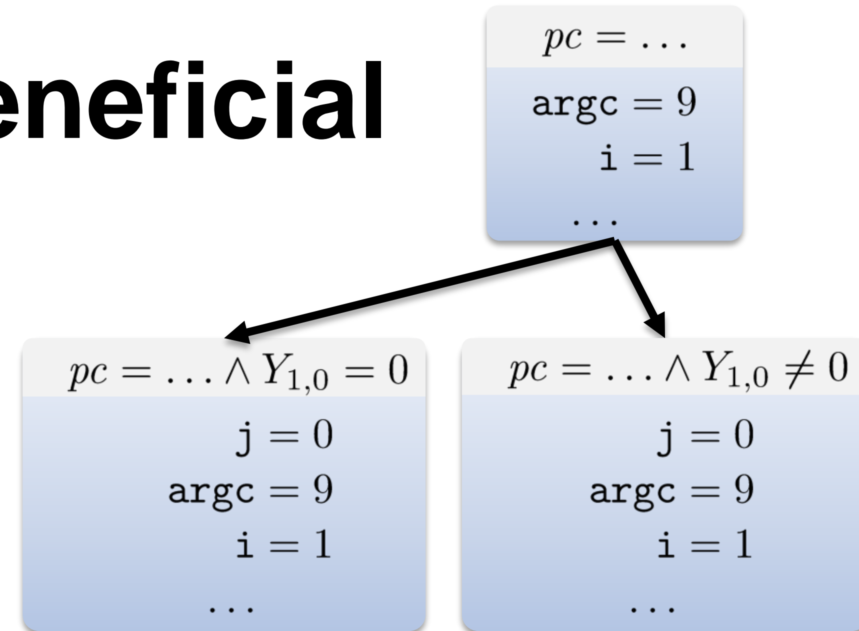
i is “hot”, leads to many extra queries

Merging Beneficial

$pc = \ldots$
$\texttt{argc} = 9$
$\texttt{i} = 1$
$\ldots$

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

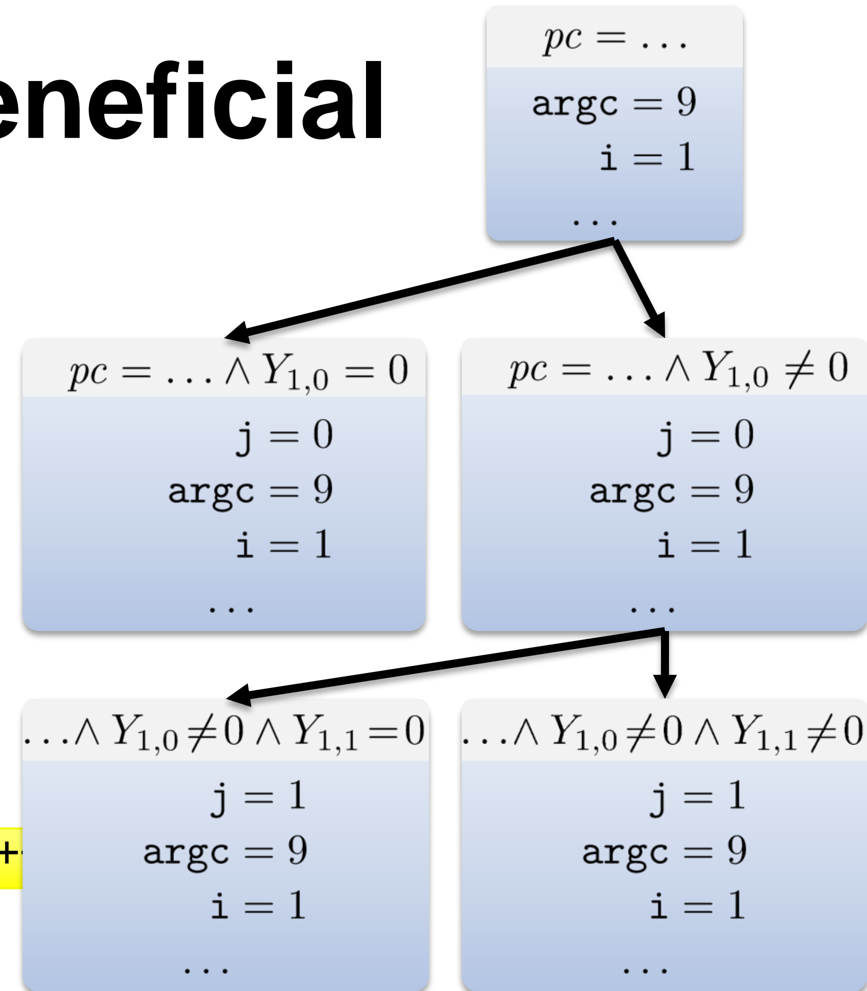
Merging Beneficial




```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

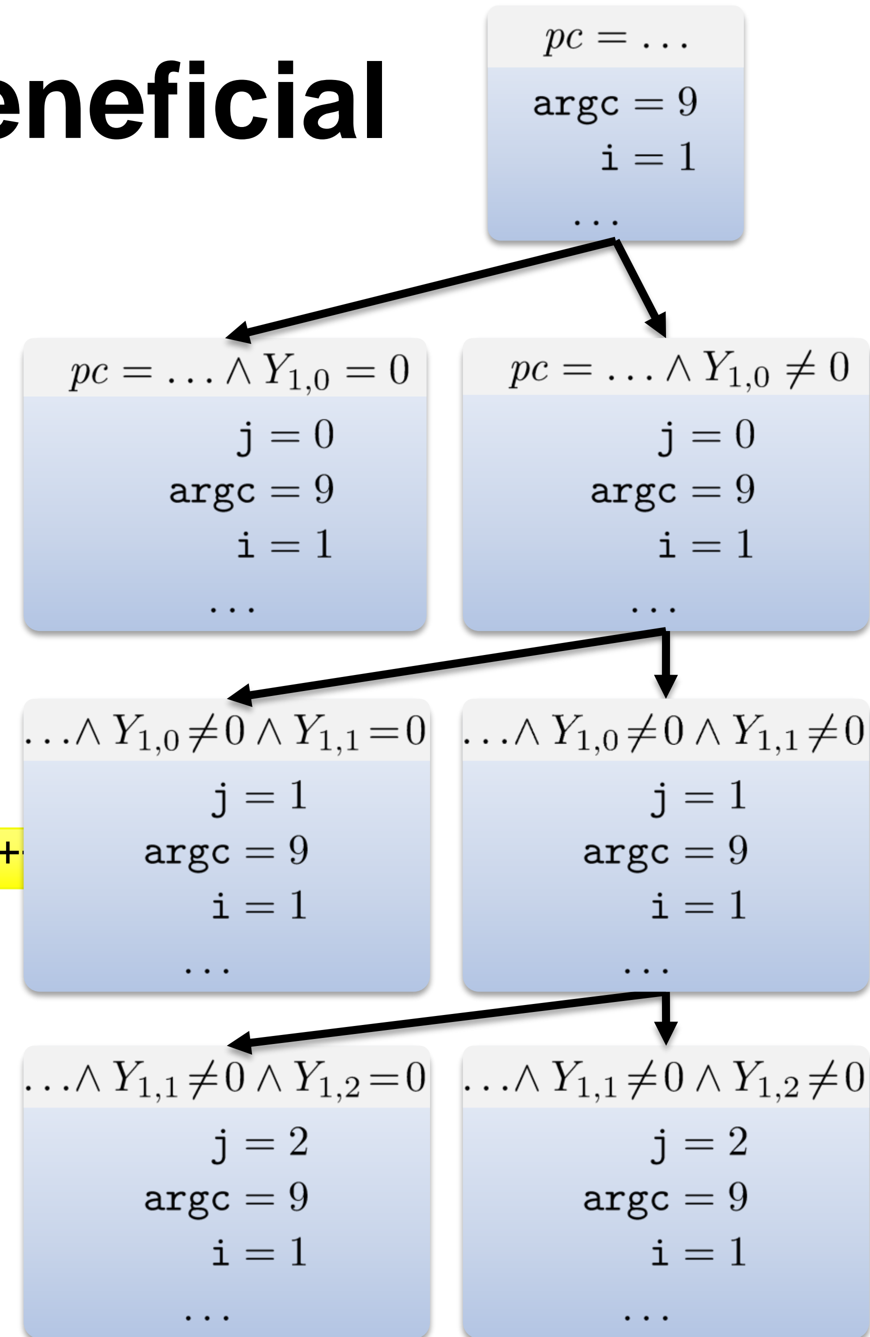
Merging Beneficial

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; +
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```



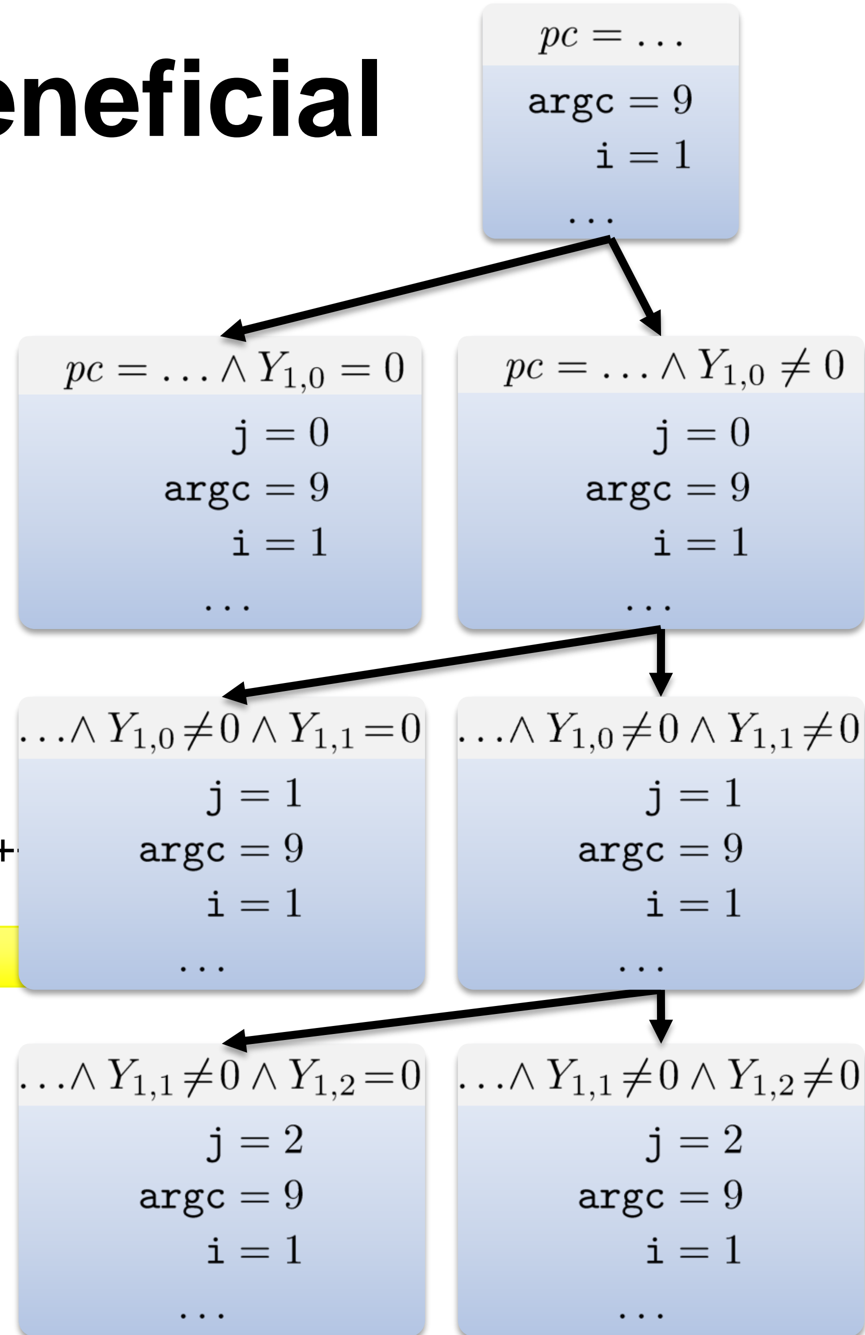
Merging Beneficial

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; +
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```



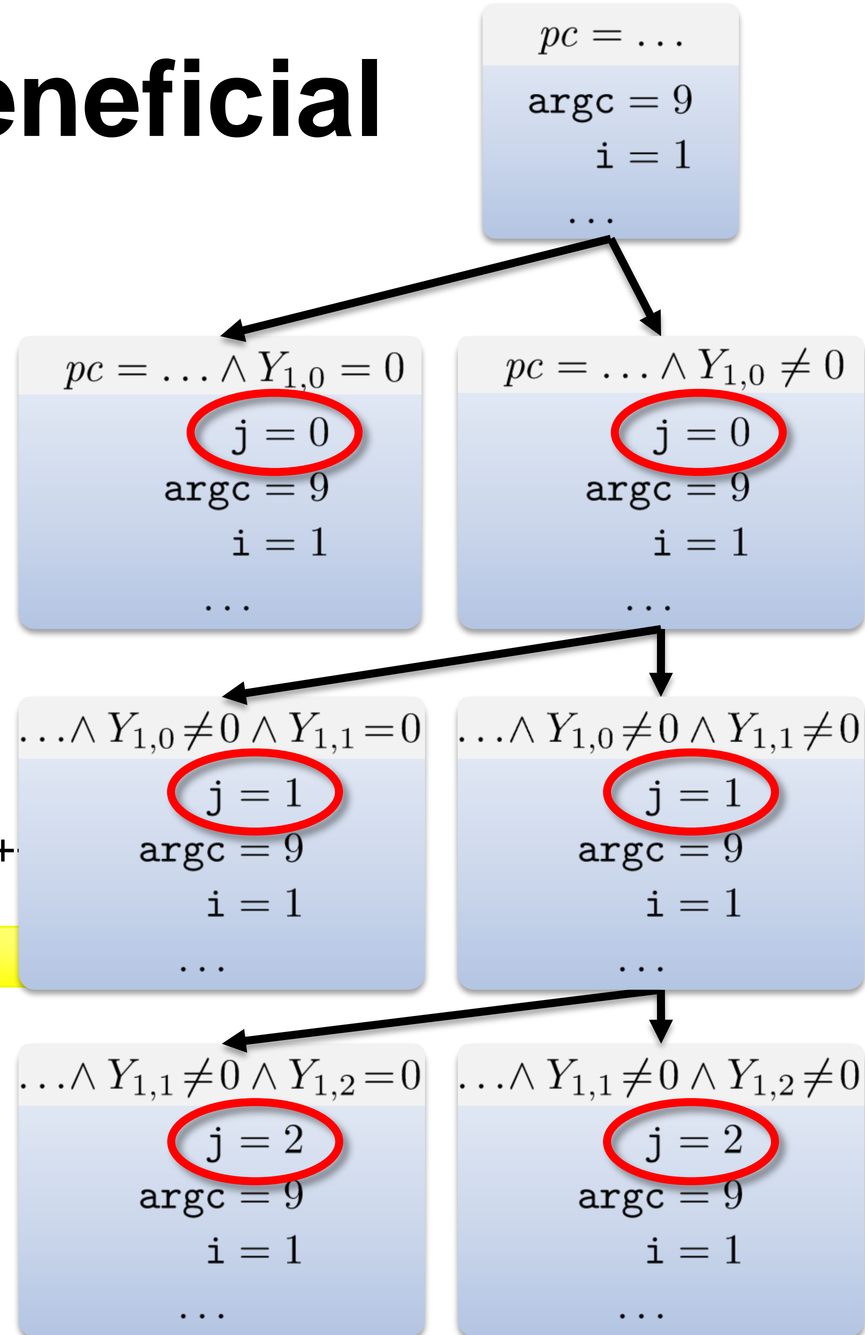
Merging Beneficial

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; +
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

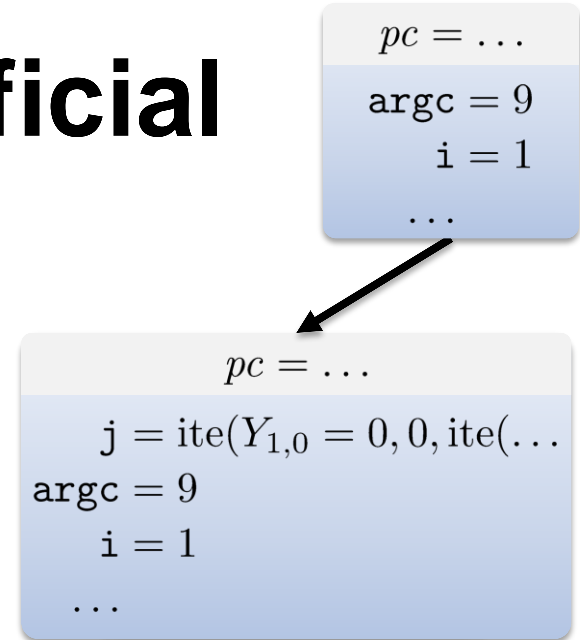


Merging Beneficial

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; +
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

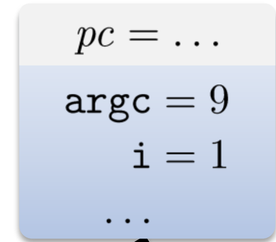


Merging Beneficial

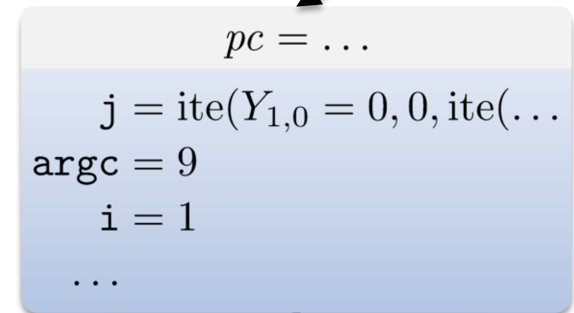


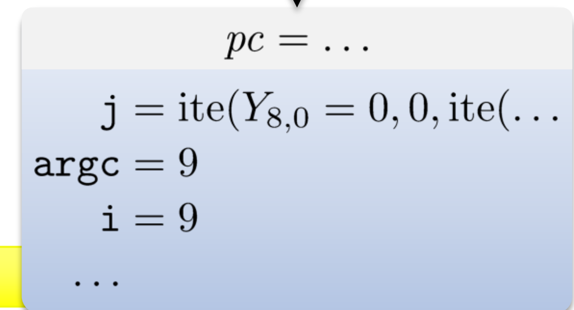

```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```

Merging Beneficial




```
void main(int argc, char **argv) {
  int r = 1, i = 1;

  if (i < argc) {
    if (argv[i][0] == 'n') {
      r = 0;
      ++i;
    }
  }

  for (; i < argc; ++i) {
    for (int j = 0; argv[i][j] != 0; ++j) {
      putchar(argv[i][j]);
    }
  }

  if (r) {
    putchar('\n');
  }
}
```



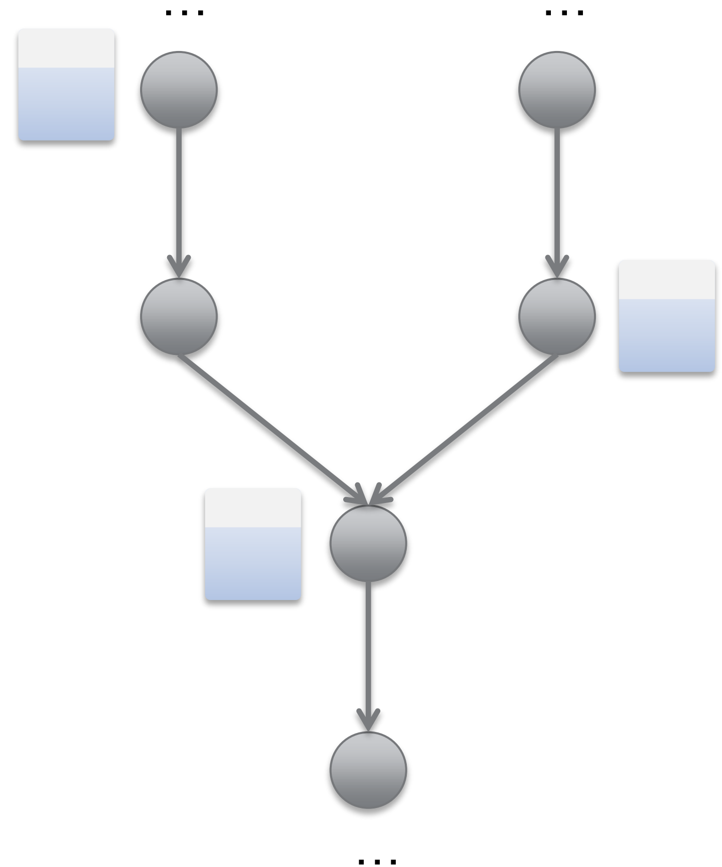


L^{argc} states reduced to 1

(L: maximum argument length)

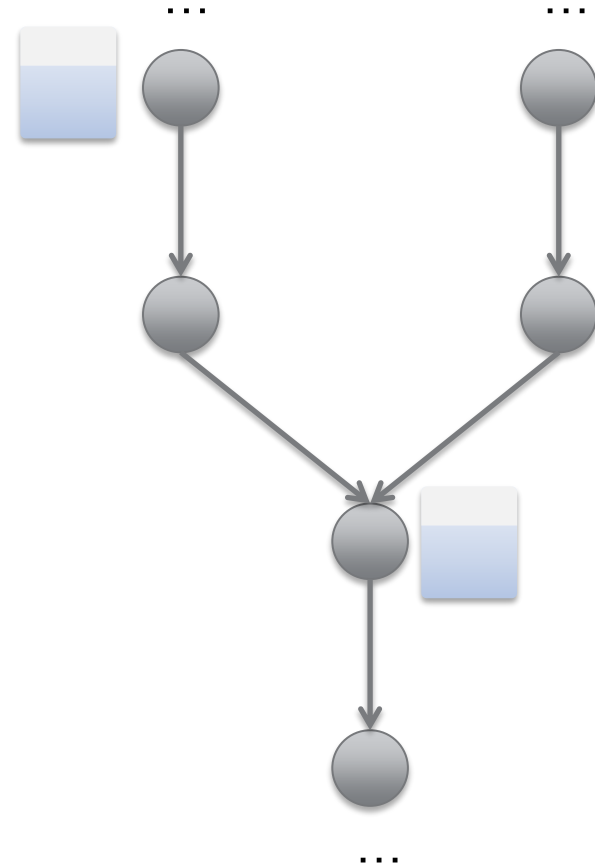
Search Strategies

- SE usually incomplete
  - *100% path coverage is impractical*
- Uses *search strategy* to reach a coverage goal
- Search strategy chooses states from a worklist



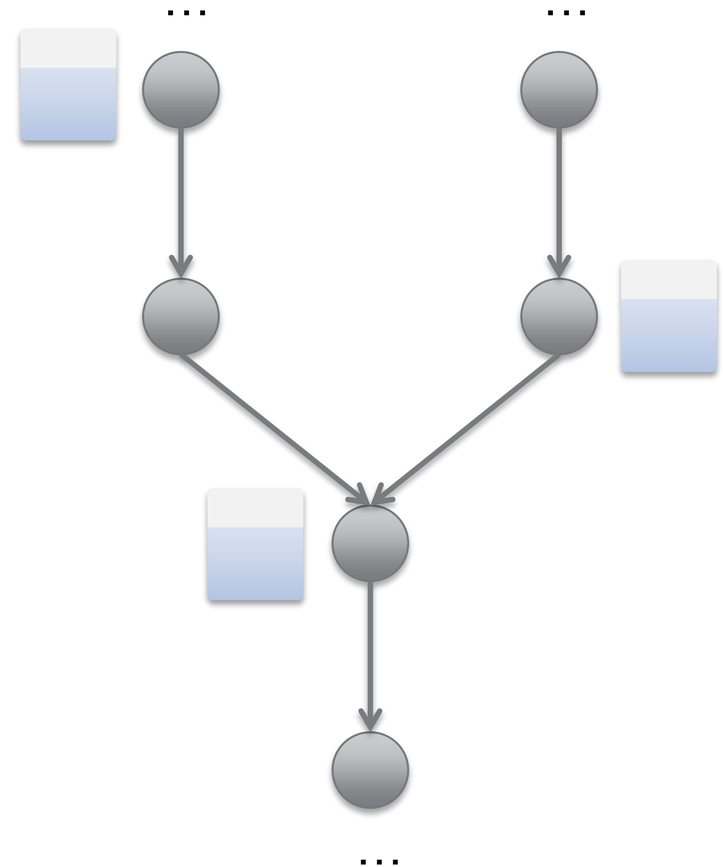
Search Strategies

- SE usually incomplete
  - *100% path coverage is impractical*
- Uses *search strategy* to reach a coverage goal
- Search strategy chooses states from a worklist



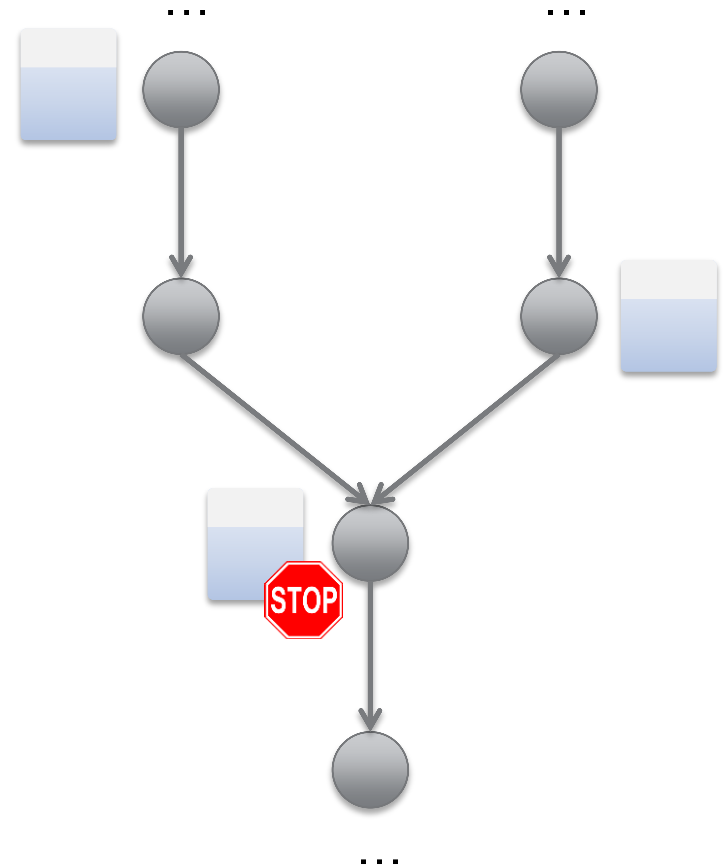
Merging Breaks Search Strategies

- Naïve state merging blocks states at join points
- Allows earlier states to "catch up"
- Thwarts strategy in reaching its goal!



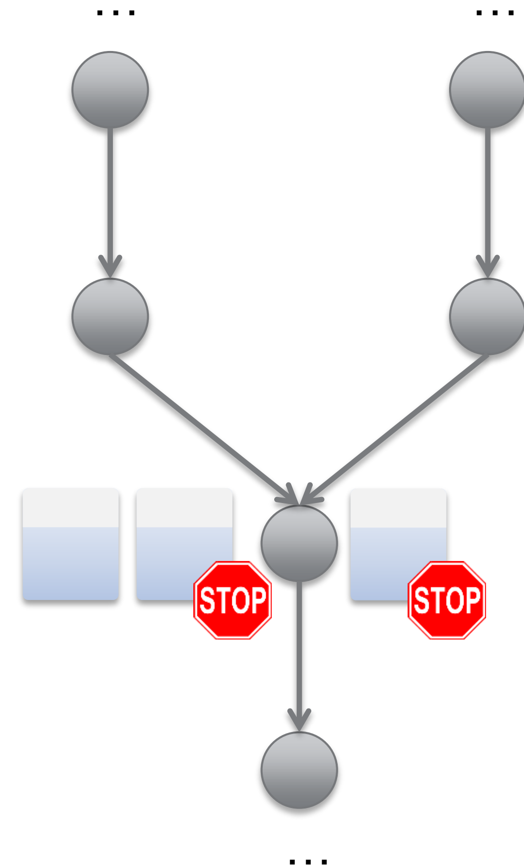
Merging Breaks Search Strategies

- Naïve state merging blocks states at join points
- Allows earlier states to “catch up”
- Thwarts strategy in reaching its goal!



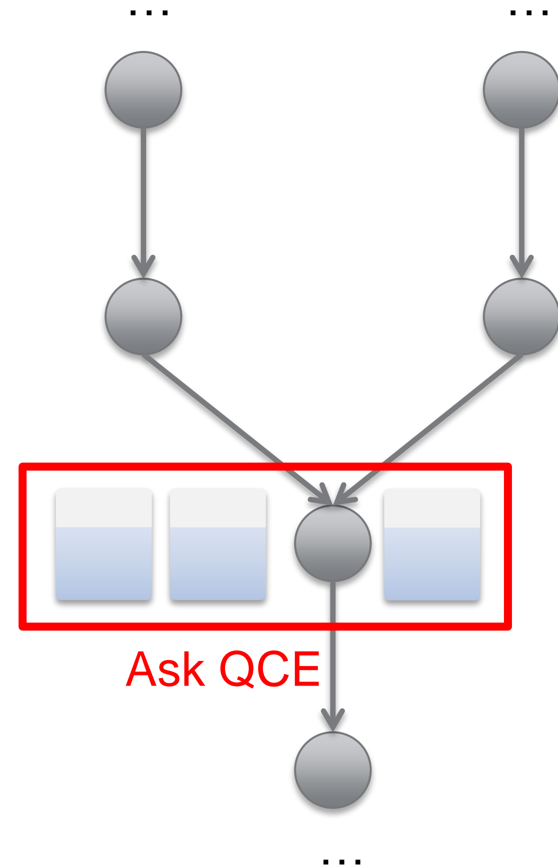
Merging Breaks Search Strategies

- Naïve state merging blocks states at join points
- Allows earlier states to "catch up"
- Thwarts strategy in reaching its goal!



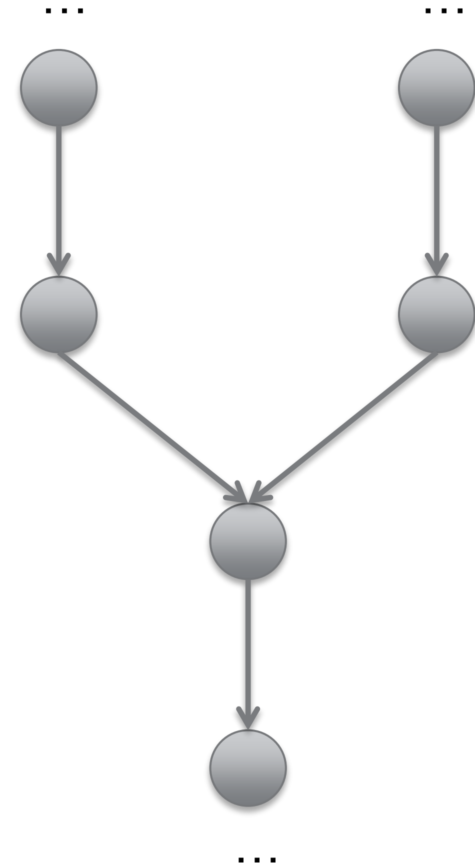
Merging Breaks Search Strategies

- Naïve state merging blocks states at join points
- Allows earlier states to “catch up”
- Thwarts strategy in reaching its goal!



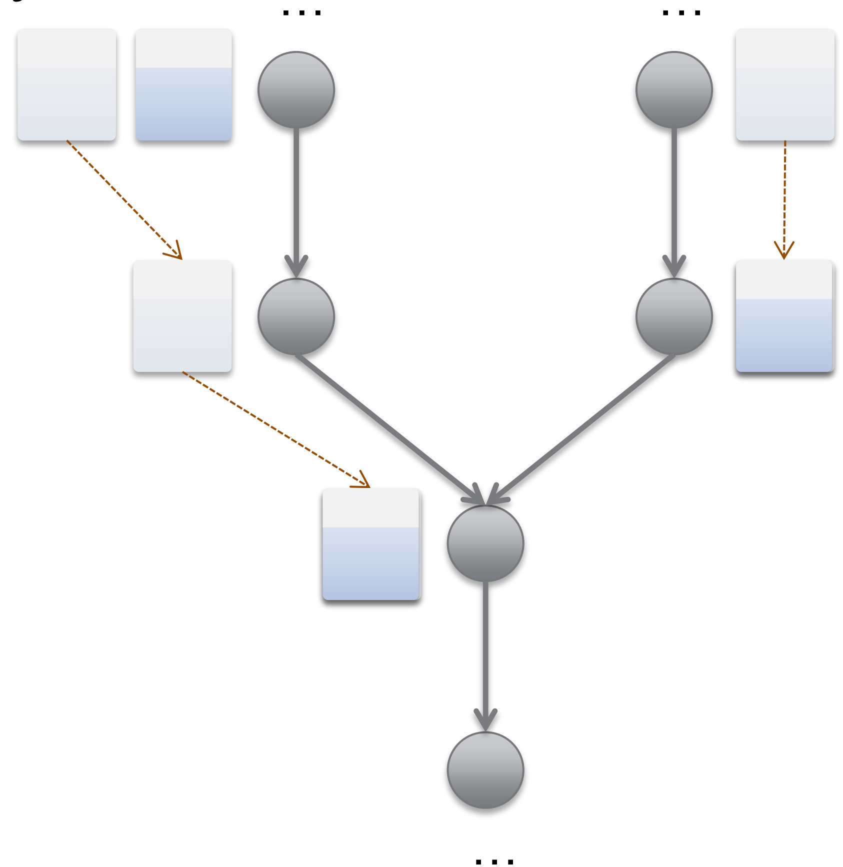
Merging Breaks Search Strategies

- Naïve state merging blocks states at join points
- Allows earlier states to "catch up"
- Thwarts strategy in reaching its goal!



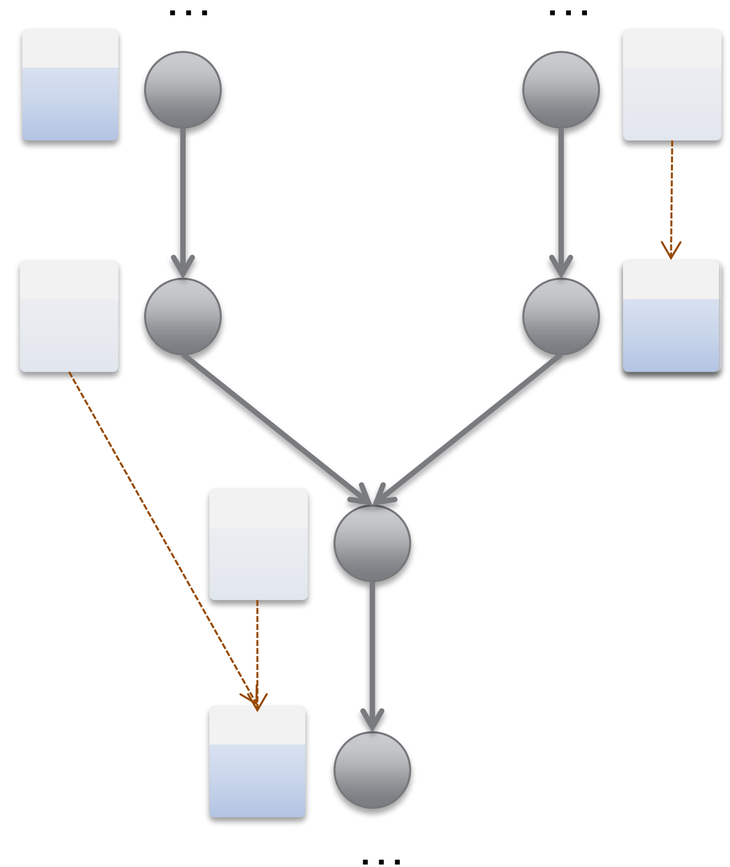
Dynamic State Merging

- Maintain bounded history of predecessor states



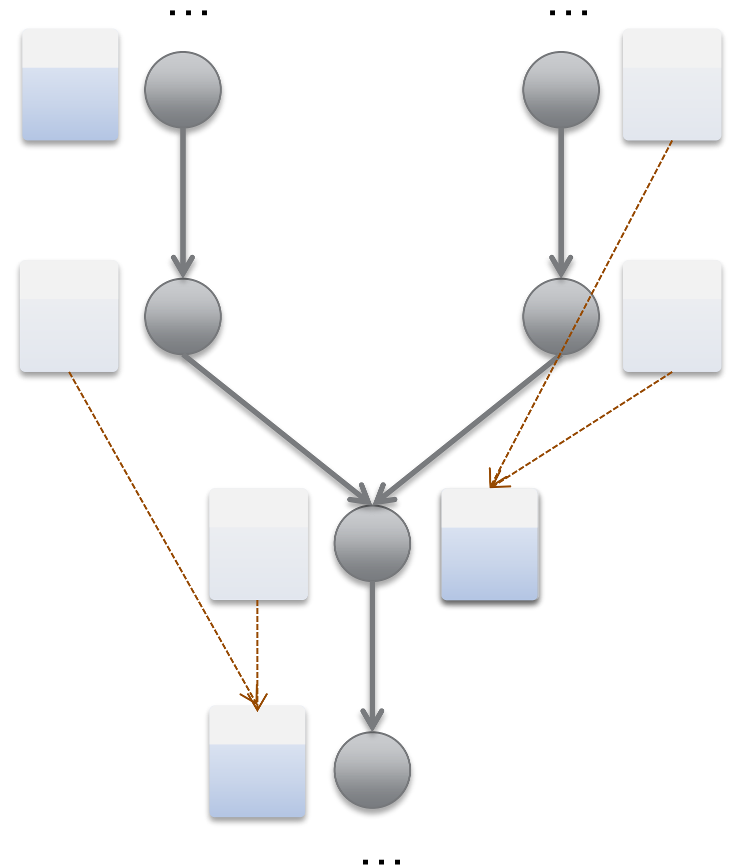
Dynamic State Merging

- Maintain bounded history of predecessor states



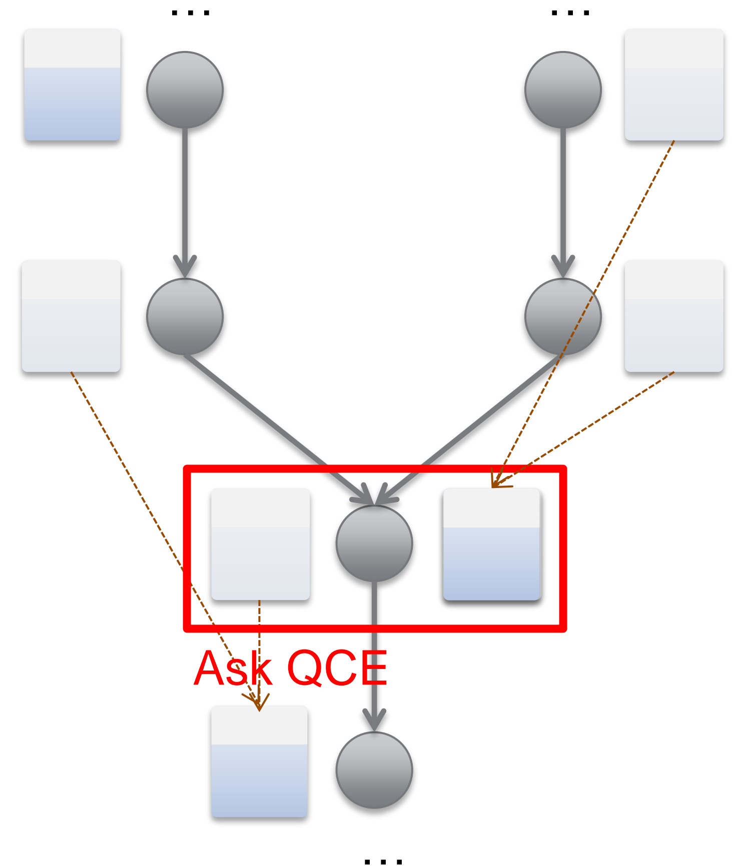
Dynamic State Merging

- Maintain bounded history of predecessor states



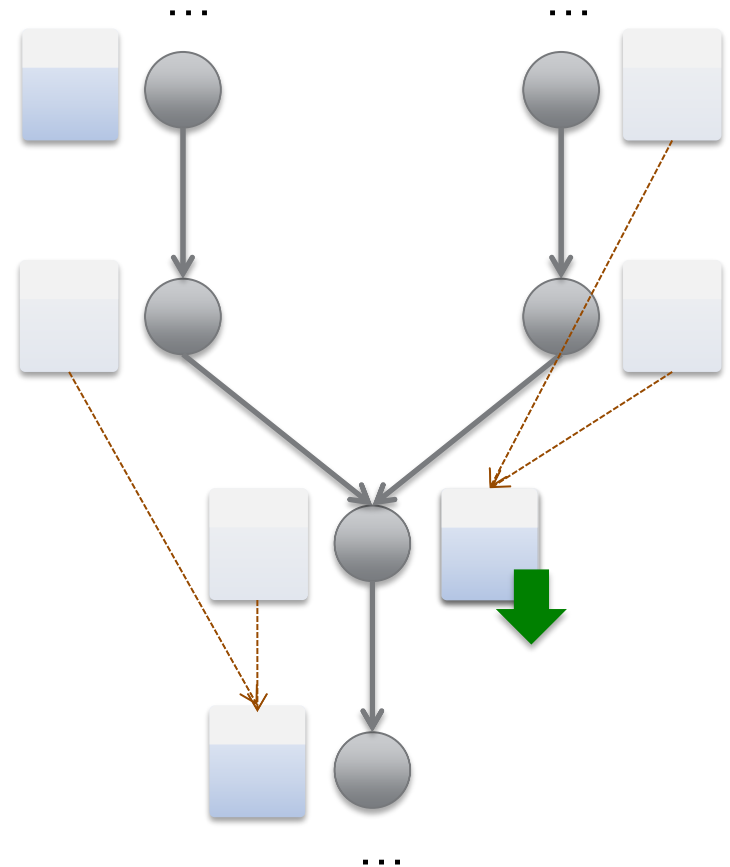
Dynamic State Merging

- Maintain bounded history of predecessor states
- QCE compares a state to predecessors of others



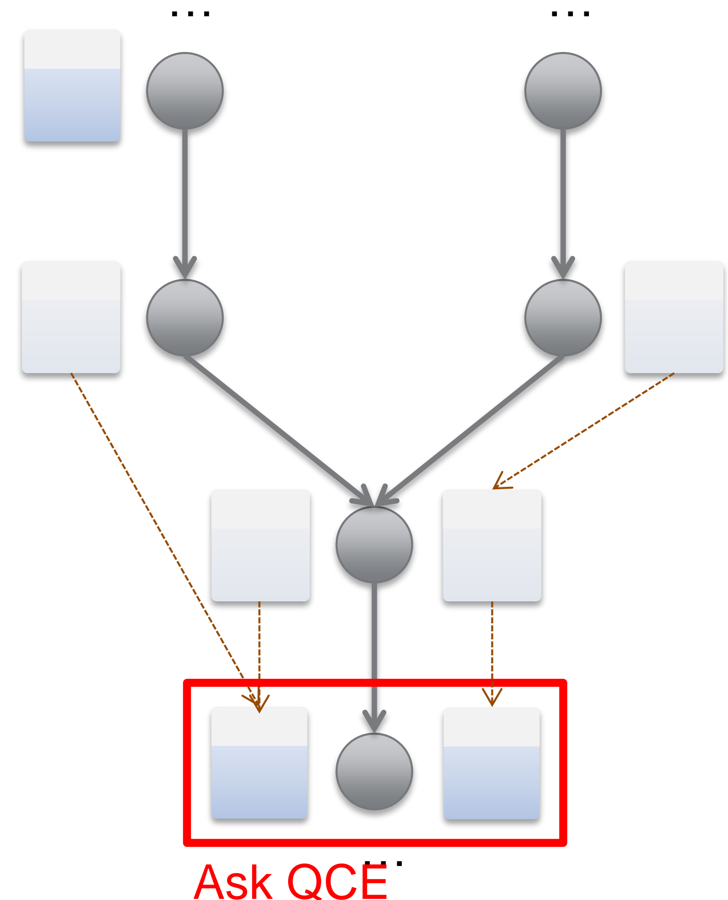
Dynamic State Merging

- Maintain bounded history of predecessor states
- QCE compares a state to predecessors of others
- Matching states are prioritized
- Original search strategy controls remaining worklist



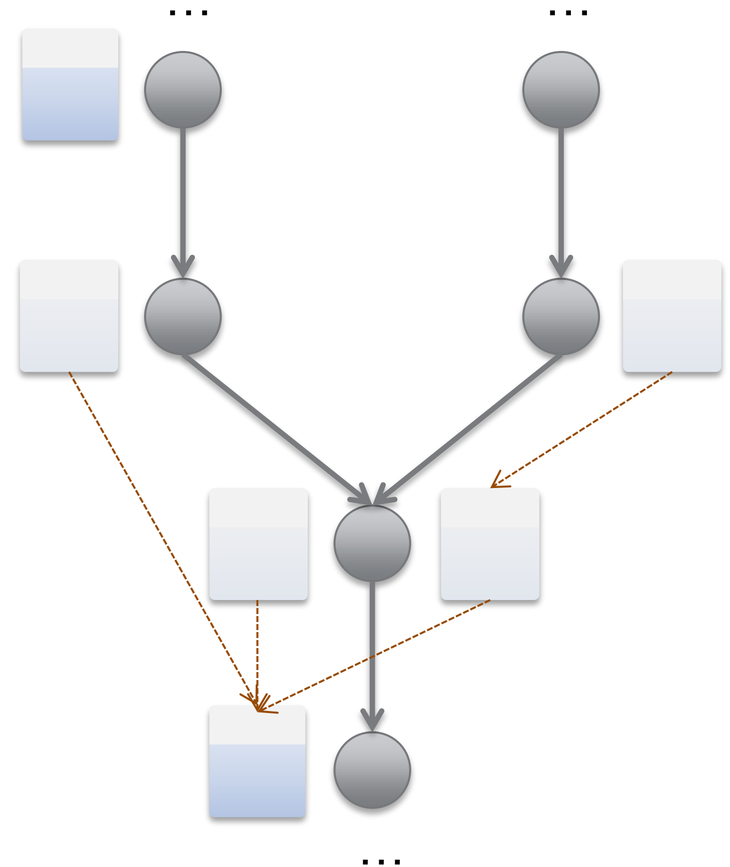
Dynamic State Merging

- Maintain bounded history of predecessor states
- QCE compares a state to predecessors of others
- Matching states are prioritized
- Original search strategy controls remaining worklist



Dynamic State Merging

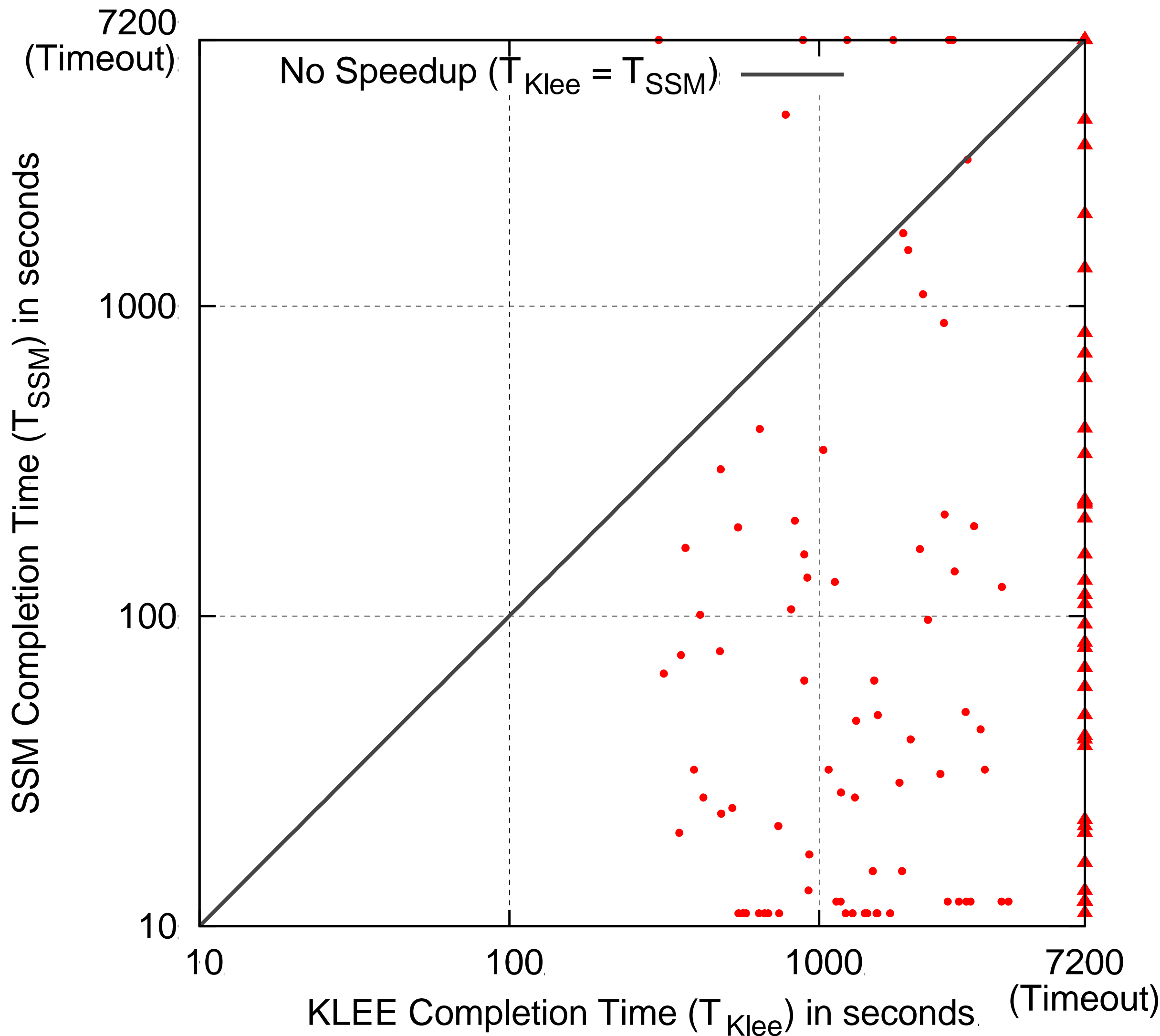
- Maintain bounded history of predecessor states
- QCE compares a state to predecessors of others
- Matching states are prioritized
- Original search strategy controls remaining worklist



Evaluation

- Our prototype
  - *Based on state-of-the-art engine KLEE*
  - *QCE implemented in LLVM – static analysis completes in seconds*
- Analysis Targets
  - *96 GNU Coreutils (echo, ls, dd, who, …)*
  - *2’000 – 10’000 executable lines of code per tool*
  - *72 KLOC in total*

Time for Exhaustive Exploration



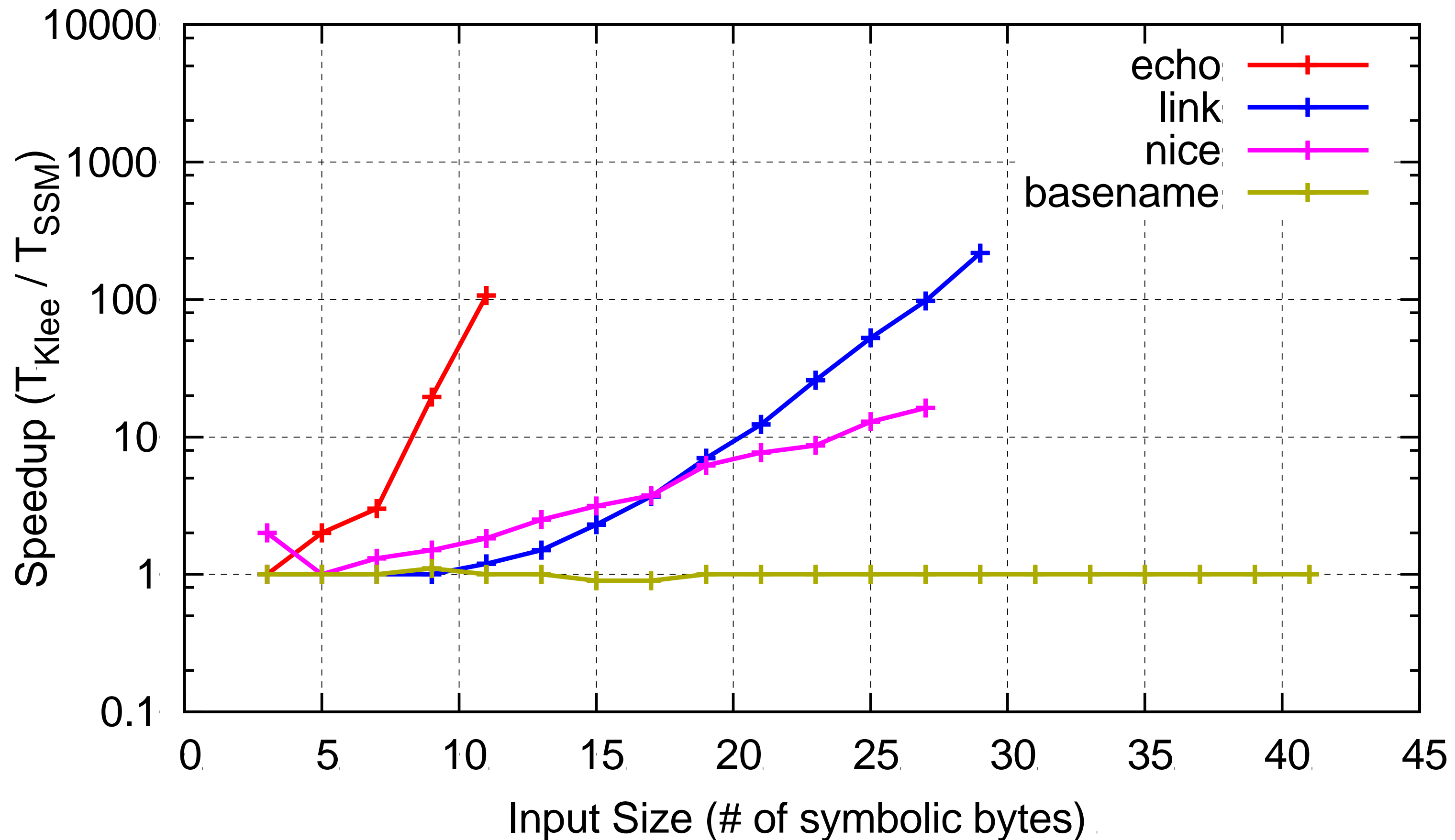
Triangle: KLEE time-out (> 2h)

QCE + SSM vs. KLEE.

Static state merging (SSM) follows topological order.

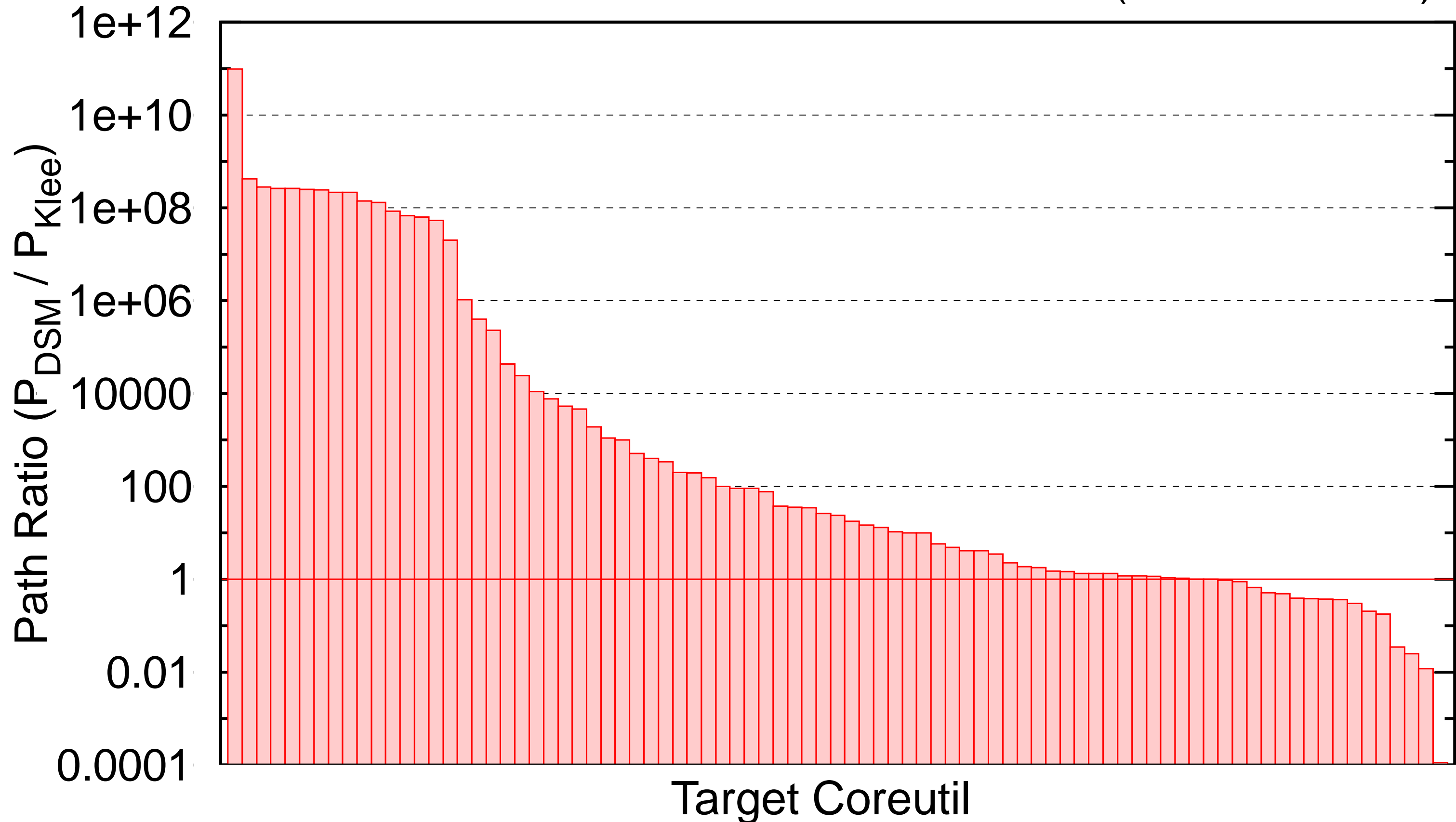
Consistent speedups

Speedup vs. Input-Size (Exhaustive)



Exponential speedups in symbolic input size

Paths in Incomplete Exploration



Up to 11 orders of magnitude more paths explored

Combating Path Explosion

- Dynamic merging (DSM + QCE)
- Static merging of small structures [Avgerinos et al., ICSE’14]
- Procedure summaries [Godefroid, POPL’07]
- Parallelization [Bucur et al, EuroSys’11]

Outline

- Symbolic Execution for Testing
- State Merging – Fighting Path Explosion
- Interpreted High-Level Code

Programming Languages

Symbolic Execution Engines

Programming Languages

Symbolic Execution Engines

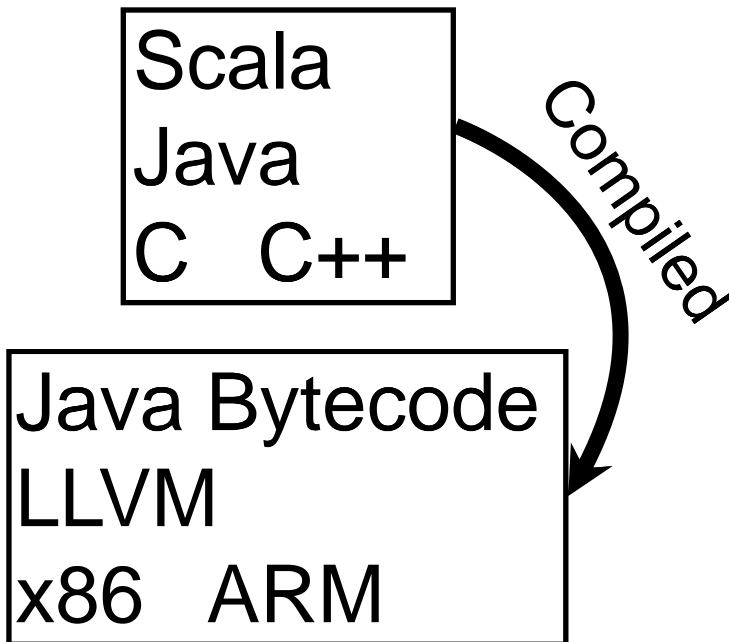
Scala

Java

C   C++

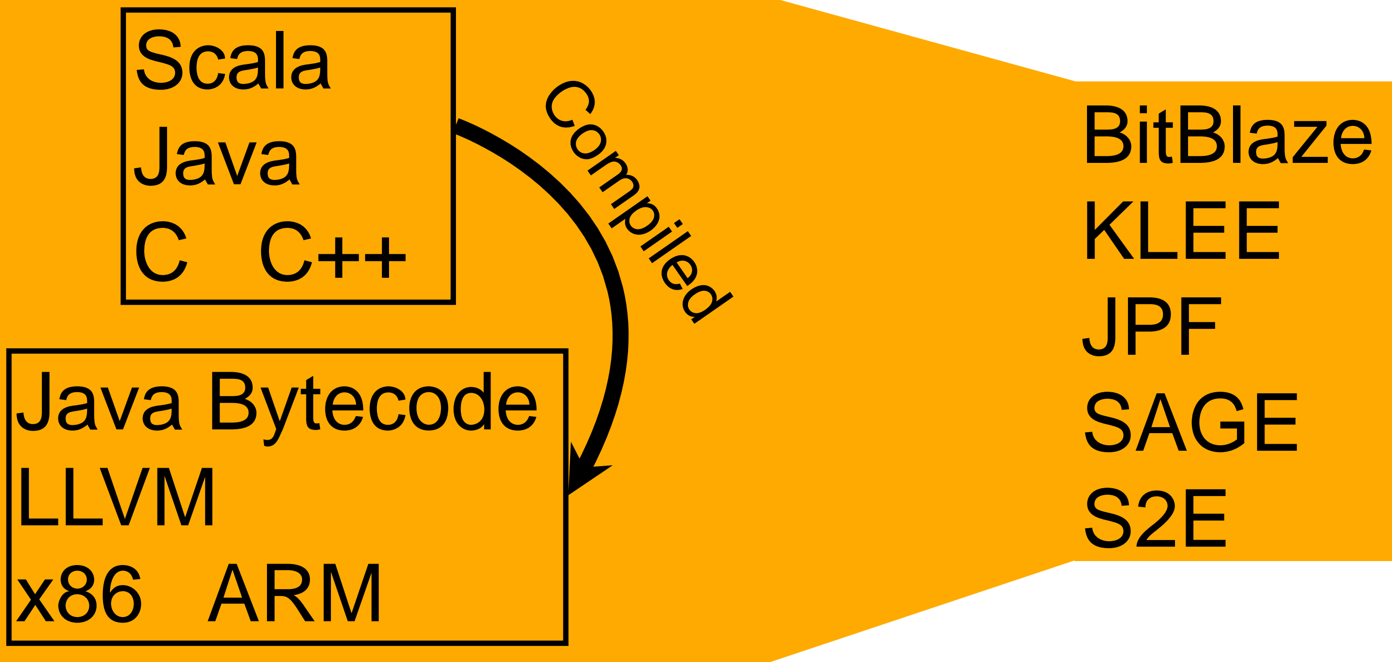
Programming Languages

Symbolic Execution Engines



Programming Languages

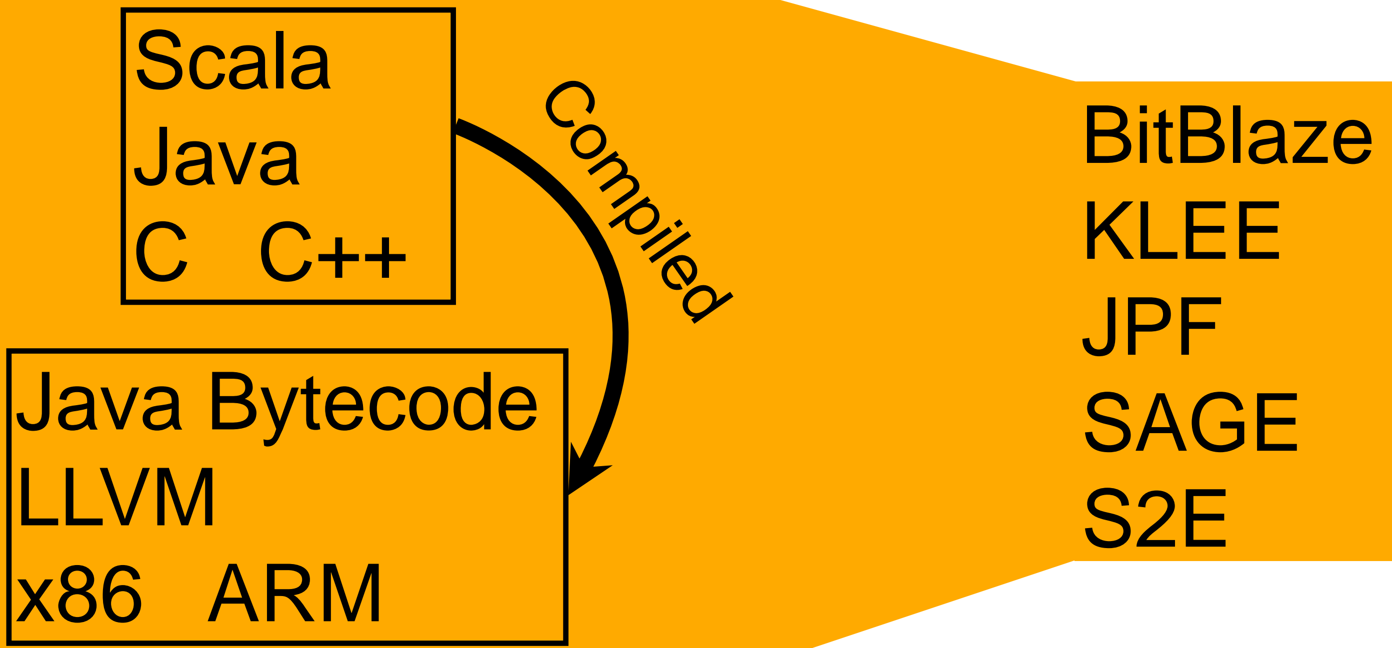
Symbolic Execution Engines



Programming Languages

Symbolic Execution Engines

Python Ruby
Lua JavaScript
Bash Perl



Programming Languages

Symbolic Execution Engines

Python Ruby
Lua JavaScript
Bash Perl

?

Scala
Java
C C++

Compiled

Java Bytecode
LLVM
x86 ARM

BitBlaze
KLEE
JPF
SAGE
S2E

Interpreted Languages

```
def parse_file(file_name):
  with open(file_name, "r") as f:
    data = f.read()
    return json.loads(data, encoding="utf-8")
```

Interpreted Languages

```
def parse_file(file_name):
    with open(file_name, "r") as f:
        data = f.read()
        return json.loads(data, encoding="utf-8")
```

Since Python 2.5 (with)

Complete File Read (f.read())

Incomplete Specification (json.loads(data, encoding="utf-8"))

Complex semantics

+

Ambiguity in specifications

+

Evolving language

+

Large standard library

+

Widespread native methods

Interpreted Languages

```
def parse_file(file_name):
    with open(file_name, "r") as f:
        data = f.read()
        return json.loads(data, encoding="utf-8")
```

Since Python 2.5 (with)

Complete File Read (f.read())

Incomplete Specification (json.loads(data, encoding="utf-8"))

Too much work:

Complex semantics
+
Ambiguity in specifications
+
Evolving language
+
Large standard library
+
Widespread native methods

"Consequently, if you were coming from Mars and tried to re-implement Python from this document alone, you might have to ***guess things*** *and in fact you would probably end up* ***implementing quite a different language****."*

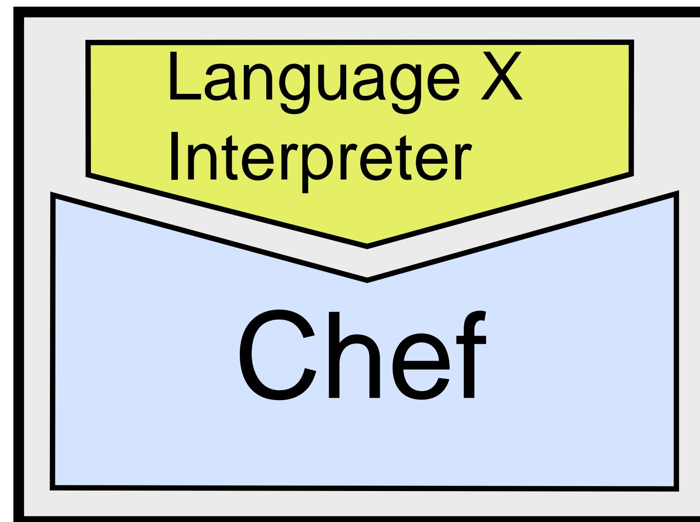
- The Python Language Reference

How can we **efficiently** obtain
a **correct** symbolic execution engine?

Key idea:
Use the language interpreter
as executable specification

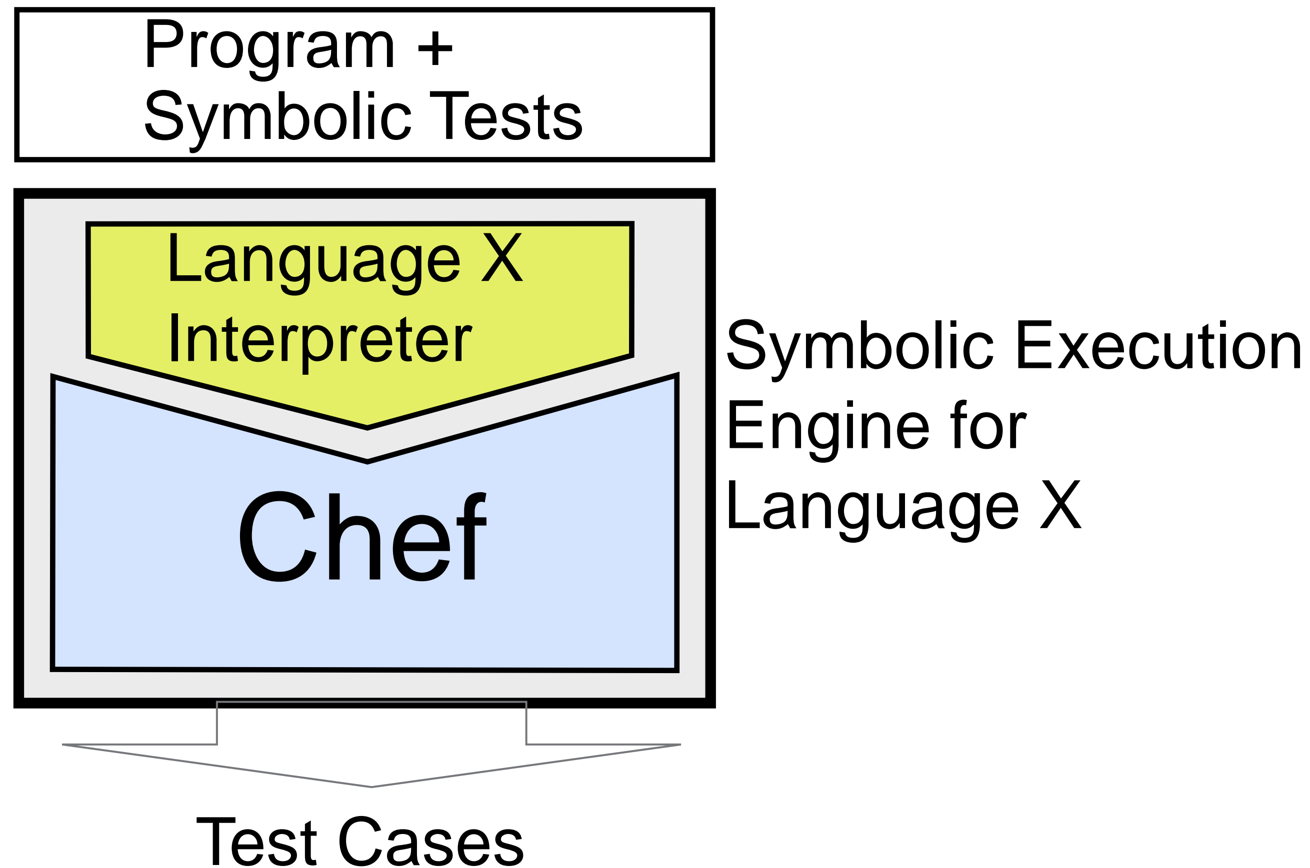
[Bucur, K, Candea, ASPLOS'14]

Key idea:
Use the language interpreter as executable specification



[Bucur, K, Candea, ASPLOS'14]

Key idea:
Use the language interpreter as executable specification



[Bucur, K, Candea, ASPLOS'14]

Chef Overview

- Built on top of the S2E symbolic execution engine for x86
- Relies on lightweight interpreter instrumentation + optimizations
- Prototyped engines for Python and Lua in 5 + 3 person-days

Testing Interpreted Programs

Naive approach: Run interpreter in stock SE engine

Testing Interpreted Programs

Naive approach: Run interpreter in stock SE engine

```
def validateEmail(email):
  pos = email.find("@")
  if pos < 1:
    raise InvalidEmailError()
  if email.rfind(".") < pos:
    raise InvalidEmailError()
```

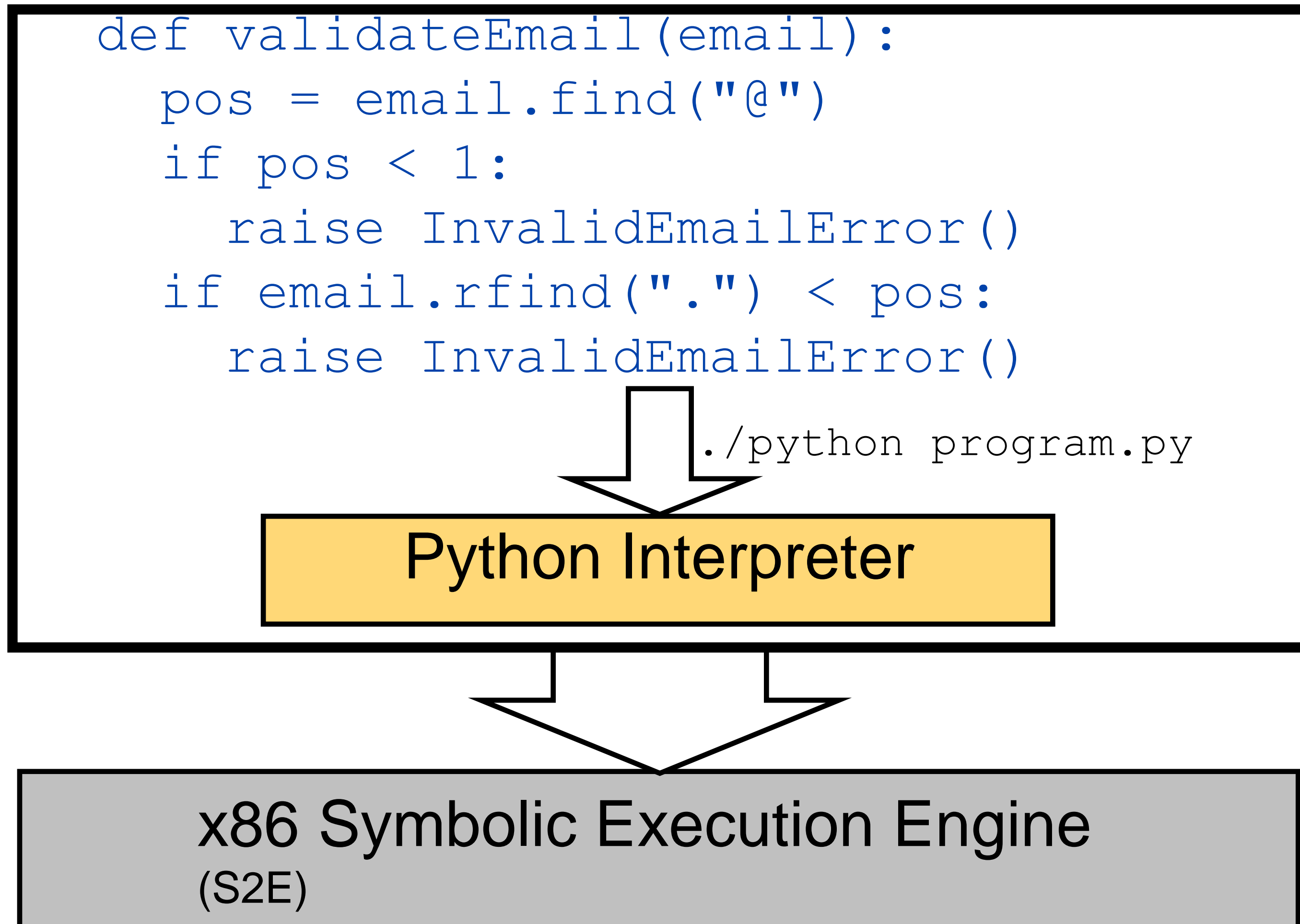
Testing Interpreted Programs

Naive approach: Run interpreter in stock SE engine

```
def validateEmail(email):
  pos = email.find("@")
  if pos < 1:
    raise InvalidEmailError()
  if email.rfind(".") < pos:
    raise InvalidEmailError()
```

Testing Interpreted Programs

Naive approach: Run interpreter in stock SE engine



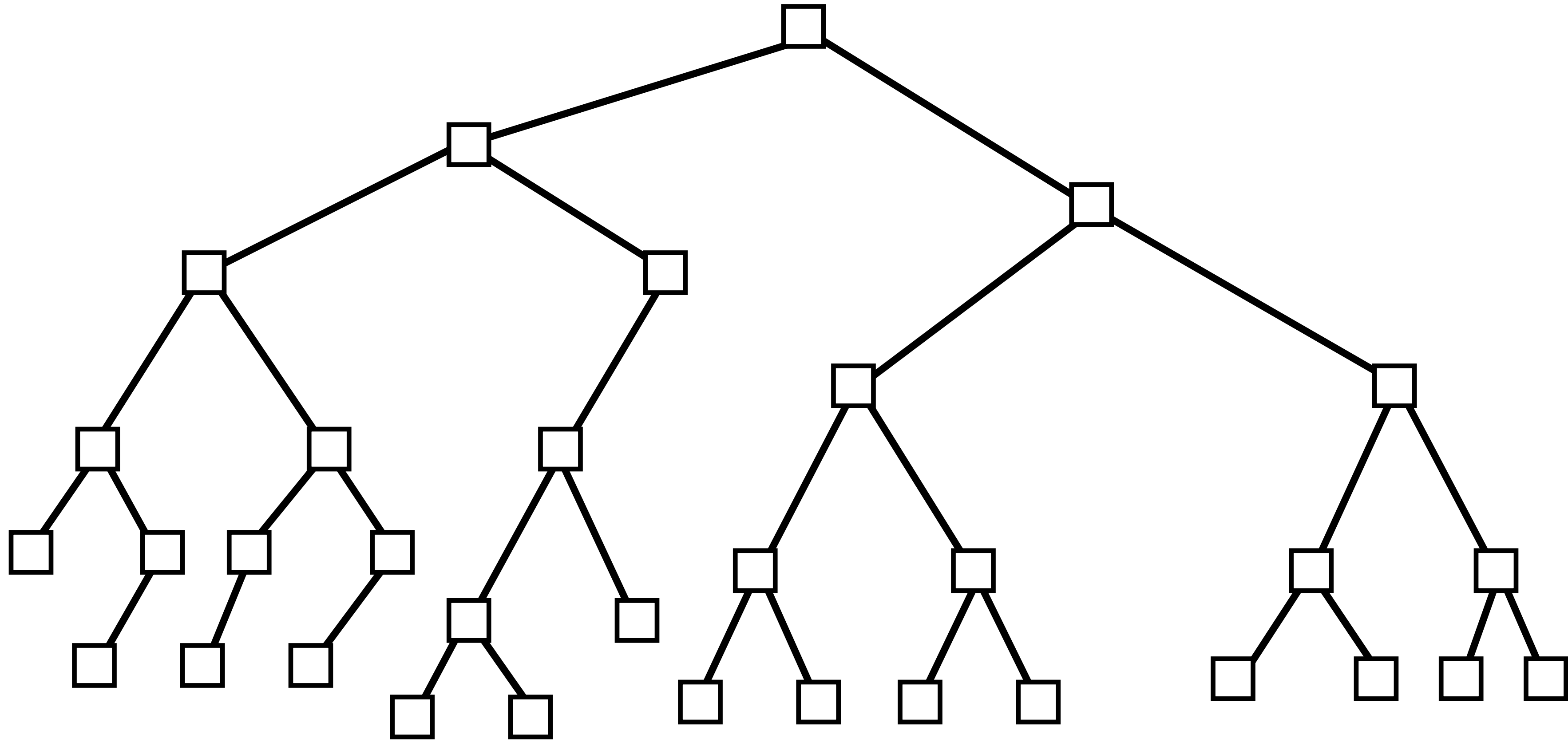
Naive approach:
Run interpreter in stock symbolic execution engine

```
pos = email.find("@")
```

```
{
    unsigned long mask;
    Py_ssize_t skip, count = 0;
    Py_ssize_t i, j, mlast, w;

    w = n - m;

    if (w < 0 || (mode == FAST_COUNT && maxcount == 0))
        return -1;

    /* look for special cases */
    if (m <= 1) {
        if (m <= 0)
            return -1;
        /* use special case for 1-character strings */
        if (mode == FAST_COUNT) {
```

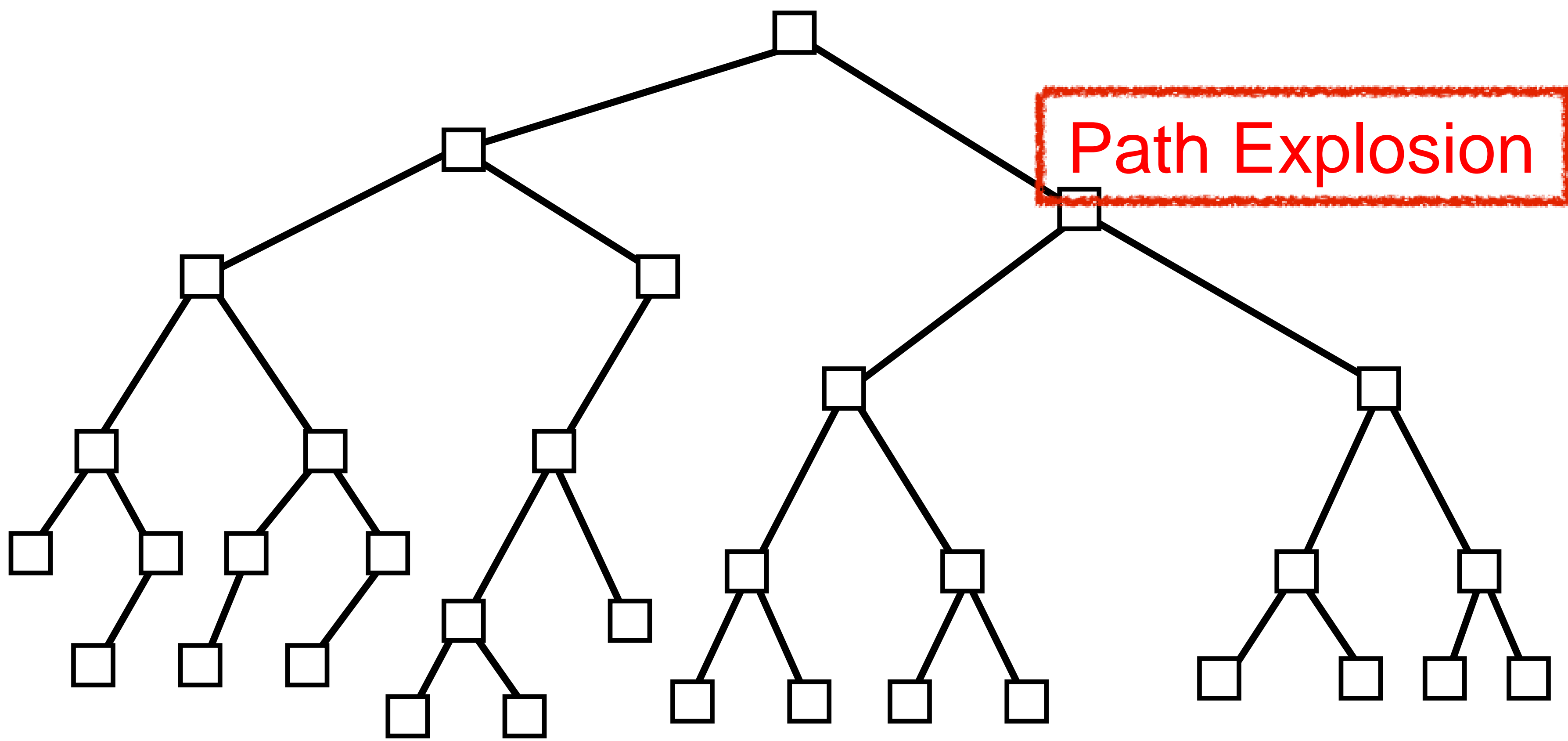
Naive approach:
Run interpreter in stock symbolic execution engine

```
pos = email.find("@")
```



Naive approach:
Run interpreter in stock symbolic execution engine

```
pos = email.find("@")
```

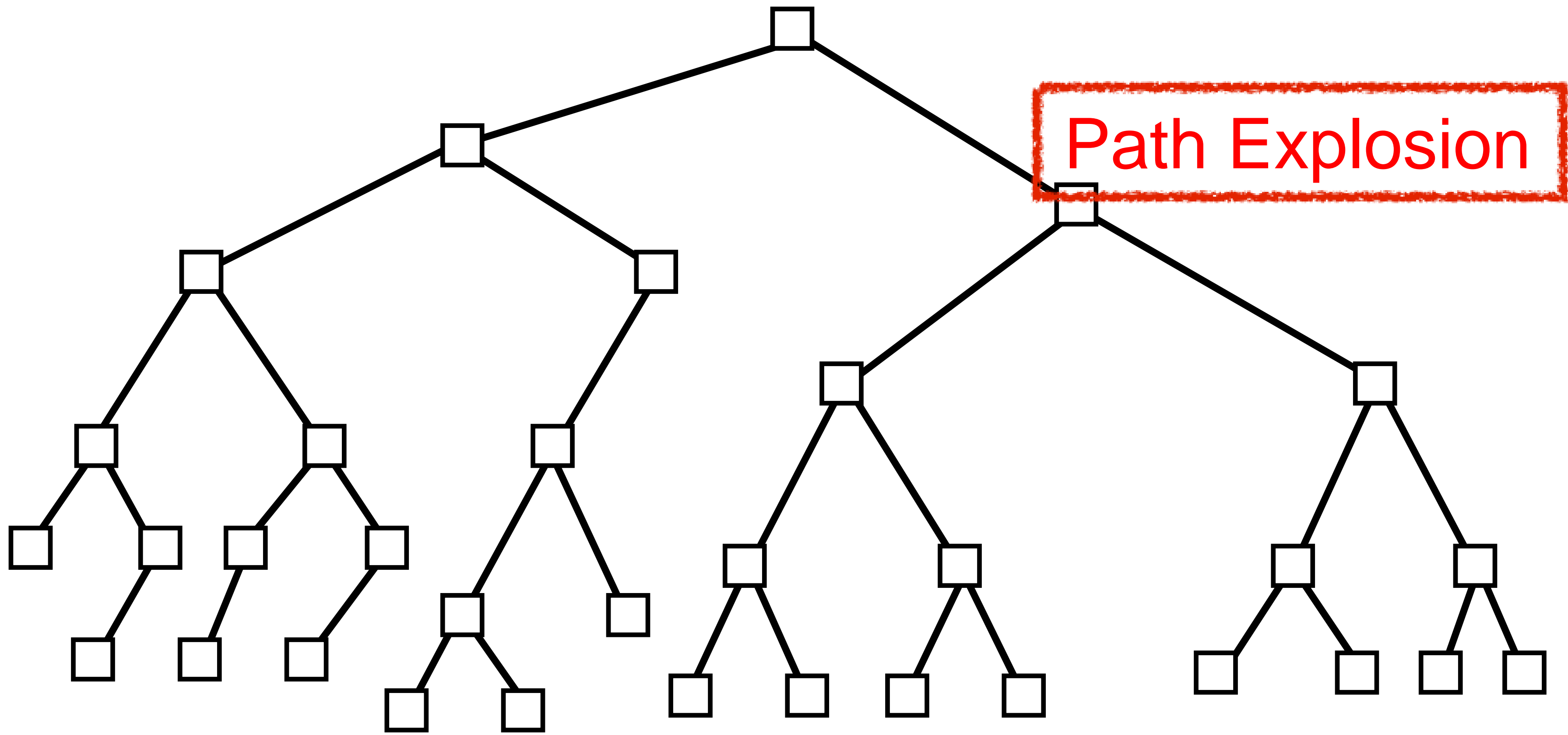


Naive approach:

~~Run interpreter in stock symbolic execution engine~~

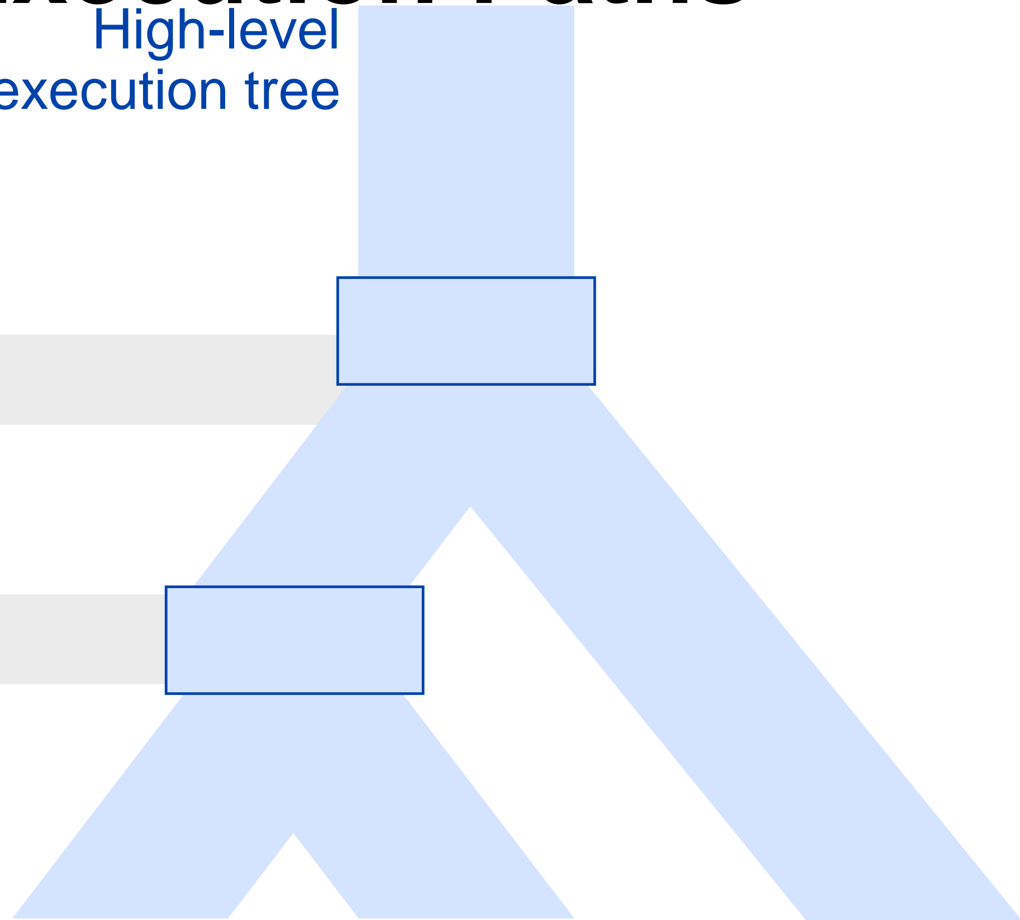
Gets lost in the details of the implementation

```
pos = email.find("@")
```



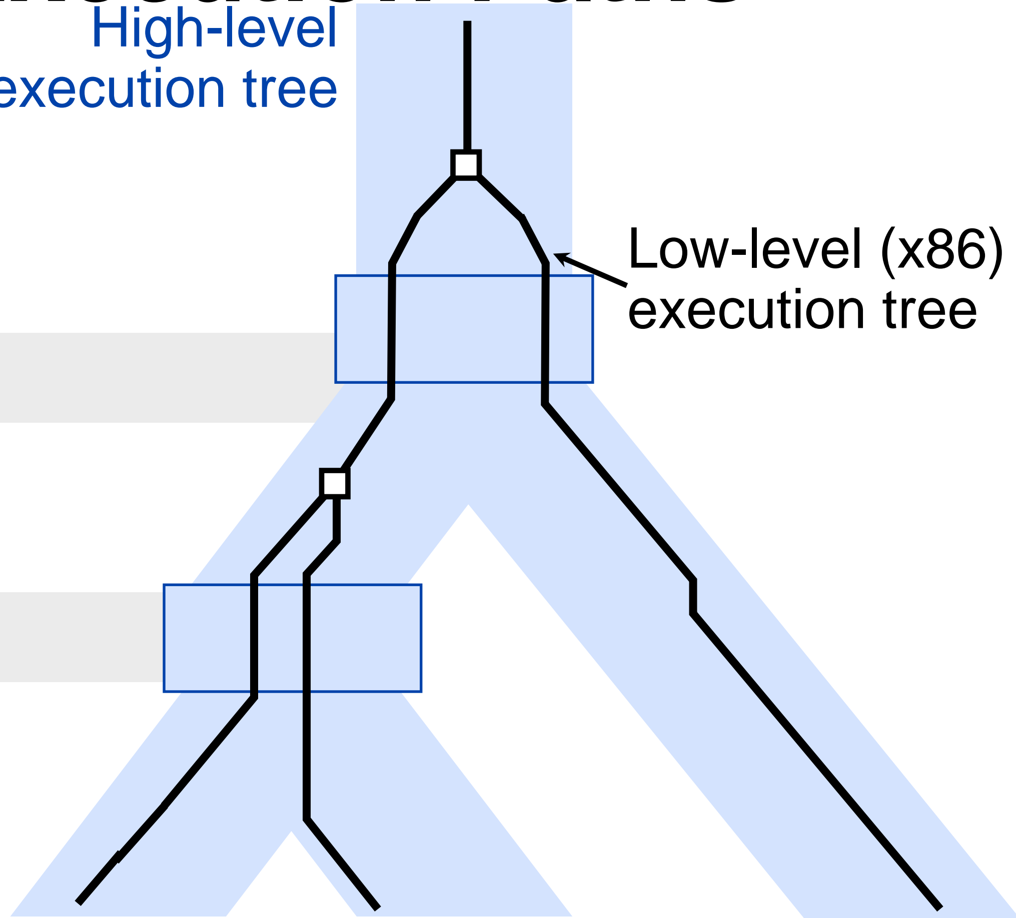
High-level Execution Paths

```
def validateEmail(email):
  pos = email.find("@")
  if pos < 1:
    raise InvalidEmailError()
  if email.rfind(".") < pos:
    raise InvalidEmailError()
```



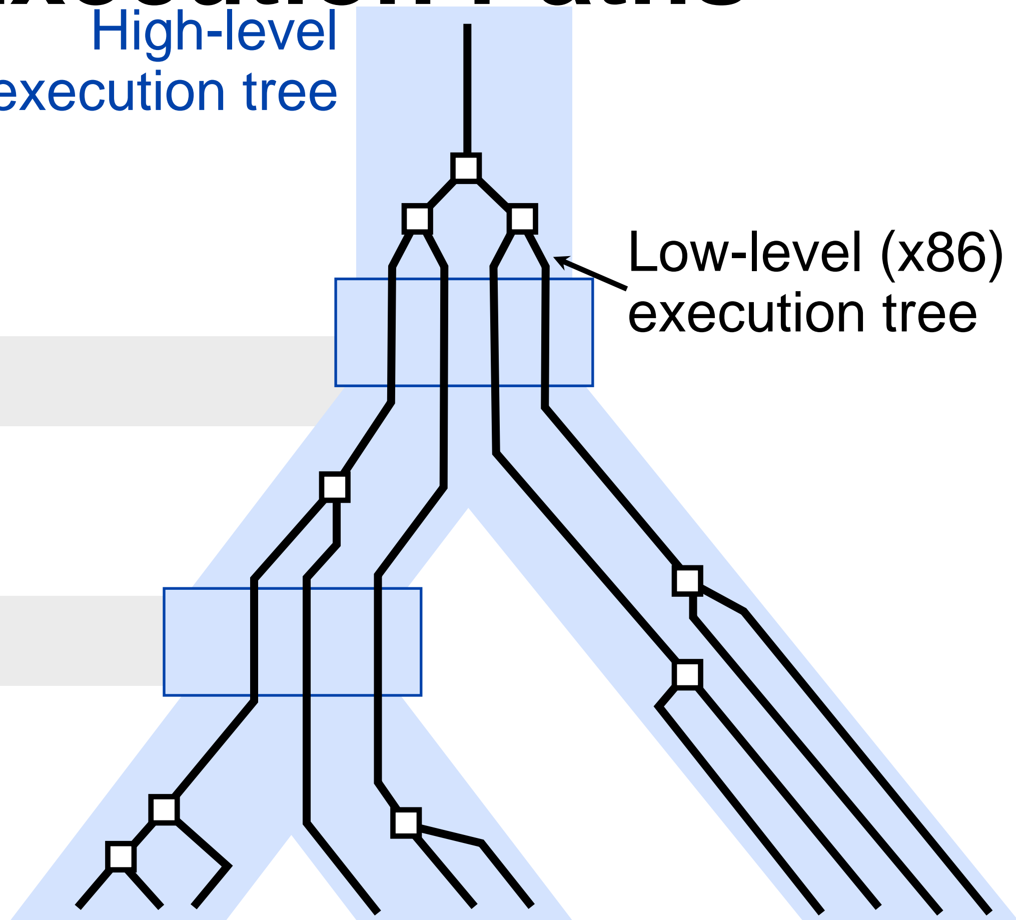
High-level Execution Paths

```
def validateEmail(email):
  pos = email.find("@")
  if pos < 1:
    raise InvalidEmailError()
  if email.rfind(".") < pos:
    raise InvalidEmailError()
```



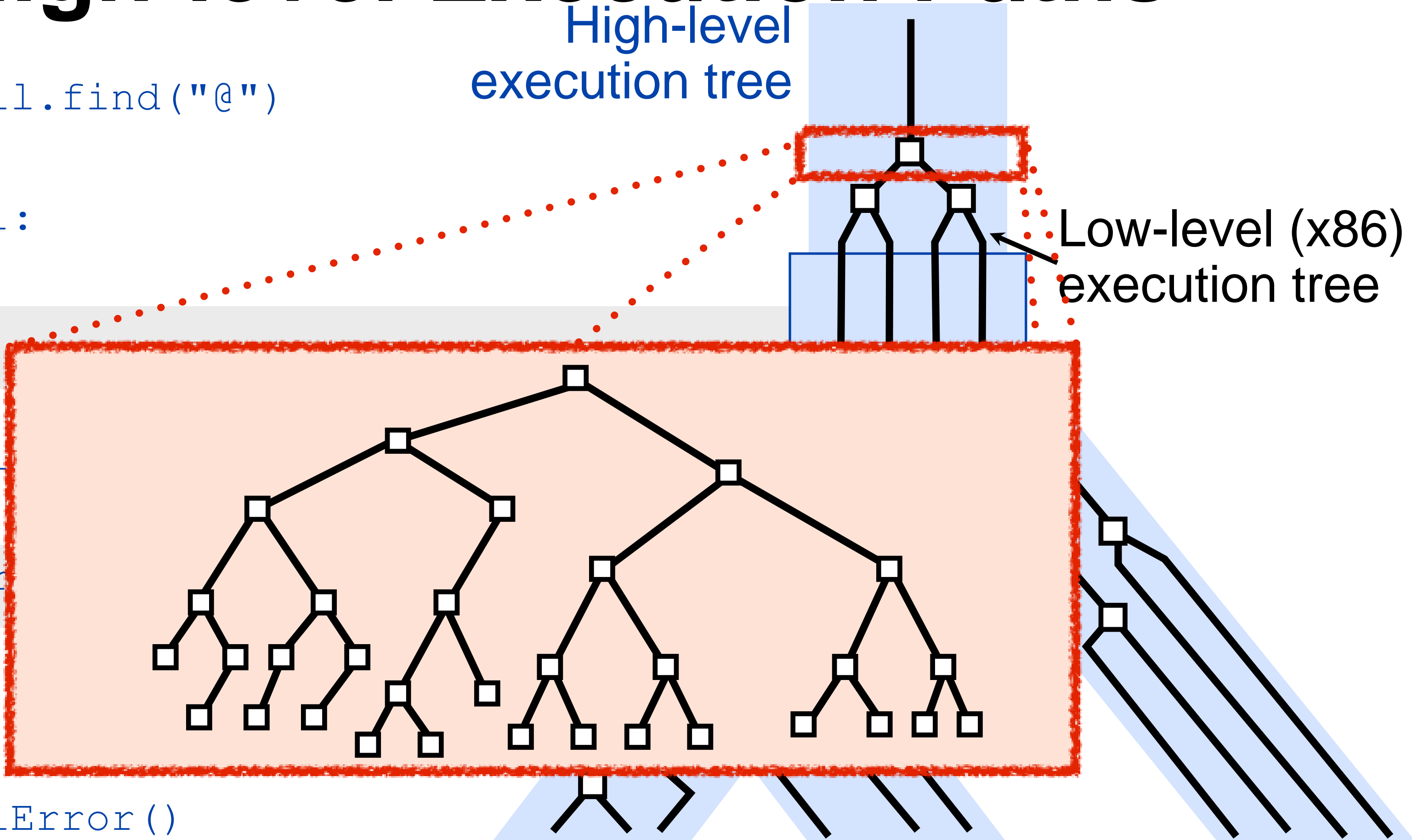
High-level Execution Paths

```
def validateEmail(email):
  pos = email.find("@")
  if pos < 1:
    raise InvalidEmailError()
  if email.rfind(".") < pos:
    raise InvalidEmailError()
```

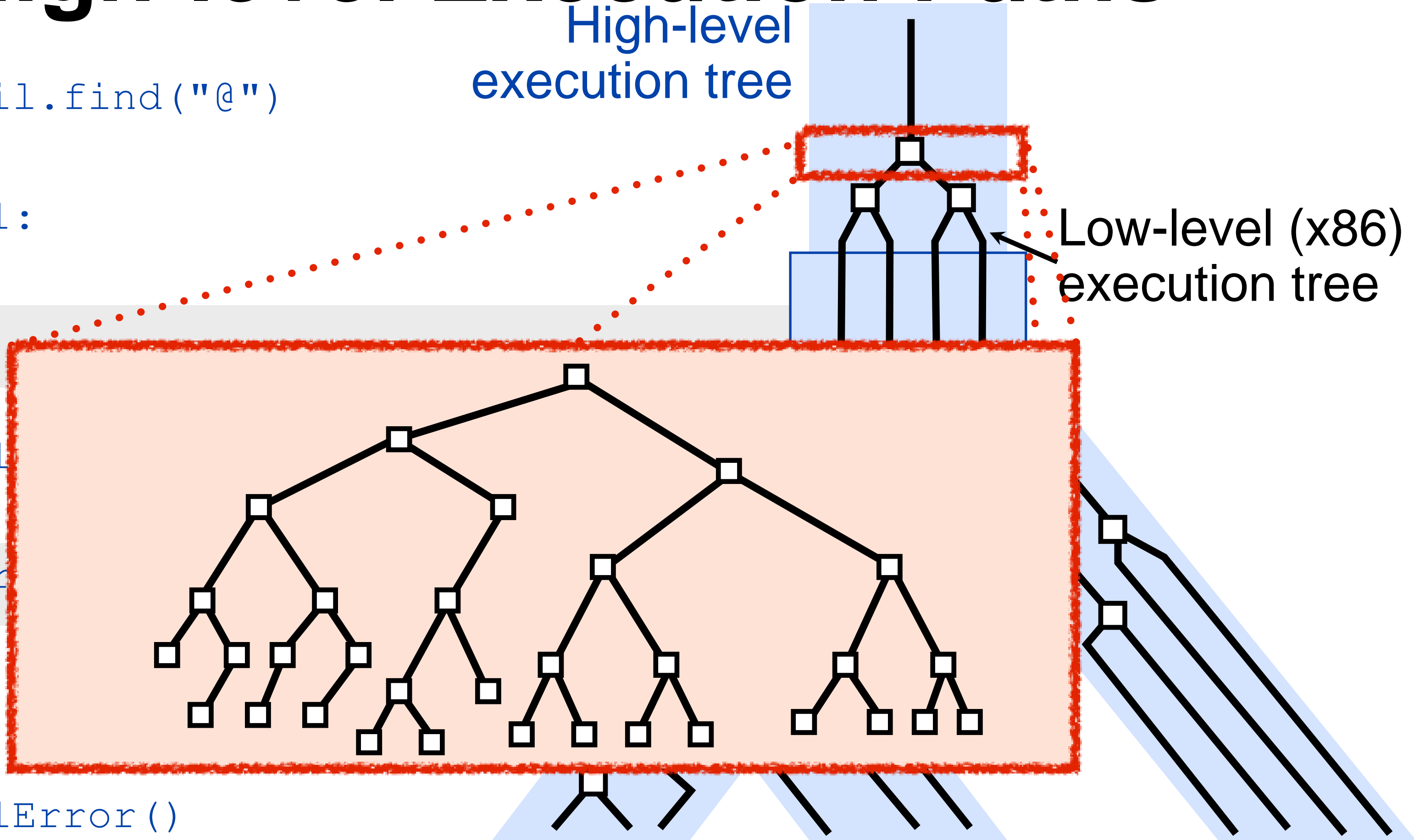


High-level Execution Paths

```
def validateEmail(email):
  pos = email.find("@")
  if pos < 1:
    raise
InvalidEmail
  if email.r
    raise
InvalidEmailError()
```

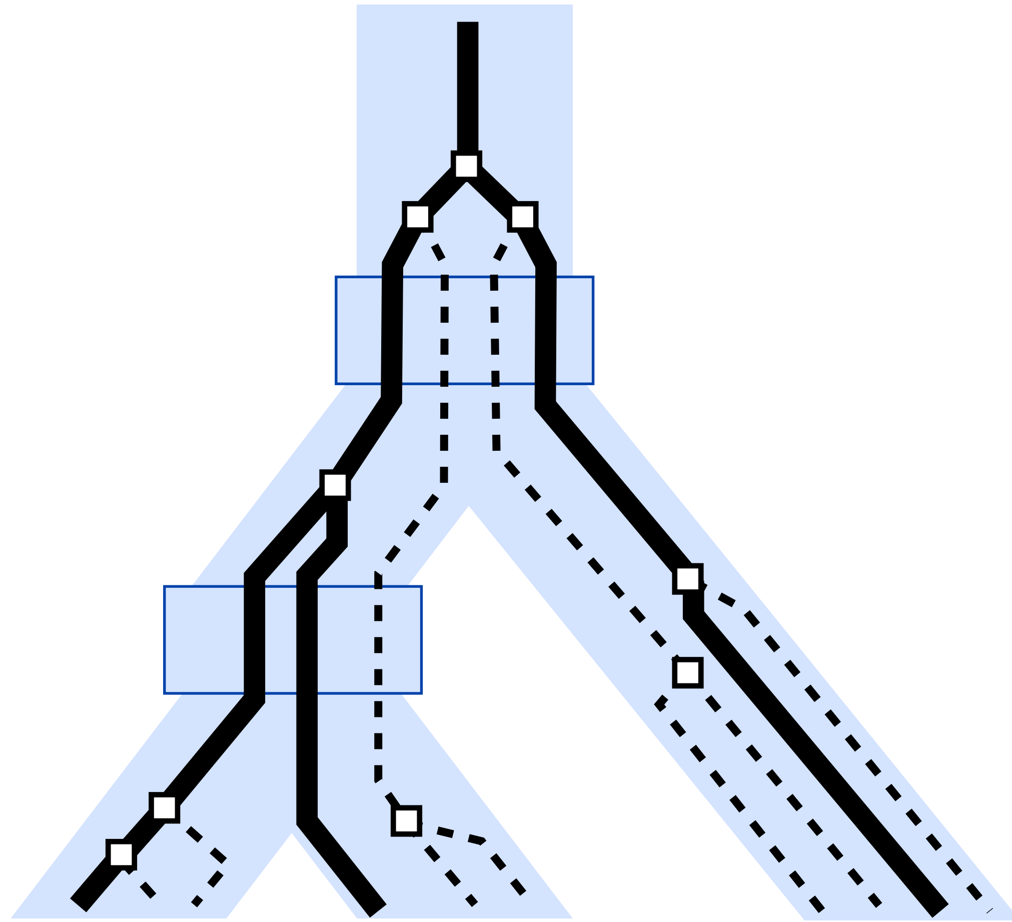


High-level Execution Paths



$$\frac{\text{3 HL paths}}{\text{10 LL paths}}$$

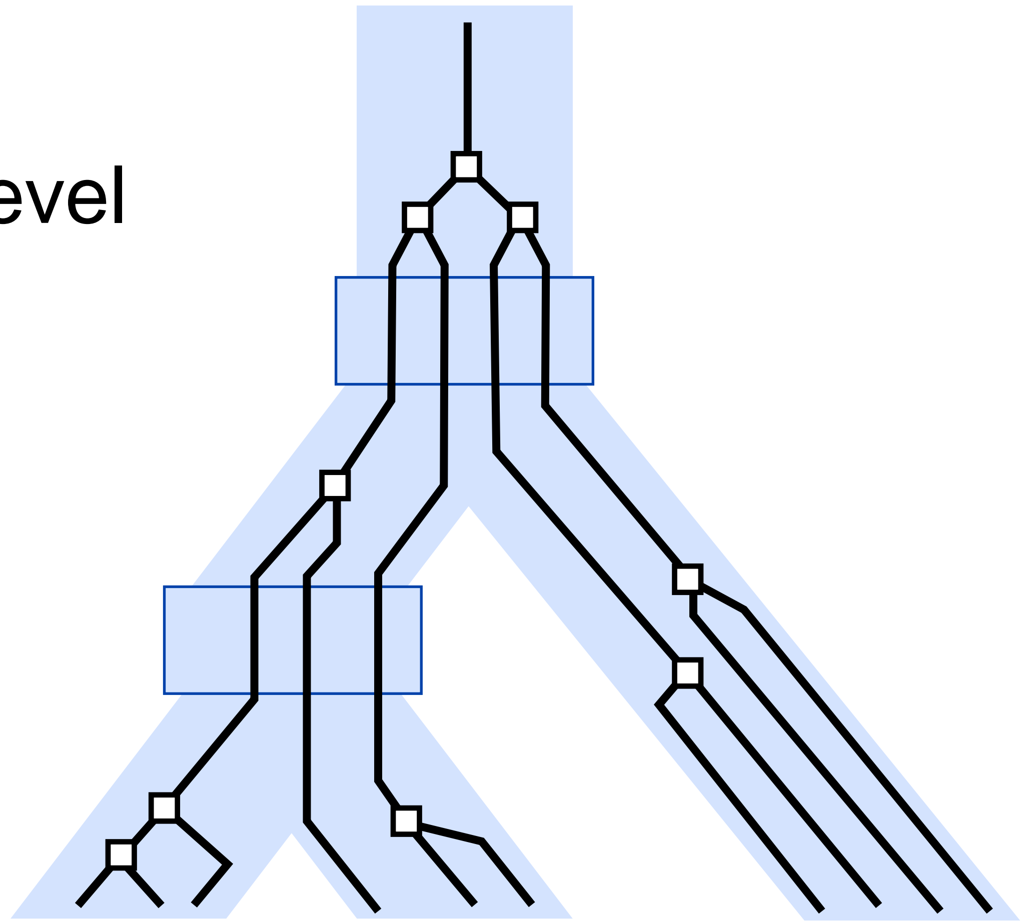
HL/LL path ratio is low due to path explosion



Goal:
Prioritize the low-level paths
that maximize the HL/LL path ratio.

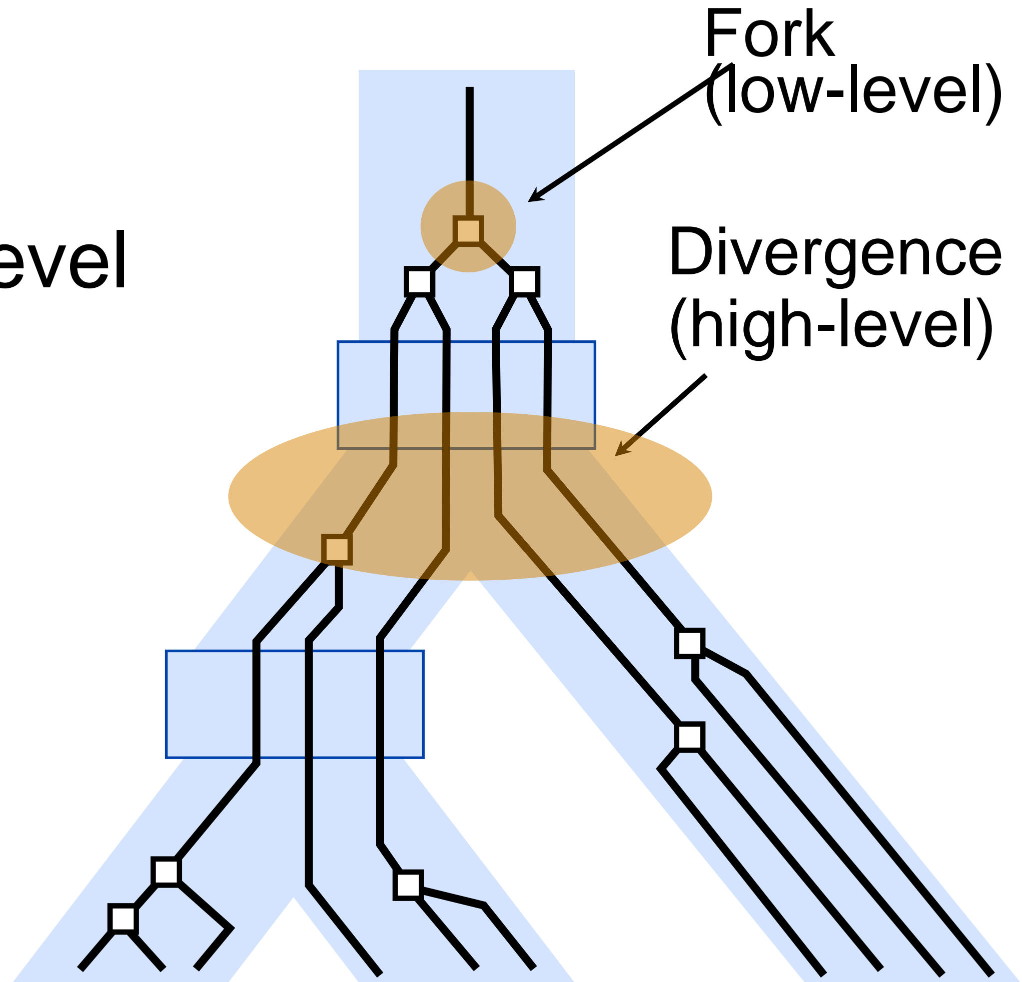
High-level Execution Paths

Alternative approach:
Select states at high-level branches



High-level Execution Paths

Alternative approach:
Select states at high-level branches

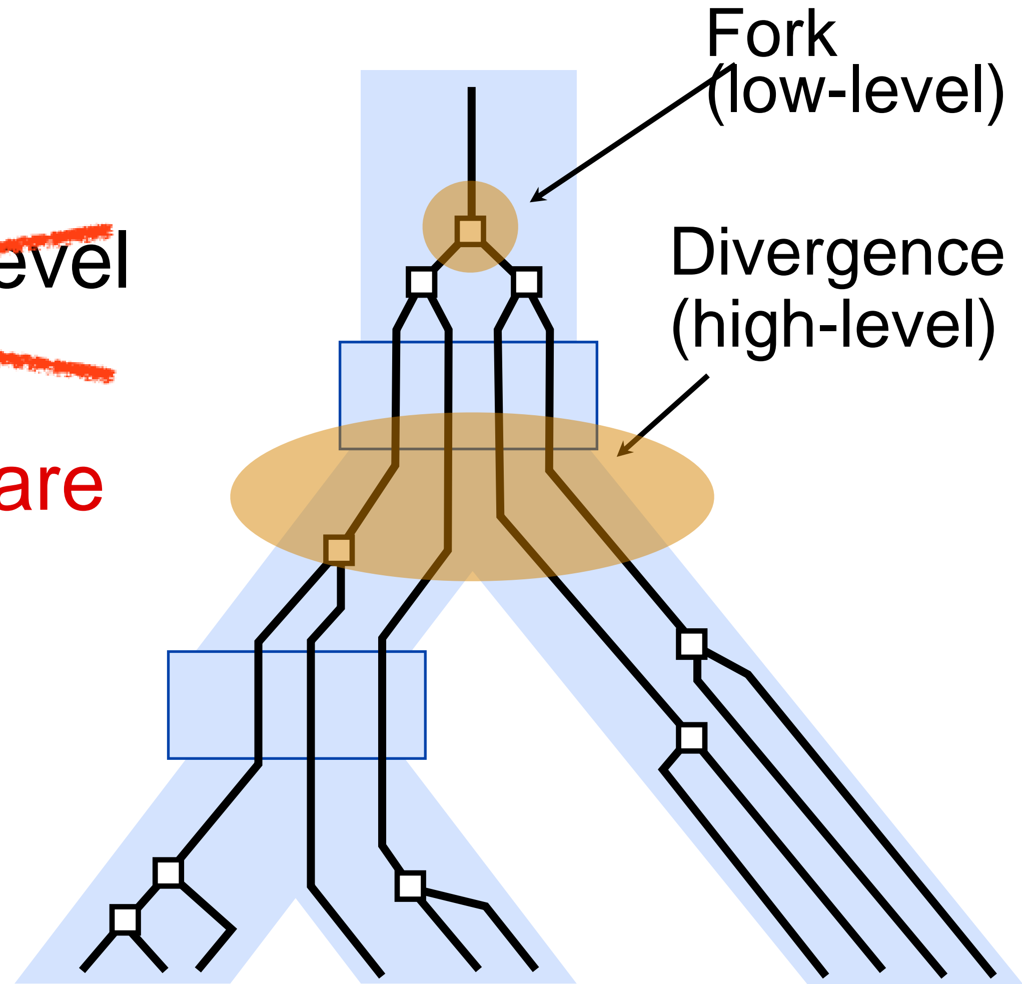


High-level Execution Paths

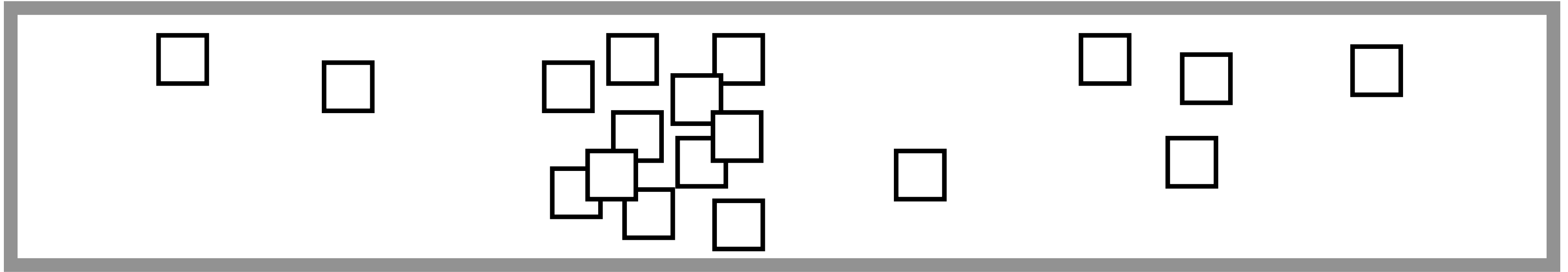
Alternative approach:

~~Select states at high-level branches~~

High-level fork points are unpredictable

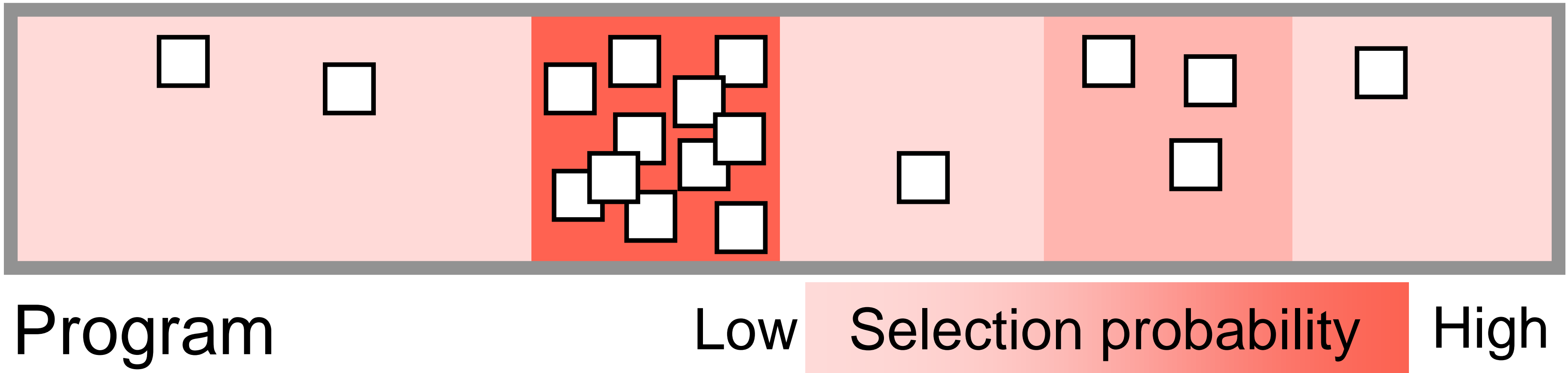


Reducing Path Explosion



Program

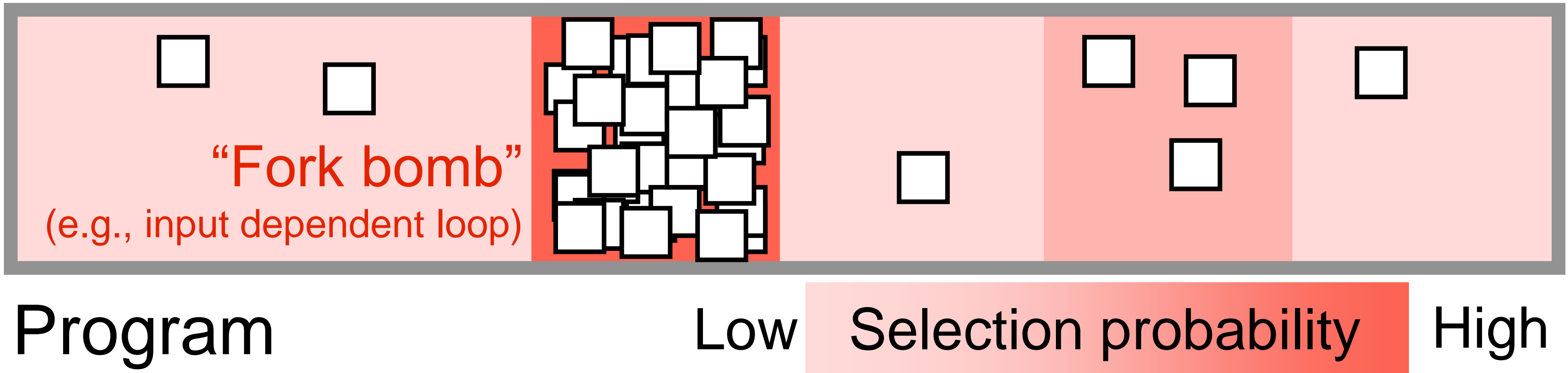
Reducing Path Explosion



Fork points clustered in hot spots

Reducing Path Explosion

Global DFS / BFS / randomized strategy

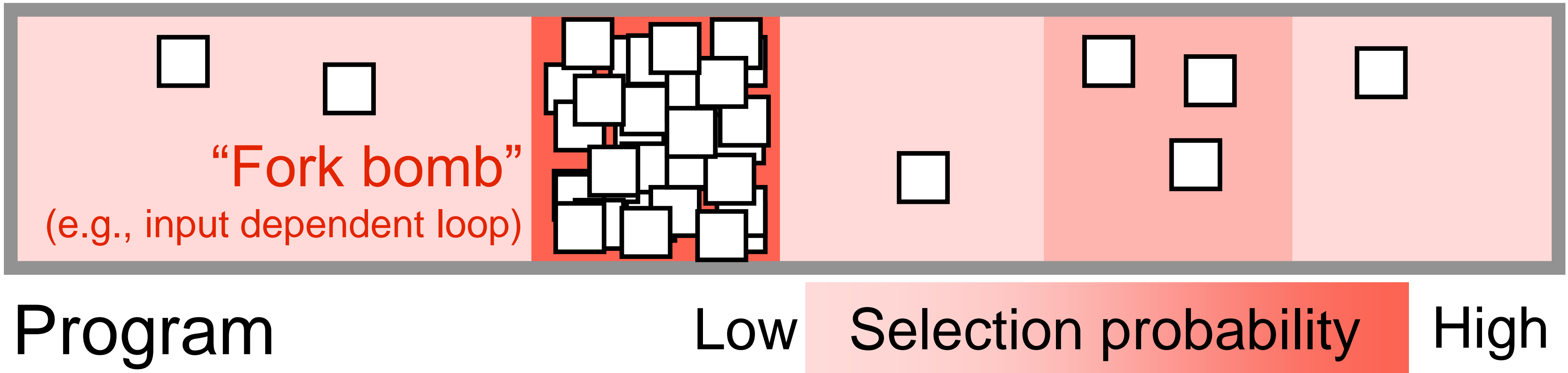


Fork points clustered in hot spots

Clusters grow bigger ⇒ Slower overall progress

Reducing Path Explosion

Global DFS / BFS / randomized strategy



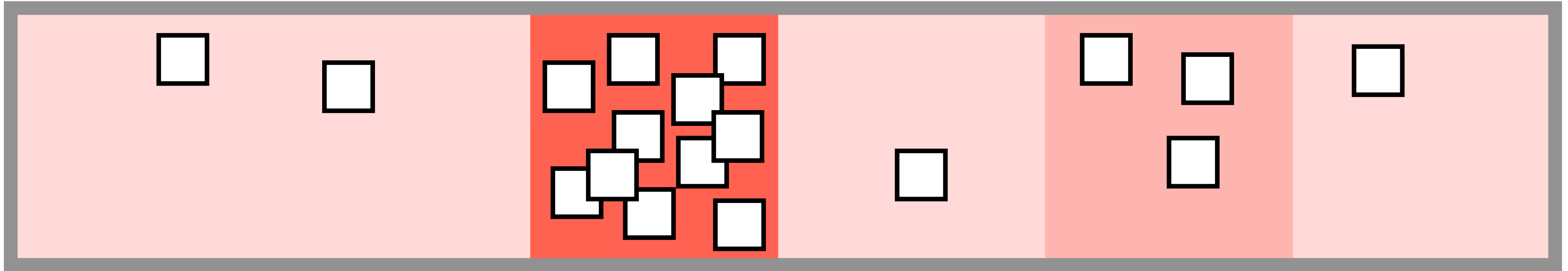
Fork points clustered in hot spots

Clusters grow bigger ⇒ Slower overall progress

Reduced state diversity

Class-Uniform Path Analysis

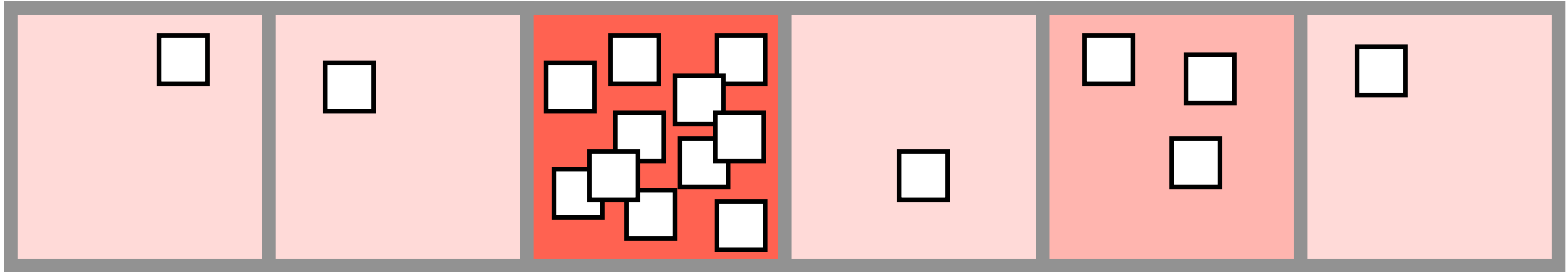
Idea: Partition the state space into groups



Program

Class-Uniform Path Analysis

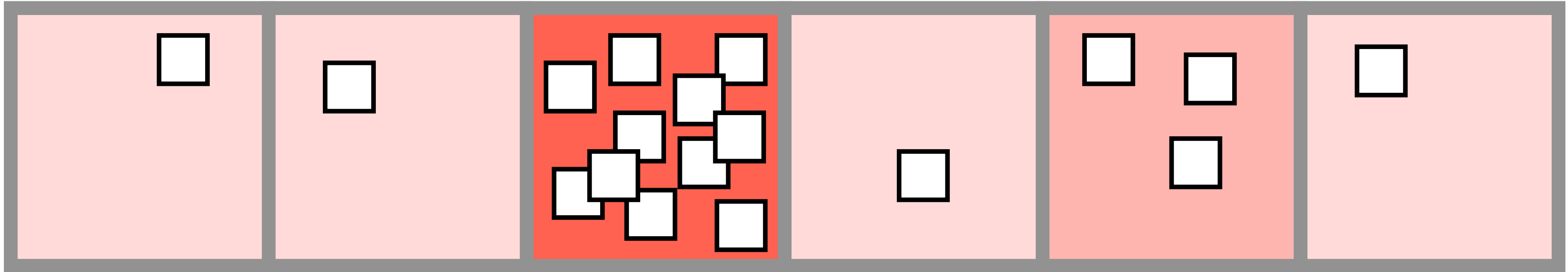
Idea: Partition the state space into groups



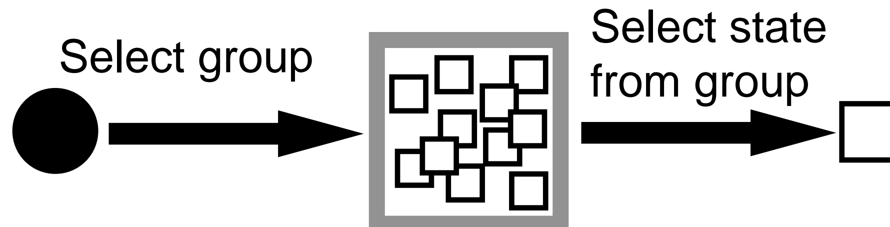
Program

Class-Uniform Path Analysis

Idea: Partition the state space into groups

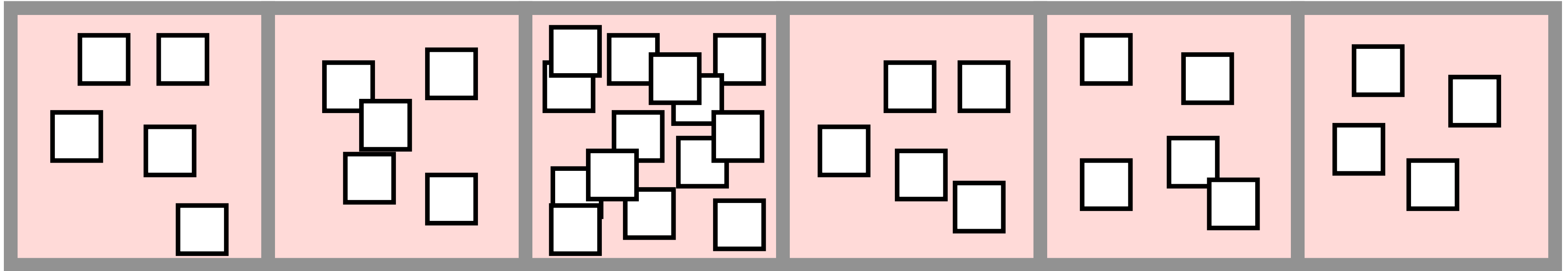


Program

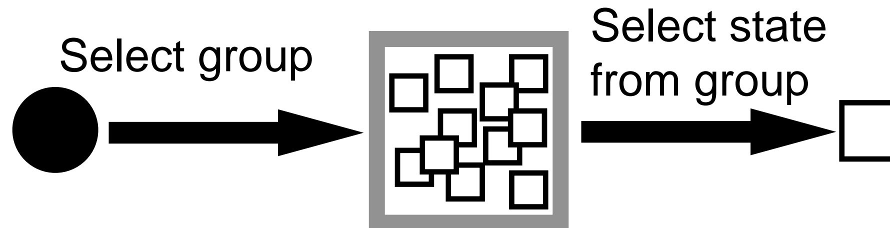


Class-Uniform Path Analysis

Idea: Partition the state space into groups



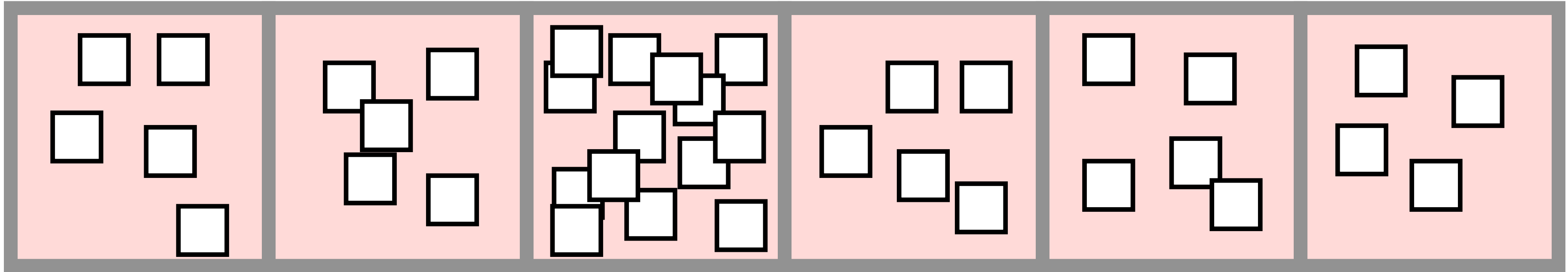
Program



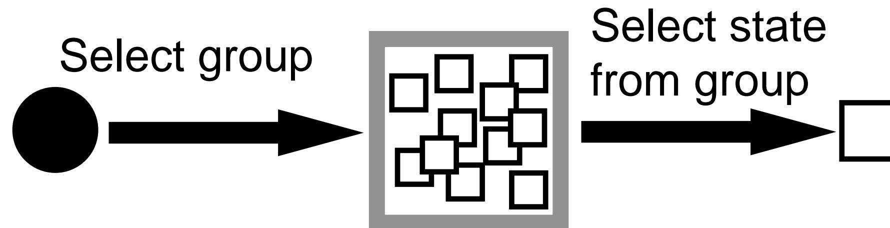
Faster progress across all groups

Class-Uniform Path Analysis

Idea: Partition the state space into groups



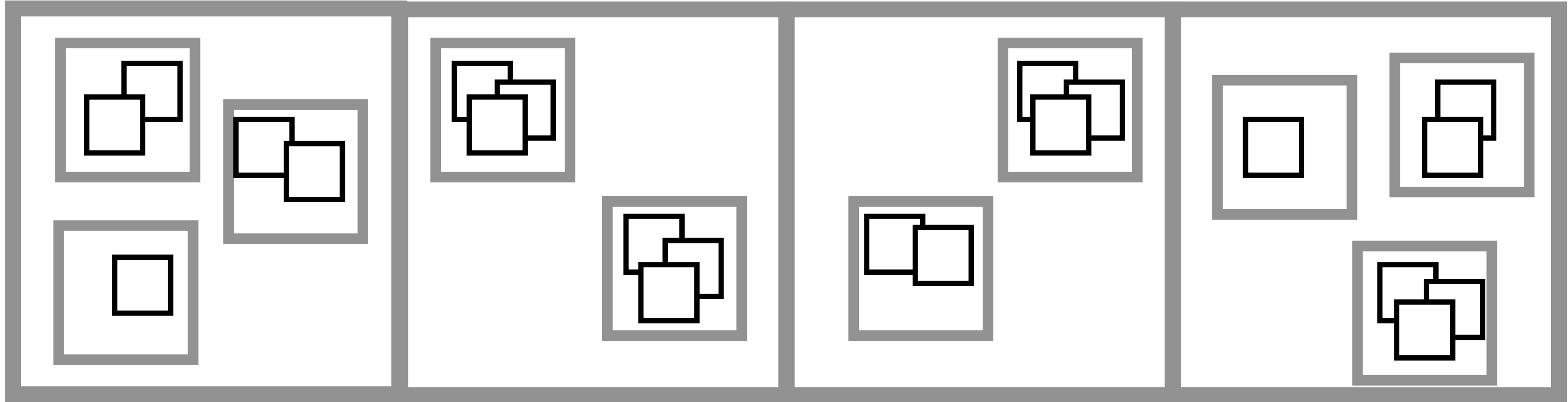
Program



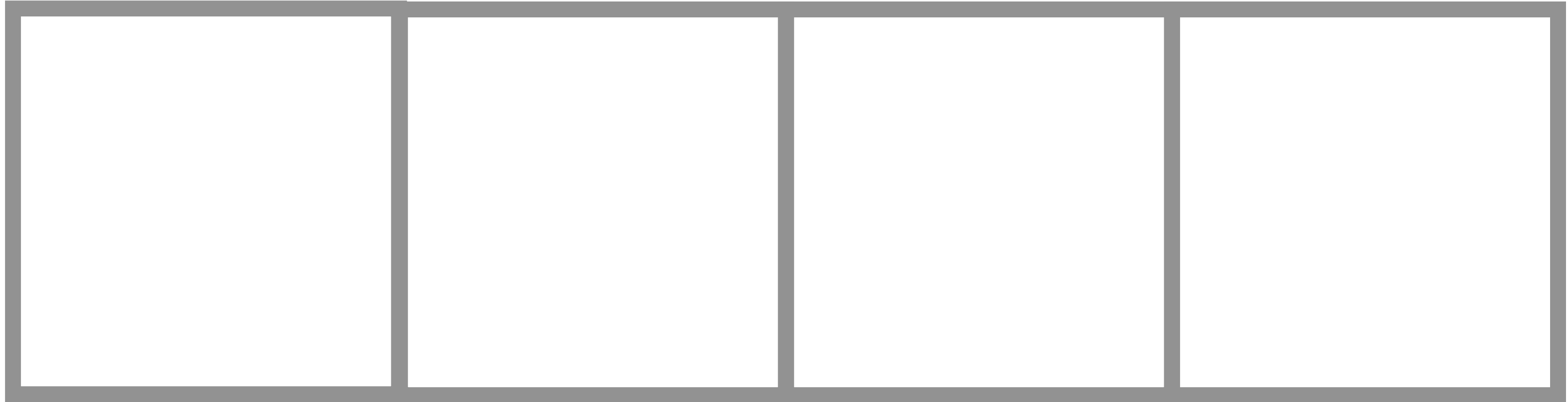
Faster progress across all groups

Increased state diversity

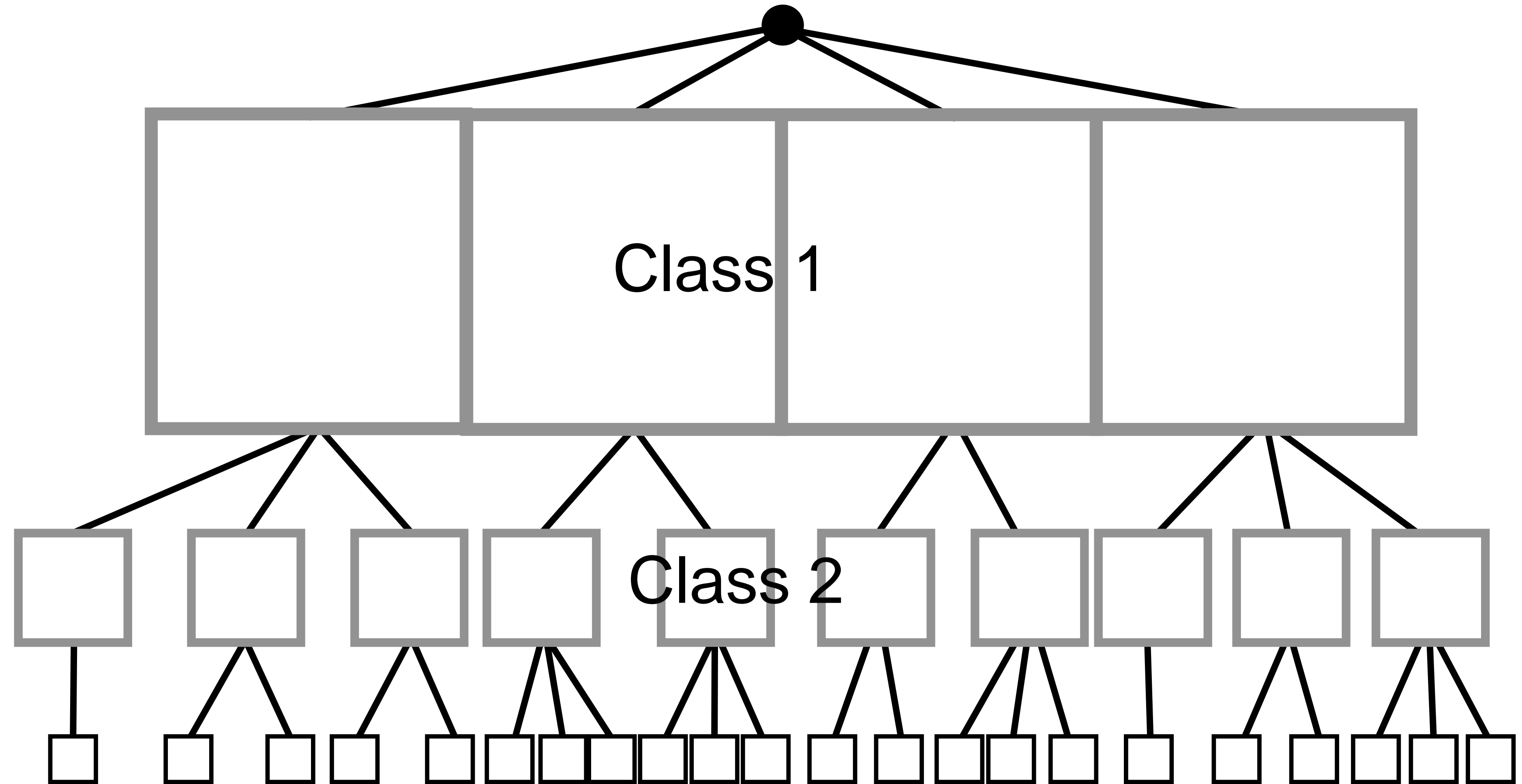
Class-Uniform Path Analysis



Class-Uniform Path Analysis

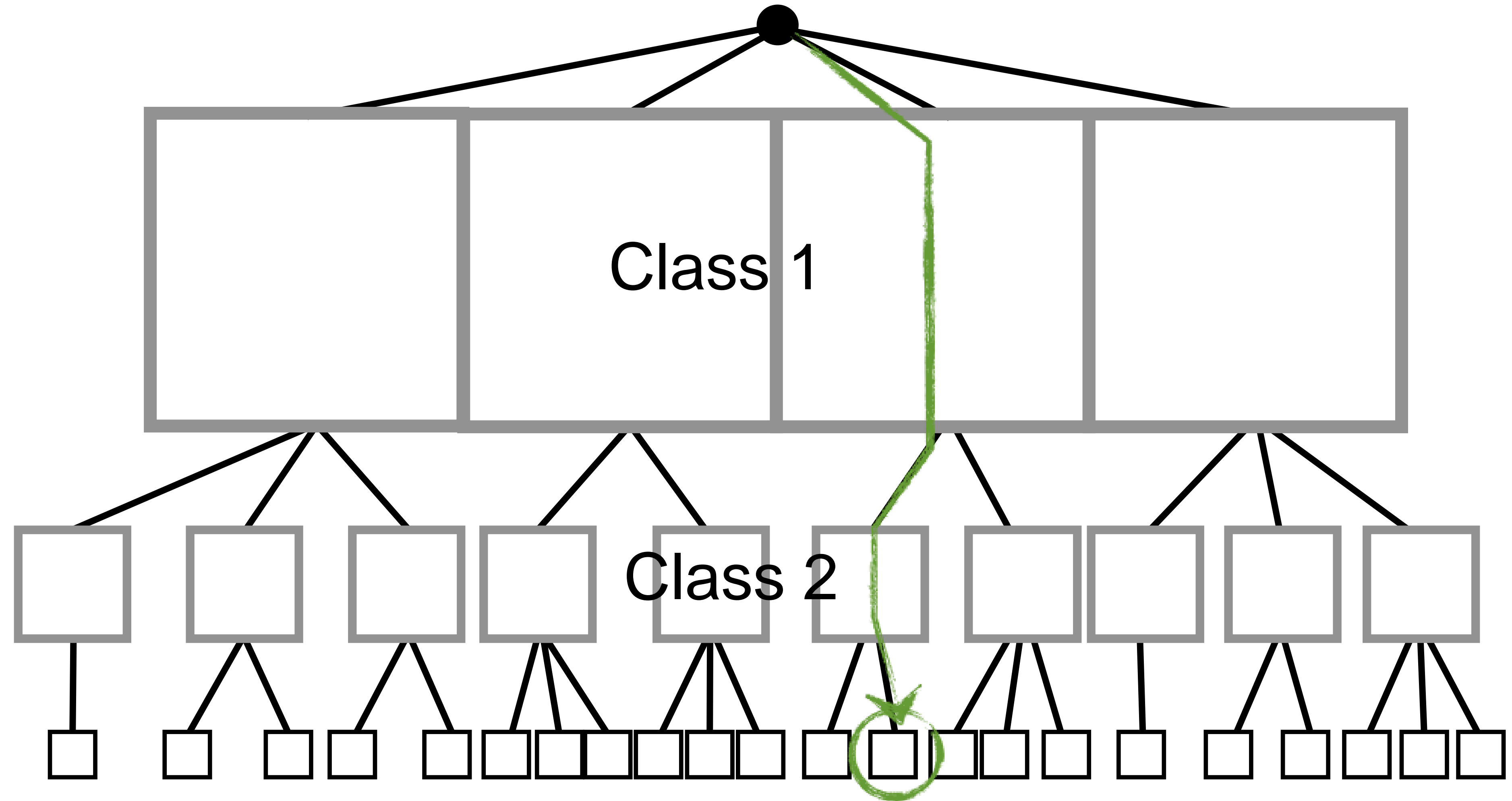


Class-Uniform Path Analysis



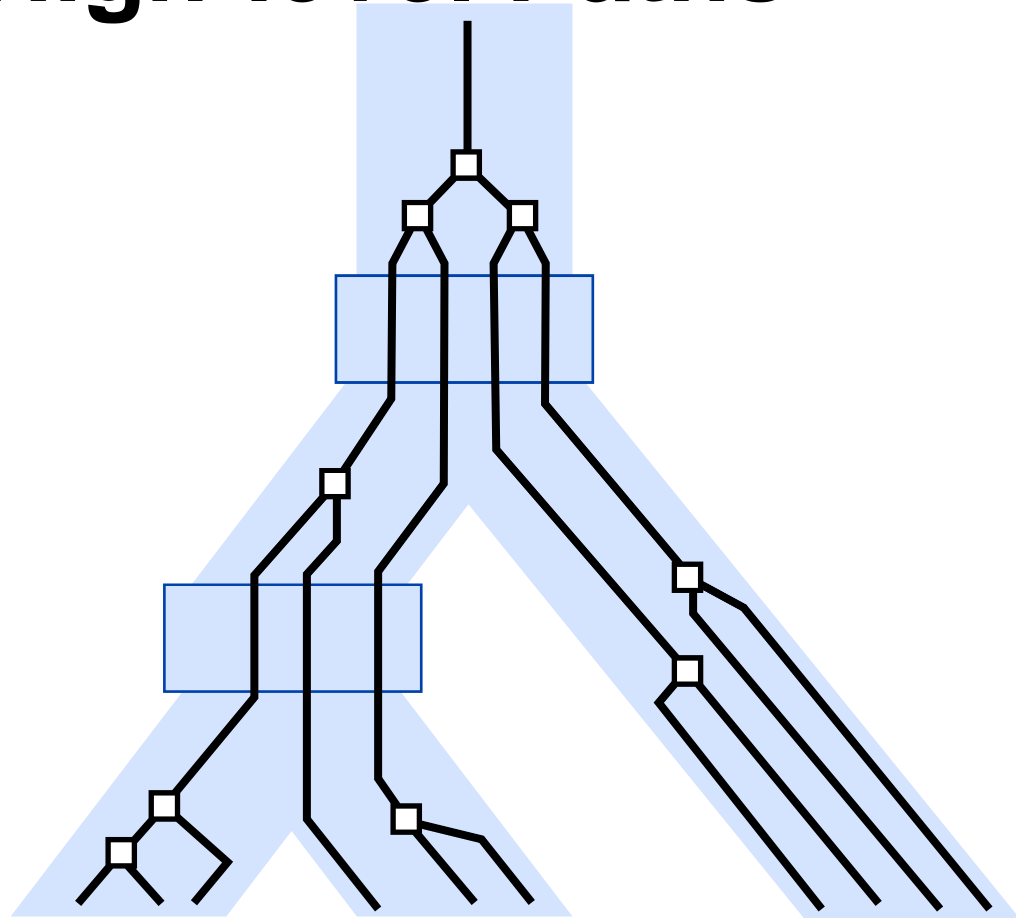
States arranged in a class hierarchy

Class-Uniform Path Analysis



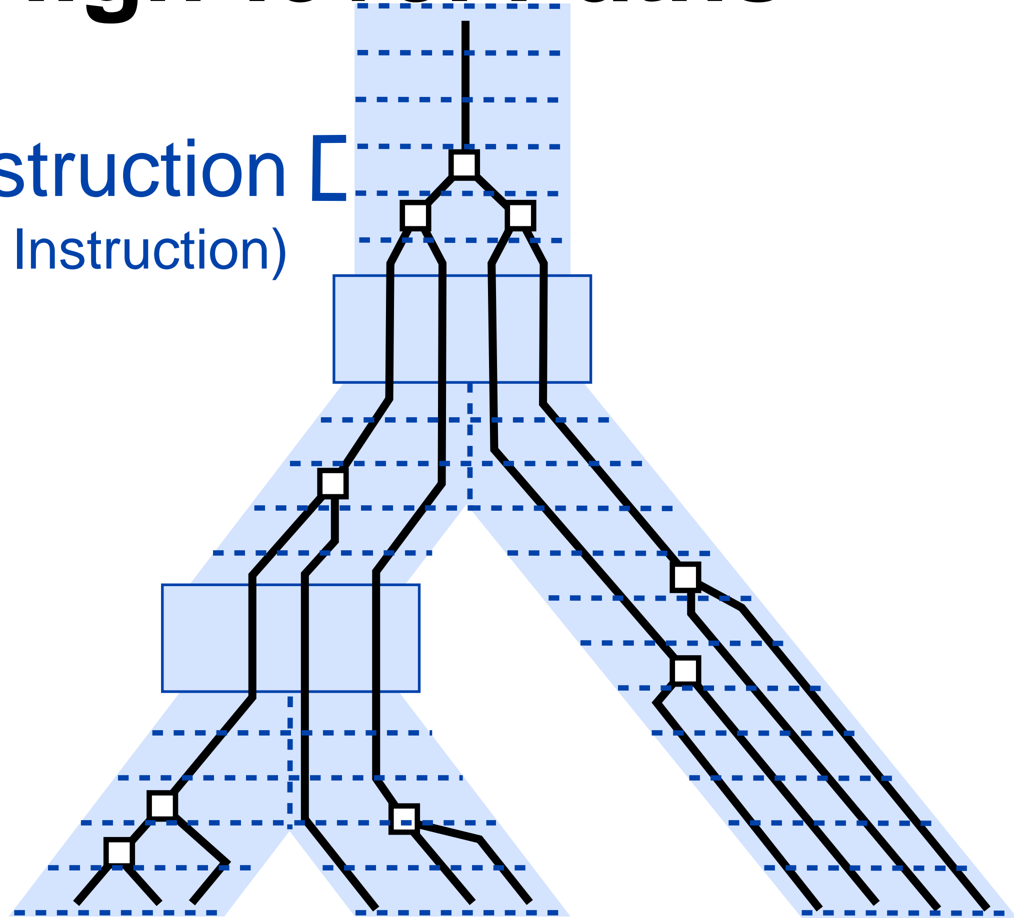
States arranged in a class hierarchy

Partitioning High-level Paths

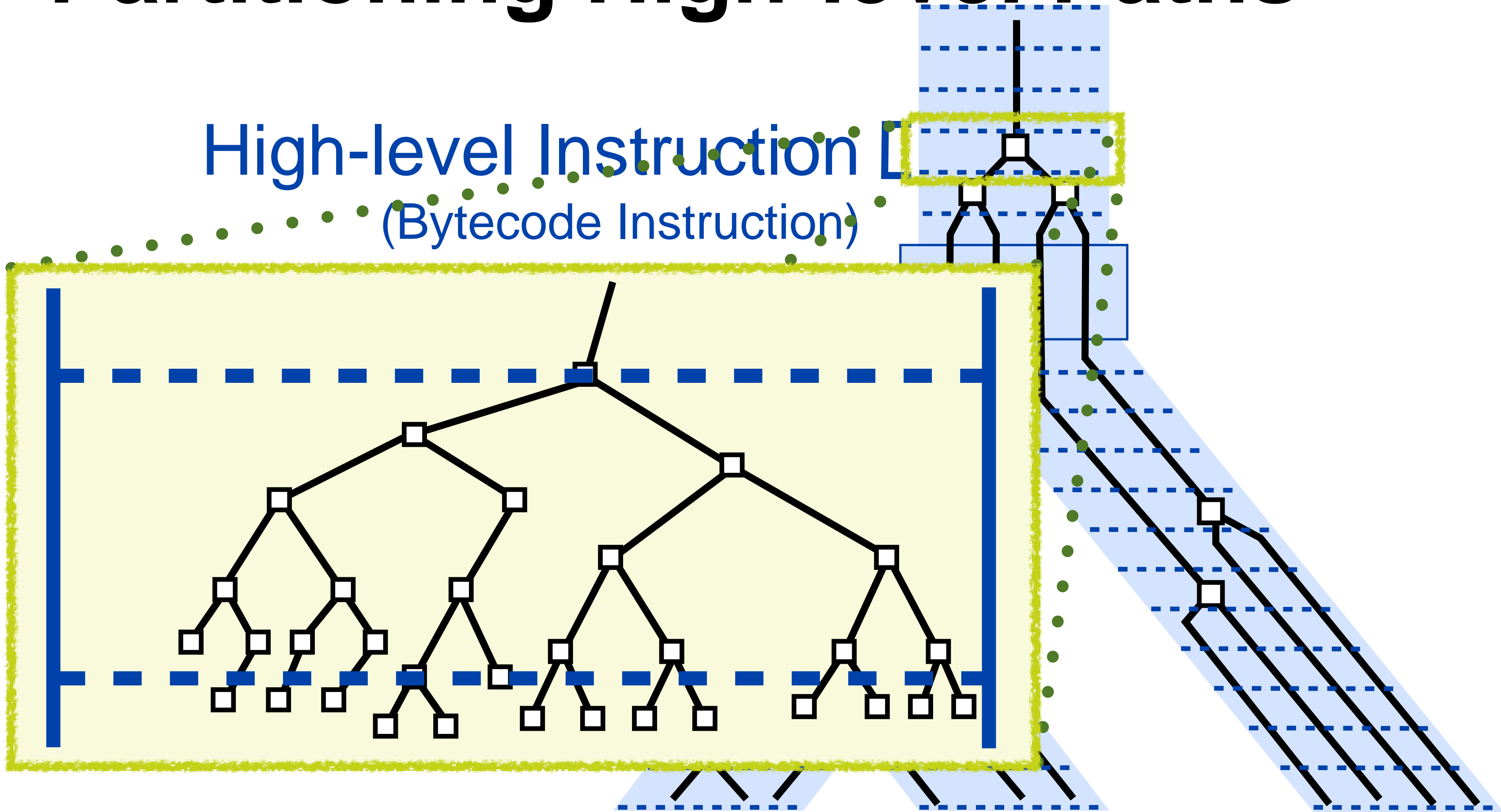


Partitioning High-level Paths

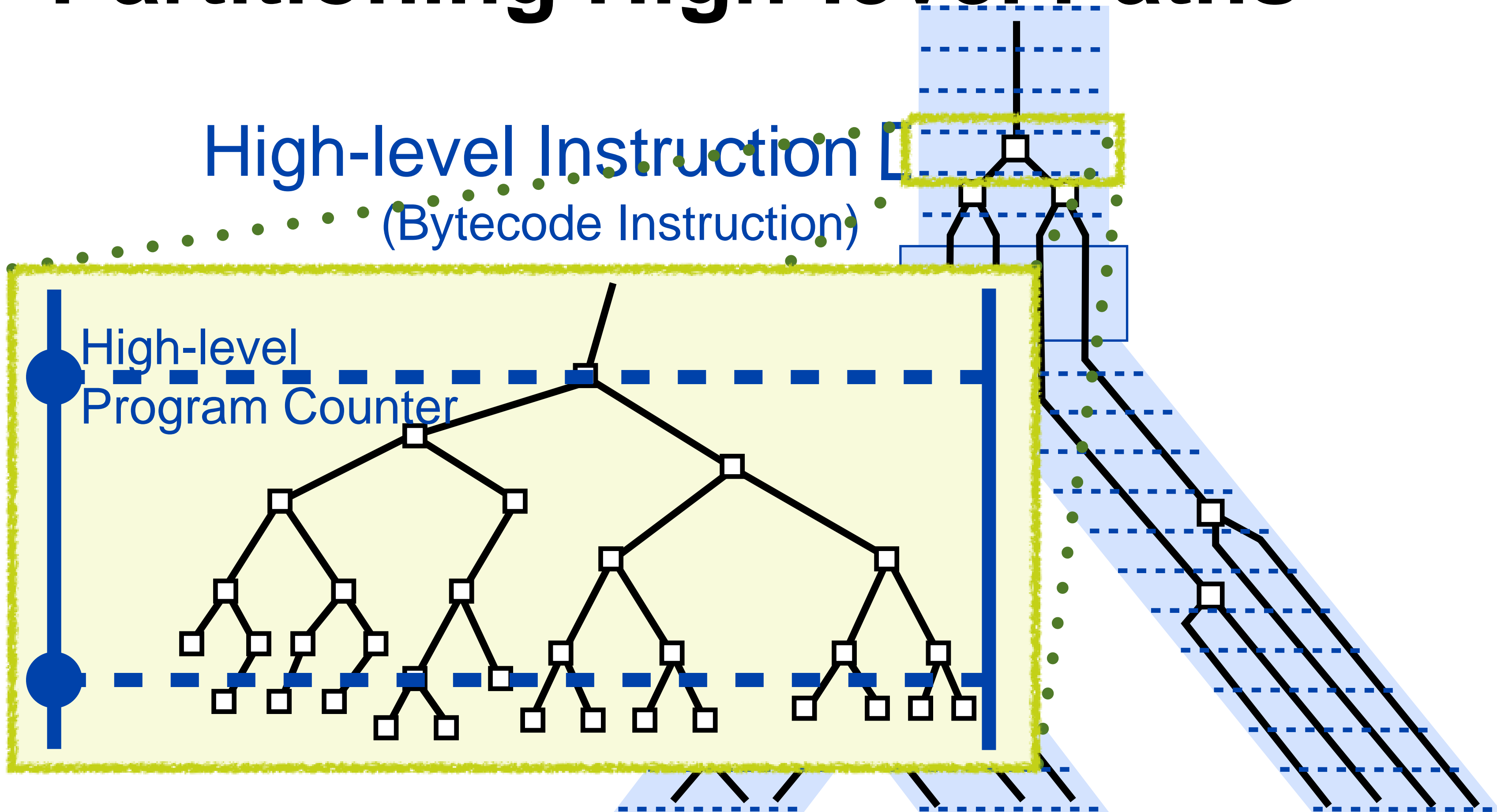
High-level Instruction
(Bytecode Instruction)



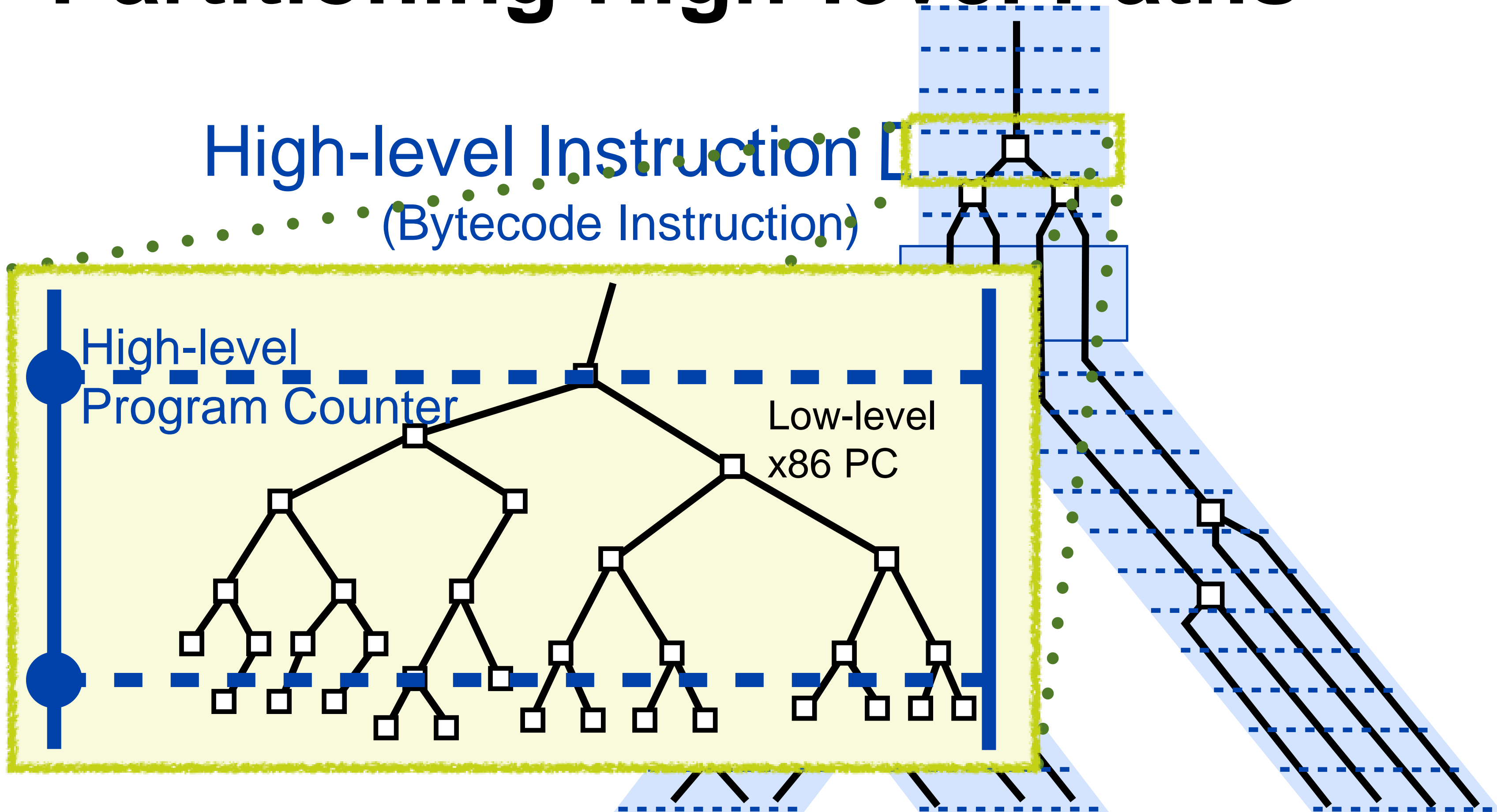
Partitioning High-level Paths



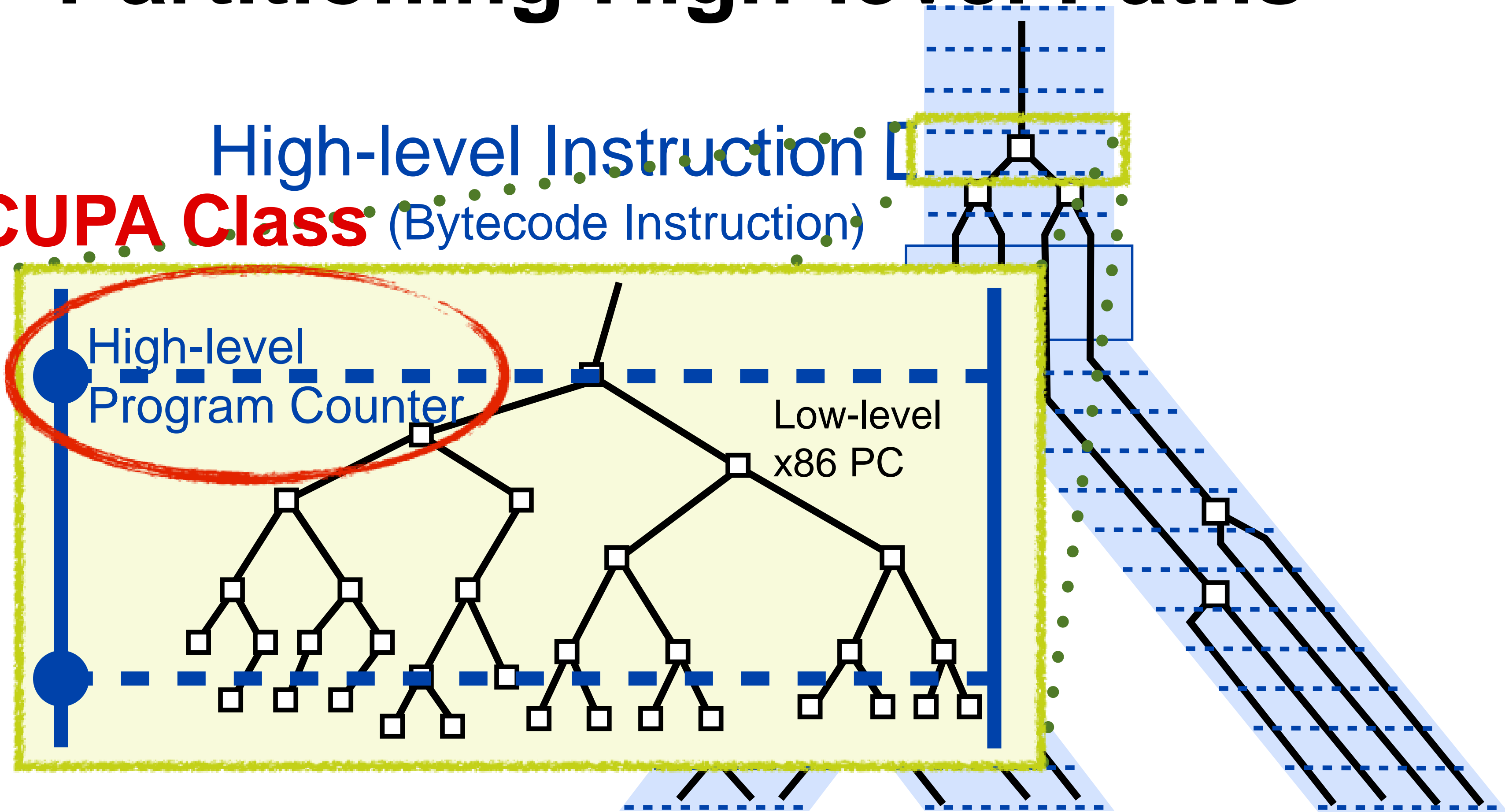
Partitioning High-level Paths



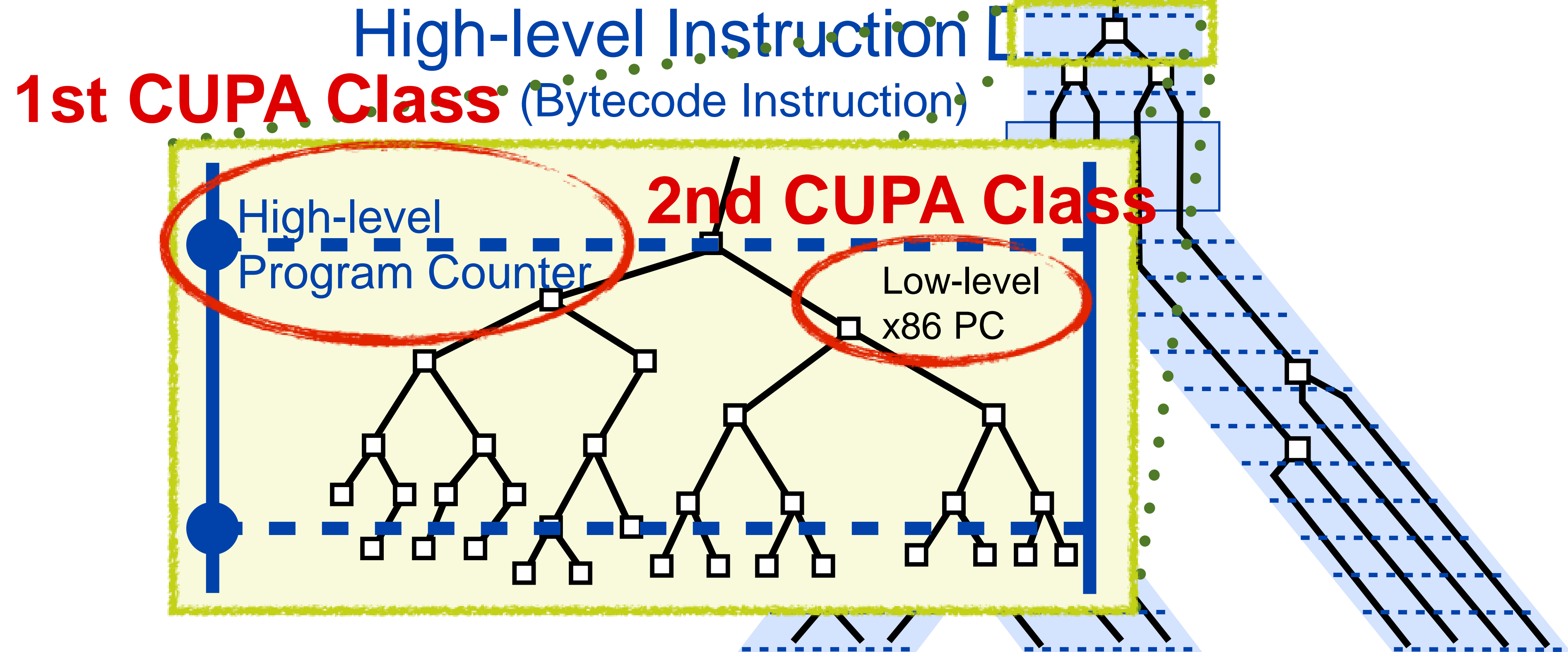
Partitioning High-level Paths



Partitioning High-level Paths



Partitioning High-level Paths



Reconstruct high-level execution tree

CUPA Classes

1. High-level PC
   - *Uniform HL instruction exploration*
   - *Obtained via instrumentation*
2. x86 PC
   - *Uniform native method exploration*
   - *Approximated as the PC of fork point*

Interpreter Loop Instrumentation

```
while (true) {
  fetch_instr(hlpc, &opcode, &params);
  switch (opcode) {
  case LOAD:
    ...
  case STORE:
    ...
  case CALL_FUNCTION:
    ...
  ...
  }
  hlpc++;
}
```

Interpreter Loop Instrumentation

```
while (true) {
  fetch_instr(hlpc, &opcode, &params);
  chef_log_hlpc(hlpc, opcode);
  switch (opcode) {
  case LOAD:
    ...
  case STORE:
    ...
  case CALL_FUNCTION:
    ...
  ...
  }
  hlpc++;
}
```

Reconstruct high-level execution tree and CFG

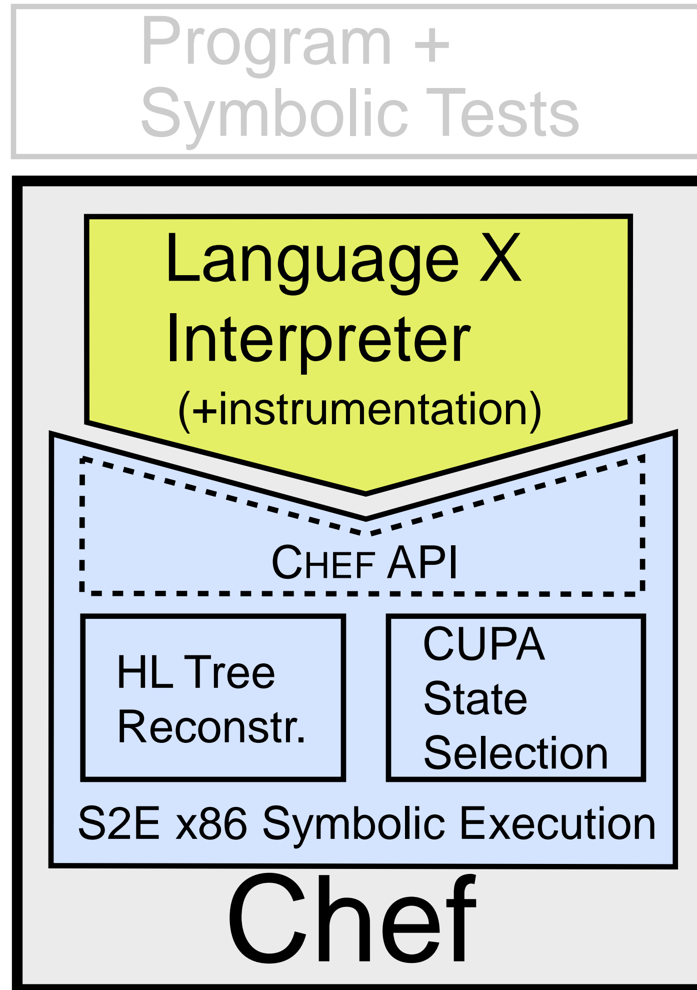
Interpreter Optimizations

```
static long
string_hash(PyStringObject *a)
{
#ifdef SYMBEX_HASHES
    return 0;
#else
    register Py_ssize_t len;
    register unsigned char *p;
    register long x;

    len = Py_SIZE(a);

    p = (char *) a->ob_sval;
    x = _Py_HashSecret.prefix;
    x ^= *p << 7;
    while (--len >= 0)
        x = (1000003*x) ^ *p++;
    x ^= Py_SIZE(a);
    x ^= _Py_HashSecret.suffix;
    if (x == -1)
        x = -2;
    return x;
#endif
}
```

Hash neutralization

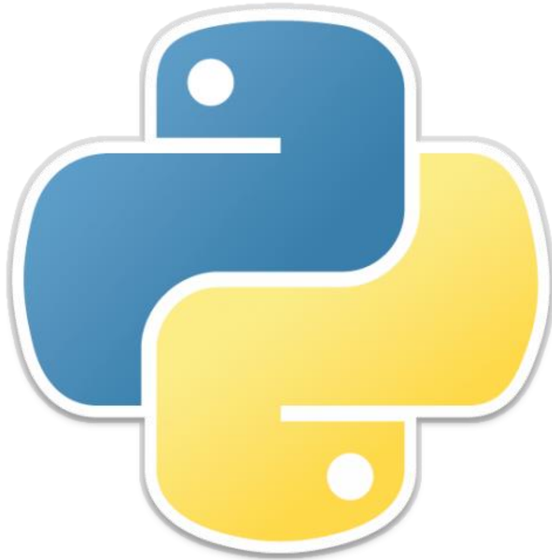
- Simple changes to interpreter source
- “Anti-optimizations” in linear performance...
- ... but exponential gains in symbolic mode

Chef Summary



Symbolic Execution
Engine for
Language X

Chef-Prototyped Engines



Python
5 person-days
321 LoC

Lua
3 person-days
277 LoC

Testing Python Packages

6 Popular Packages

- argparse
- ConfigParser
- HTMLParser
- simplejson
- unicodecsv
- xlrd

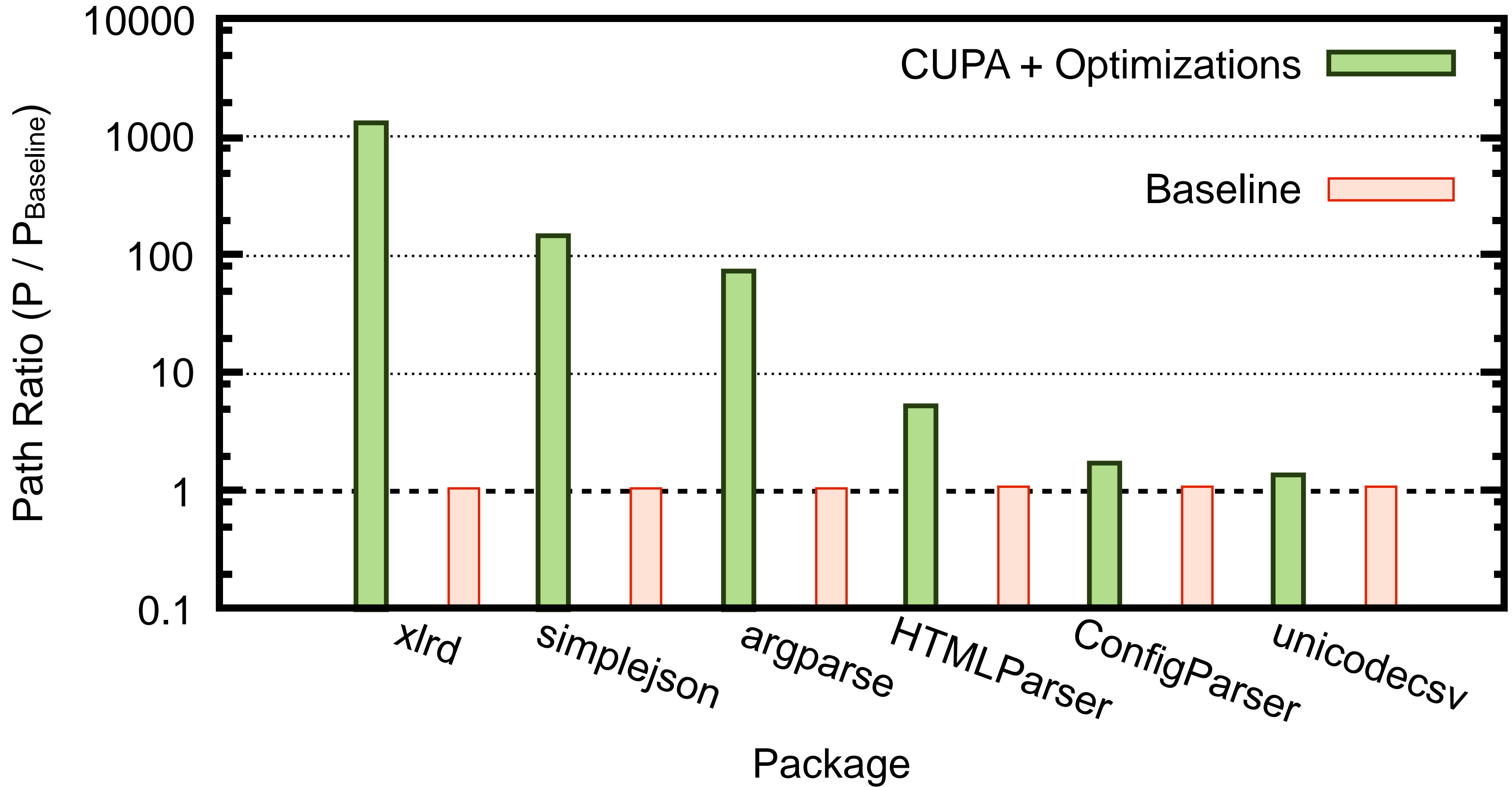
10.9K lines of Python code

30 min. / package

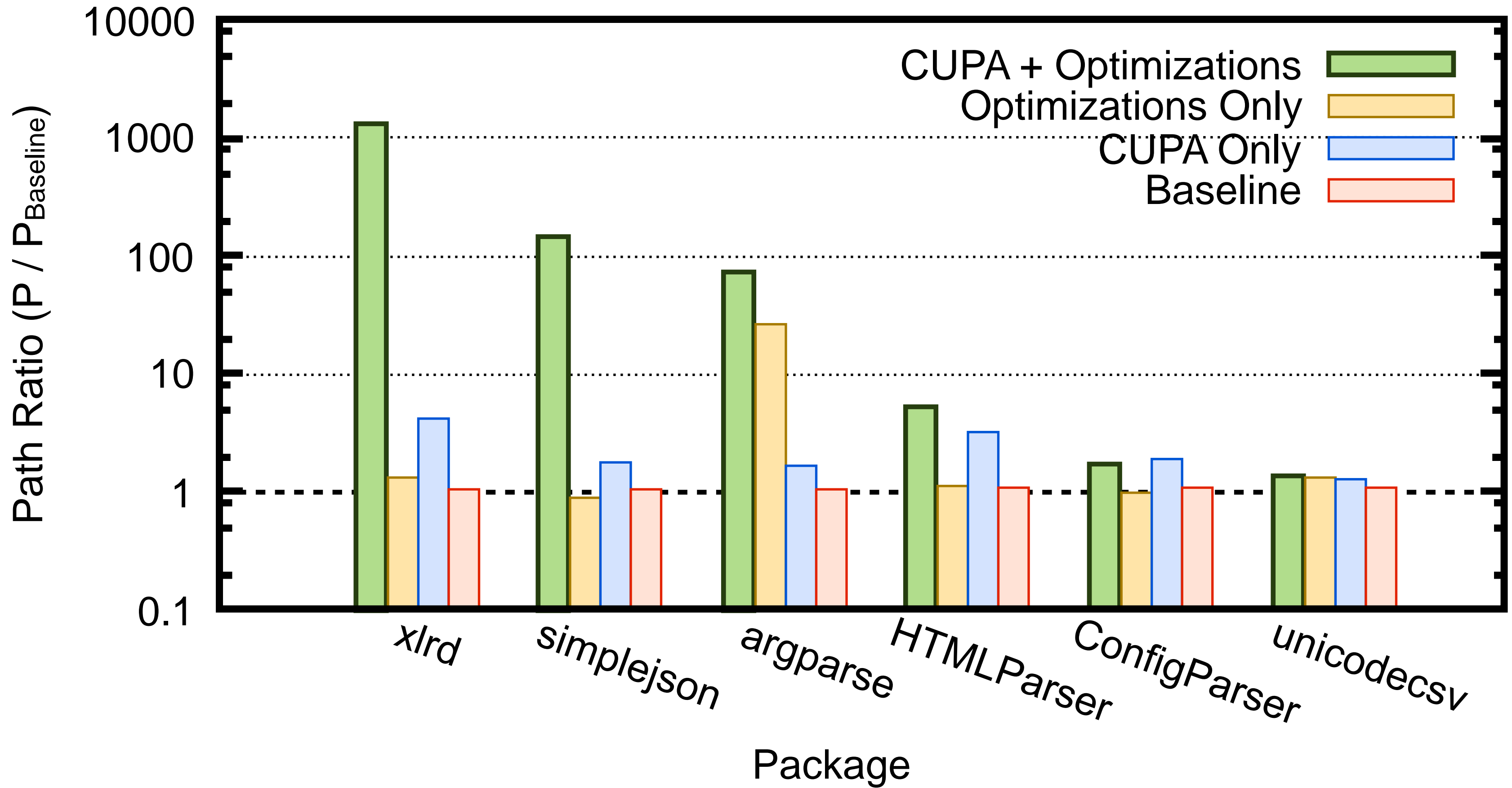
> 7,000 tests generated

4 undocumented exceptions found

Efficiency



Efficiency



Symbolic Execution

- Path-wise under-approximate program analysis
  - *Mixes concrete and symbolic reasoning*
- Automatic test case-generation
- Major-challenge: path explosion
- Solutions:
  - *State merging*
  - *Domain-specific optimizations*
  - *…*