

Dynamic Symbolic Execution

Testing, Quality Assurance, and Maintenance
Fall 2023

Prof. Arie Gurfinkel

based on slides by Prof. Johannes Kinder and others

Problems with Scaling Symbolic Execution

Code that is hard to analyze

Path explosion

- Complex control flow
- Loops
- Procedures

Environment (what are the inputs to the program under test?)

- pointers, data structures, ...
- files, data bases, ...
- threads, thread schedules, ...
- sockets, ...

Some code is hard to analyze symbolically

```
int obscure(int x, int y) {  
    if (x == complex(y))  
        error();  
    return 0;  
}
```

It might be very hard to statically generate values for **x** and **y** such that **x == complex(y)**

Sources of complexity:

- Virtual functions (function pointers)
- Cryptographic functions
- Non-linear integer or floating point arithmetic
- System calls
- ...

Directed Automated Random Testing

DART, aka

- concolic testing (**con**crete + **sy**mbolic)
- Dynamic Symbolic Execution (DSE)

Use concrete execution when symbolic execution is hard

- use concrete execution to guide symbolic execution to useful parts of the code

Run 1:

- Use random inputs: $x=33$, $y=42$
- `complex(42)` returns `567`, else branch taken
- execute same path symbolically
- `x == complex(y)` is too hard, so simplify into `x == 567`

Run 2

- Use inputs from symex in Run1: $x=567$, $y=42$
- then-branch is taken, all branches covered, error found

```
int obscure(int x, int y) {  
    if (x == complex(y))  
        error();  
    return 0;  
}
```



Flavors of Symbolic Execution

Static (Classical) symbolic execution

- Simulate execution on program source code
- Computes strongest post-conditions from entry point

Dynamic symbolic execution (DSE)

- Run / interpret the program with concrete state
- Symbolic state computed in parallel with concrete execution (“concolic”)
- Solver generates new concrete inputs to increase coverage

Two main flavors of DSE:

- **EXE-style** [Cadar et al. ‘06] vs. **DART-style** [Godefroid et al. ‘05]

Many successful tools

- **EXE**: KLEE (Imperial), SPF (NASA), Cloud9, S2E (EPFL)
- **DART**: SAGE, PEX (Microsoft), CUTE (UIUC), CREST (Berkeley)

EXE Algorithm: Intuition

Run the program with concrete inputs

- e.g., use fuzzing or some seed to get initial inputs

In parallel to concrete execution, run symbolic execution

- symbolic execution follows concrete execution
- keeps track of what values are inputs (symbolic) or not inputs (concrete)
- computes path condition in terms of inputs

At every branch point

- concrete execution takes one branch (say branch1)
- symbolic execution updates current input to force execution into a different branch (say branch2)
 - this is done by updating current path condition to go to branch2
 - creates NEW concrete state that follows into branch2
- both states (for branch1 and branch2) are explored

EXE Algorithm: Details

Program state is a tuple (`ConcreteState`, `SymbolicState`)

Initially all input variables are symbolic

At each execution step

- update the concrete state by executing a program instruction concretely
- update symbolic state executing symbolically
- if the last instruction was a branch
 - if path condition now is consistent with negation of the branch condition, then
 - fork current execution state into two
 - compute NEW concrete state to match the NEW path condition

EXE

$pc = \text{true}$

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

```
1  int proc(int x) {  
2  
3    int r = 0  
4  
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  
4  
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

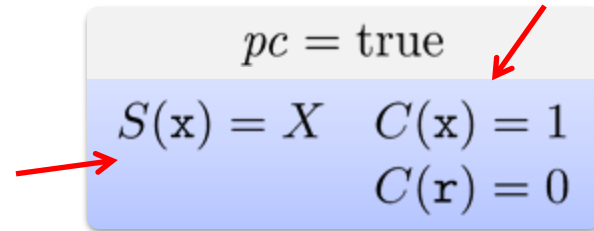


$pc = \text{true}$	
$S(x) = X$	$C(x) = 1$
	$C(r) = 0$

EXE

```
1  int proc(int x) {  
2  
3    int r = 0  
4  
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



EXE

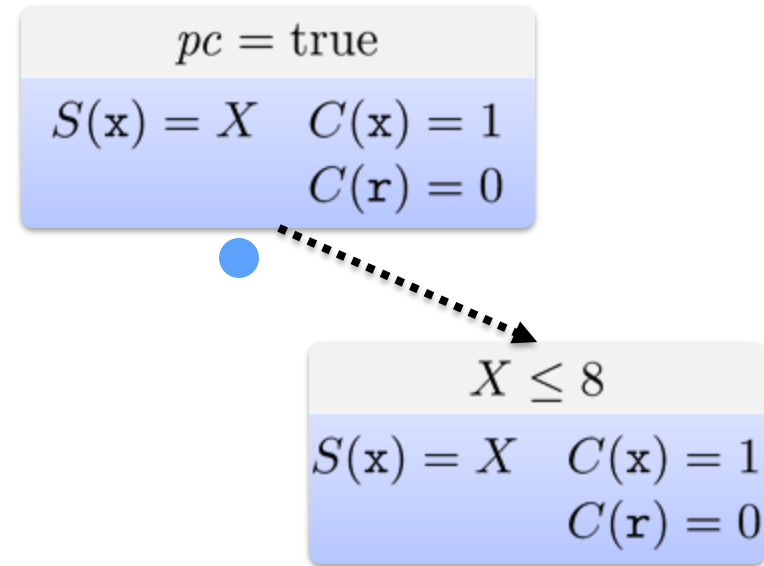
```
1 int proc(int x) {  
2  
3     int r = 0  
4  
5     if (x > 8 ●) {  
6         r = x - 7  
7     }  
8  
9     if (x < 5 ●) {  
10        r = x - 2  
11    }  
12  
13    return r  
    }
```

$pc = \text{true}$

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

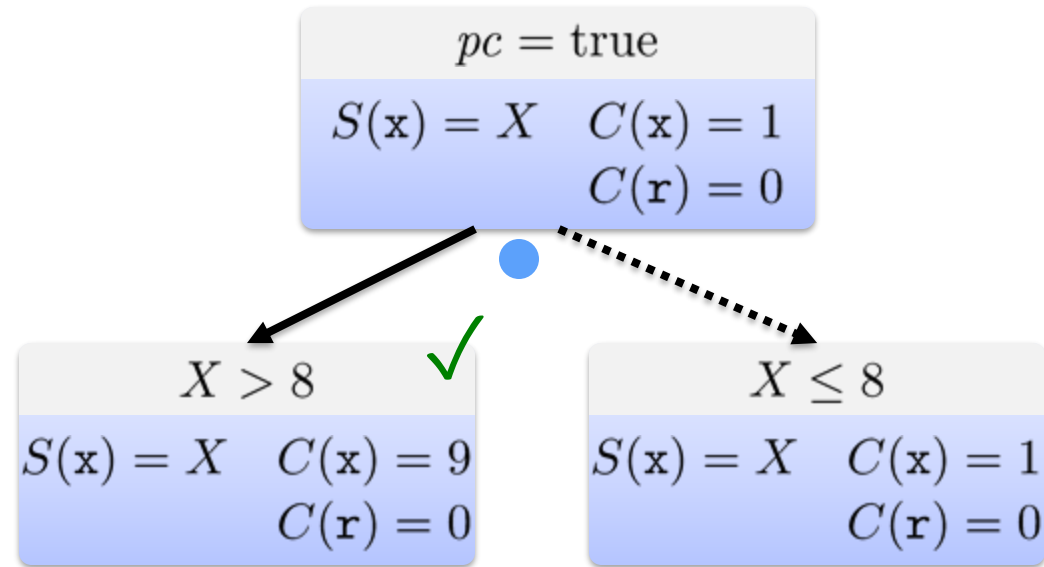
EXE

```
1 int proc(int x) {  
2  
3   int r = 0  
4  
5   if (x > 8 ●) {  
6     r = x - 7  
7   }  
8  
9   if (x < 5 ●) {  
10    r = x - 2  
11  }  
12  
13  return r  
}
```



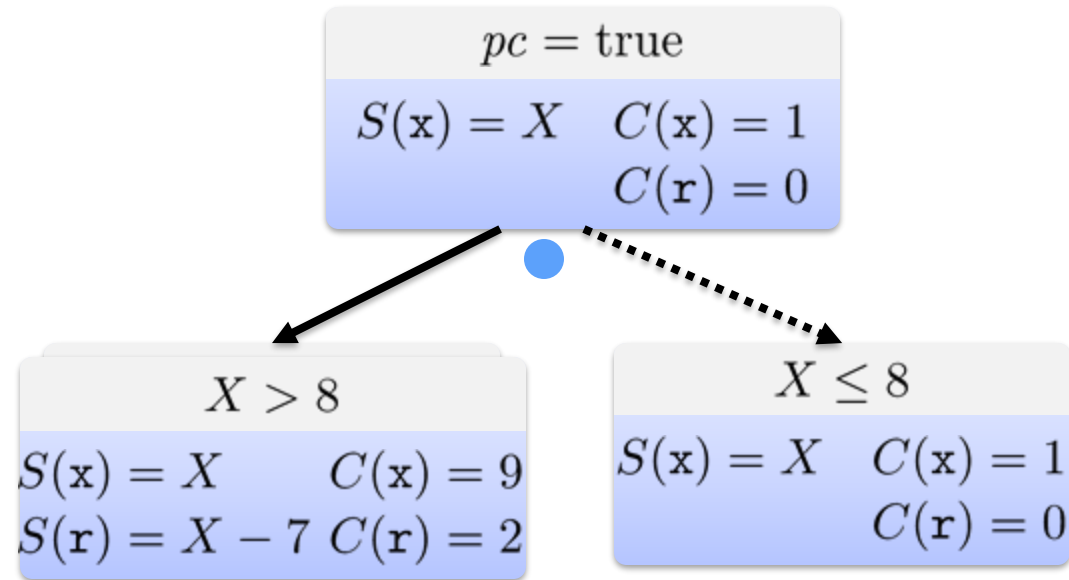
EXE

```
1 int proc(int x) {  
2  
3   int r = 0  
4  
5   if (x > 8 ●) {  
6     r = x - 7  
7   }  
8  
9   if (x < 5 ●) {  
10    r = x - 2  
11  }  
12  
13  return r  
}
```



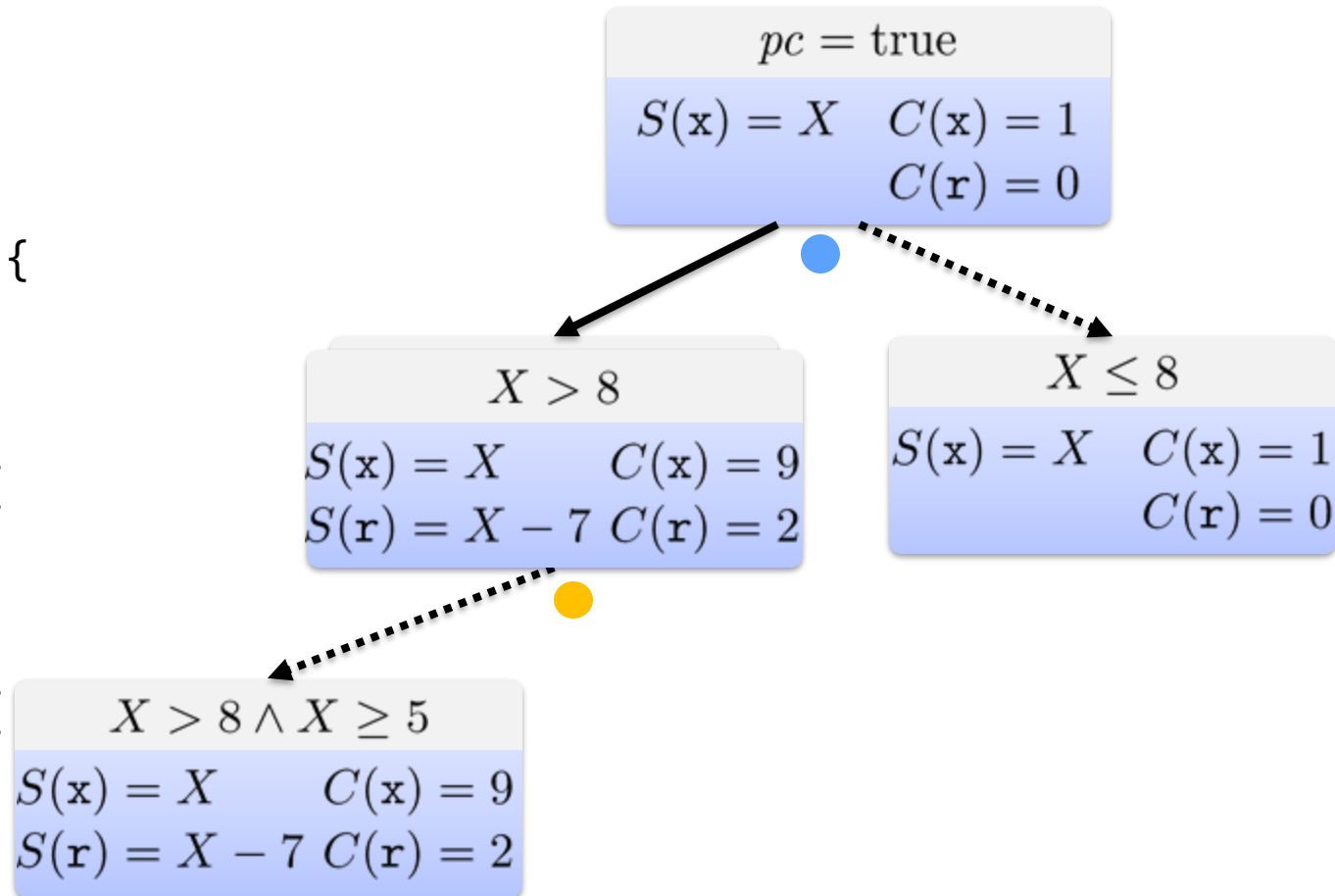
EXE

```
1 int proc(int x) {  
2  
3   int r = 0  
4  
5   if (x > 8 ●) {  
6     r = x - 7  
7   }  
8  
9   if (x < 5 ●) {  
10    r = x - 2  
11  }  
12  
13  return r  
}
```



EXE

```
1 int proc(int x) {  
2  
3   int r = 0  
4  
5   if (x > 8 ●) {  
6     r = x - 7  
7   }  
8  
9   if (x < 5 ●) {  
10    r = x - 2  
11  }  
12  
13  return r  
}
```

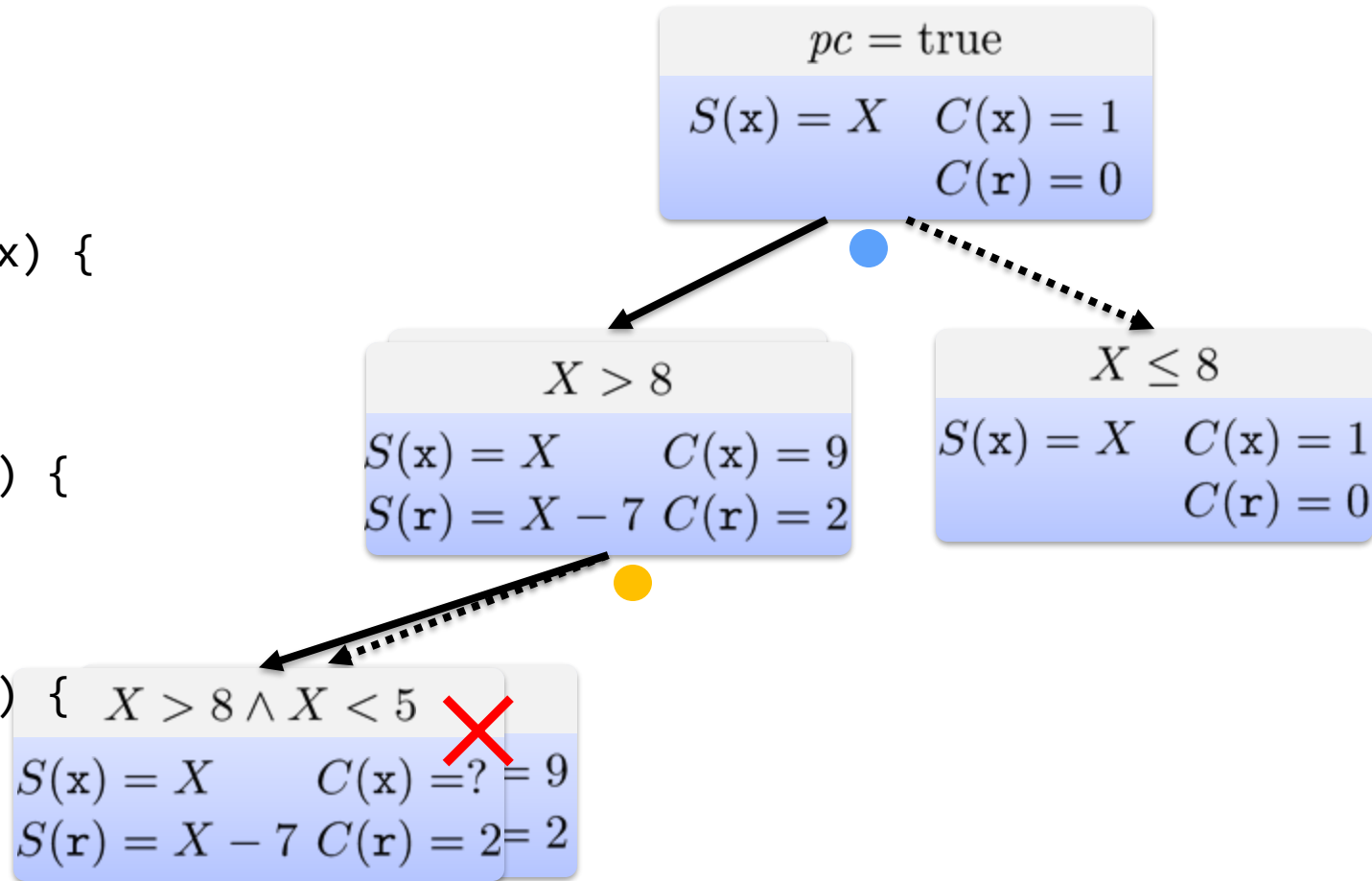


EXE

```

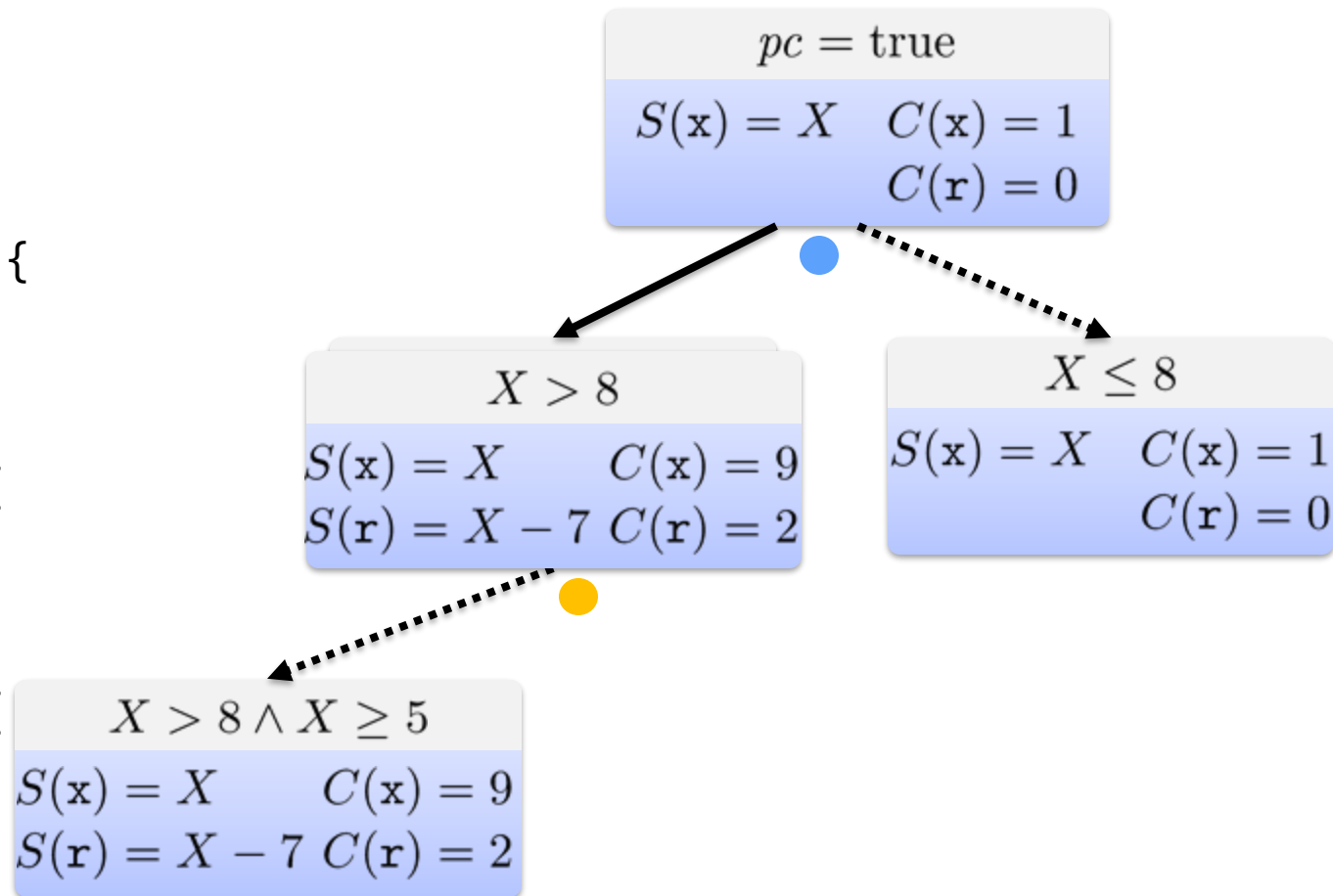
1 int proc(int x) {
2
3   int r = 0
4
5   if (x > 8 ●) {
6     r = x - 7
7   }
8
9   if (x < 5 ●) {
10    r = x - 2
11  }
12
13  return r
14 }

```



EXE

```
1 int proc(int x) {  
2  
3   int r = 0  
4  
5   if (x > 8 ●) {  
6     r = x - 7  
7   }  
8  
9   if (x < 5 ●) {  
10    r = x - 2  
11  }  
12  
13  return r  
}
```

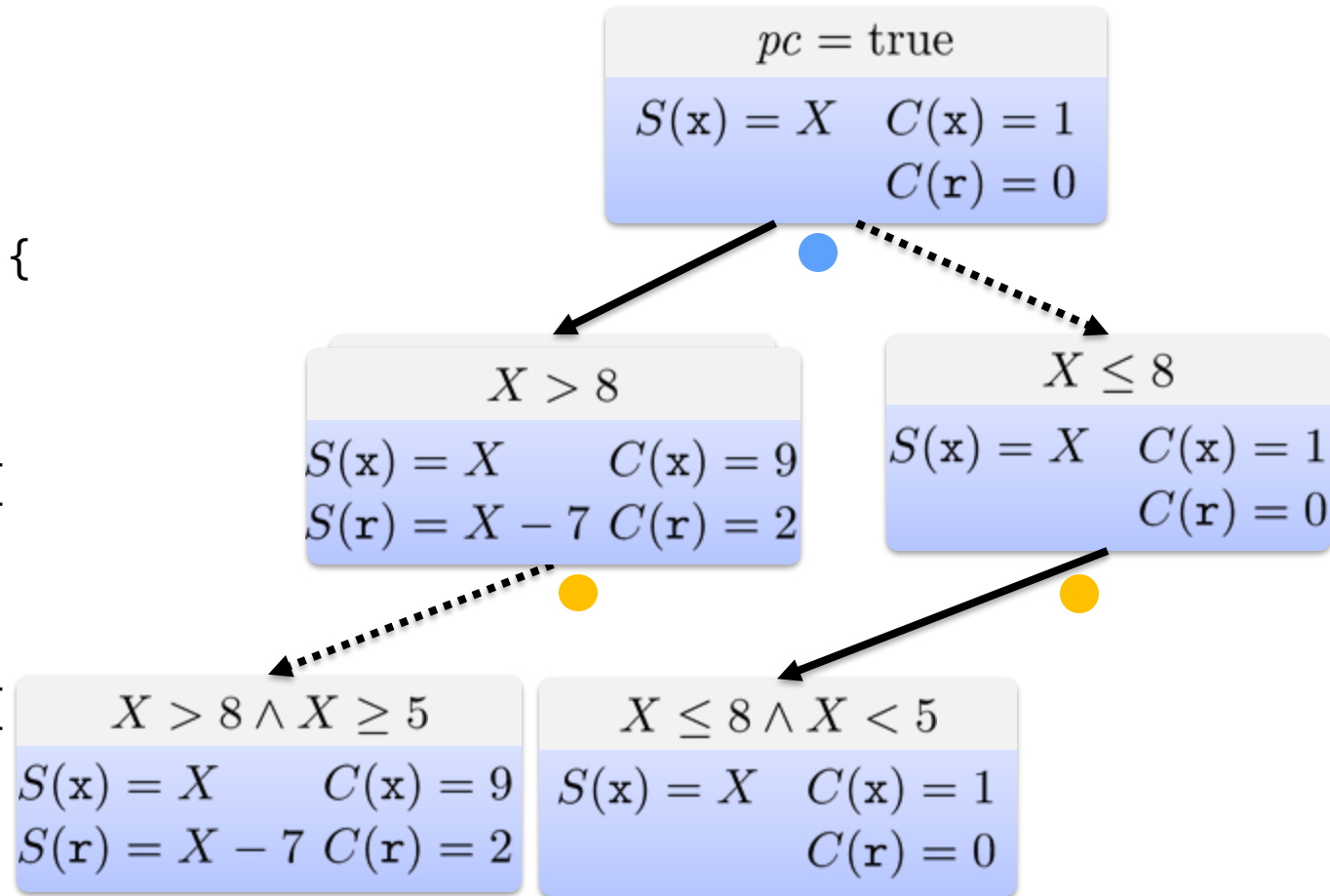


EXE

```

1 int proc(int x) {
2
3   int r = 0
4
5   if (x > 8 ●) {
6     r = x - 7
7   }
8
9   if (x < 5 ●) {
10    r = x - 2
11  }
12
13  return r
14 }

```

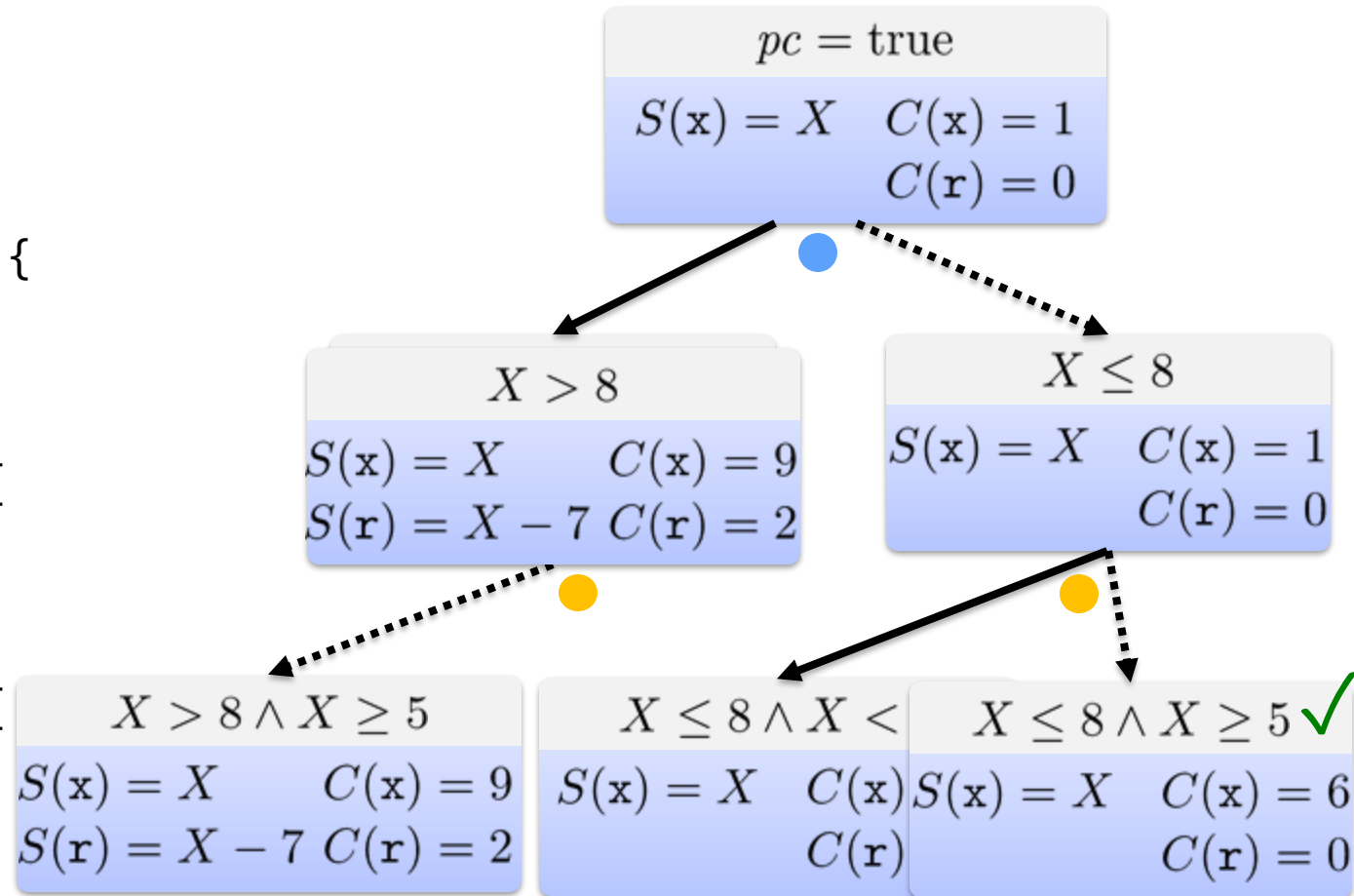


EXE

```

1 int proc(int x) {
2
3   int r = 0
4
5   if (x > 8 ●) {
6     r = x - 7
7   }
8
9   if (x < 5 ●) {
10    r = x - 2
11  }
12
13  return r
14 }

```

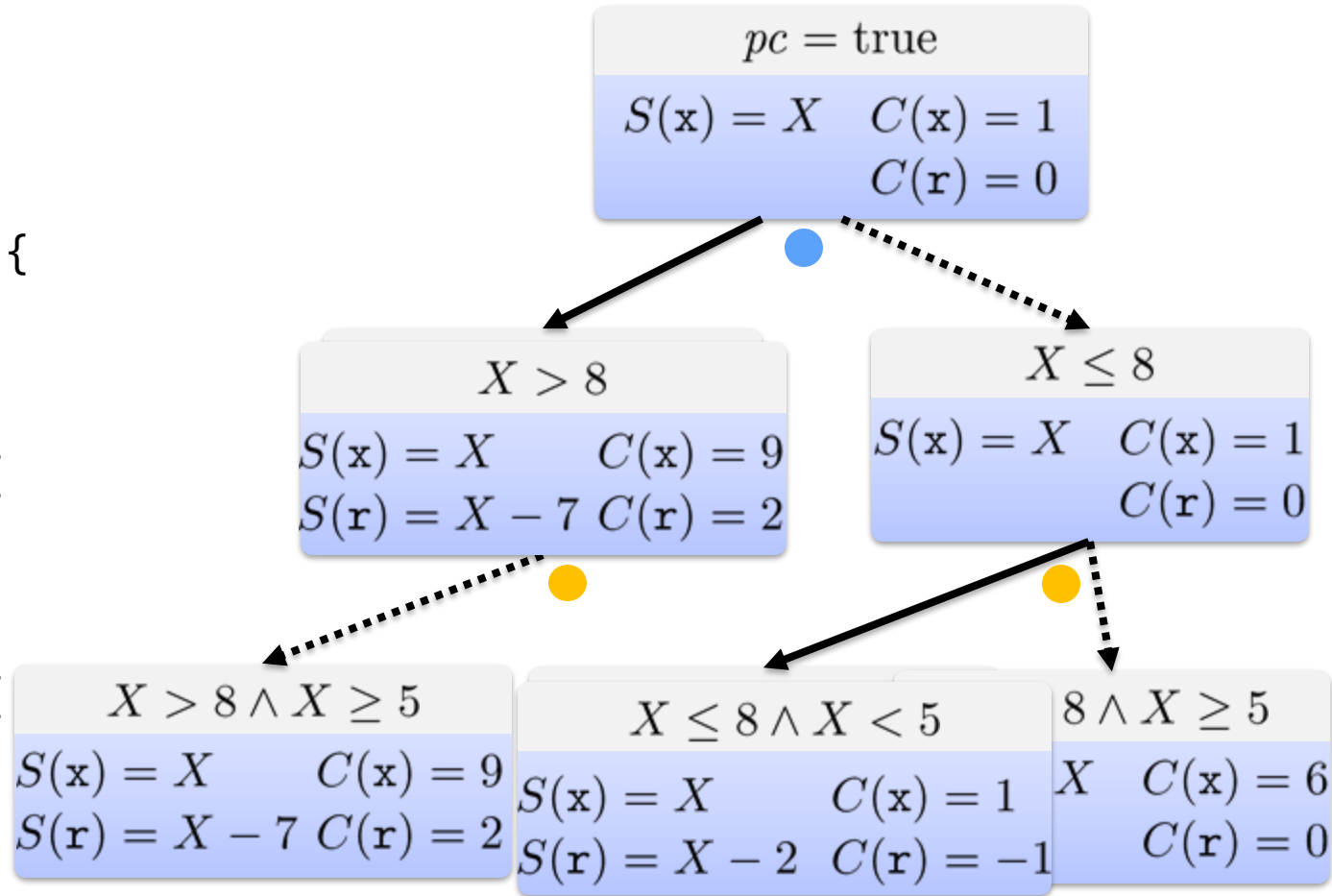


EXE

```

1  int proc(int x) {
2
3      int r = 0
4
5      if (x > 8 ●) {
6          r = x - 7
7      }
8
9      if (x < 5 ●) {
10         r = x - 2
11     }
12
13     return r
14 }

```

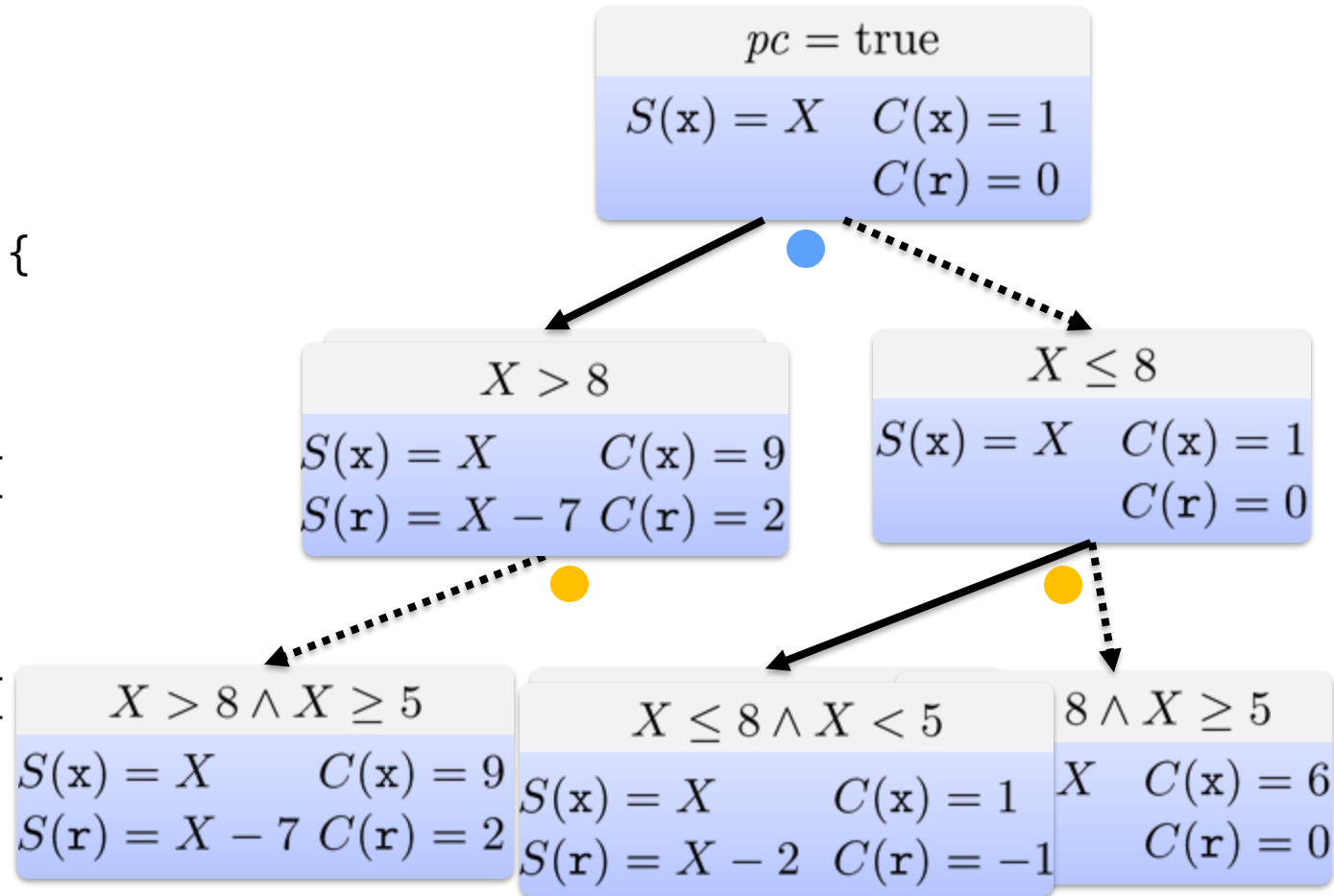


EXE

```

1 int proc(int x) {
2
3   int r = 0
4
5   if (x > 8 ●) {
6     r = x - 7
7   }
8
9   if (x < 5 ●) {
10    r = x - 2
11  }
12
13  return r
14 }

```



Satisfying assignments:

$X = 9$

$X = 1$

$X = 6$

Test cases:

proc(9)

proc(1)

proc(6)

EXE: Implementation Considerations

Must maintain multiple execution states

- each execution state has symbolic and concrete components
- must switch between different execution states
- usually implemented using a special-purpose virtual machine

KLEE uses LLVM bitcode interpreter to maintain and update state

- requires programs to be compiled by Clang into LLVM bitcode
- code that cannot be compiled, must be modeled by the user
 - e.g., assembly, system calls, third-party libraries, etc.

klee.github.io/

S2E uses QEMU virtual machine to fork and restore the entire machine state, including OS

- executes symbolically any executable code
- executes concretely system code (i.e., OS, assembly, third-party)
- uses LLVM under-the-hood for symbolic execution

s2e.systems

DART: Intuition

Test the program concretely, and record execution paths taken

- inputs are either random (fuzzing), user-provided (tests), or mixed
- the program is instrumented to record execution path
- instrumentation can be at binary level, enough to track branching decisions
 - same idea as in typical coverage techniques

For each run in the previous step

- re-execute symbolically by computing the path condition
- at every branch condition, adjust inputs to force different branch than taken
- this step generates new inputs that cover new execution paths

Repeat the first step using inputs generated in second step

- repeat until all bugs are found, or all resources are used up

DART: Details

```
formula Seen := False
while ¬Seen is satisfiable do
  find input x that satisfies ¬Seen
  execute P(x) and record path condition C
  // important: x satisfies C
  Seen := Seen ∨ C // record C as seen
done
```

Seen is a formula in SMT-LIB format, initially *False*

P is a program being tested, with input *x*

- *x* represents possible inputs, it can be one or multiple values, command line parameters, input files, etc...

C is a path condition in SMT-LIB, computed using symbolic execution

- based on the program path that is executed using input *x*

DART

```
1 int proc(int x) {  
2  
3   int r = 0  
4  
5   if (x > 8 ●) {  
6     r = x - 7  
7   }  
8  
9   if (x < 5 ●) {  
10    r = x - 2  
11  }  
12  
13  return r  
}
```

$pc = \text{true}$

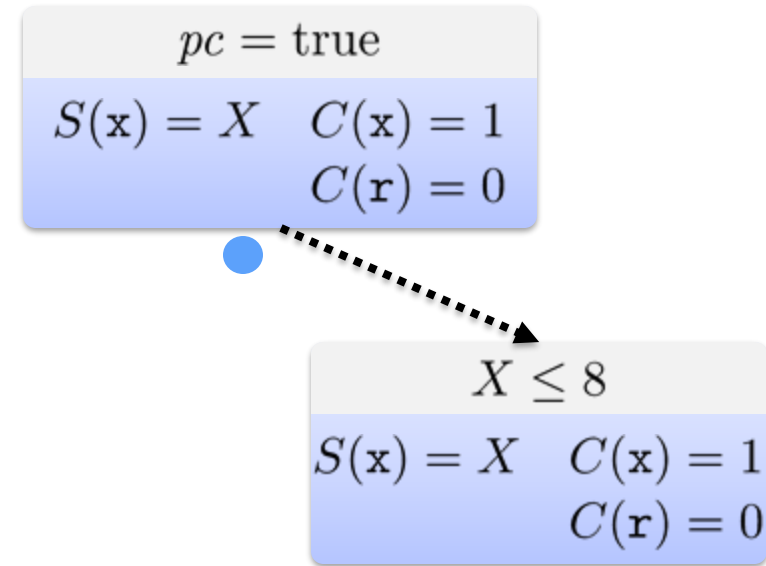
$S(x) = X \quad C(x) = 1$
 $C(r) = 0$

Test cases:

proc(1)

DART

```
1 int proc(int x) {  
2  
3   int r = 0  
4  
5   if (x > 8 ●) {  
6     r = x - 7  
7   }  
8  
9   if (x < 5 ●) {  
10    r = x - 2  
11  }  
12  
13  return r  
}
```

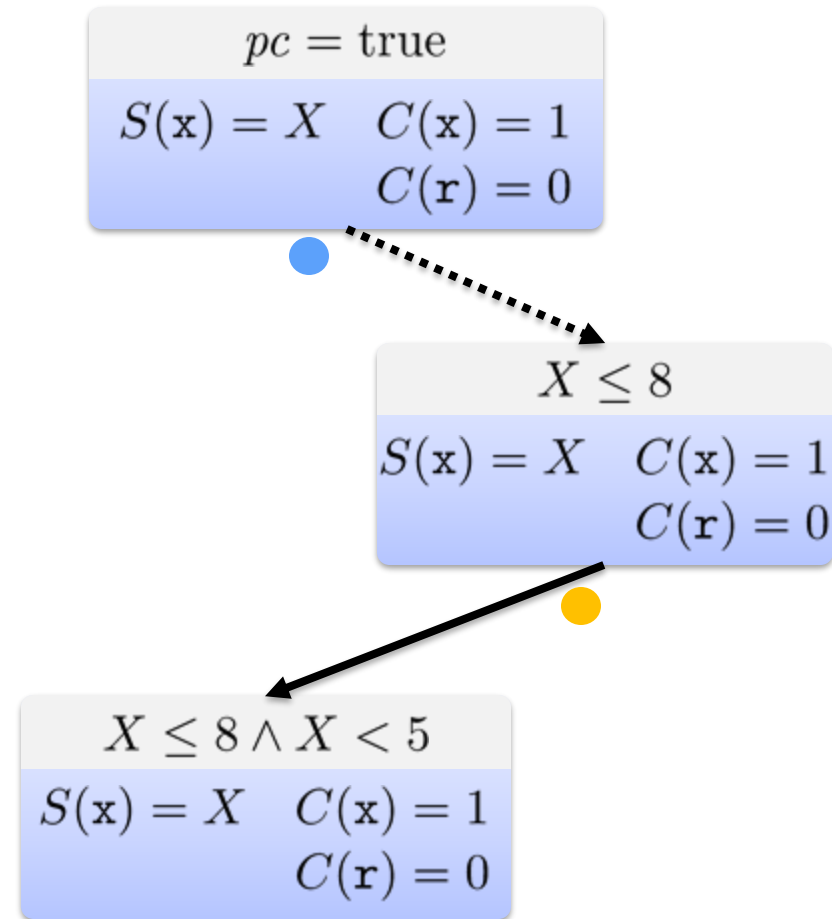


Test cases:

proc(1)

DART

```
1 int proc(int x) {  
2  
3   int r = 0  
4  
5   if (x > 8 ●) {  
6     r = x - 7  
7   }  
8  
9   if (x < 5 ●) {  
10    r = x - 2  
11  }  
12  
13  return r  
}
```

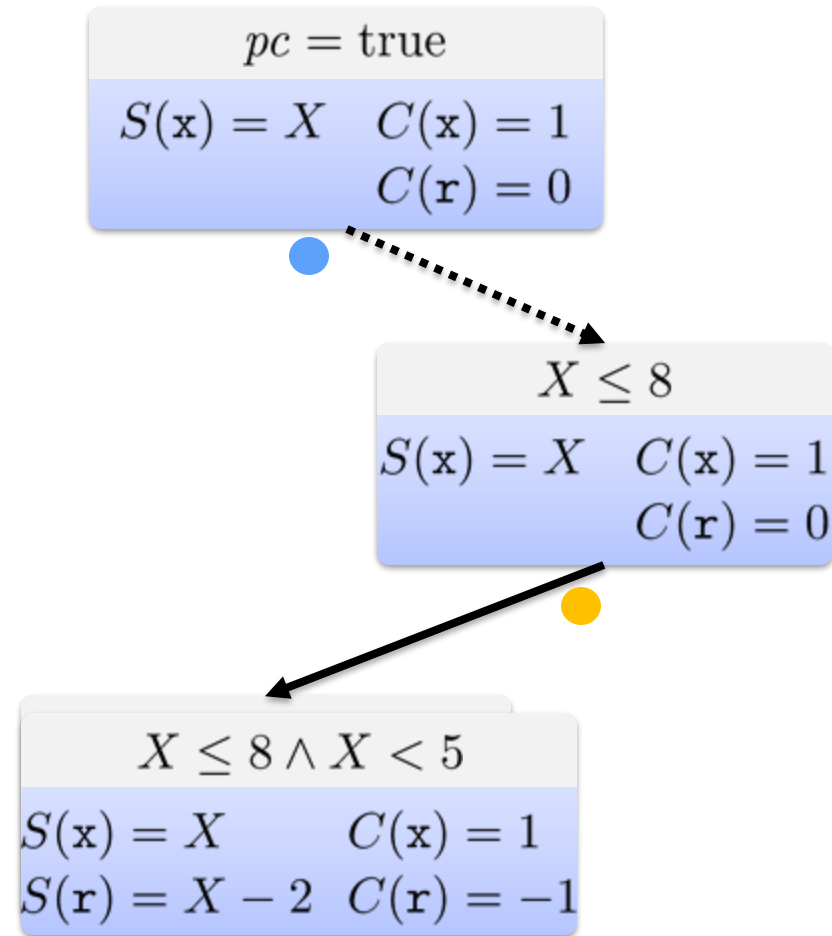


Test cases:

`proc(1)`

DART

```
1 int proc(int x) {  
2  
3   int r = 0  
4  
5   if (x > 8 ●) {  
6     r = x - 7  
7   }  
8  
9   if (x < 5 ●) {  
10    r = x - 2  
11  }  
12  
13  return r  
}
```

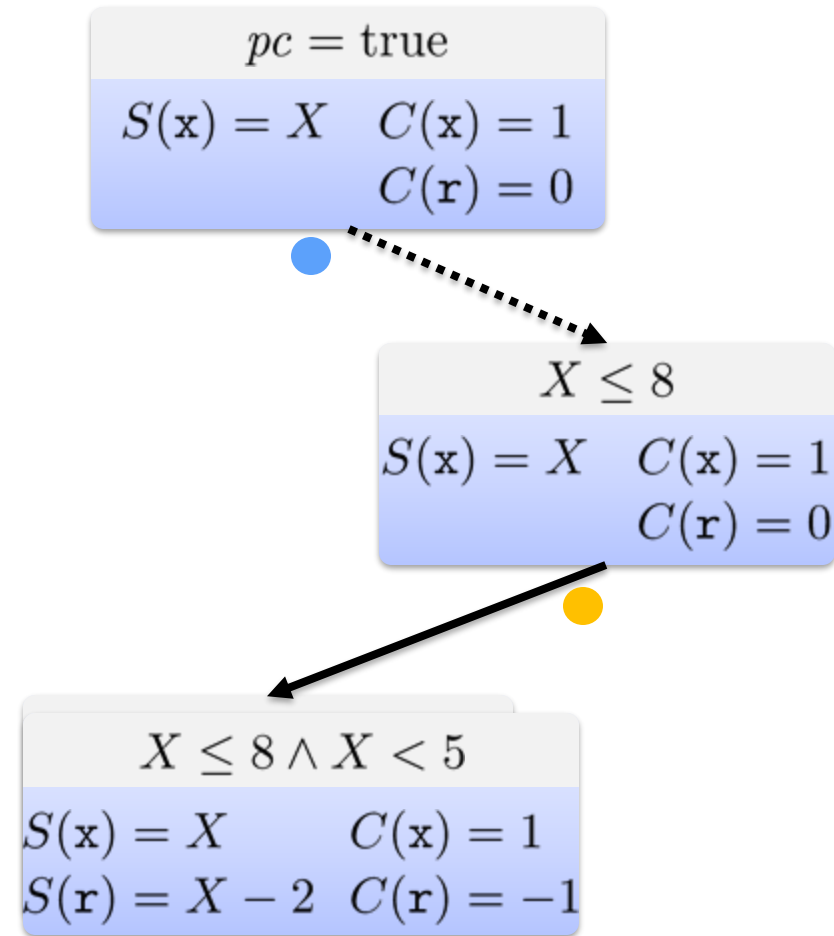


Test cases:

`proc(1)`

DART

```
1 int proc(int x) {  
2  
3   int r = 0  
4  
5   if (x > 8 ●) {  
6     r = x - 7  
7   }  
8  
9   if (x < 5 ●) {  
10    r = x - 2  
11  }  
12  
13  return r  
}
```



New path condition:

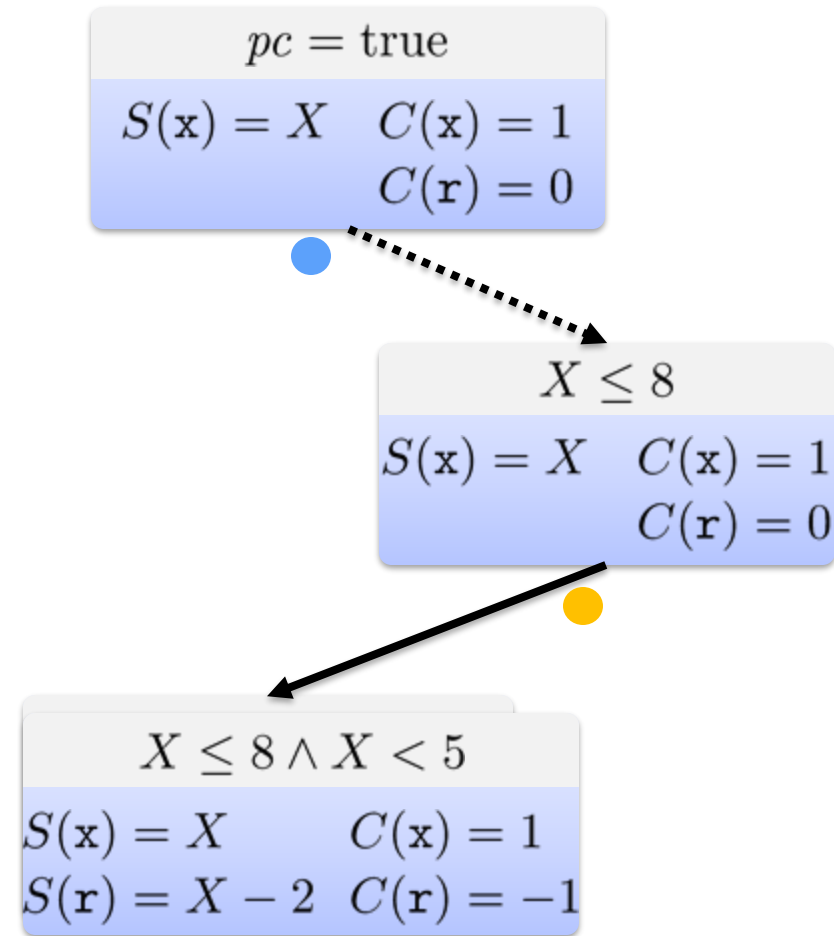
$$X \leq 8 \wedge \neg(X < 5)$$

Test cases:

proc(1)

DART

```
1 int proc(int x) {  
2  
3   int r = 0  
4  
5   if (x > 8 ●) {  
6     r = x - 7  
7   }  
8  
9   if (x < 5 ●) {  
10    r = x - 2  
11  }  
12  
13  return r  
}
```



New path condition:

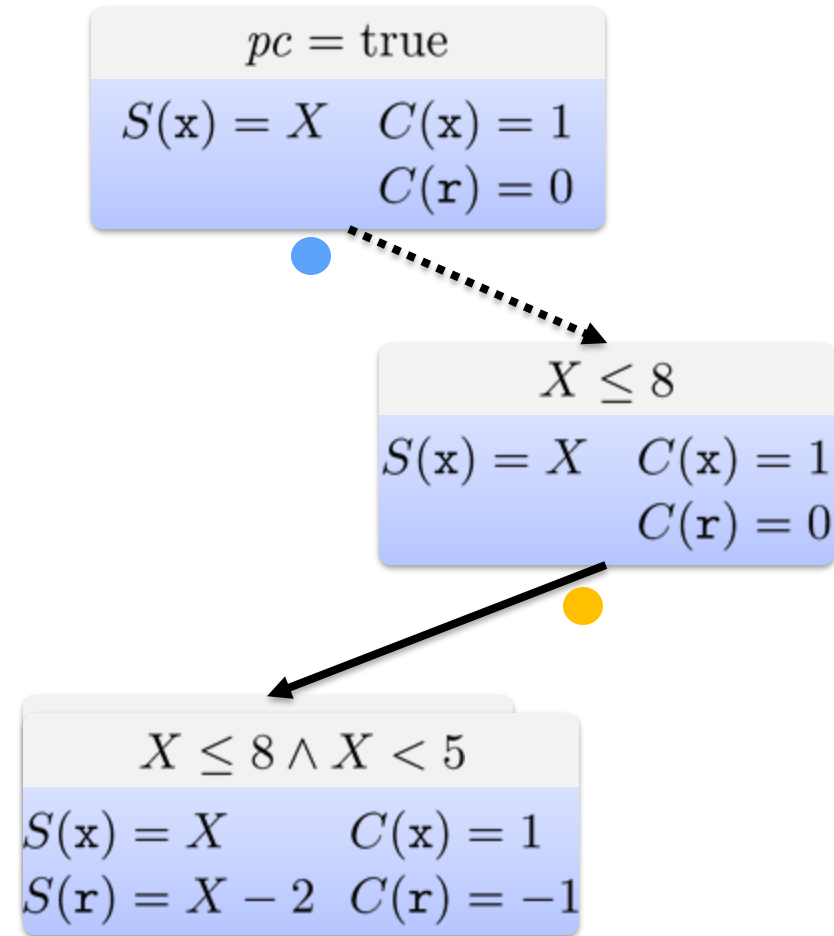
$$X \leq 8 \wedge \neg(X < 5) \checkmark$$

Test cases:

proc(1)

DART

```
1 int proc(int x) {  
2  
3   int r = 0  
4  
5   if (x > 8 ●) {  
6     r = x - 7  
7   }  
8  
9   if (x < 5 ●) {  
10    r = x - 2  
11  }  
12  
13  return r  
}
```



New path condition:

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

Test cases:

proc(1)

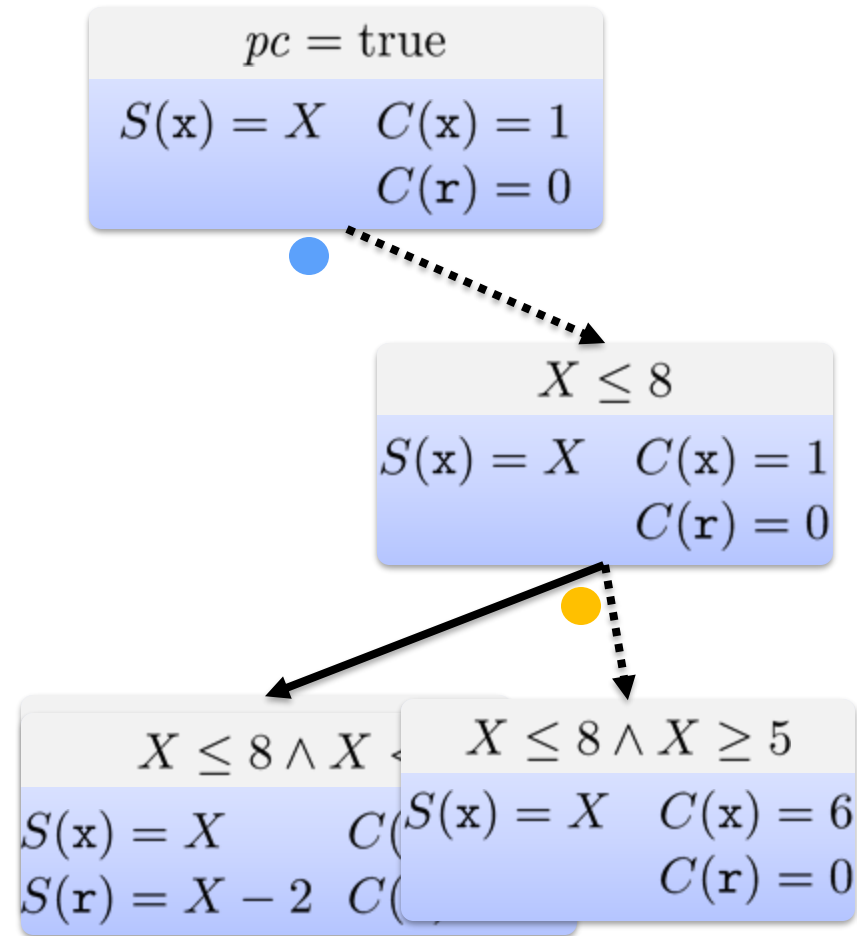
proc(6)

DART

```

1 int proc(int x) {
2
3   int r = 0
4
5   if (x > 8 ●) {
6     r = x - 7
7   }
8
9   if (x < 5 ●) {
10    r = x - 2
11  }
12
13  return r
14 }

```



New path condition:

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

Test cases:

proc(1)

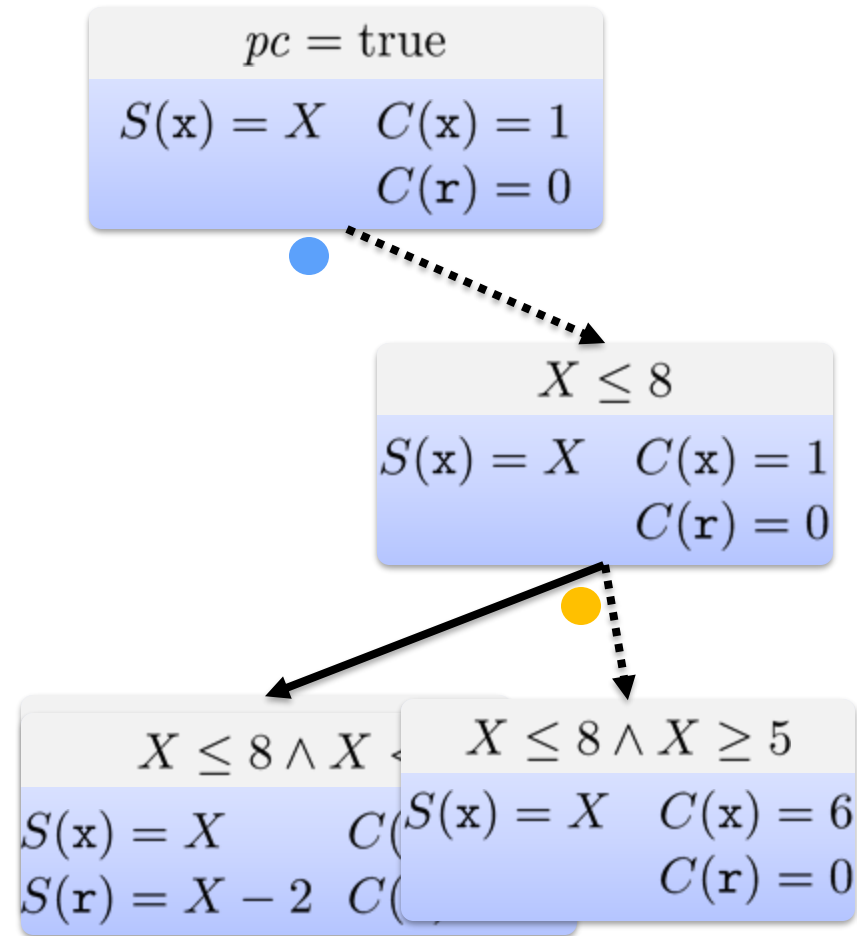
proc(6)

DART

```

1 int proc(int x) {
2
3   int r = 0
4
5   if (x > 8 ●) {
6     r = x - 7
7   }
8
9   if (x < 5 ●) {
10    r = x - 2
11  }
12
13  return r
14 }

```



New path condition:

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

Test cases:

proc(1)

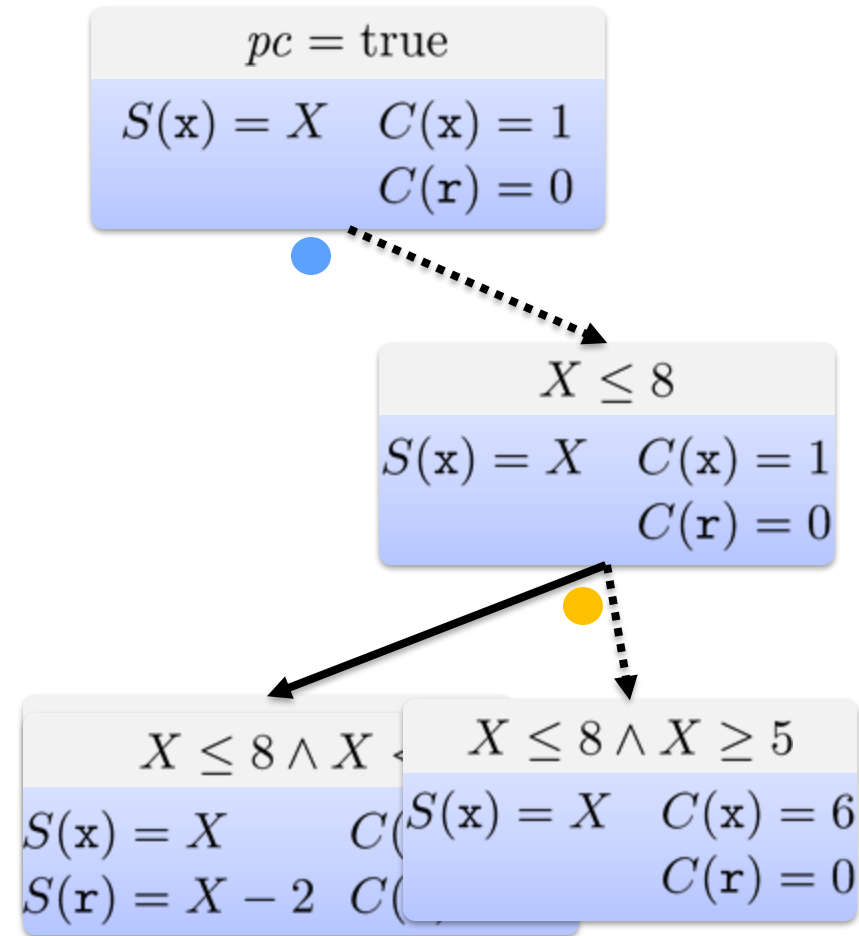
proc(6)

DART

```

1 int proc(int x) {
2
3   int r = 0
4
5   if (x > 8 ●) {
6     r = x - 7
7   }
8
9   if (x < 5 ●) {
10    r = x - 2
11  }
12
13  return r
14 }

```



New path condition:

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

Test cases:

proc(1)

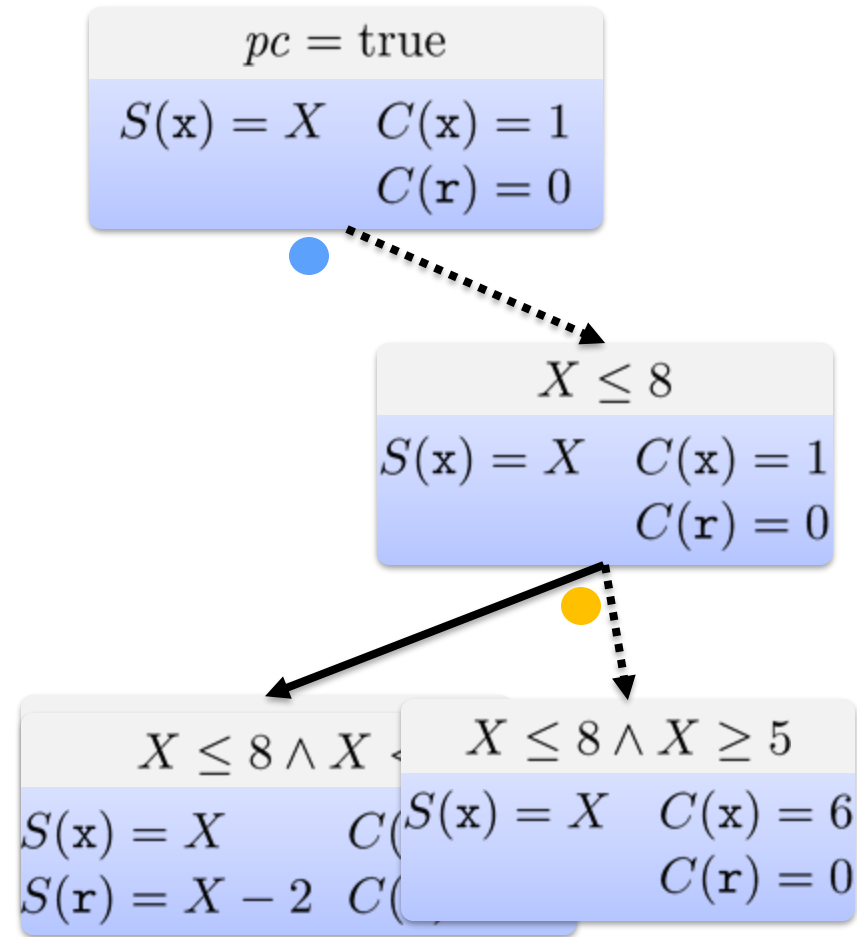
proc(6)

DART

```

1 int proc(int x) {
2
3   int r = 0
4
5   if (x > 8 ●) {
6     r = x - 7
7   }
8
9   if (x < 5 ●) {
10    r = x - 2
11  }
12
13  return r
14 }

```



New path condition:

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

Test cases:

proc(9)

proc(1)

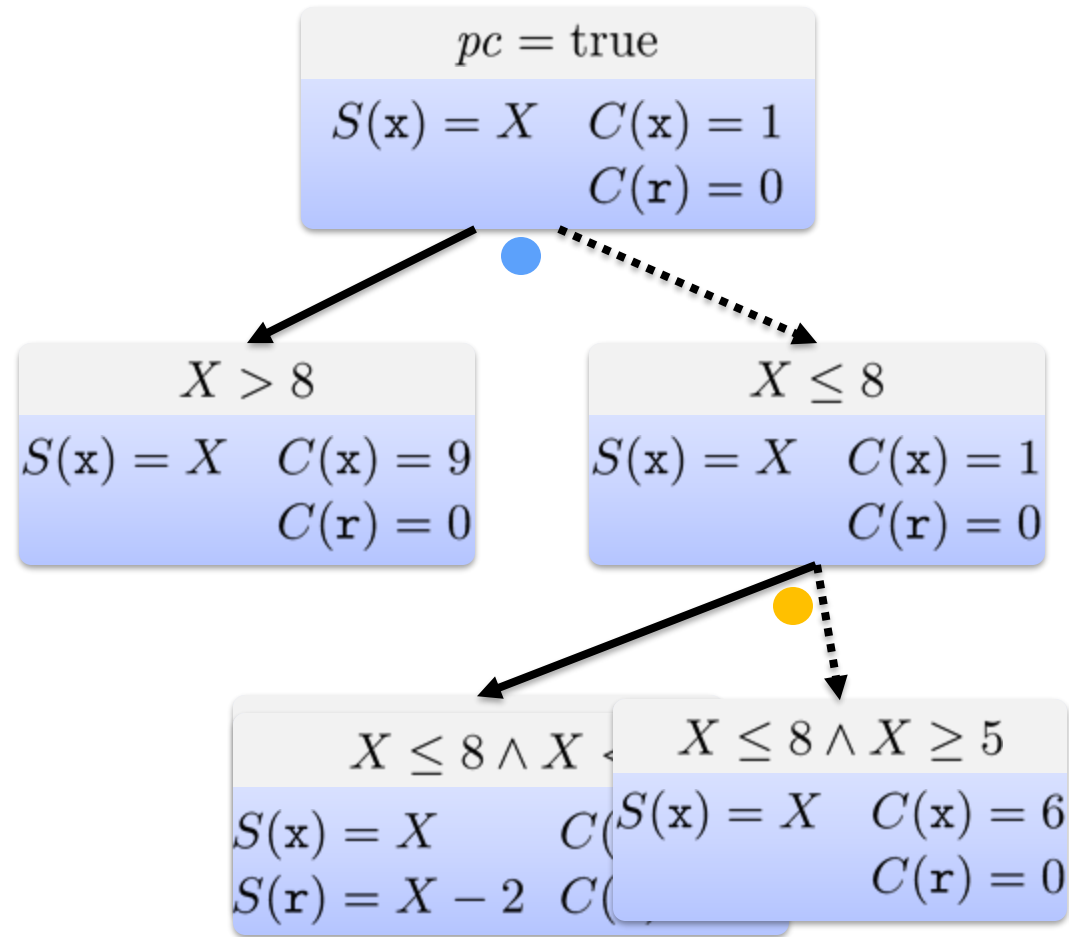
proc(6)

DART

```

1 int proc(int x) {
2
3   int r = 0
4
5   if (x > 8 ●) {
6     r = x - 7
7   }
8
9   if (x < 5 ●) {
10    r = x - 2
11  }
12
13  return r
14 }

```



New path condition:
 $\neg(X \leq 8)$

Test cases:

proc(9)

proc(1)

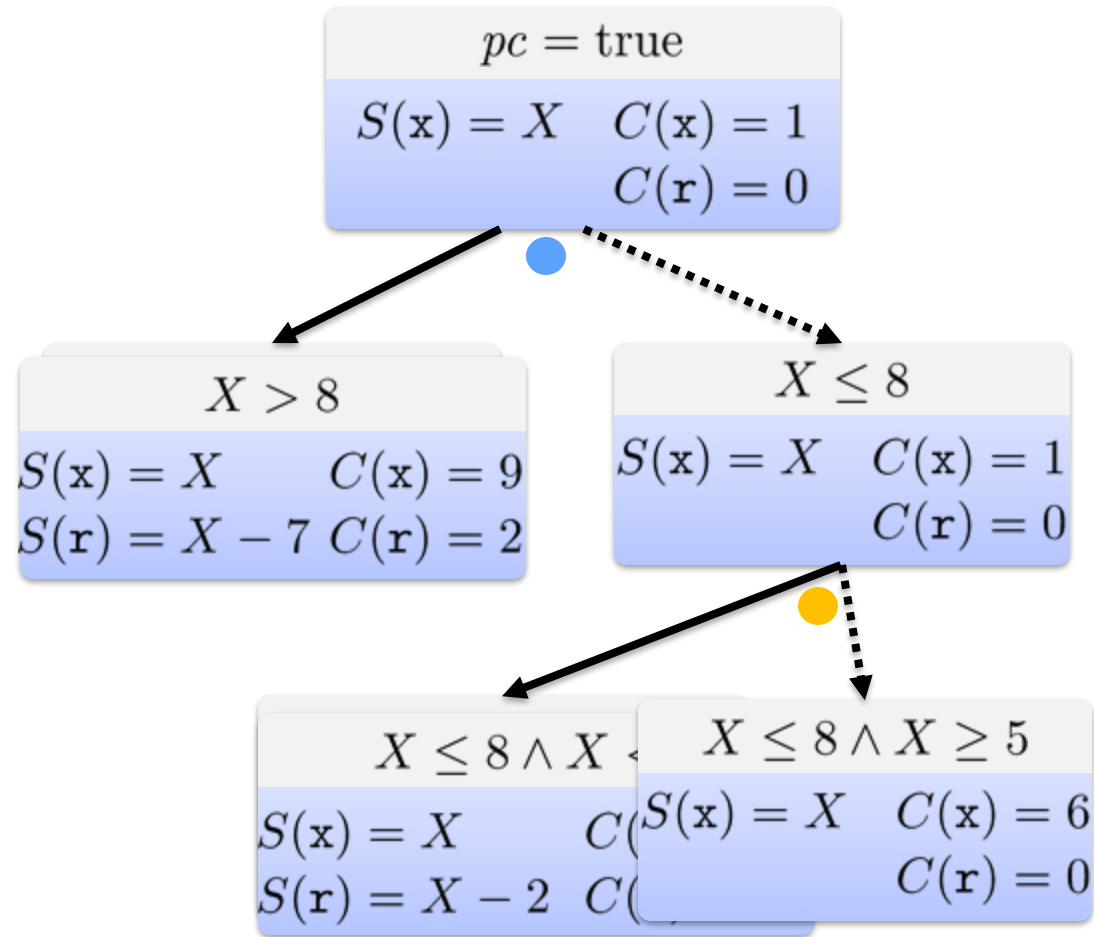
proc(6)

DART

```

1 int proc(int x) {
2
3   int r = 0
4
5   if (x > 8 ●) {
6     r = x - 7
7   }
8
9   if (x < 5 ●) {
10    r = x - 2
11  }
12
13  return r
14 }

```



New path condition:
 $\neg(X \leq 8)$

Test cases:

proc(9)

proc(1)

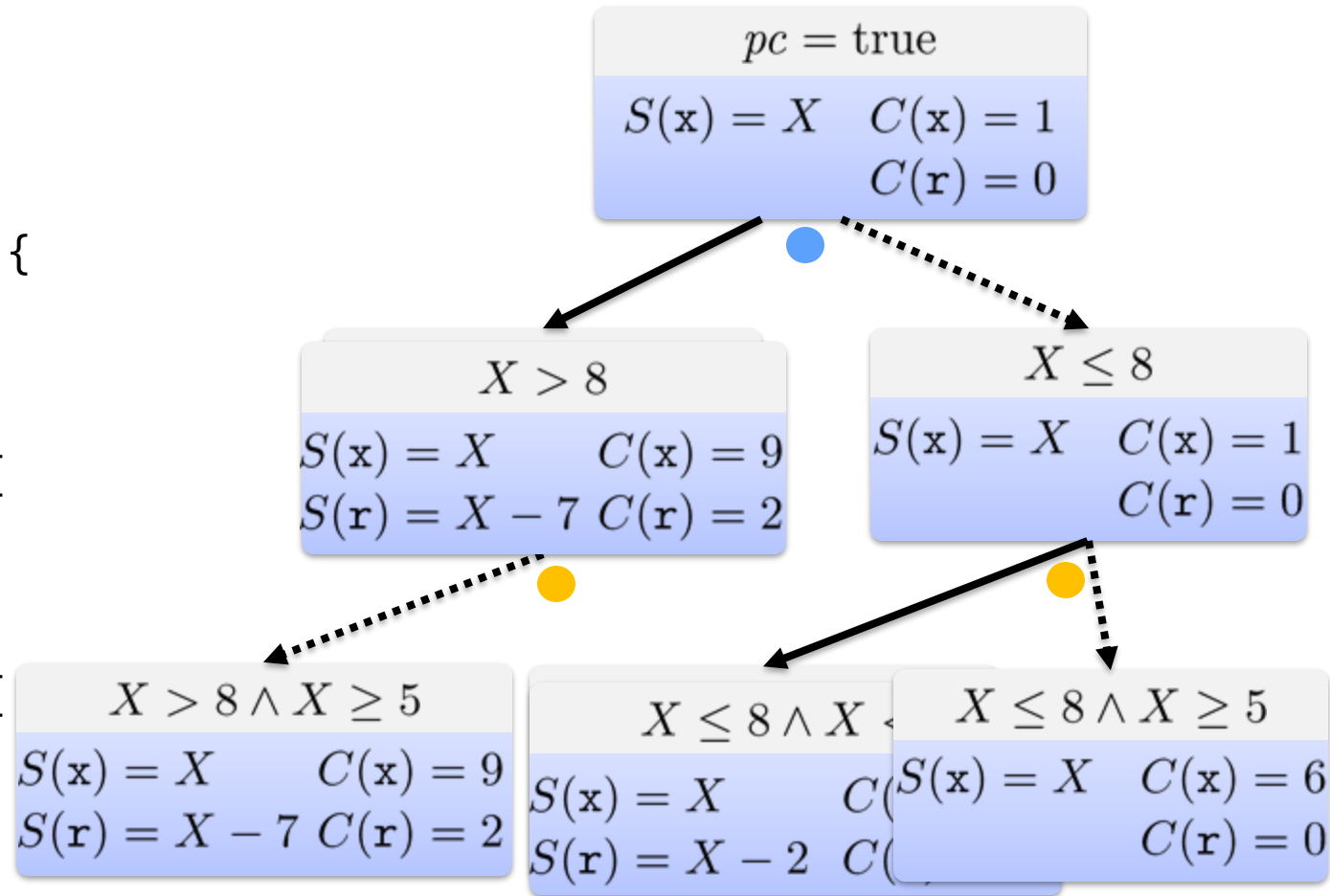
proc(6)

DART

```

1 int proc(int x) {
2
3   int r = 0
4
5   if (x > 8 ●) {
6     r = x - 7
7   }
8
9   if (x < 5 ●) {
10    r = x - 2
11  }
12
13  return r
14 }

```



New path condition:

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

Test cases:

proc(9)

proc(1)

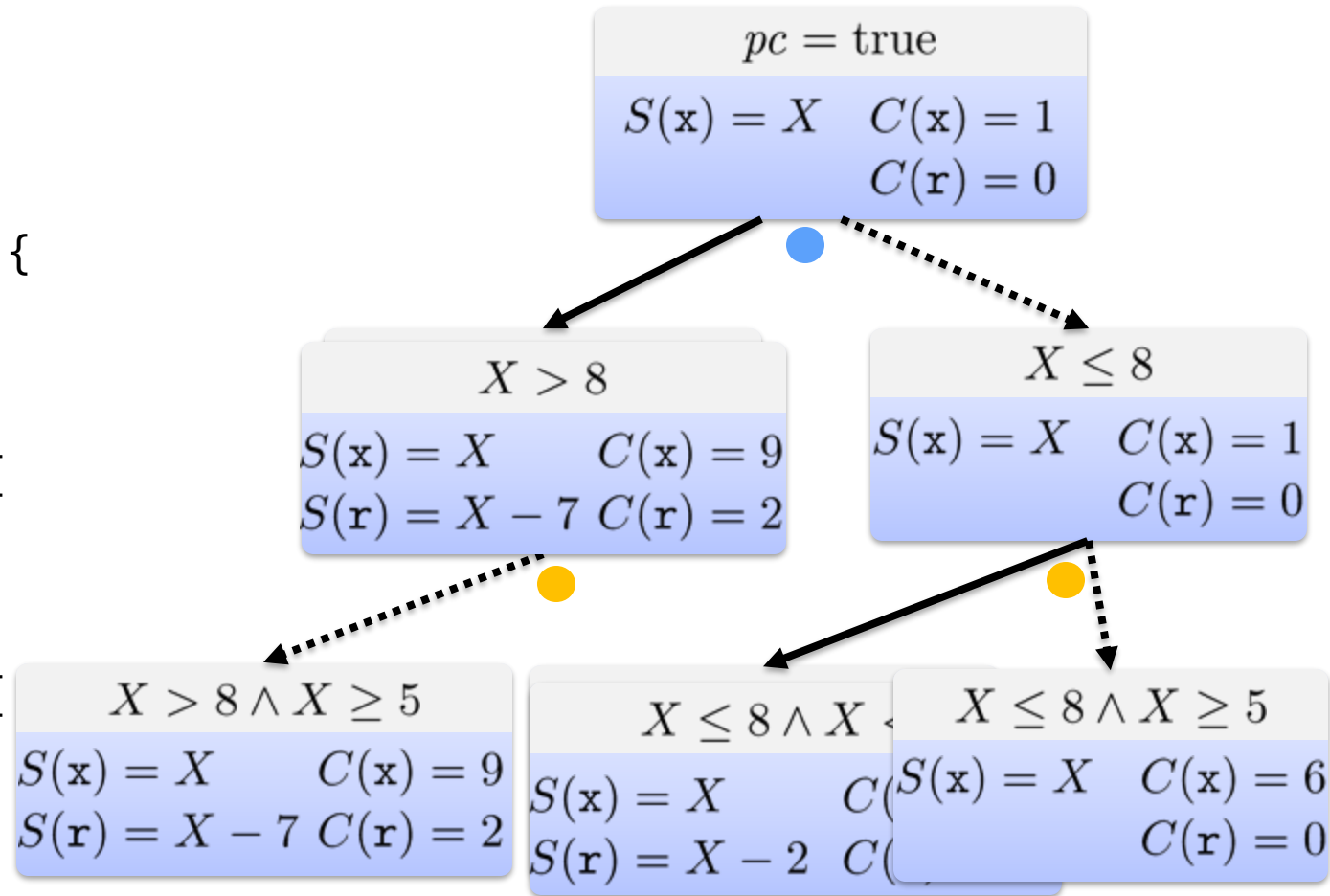
proc(6)

DART

```

1 int proc(int x) {
2
3   int r = 0
4
5   if (x > 8 ●) {
6     r = x - 7
7   }
8
9   if (x < 5 ●) {
10    r = x - 2
11  }
12
13  return r
14 }

```



New path condition:

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

Test cases:

proc(9)

proc(1)

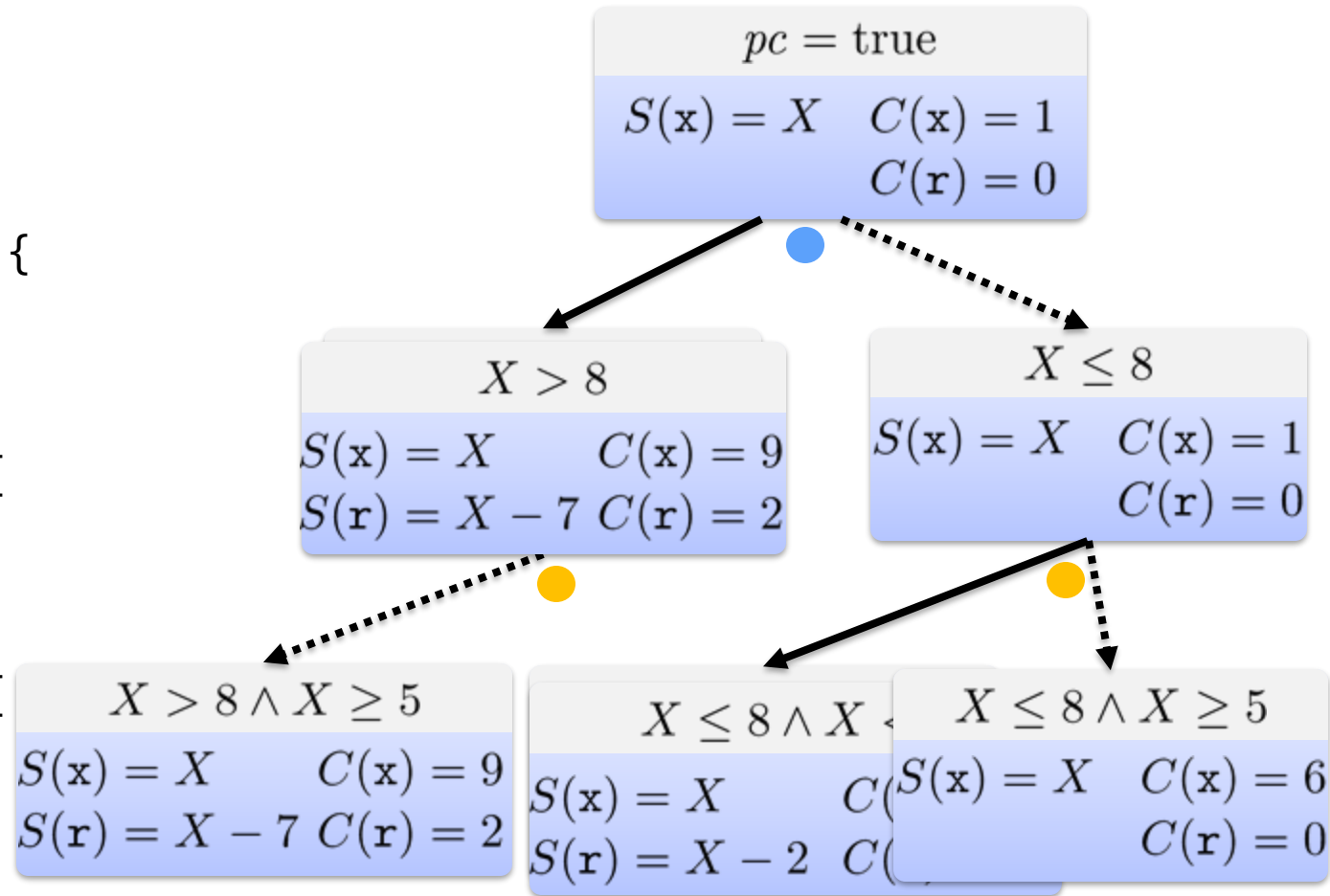
proc(6)

DART

```

1 int proc(int x) {
2
3   int r = 0
4
5   if (x > 8 ●) {
6     r = x - 7
7   }
8
9   if (x < 5 ●) {
10    r = x - 2
11  }
12
13  return r
14 }

```



New path condition:

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

Test cases:

proc(9)

proc(1)

proc(6)

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

Data	

DART: Another Example

Code to generate inputs for:


```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

Data	
	null

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

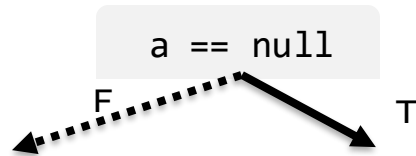


Data	
null	

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

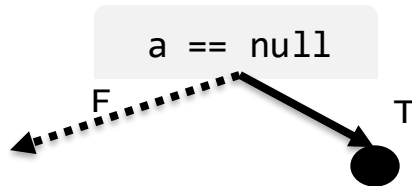


Execute&Monitor	
Data	
null	

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

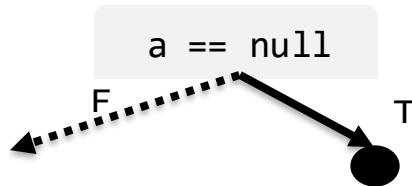


Execute&Monitor	
Data	Observed constraints
null	a==null

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```



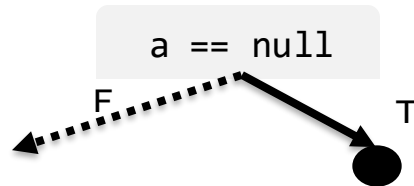
Choose next path

Data	Observed constraints
null	a==null

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```



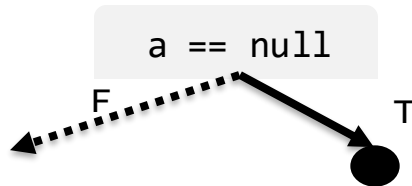
Choose next path

Constraints to solve	Data	Observed constraints
	null	a==null
a!=null		

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

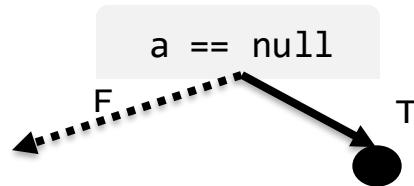


Solve		
Constraints to solve	Data	Observed constraints
	null	a==null
a!=null		

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

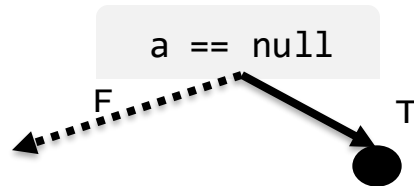


Solve		
Constraints to solve	Data	Observed constraints
	null	a==null
a!=null	{}	

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

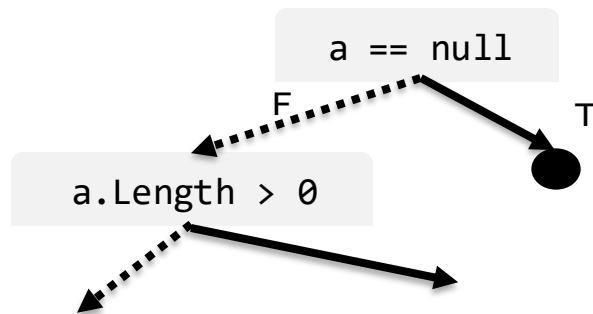


Constraints to solve	Execute&Monitor	
	Data	Observed constraints
	null	a==null
a!=null	{}	

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

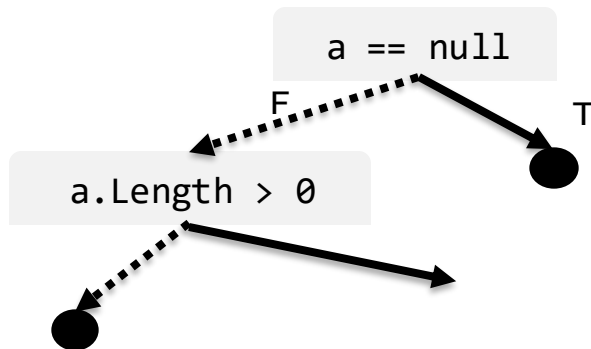


Constraints to solve	Execute&Monitor	
	Data	Observed constraints
	null	a==null
a!=null	{}	

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

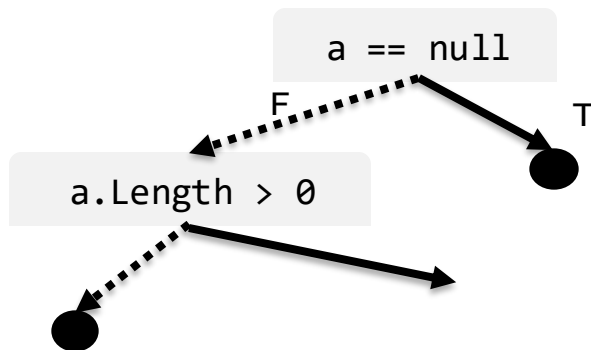


Constraints to solve	Execute&Monitor	
	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

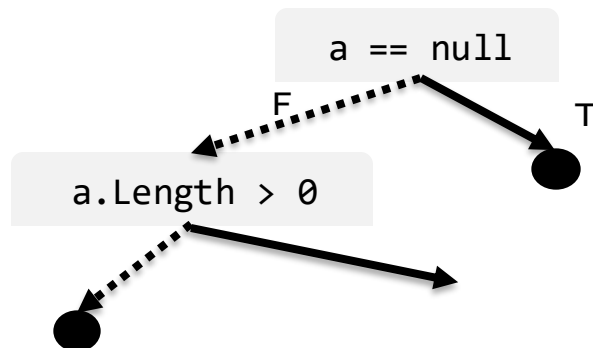


Choose next path		
Constraints to solve	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```



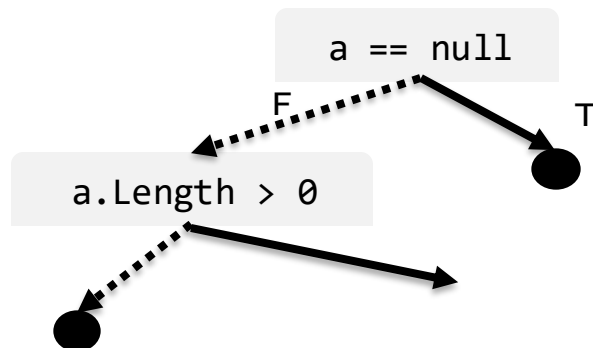
Choose next path

Constraints to solve	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0		

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

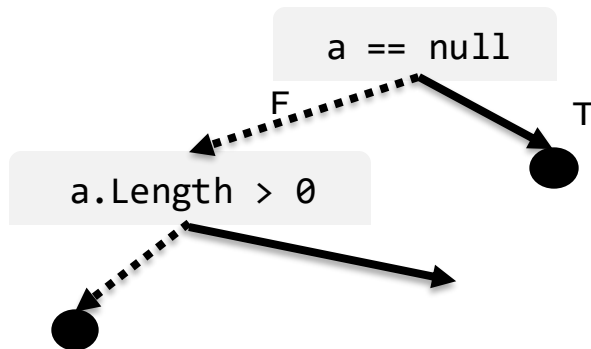


Solve		
Constraints to solve	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0		

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

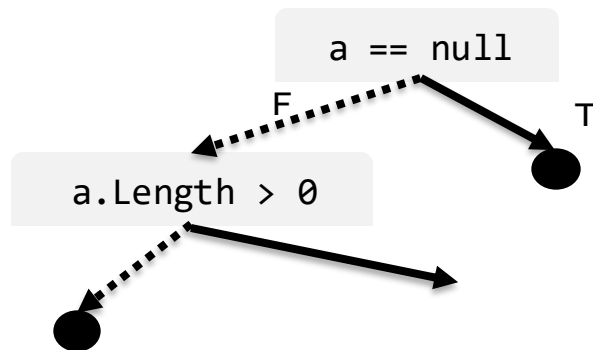


Solve		
Constraints to solve	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0	{0}	

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

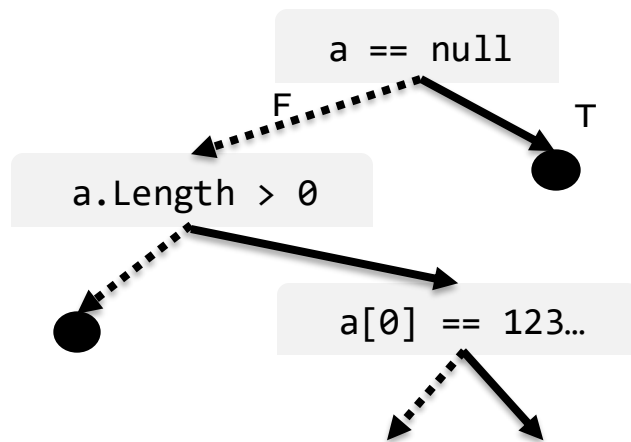


Constraints to solve	Execute&Monitor	
	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0	{0}	

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

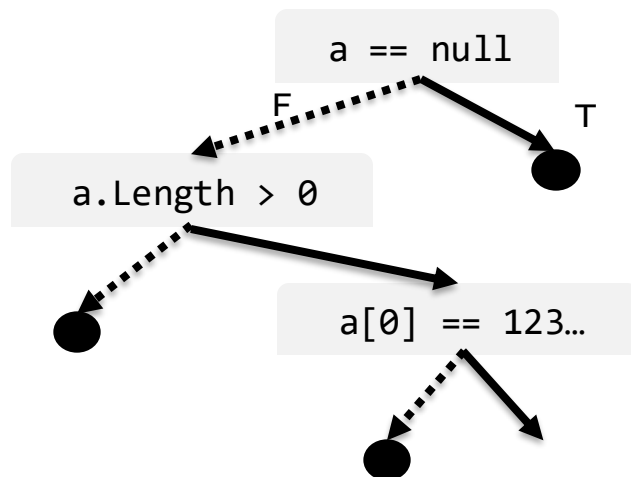


Constraints to solve	Execute&Monitor	
	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0	{0}	

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

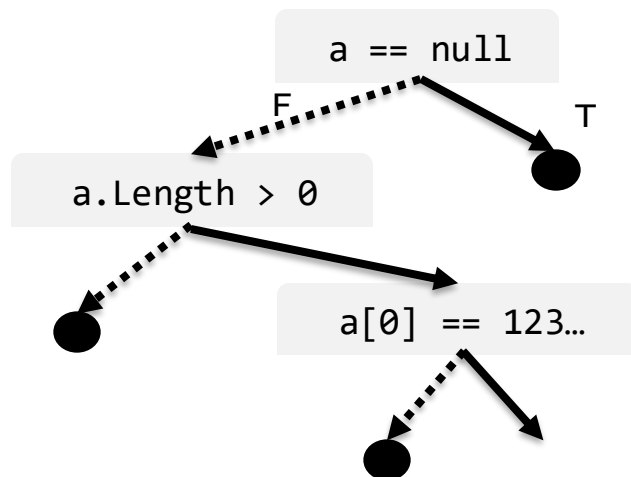


Constraints to solve	Execute&Monitor	
	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0	{0}	a!=null && a.Length>0 && a[0]!=1234567890

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```



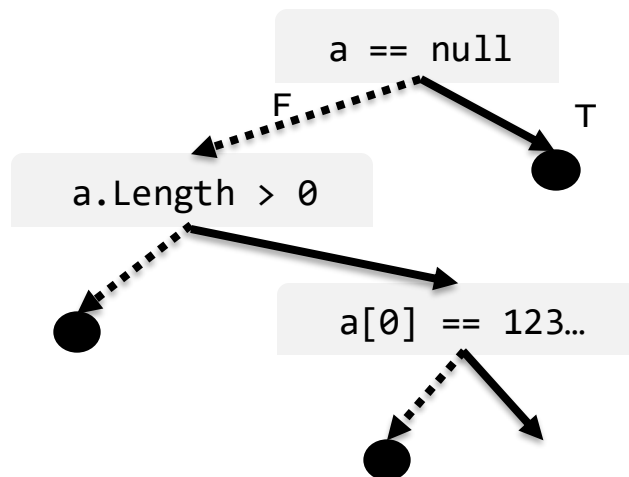
Choose next path

Constraints to solve	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0	{0}	a!=null && a.Length>0 && a[0]!=1234567890

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```



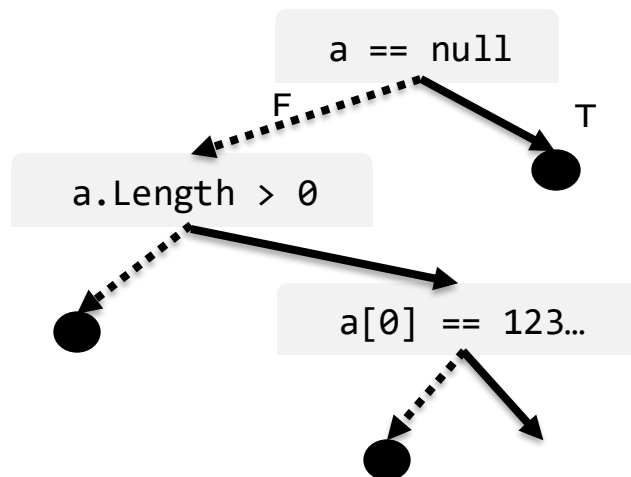
Choose next path

Constraints to solve	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0	{0}	a!=null && a.Length>0 && a[0]!=1234567890
a!=null && a.Length>0 && a[0]==1234567890		

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

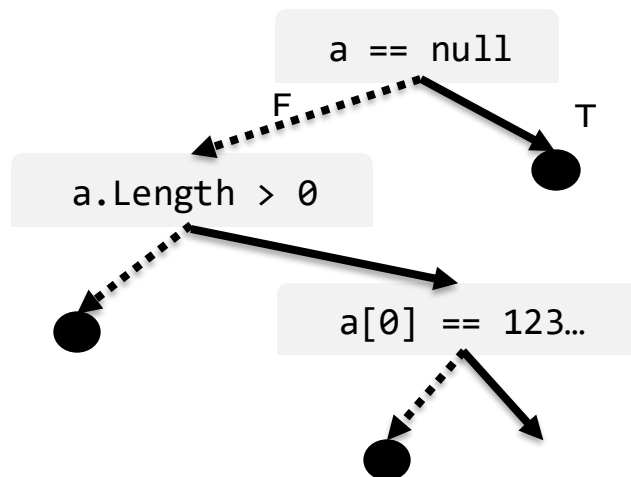


Solve		
Constraints to solve	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0	{0}	a!=null && a.Length>0 && a[0]!=1234567890
a!=null && a.Length>0 && a[0]==1234567890		

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

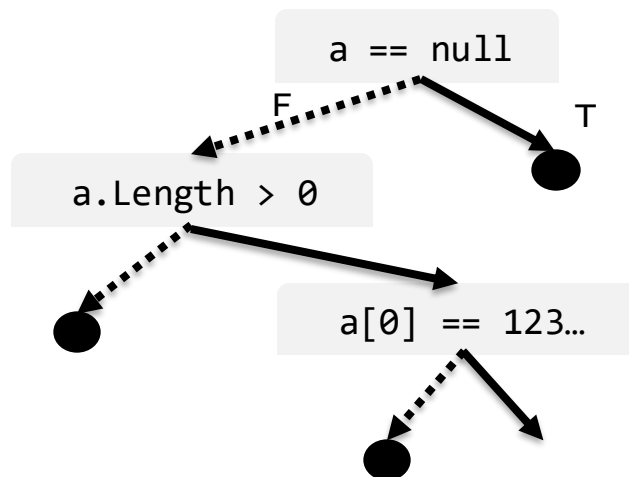


Solve		
Constraints to solve	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0	{0}	a!=null && a.Length>0 && a[0]!=1234567890
a!=null && a.Length>0 && a[0]==1234567890	{123..}	

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

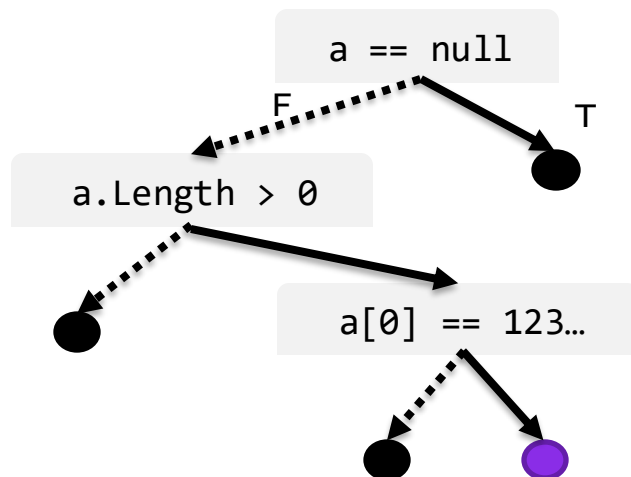


Constraints to solve	Execute&Monitor	
	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0	{0}	a!=null && a.Length>0 && a[0]!=1234567890
a!=null && a.Length>0 && a[0]==1234567890	{123..}	

DART: Another Example

Code to generate inputs for:

```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```

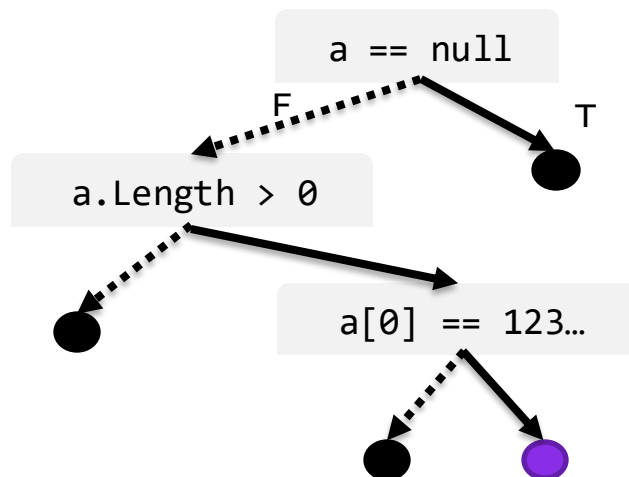


Constraints to solve	Execute&Monitor	
	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0	{0}	a!=null && a.Length>0 && a[0]!=1234567890
a!=null && a.Length>0 && a[0]==1234567890	{123..}	a!=null && a.Length>0 && a[0]==1234567890

DART: Another Example

Code to generate inputs for:

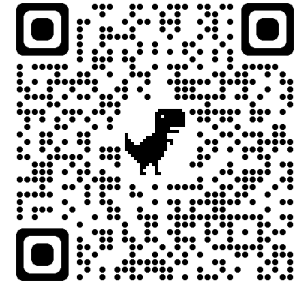
```
void CoverMe(int [] a) {  
  
    if(a == null) return;  
    if(a.Length > 0  
        if(a[0] == 1234567890)  
            throw new Exception("bug");  
}
```



Constraints to solve	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0	{0}	a!=null && a.Length>0 && a[0]!=1234567890
a!=null && a.Length>0 && a[0]==1234567890	{123..}	a!=null && a.Length>0 && a[0]==1234567890

Done: There is no path left.

Zero to Crash in 10 Generations



Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser:

```
00000000h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000060h: 00 00 00 00 ; ....
```

Generation 0 – seed file

Zero to Crash in 10 Generations

Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 00 00 00 00 00 00 00 00 00 00 00 ; RTFF.....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000060h: 00 00 00 00 ; ....
```

Generation 1

Zero to Crash in 10 Generations

Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 00 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF... *** ...
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000060h: 00 00 00 00 ; ....
```

Generation 2

Zero to Crash in 10 Generations

Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF*** ...
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000060h: 00 00 00 00 ; ....
```

Generation 3

Zero to Crash in 10 Generations

Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ...
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 00 00 00 ; ...strh.....
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000060h: 00 00 00 00 ; ....
```

Generation 4

Zero to Crash in 10 Generations

Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ...
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh...vids
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000060h: 00 00 00 00 ; ....
```

Generation 5

Zero to Crash in 10 Generations

Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ...
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 00 00 00 00 ; ....strf.....
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000060h: 00 00 00 00 ; ....
```

Generation 6

Zero to Crash in 10 Generations

Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ...
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000060h: 00 00 00 00 ; ....
```

Generation 7

Zero to Crash in 10 Generations

Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ...
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(..
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 C9 9D E4 4E ; .....E4N
00000060h: 00 00 00 00 ; ....
```

Generation 8

Zero to Crash in 10 Generations

Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ...
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(..
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 ; .....
00000060h: 00 00 00 00 ; ....
```

Generation 9

Zero to Crash in 10 Generations

Starting with 100 zero bytes ...

SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ...
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 B2 75 76 3A 28 00 00 00 ; ....strf2uv:(..
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ; .....
00000060h: 00 00 00 00 ; .....
```

Generation 10

Example: SAGE Generational Input Generation

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt > 3) crash();
}
```

Example: SAGE Generational Input Generation

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt > 3) crash();
}

input = "good"
```

Example: SAGE Generational Input Generation

```
void top(char input[4])  
{  
    int cnt = 0;  
    if (input[0] == 'b') cnt++;  
    if (input[1] == 'a') cnt++;  
    if (input[2] == 'd') cnt++;  
    if (input[3] == '!') cnt++;  
    if (cnt > 3) crash();  
}
```

`input = "good"`

Path constraint:

$I_0 \neq \text{'b'}$

Example: SAGE Generational Input Generation

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt > 3) crash();
}
```

`input = "good"`

Path constraint:

$I_0 \neq \text{'b'}$

$I_1 \neq \text{'a'}$

Example: SAGE Generational Input Generation

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt > 3) crash();
}
```

`input = "good"`

Path constraint:

$I_0 \neq \text{'b'}$

$I_1 \neq \text{'a'}$

$I_2 \neq \text{'d'}$

Example: SAGE Generational Input Generation

```
void top(char input[4])  
{  
    int cnt = 0;  
    if (input[0] == 'b') cnt++;  
    if (input[1] == 'a') cnt++;  
    if (input[2] == 'd') cnt++;  
    if (input[3] == '!') cnt++;  
    if (cnt > 3) crash();  
}
```

`input = "good"`

Path constraint:

$I_0 \neq \text{'b'}$

$I_1 \neq \text{'a'}$

$I_2 \neq \text{'d'}$

$I_3 \neq \text{'!'}$

Example: SAGE Generational Input Generation

```
void top(char input[4])  
{  
    int cnt = 0;  
    if (input[0] == 'b') cnt++;  
    if (input[1] == 'a') cnt++;  
    if (input[2] == 'd') cnt++;  
    if (input[3] == '!') cnt++;  
    if (cnt > 3) crash();  
}
```

`input = "good"`

Path constraint:

$I_0 \neq \text{'b'}$

$I_1 \neq \text{'a'}$

$I_2 \neq \text{'d'}$

$I_3 \neq \text{'!'}$

Negate each constraint in path
constraint
Solve new constraint → new input

Example: SAGE Generational Input Generation

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt > 3) crash();
}
```

`input = "good"`

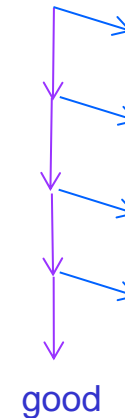
Path constraint:

$I_0 \neq \text{'b'}$

$I_1 \neq \text{'a'}$

$I_2 \neq \text{'d'}$

$I_3 \neq \text{'!'}$



Negate each constraint in path
constraint
Solve new constraint → new input

Example: SAGE Generational Input Generation

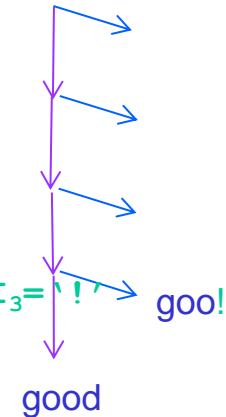
```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt > 3) crash();
}
```

```
input = "good"
```

Path constraint:

$$I_0 \neq 'b'$$
$$I_1 \neq 'a'$$
$$I_2 \neq 'd'$$

$I_3 \neq \backslash ! ' \rightarrow I_3 = \backslash ! ' \rightarrow \text{goo!}$



Negate each constraint in path
constraint
Solve new constraint → new input

Example: SAGE Generational Input Generation

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt > 3) crash();
}
```

input = "good"

Path constraint:

$I_0 \neq \text{'b'}$

$I_1 \neq \text{'a'}$

$I_2 \neq \text{'d'}$ → $I_2 = \text{'d'}$ → godd

$I_3 \neq \text{'!'}$ → $I_3 = \text{'!'}$ → goo!

good

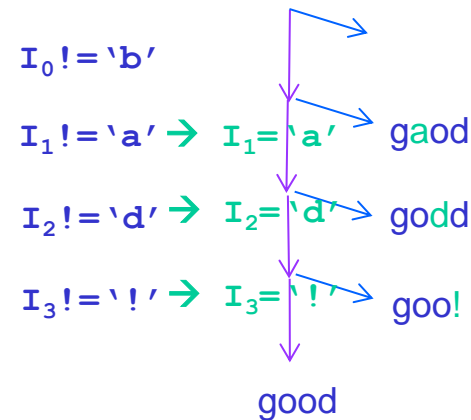
Negate each constraint in path
constraint
Solve new constraint → new input

Example: SAGE Generational Input Generation

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt > 3) crash();
}
```

input = "good"

Path constraint:



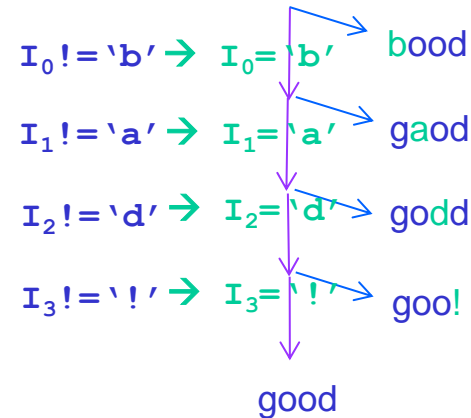
Negate each constraint in path
constraint
Solve new constraint → new input

Example: SAGE Generational Input Generation

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt > 3) crash();
}
```

input = "good"

Path constraint:



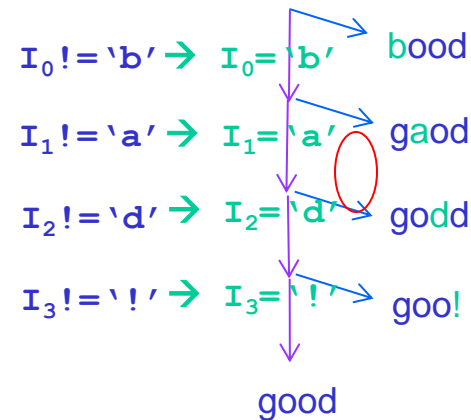
Negate each constraint in path
constraint
Solve new constraint → new input

Example: SAGE Generational Input Generation

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt > 3) crash();
}
```

input = "good"

Path constraint:



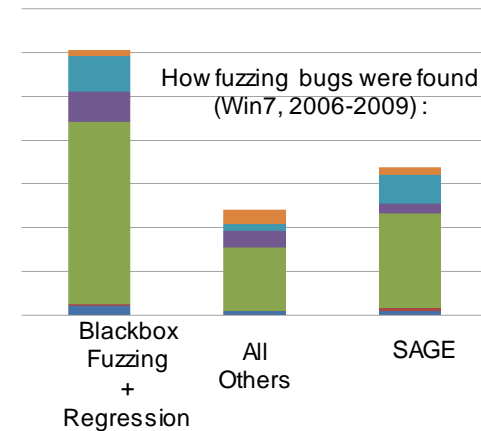
Gen 1

Negate each constraint in path
constraint
Solve new constraint → new input

Whitebox File Fuzzing

SAGE @ Microsoft:

- 1st whitebox fuzzer for security testing
- 400+ machine years (since 2008) →
- 3.4+ Billion constraints
- 100s of apps, 100s of security bugs
- Example: Win7 file fuzzing
 - ~1/3 of **all** fuzzing bugs found by SAGE →
(missed by everything else...)
- Bug fixes shipped (quietly) to 1 Billion+ PCs
- Millions of dollars saved
 - for Microsoft + time/energy for the world



DART: Implementation Considerations

Instructions are instrumented or recorded

- Concrete program execution proceeds normally

Path condition is computed based on concrete execution

- e.g., **SAGE** separates tracing and symbolic execution
- e.g., **CUTE/CREST** instruments the program to compute path condition on-the-fly during execution

Generational search

- new path conditions are computed offline for each branch point
- many branch-points means that many new inputs can be generated from a single concrete execution
- easy to parallelize and distribute in the cloud

Comparing EXE vs. DART

EXE	DART
<ul style="list-style-type: none">• Fine-grained control of execution• Shallow exploration<ul style="list-style-type: none">• Many queries early on• Online SE<ul style="list-style-type: none">• SE and interpretation in lockstep	<ul style="list-style-type: none">• Complete execution from first step• Deep exploration<ul style="list-style-type: none">• One query per run• Offline SE possible<ul style="list-style-type: none">• Execute along recorded trace

Loops, Recursion, ...

Symbolic execution can only reason about finitely many steps of execution

- but programs have loops and may execute for arbitrary many steps!

Symbolic execution dynamically unrolls loops

- treat each loop as a branch condition
 - either exit loop, or repeat
- some loops run few iterations – they are explored completely
- some loops run unbounded number of iterations
 - symbolic execution either gets stuck or skips such loops

Some loops are easy...

- only input-dependent loops are problematic
- loops with known iteration count are executed concretely

Input-dependent Loops

Unrolling in EXE – online SE

- Every iteration of the loop forks execution
- Search algorithm decides whether to continue unrolling loop or to break out

Unrolling in DART – offline SE

- Concrete input determines iterations / unrollings
- Search algorithm can flip one of the loop branches to change the number of iterations

Naïve search algorithms can get stuck in loops

DSE: Concretization

DSE executes the program both concretely and symbolically

- concrete execution is “easy”
- symbolic execution can be hard
- what to do when symbolic reasoning is too hard?!

Concretization – selectively replace symbolic inputs by concrete values

- **simplifies** symbolic reasoning
 - at the limit, all symbols are replaced, and symbolic reasoning is trivial
- somehow must decide **WHAT** to concretize
 - not always clear what makes reasoning hard
- potential source of **unsoundness**
 - all decisions (like concretization) must be recorded in the path condition
- definite source of **incompleteness**
 - bugs can be missed since some inputs are no longer symbolic

Example of Concretization in EXE

true

$$C(X) = 5$$

$$S(m) = X + 2 \quad C(m) = 7$$

$$S(\text{size}) = Y \quad C(\text{size}) = 256$$

if ($m * m > \text{size}$) {

...

Example of Concretization in EXE

true

$$C(X) = 5$$

$$S(m) = X + 2 \quad C(m) = 7$$

$$S(\text{size}) = Y \quad C(\text{size}) = 256$$

if ($m * m > \text{size}$) {

...

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

Symbolic multiplication is difficult for most SMT solvers

- select variable X to be concretized
- use its current concrete value $C(X) = 5$

Example of Concretization in EXE

true

$$C(X) = 5$$

$$S(m) = X + 2$$

$$C(m) = 7$$

$$S(\text{size}) = Y$$

$$C(\text{size}) = 256$$

if ($m * m > \text{size}$) {

...

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

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

Example of Concretization in EXE

true

$$C(X) = 5$$

$$S(m) = X + 2$$

$$C(m) = 7$$

$$S(\text{size}) = Y$$

$$C(\text{size}) = 256$$

if ($m * m > \text{size}$) {

...

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

$$49 > Y$$

To force execution into the then-branch of the if-statement

- update the path condition with current symbolic branch condition
- solve for symbolic inputs (in this case, Y)
- assume that SMT solver found a solution $Y=48$

Example of Concretization in EXE

true

$$\begin{array}{ll} C(X) = 5 \\ S(m) = X + 2 & C(m) = 7 \\ S(\text{size}) = Y & C(\text{size}) = 256 \end{array}$$

$49 > Y$

$$\begin{array}{ll} C(X) = 5 \\ S(m) = X + 2 & C(m) = 7 \\ S(\text{size}) = Y & C(\text{size}) = 48 \end{array}$$

if ($m * m > \text{size}$) {

...

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

$$49 > Y$$

Example of Concretization in EXE

true

$$\begin{array}{ll} S(m) = X + 2 & C(X) = 5 \\ S(size) = Y & C(m) = 7 \\ & C(size) = 256 \end{array}$$

if ($m * m > size$) {

...
if ($m < 5$) {
...
}

$49 > Y$

$$\begin{array}{ll} S(m) = X + 2 & C(X) = 5 \\ S(size) = Y & C(m) = 7 \\ & C(size) = 48 \end{array}$$

Example of Concretization in EXE

true

$$\begin{array}{ll} S(m) = X + 2 & C(X) = 5 \\ S(size) = Y & C(m) = 7 \\ & C(size) = 256 \end{array}$$

if ($m * m > size$) {

...
if ($m < 5$) {
...

$$X + 2 < 5$$

$49 > Y$

$$\begin{array}{ll} & C(X) = 5 \\ S(m) = X + 2 & C(m) = 7 \\ S(size) = Y & C(size) = 48 \end{array}$$

Example of Concretization in EXE

true

$$\begin{array}{ll} S(m) = X + 2 & C(X) = 5 \\ S(size) = Y & C(m) = 7 \\ & C(size) = 256 \end{array}$$

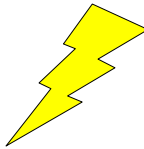
$49 > Y$

$$\begin{array}{ll} S(m) = X + 2 & C(X) = 5 \\ S(size) = Y & C(m) = 7 \\ & C(size) = 48 \end{array}$$

if ($m * m > size$) {

...
if ($m < 5$) {
...

$$X + 2 < 5$$



Solution diverges from expected path!
(e.g., if SMT solver returns $X = 2$)

Example of Concretization in EXE

true

$$\begin{array}{ll} S(m) = X + 2 & C(X) = 5 \\ S(size) = Y & C(m) = 7 \\ & C(size) = 256 \end{array}$$

if ($m * m > size$) {

...
if ($m < 5$) {
...

Concretization constraint

$49 > Y \wedge X = 5$

$$\begin{array}{ll} C(X) = 5 \\ S(m) = X + 2 & C(m) = 7 \\ S(size) = Y & C(size) = 48 \end{array}$$

Root cause of the unsoundness

- concretization decided that $X=5$
- However, this was not recorded in the path condition
- Thus, X could change to a different value later
- unsound!

Concretization: Implementation Details

Input

- concrete and symbolic states C and S
- a symbolic expression E to evaluate

Algorithm

- choose variables x_1, \dots, x_k in E to concretize
- replace x_i by $C(x_i)$ in E
- $v := S.eval(E); CC := x_1 = C(x_1) \wedge \dots \wedge x_k = C(x_k)$
- add *concretization constraint* CC to the path condition
- return v

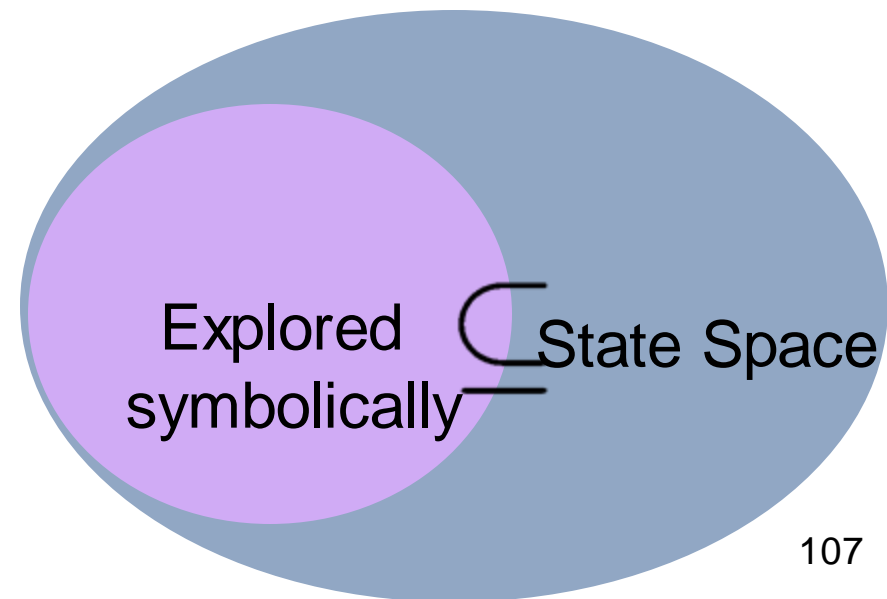
Soundness & Completeness

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



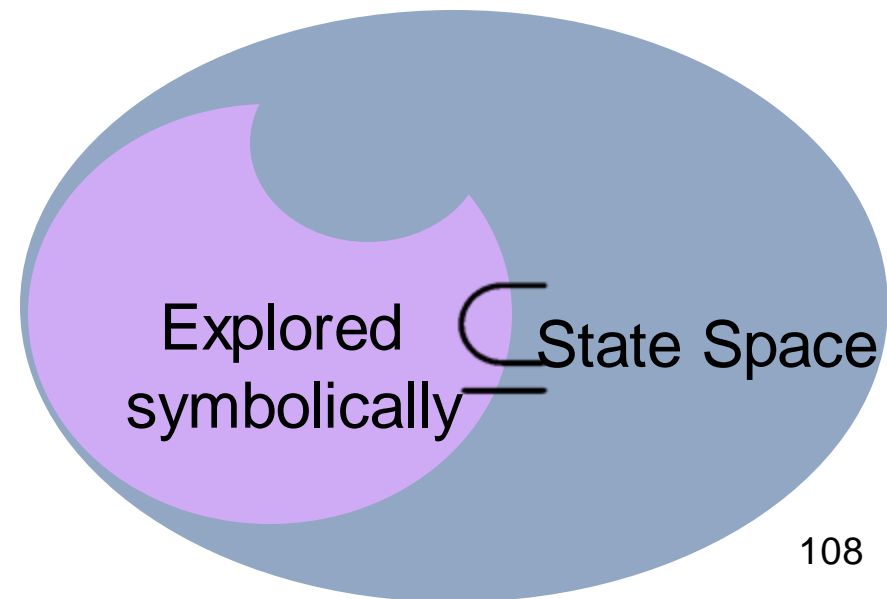
Soundness & Completeness

Symbolic Execution = Underapproximates

- Soundness = does not include infeasible behavior
- Completeness = explores all behavior

Concretization restricts state covered by path

- Remains sound
- Loses (eventual) completeness



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

Challenges for Symbolic Execution

Expensive to create representations

Expensive to reason about expressions

- Although modern SAT/SMT solvers help!

Problems with function calls – need to keep track of calling contexts

- Called *interprocedural analysis*

Problem with handling loops

- often unroll them up to a certain depth rather than dealing with termination or loop invariants

Aliasing – leads to a massive blow-up in the number of paths