Lecture 6 – Dynamic Symbolic Execution (DSE)

AAA705: Software Testing and Quality Assurance

Jihyeok Park

PLRG

2024 Spring

Recall – Search Based Software Engineering (SBSE)

- **Search Based Software Engineering (SBSE)**
  - Fitness Landscape
  - Local Search
    - Hill Climbing
    - Simulated Annealing
    - Tabu Search
  - Genetic Algorithms
    - Selection Strategies
    - Crossover Operators
    - Mutation Operators
  - Bio-inspired Algorithms
    - Particle Swarm Optimization (PSO)
    - Ant Colony Optimization (ACO)
- **Search Based Software Testing (SBST)**
  - Alternating Variable Method (AVM)

Recall – White-Box (Structural) Testing

Sometimes called **structural testing** because it uses the **internal structure** of the program to derive test cases.

- **Coverage Criteria**
  - The adequacy of a test suite is measured in terms of the **coverage** of the program's internal structure.
- **Search Based Software Testing (SBST)**
  - A technique that uses **meta-heuristic search** algorithms to maximize/minimize a certain **fitness function**.
- **Dynamic Symbolic Execution (DSE)**
  - A technique that systematically explores the input space using **symbolic execution** with **dynamic analysis**.

Let's focus on the **Dynamic Symbolic Execution (DSE)** technique.

Contents

Contents

Limitations of Random Testing

- Random testing has a **limitation** that it sometimes fails or takes a long time to find bugs if they can only be triggered **under specific conditions**.
- For example, consider the following program.

```
void testme (int x) {
  if (x == 93589) {
    ERROR
  }
}
```

- The bug can only be triggered when the input is `x = 93589`.
- It means that the probability of triggering the bug is as follows when the integer input is randomly generated:

$$\frac{1}{2^{32}} \approx 0.00000000023283\%$$

Symbolic Execution

PLRG

- **Symbolic execution** is a program analysis technique that explores the paths of a program with **symbolic values** as inputs and collects **constraints** on the inputs.

- **1976** – A system to generate test data and symbolically execute programs (Lori Clarke).

- **1977** – Symbolic execution and program testing (James King).

- **2005 to present** – Practical symbolic execution
  - Using advanced constraint solvers (SMT solvers)
  - Heuristics to control exponential path explosion
  - Heap modeling and reasoning about complex data structures
  - Environment modeling
  - Dealing with solver limitations

Symbolic Execution – Basic Idea

1. Execute the program on **symbolic values** Ω.

2. A **symbolic state** $\sigma \in \mathbb{S} = \mathbb{X} \xrightarrow{\text{fin}} \Omega$ is a finite mapping from variables $\mathbb{X}$ to symbolic values Ω.

3. A **path condition** $\Phi = \phi_1 \wedge \ldots \wedge \phi_n$ is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

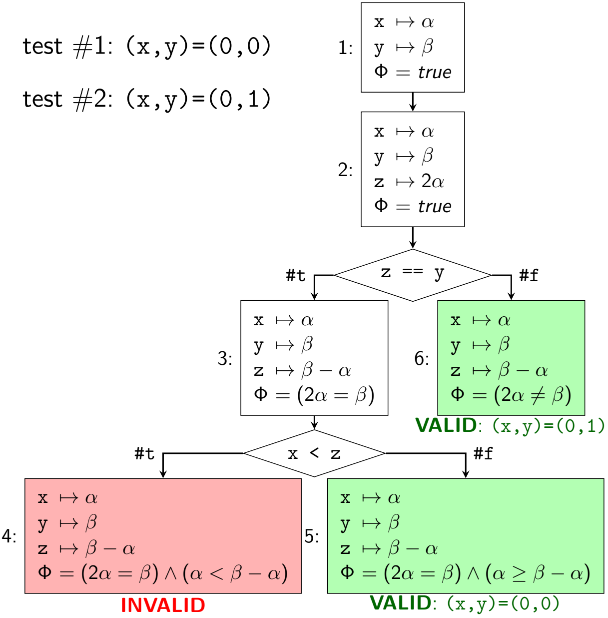
4. All paths in the program form its **execution tree**, in which some paths are **feasible** and some are **infeasible**.

Symbolic Execution – Basic Idea

```
void f(int x, int y) {
  /* 1 */
  int z = 2 * x;
  /* 2 */
  if (z == y) {
    z = y - x;
    /* 3 */
    if (x < z) {
      /* 4 */
      ERROR;
    } else {
      /* 5 */
    }
  } else {
    /* 6 */
  }
}
```

test #1: (x,y)=(0,0)

test #2: (x,y)=(0,1)



Satisfiability Modulo Theories (SMT)

- Then, how to check the **satisfiability** of a path condition Φ?

- Most symbolic execution tools use **Satisfiability Modulo Theories (SMT)** solvers (e.g., Z3, CVC4) to check it.

- An SMT solver takes a **first-order logic formula** and returns whether it is **satisfiable** or not using various background theories, such as arithmetic, arrays, bit-vectors, algebraic data types, etc.

Satisfiability Modulo Theories (SMT) – Example

- Check the satisfiability of the following formula using an SMT solver.

$$b + 2 = c \wedge f(\mathit{read}(\mathit{write}(a, b, 3), c - 2)) \neq f(c - b + 1)$$

- **Substitute** c by $b + 2$ in the second part of the formula.

$$b + 2 = c \wedge f(\mathit{read}(\mathit{write}(a, b, {\color{red}b + 2} - 2), b)) \neq f({\color{red}b + 2} - b + 1)$$

- **Arithmetic simplification** of the formula.

$$b + 2 = c \wedge f(\mathit{read}(\mathit{write}(a, {\color{red}b}, 3), b)) \neq f({\color{red}3})$$

- **Applying array theory axiom** – $\forall a, i, v.\ \mathit{read}(\mathit{write}(a, i, v), i) = v$.

$$b + 2 = c \wedge {\color{red}f(3) \neq f(3)} \qquad ({\color{red}\text{UNSAT}})$$

Satisfiability Modulo Theories (SMT) – Example

Z3[1] is one of the most popular SMT solvers developed by Microsoft Research and used in many symbolic execution tools.

For example, the following Z3 script returns `unsat`.

```
(declare-const a (Array Int Int))
(declare-const b Int)
(declare-const c Int)
(declare-fun f (Int) Int)
(assert (= (+ b 2) c))
(assert (not (= (f (select (store a b 3) (- c 2))) (f (+ (- c b) 1)))))
(check-sat) ; => unsat
```

Or, the following script returns a satisfying assignment $x = 0$ and $y = 0$.

```
(declare-const x Int)
(declare-const y Int)
(assert (and (= (* 2 x) y) (>= x (- y x))))
(check-sat) ; => sat
(get-model) ; => (x, y) = (0, 0)
```

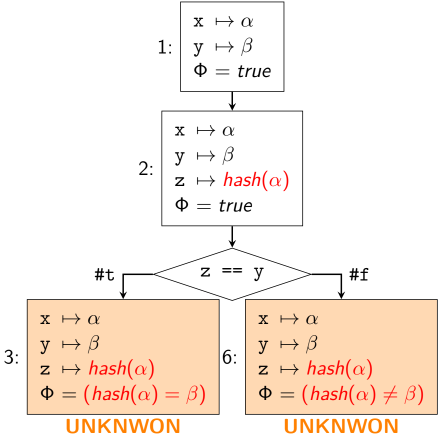
[1] https://github.com/Z3Prover/z3

Limitations of Symbolic Execution

PLRG

```
void f(int x, int y) {
  /* 1 */
  int z = hash(x);
  /* 2 */
  if (z == y) {
    z = y - x;
    /* 3 */
    if (x < z) {
      /* 4 */
      ERROR;
    } else {
      /* 5 */
    }
  } else {
    /* 6 */
  }
}
```

We cannot solve path condition Φ containing the `hash` function using SMT solver.



Contents

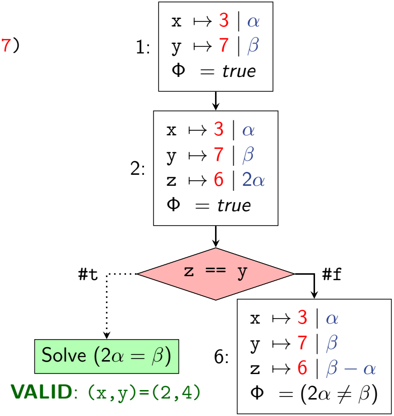
Dynamic Symbolic Execution (DSE)

- **Dynamic Symbolic Execution (DSE)** is a technique that combines **concrete execution** with **symbolic execution** to overcome the limitations of symbolic execution.
- It is sometimes called **concolic testing** because it combines both **conc**rete and symb**olic** execution to generate **test cases**.
- It stores both the **concrete** values and the **symbolic** values during the execution of the program, and solves the path condition to **guide execution** at branch points.
- The concrete values are used to **simplify** the path condition.

Dynamic Symbolic Execution (DSE) – Example

```
void f(int x, int y) {
  /* 1 */
  int z = 2 * x;
  /* 2 */
  if (z == y) {
    z = y - x;
    /* 3 */
    if (x < z) {
      /* 4 */
      ERROR;
    } else {
      /* 5 */
    }
  } else {
    /* 6 */
  }
}
```

test #1: (x,y)=(3,7)

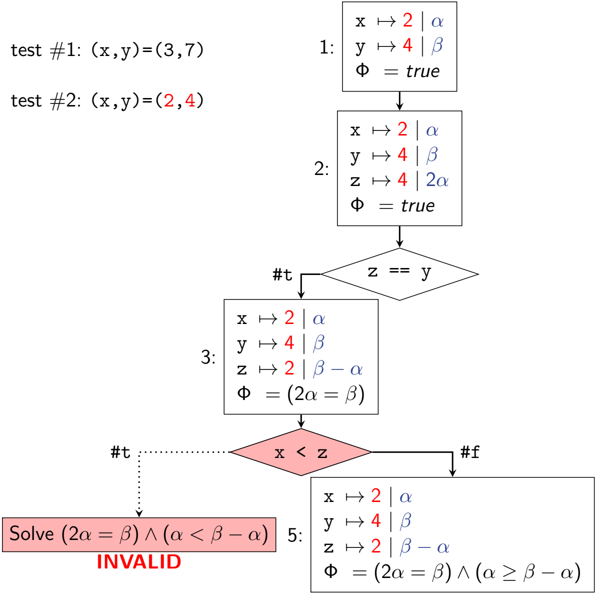


Dynamic Symbolic Execution (DSE) – Example

```
void f(int x, int y) {
  /* 1 */
  int z = 2 * x;
  /* 2 */
  if (z == y) {
    z = y - x;
    /* 3 */
    if (x < z) {
      /* 4 */
      ERROR;
    } else {
      /* 5 */
    }
  } else {
    /* 6 */
  }
}
```

test #1: (x,y)=(3,7)

test #2: (x,y)=(2,4)



Dynamic Symbolic Execution (DSE)

Algorithm 1. Concolic Testing

Input : Program P, budget N, initial input v_0
Output : The number of branches covered

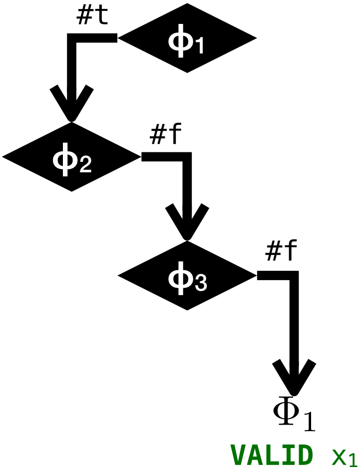
$T \leftarrow \langle\rangle$
$v \leftarrow v_0$
for $m = 1$ to N **do**
    $\Phi_m \leftarrow \mathsf{RunProgram}(P, v)$
    $T \leftarrow T \cdot \Phi_m$
    repeat
        $(\Phi, \phi_i) \leftarrow \mathsf{Choose}(T)$ $(\Phi = \phi_1 \wedge \cdots \wedge \phi_n)$
    until $\mathsf{SAT}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$
    $v \leftarrow \mathsf{model}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$
return $|\mathsf{Branches}(T)|$

Dynamic Symbolic Execution (DSE)

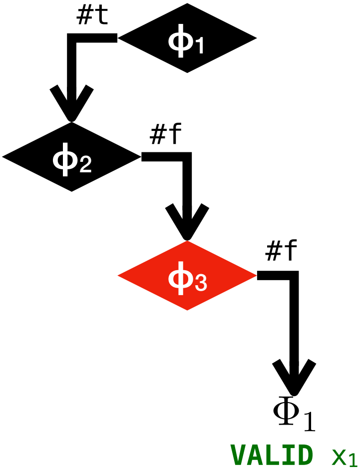
In each iteration, DSE **chooses** a path to explore based on the path condition Φ using a specific **search heuristic**.

- **Random Search** – Randomly selects a branch from the most recently visited execution path.
- **Control-Flow Directed Search (CFDS)** – Selects the uncovered branch **closest** to the last branch in the current execution path.
- **Context-Guided Search (CGS)** – Performs a **breadth-first search (BFS)** on the execution tree by excluding branches whose **contexts** (i.e., the last d preceding branches) are already explored.
- **Depth-First Search (DFS)**
- etc.

This is the **control-flow directed search (CFDS)** heuristic.

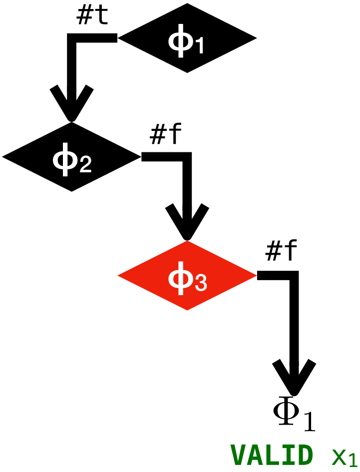


This is the **control-flow directed search (CFDS)** heuristic.



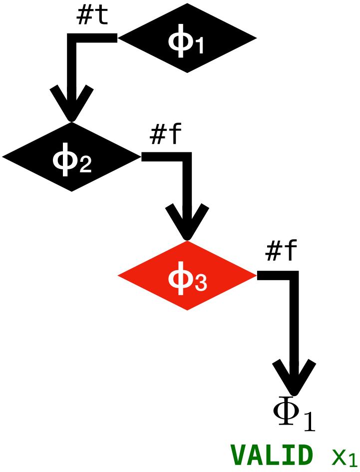
Search Heuristics in DSE

This is the **control-flow directed search (CFDS)** heuristic.



Solve $\phi_1 \land \neg\phi_2 \land \phi_3$

This is the **control-flow directed search (CFDS)** heuristic.

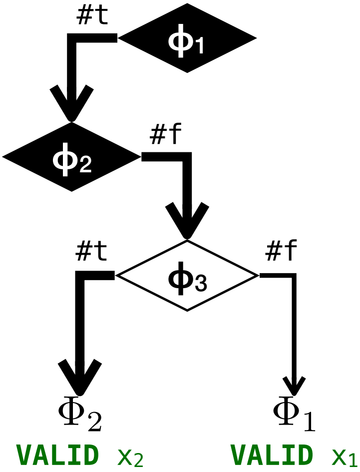


Solve $\phi_1 \land \neg\phi_2 \land \phi_3$

VALID x_2

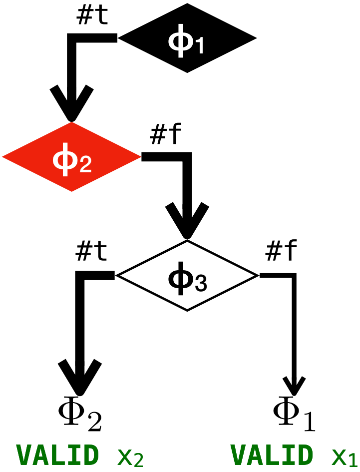
Search Heuristics in DSE

This is the **control-flow directed search (CFDS)** heuristic.



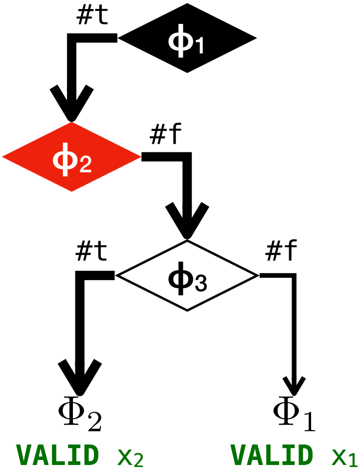
Search Heuristics in DSE

This is the **control-flow directed search (CFDS)** heuristic.



Search Heuristics in DSE

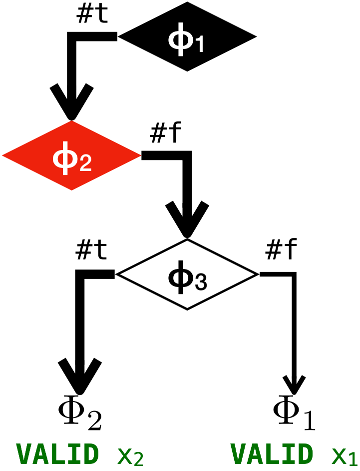
This is the **control-flow directed search (CFDS)** heuristic.



`Solve` $\phi_1 \land \phi_2$

Search Heuristics in DSE

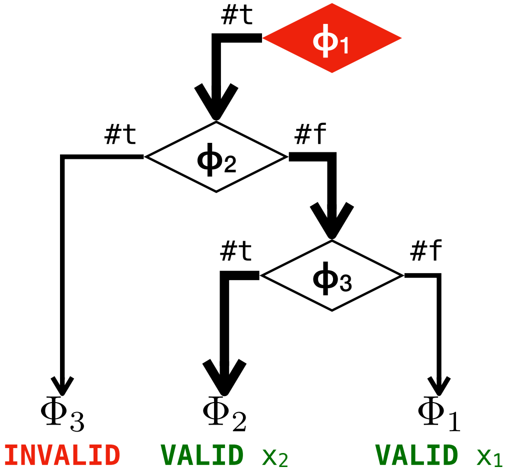
This is the **control-flow directed search (CFDS)** heuristic.



Solve $\phi_1 \land \phi_2$

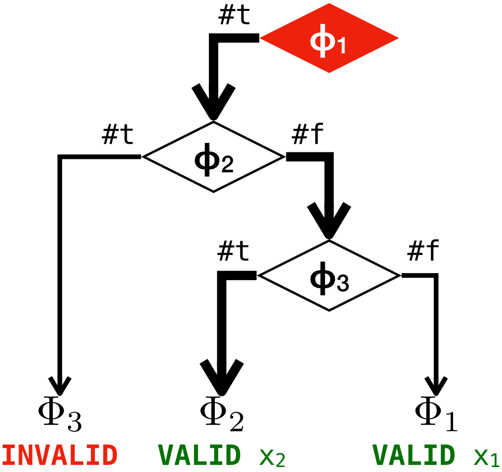
INVALID

This is the **control-flow directed search (CFDS)** heuristic.



Search Heuristics in DSE

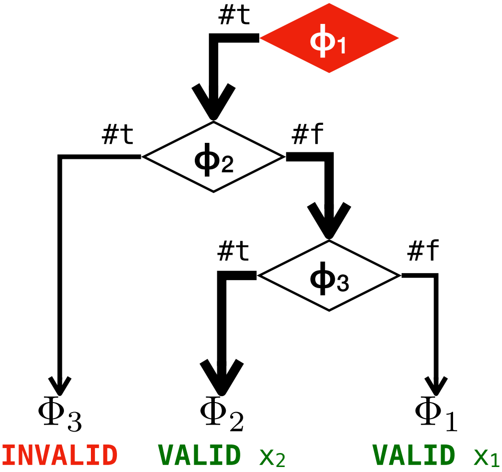
This is the **control-flow directed search (CFDS)** heuristic.



Solve $\neg\phi_1$

Search Heuristics in DSE

This is the **control-flow directed search (CFDS)** heuristic.

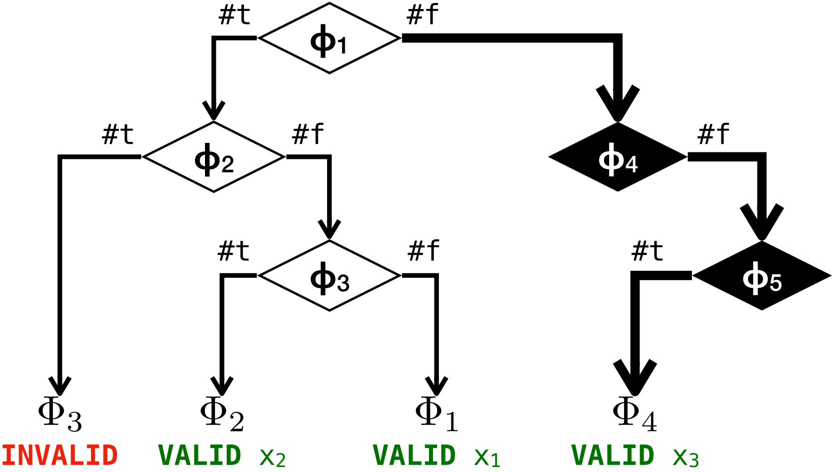


Solve $\neg\phi_1$

VALID x_3

Search Heuristics in DSE

This is the **control-flow directed search (CFDS)** heuristic.



Learning Search Heuristics in DSE

[ICSE'18] S. Cha et al., "Automatically Generating Search Heuristics for Concolic Testing"

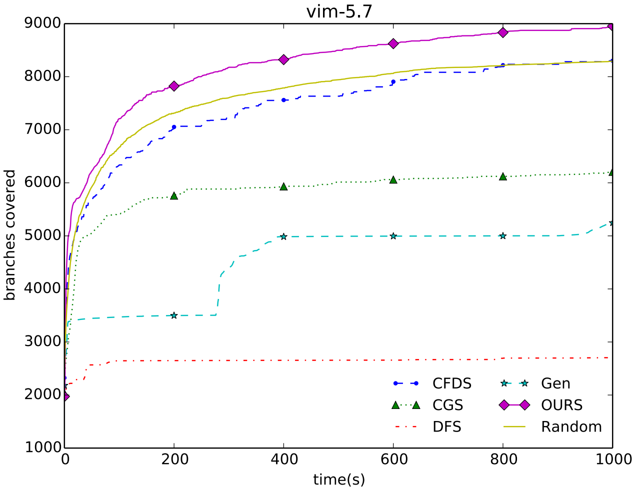
$$Choose_\theta(\langle \Phi_1, \ldots, \Phi_m \rangle) = (\Phi_m, \operatorname*{argmax}_{\phi_j \in \Phi_m} score_\theta(\phi_j))$$

Followings are 12 static and 28 dynamic branch features used for learning.

#	Description
1	branch in the main function
2	true branch of a loop
3	false branch of a loop
4	nested branch
5	branch containing external function calls
6	branch containing integer expressions
7	branch containing constant strings
8	branch containing pointer expressions
9	branch containing local variables
10	branch inside a loop body
11	true branch of a case statement
12	false branch of a case statement
13	first 10% branches of a path
14	last 10% branches of a path
15	branch appearing most frequently in a path

Learning Search Heuristics in DSE

[ICSE'18] S. Cha et al., "Automatically Generating Search Heuristics for Concolic Testing"



Example – Hash Function

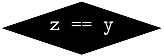
```
void f(int x, int y) {
  *
  int z = hash(x);

  if (z == y) {
    z = y - x;

    if (x < z) {

      ERROR;
    } else {

    }
  } else {

  }
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	3	α
y	7	β
z		
Φ	*true*	

Example – Hash Function

```
void f(int x, int y) {

  int z = hash(x);
  *
  if (z == y) {
    z = y - x;

    if (x < z) {

      ERROR;
    } else {

    }
  } else {

  }
}
```

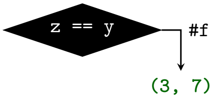
$\mathbb{X}$	$\mathbb{V}$	Ω
x	3	α
y	7	β
z	182039482	$hash(\alpha)$
Φ	*true*	



Example – Hash Function

```
void f(int x, int y) {

  int z = hash(x);

  if (z == y) {
    z = y - x;

    if (x < z) {

      ERROR;
    } else {

    }
  } else {
    *
  }
}
```

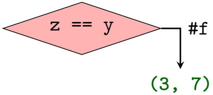
$\mathbb{X}$	$\mathbb{V}$	Ω
x	3	α
y	7	β
z	182039482	$hash(\alpha)$
Φ	$(hash(\alpha) \neq \beta)$	



Example – Hash Function

```
void f(int x, int y) {

  int z = hash(x);

  if (z == y) {
    z = y - x;

    if (x < z) {

      ERROR;
    } else {

    }
  } else {
    *
  }
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	3	α
y	7	β
z	182039482	$hash(\alpha)$
Φ	$(hash(\alpha) \neq \beta)$	



We can utilize the current concrete values.

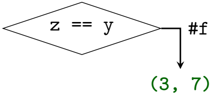
$(hash(\alpha) = \beta)$ is **SAT**

when `(x, y) = (3, 182039482)`.

Example – Hash Function

```
void f(int x, int y) {
  *
  int z = hash(x);

  if (z == y) {
    z = y - x;

    if (x < z) {

      ERROR;
    } else {

    }
  } else {

  }
}
```

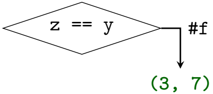
$\mathbb{X}$	$\mathbb{V}$	Ω
x	3	α
y	182039482	β
z		
Φ	*true*	



Example – Hash Function

```
void f(int x, int y) {

  int z = hash(x);
  *
  if (z == y) {
    z = y - x;

    if (x < z) {

      ERROR;
    } else {

    }
  } else {

  }
}
```

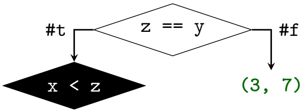
$\mathbb{X}$	$\mathbb{V}$	Ω
x	3	α
y	182039482	β
z	182039482	$hash(\alpha)$
Φ	*true*	



Example – Hash Function

```
void f(int x, int y) {

  int z = hash(x);

  if (z == y) {
    z = y - x;
     *
    if (x < z) {

      ERROR;
    } else {

    }
  } else {

  }
}
```

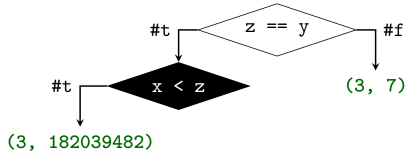
$\mathbb{X}$	$\mathbb{V}$	Ω
x	3	α
y	182039482	β
z	182039479	$\beta - \alpha$
Φ	$(hash(\alpha) = \beta)$	



Example – Hash Function

```
void f(int x, int y) {

  int z = hash(x);

  if (z == y) {
    z = y - x;

    if (x < z) {
      *
      ERROR;
    } else {

    }
  } else {

  }
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	3	α
y	182039482	β
z	182039479	$\beta - \alpha$
Φ	$(hash(\alpha) = \beta) \land (\alpha < \beta - \alpha)$	

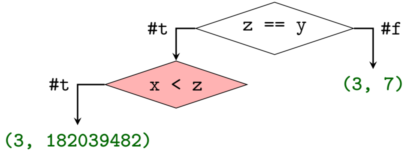


We found an error!

Example – Hash Function

```
void f(int x, int y) {

  int z = hash(x);

  if (z == y) {
    z = y - x;

    if (x < z) {
      *
      ERROR;
    } else {

    }
  } else {

  }
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	3	α
y	182039482	β
z	182039479	$\beta - \alpha$
Φ	$(hash(\alpha) = \beta) \wedge (\alpha < \beta - \alpha)$	



Unfortunately,
$(hash(\alpha) = \beta) \wedge (\alpha \geq \beta - \alpha)$ is
UNKNOWN.

Example – Loops

```
void f(int x) {
  *
  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {

    if (arr[i] == x) break;
    i++;

  }

  return i;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr		
i		
Φ	*true*	

Example – Loops

```
void f(int x) {

  int arr[] = {3, 7, 2};
  int i = 0;
   *
  while (i < 3) {

    if (arr[i] == x) break;
    i++;

  }

  return i;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr	$\{3, 7, 2\}$	
i	0	
Φ	$true$	

Example – Loops

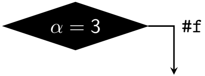
```
void f(int x) {

  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {
    *
    if (arr[i] == x) break;
    i++;

  }

  return i;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr	$\{3, 7, 2\}$	
i	0	
Φ	*true*	

Example – Loops

```
void f(int x) {

  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {

    if (arr[i] == x) break;
    i++;
    *
  }

  return i;
}
```

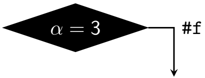
$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr	$\{3, 7, 2\}$	
i	1	
Φ	$(\alpha \neq 3)$	



Example – Loops

```
void f(int x) {

  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {
    *
    if (arr[i] == x) break;
    i++;

  }

  return i;
}
```

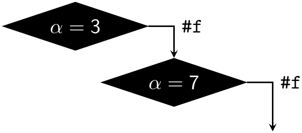
$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr	$\{3, 7, 2\}$	
i	1	
Φ	$(\alpha \neq 3)$	



Example – Loops

```
void f(int x) {

  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {

    if (arr[i] == x) break;
    i++;
    *
  }

  return i;
}
```

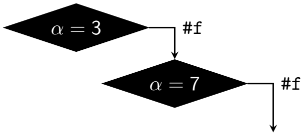
$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr	$\{3, 7, 2\}$	
i	2	
Φ	$(\alpha \neq 3) \wedge (\alpha \neq 7)$	



Example – Loops

```
void f(int x) {

  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {
    *
    if (arr[i] == x) break;
    i++;

  }

  return i;
}
```

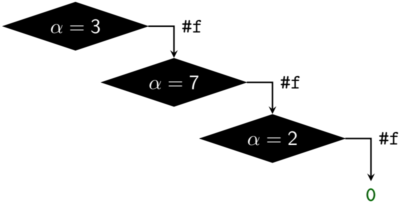
$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr	$\{3, 7, 2\}$	
i	2	
Φ	$(\alpha \neq 3) \wedge (\alpha \neq 7)$	



Example – Loops

```
void f(int x) {

  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {

    if (arr[i] == x) break;
    i++;
    *
  }

  return i;
}
```

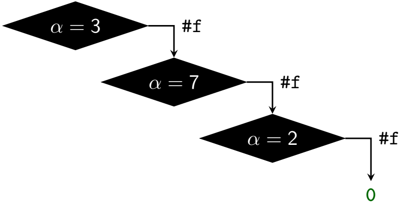
$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr	$\{3, 7, 2\}$	
i	3	
Φ	$(\alpha \neq 3) \wedge (\alpha \neq 7) \wedge (\alpha \neq 2)$	



Example – Loops

```
void f(int x) {

  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {

    if (arr[i] == x) break;
    i++;

  }
   *
  return i;
}
```

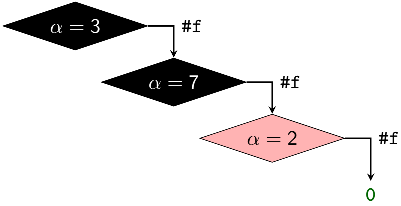
$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr	$\{3, 7, 2\}$	
i	3	
Φ	$(\alpha \neq 3) \land (\alpha \neq 7) \land (\alpha \neq 2)$	



Example – Loops

```
void f(int x) {

  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {

    if (arr[i] == x) break;
    i++;

  }
   *
  return i;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr	$\{3, 7, 2\}$	
i	3	
Φ	$(\alpha \neq 3) \wedge (\alpha \neq 7) \wedge (\alpha \neq 2)$	



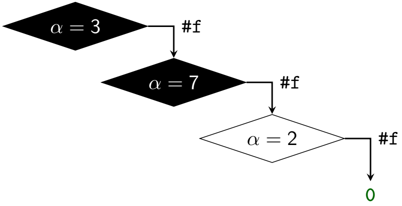
$(\alpha \neq 3) \wedge (\alpha \neq 7) \wedge (\alpha = 2)$ is **SAT**

when `x = 2`

Example – Loops

```
void f(int x) {
  *
  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {

    if (arr[i] == x) break;
    i++;

  }

  return i;
}
```

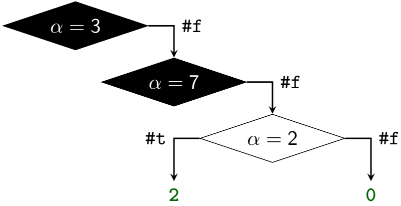
$\mathbb{X}$	$\mathbb{V}$	Ω
x	2	α
arr		
i		
Φ	*true*	



Example – Loops

```
void f(int x) {

  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {

    if (arr[i] == x) break;
    i++;

  }
   *
  return i;
}
```

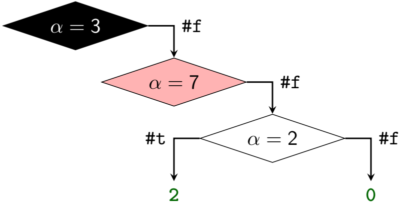
$\mathbb{X}$	$\mathbb{V}$	Ω
x	2	α
arr	$\{3, 7, 2\}$	
i	2	
Φ	$(\alpha \neq 3) \land (\alpha \neq 7) \land (\alpha = 2)$	



Example – Loops

```
void f(int x) {

  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {

    if (arr[i] == x) break;
    i++;

  }
   *
  return i;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	2	α
arr	$\{3, 7, 2\}$	
i	2	
Φ	$(\alpha \neq 3) \wedge (\alpha \neq 7) \wedge (\alpha = 2)$	



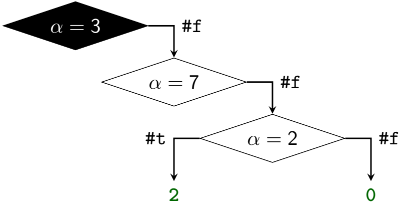
$(\alpha \neq 3) \wedge (\alpha = 7)$ is **SAT**

when `x = 7`

Example – Loops

```
void f(int x) {
  *
  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {

    if (arr[i] == x) break;
    i++;

  }

  return i;
}
```

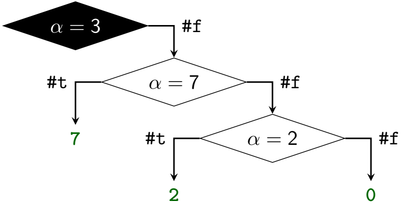
$\mathbb{X}$	$\mathbb{V}$	Ω
x	7	α
arr		
i		
Φ	*true*	



Example – Loops

```
void f(int x) {

  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {

    if (arr[i] == x) break;
    i++;

  }
   *
  return i;
}
```

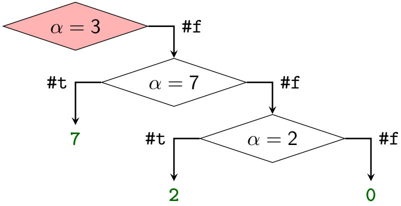
$\mathbb{X}$	$\mathbb{V}$	Ω
x	7	α
arr	$\{3, 7, 2\}$	
i	1	
Φ	$(\alpha \neq 3) \land (\alpha = 7)$	



Example – Loops

```
void f(int x) {

  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {

    if (arr[i] == x) break;
    i++;

  }
   *
  return i;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	7	α
arr	$\{3, 7, 2\}$	
i	1	
Φ	$(\alpha \neq 3) \wedge (\alpha = 7)$	



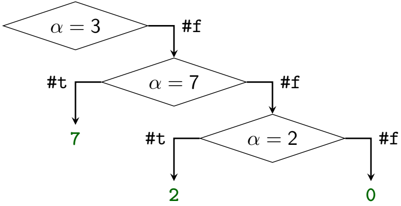
$(\alpha = 3)$ is **SAT**

when `x = 3`

Example – Loops

```
void f(int x) {
  *
  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {

    if (arr[i] == x) break;
    i++;

  }

  return i;
}
```

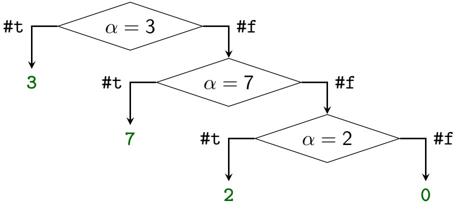
$\mathbb{X}$	$\mathbb{V}$	Ω
x	3	α
arr		
i		
Φ	*true*	



Example – Loops

```
void f(int x) {

  int arr[] = {3, 7, 2};
  int i = 0;

  while (i < 3) {

    if (arr[i] == x) break;
    i++;

  }
  *
  return i;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	3	α
arr	$\{3, 7, 2\}$	
i	0	
Φ	$(\alpha = 3)$	



Example – Data Structures

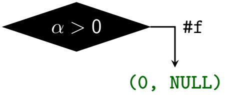
```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {
  *
  if (x > 0)

    if (p != NULL)

      if (x*2+1 == p->data)

        if (p->next == p)

          ERROR;

  return 0;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
p	*NULL*	β
Φ	*true*	

Example – Data Structures

```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {

  if (x > 0)

    if (p != NULL)

      if (x*2+1 == p->data)

        if (p->next == p)

          ERROR;
  *
  return 0;
}
```

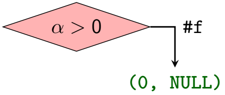
$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
p	$NULL$	β
Φ	$(\alpha \leq 0)$	



Example – Data Structures


```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {

  if (x > 0)

    if (p != NULL)

      if (x*2+1 == p->data)

        if (p->next == p)

          ERROR;
  *
  return 0;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
p	$NULL$	β
Φ	$(\alpha \leq 0)$	

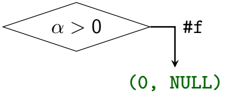


$(\alpha > 0)$ is **SAT**
when $(x, p) = (42, NULL)$

Example – Data Structures

```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {
  *
  if (x > 0)

    if (p != NULL)

      if (x*2+1 == p->data)

        if (p->next == p)

          ERROR;

  return 0;
}
```

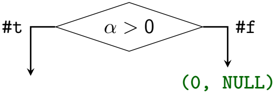
$\mathbb{X}$	$\mathbb{V}$	Ω
x	42	α
p	*NULL*	β
Φ	*true*	



Example – Data Structures

```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {

  if (x > 0)
    *
    if (p != NULL)

      if (x*2+1 == p->data)

        if (p->next == p)

          ERROR;

  return 0;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	42	α
p	*NULL*	β
Φ	$(\alpha > 0)$	

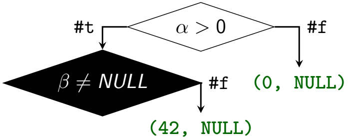


Example – Data Structures

PLRG

```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {

  if (x > 0)

    if (p != NULL)

      if (x*2+1 == p->data)

        if (p->next == p)

          ERROR;
  *
  return 0;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	42	α
p	$NULL$	β
Φ	$(\alpha > 0) \land (\beta = NULL)$	

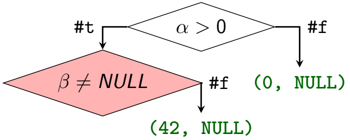


Example – Data Structures

PLRG

```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {

  if (x > 0)

    if (p != NULL)

      if (x*2+1 == p->data)

        if (p->next == p)

          ERROR;
  *
  return 0;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	42	α
p	$NULL$	β
Φ	$(\alpha > 0) \land (\beta = NULL)$	



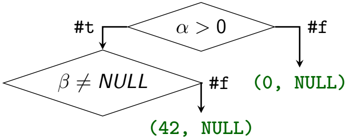
$(\alpha > 0) \land (\beta \neq NULL)$ is **SAT**

when $(x, p) = (42, \text{0xA0BF} : \boxed{0 \mid NULL})$

Example – Data Structures

```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {
  *
  if (x > 0)

    if (p != NULL)

      if (x*2+1 == p->data)

        if (p->next == p)

          ERROR;

  return 0;
}
```

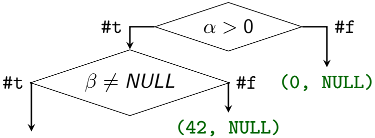
$\mathbb{X}$	$\mathbb{V}$	Ω
x	42	α
p	0xA0BF : $\boxed{0 \mid \mathit{NULL}}$	β
Φ	*true*	



Example – Data Structures

```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {

  if (x > 0)

    if (p != NULL)
      *
      if (x*2+1 == p->data)

        if (p->next == p)

          ERROR;

  return 0;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	42	α
p	0xA0BF : | 0 | *NULL* |	β
Φ	$(\alpha > 0) \wedge (\beta \neq NULL)$	

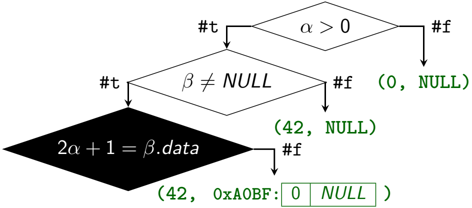


Example – Data Structures

PLRG

```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {

  if (x > 0)

    if (p != NULL)

      if (x*2+1 == p->data)

        if (p->next == p)

          ERROR;
  *
  return 0;
}
```

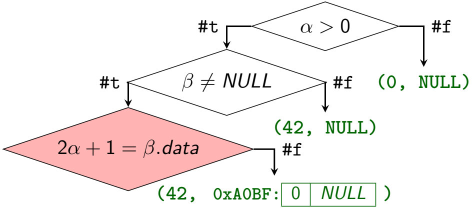
$\mathbb{X}$	$\mathbb{V}$	Ω
x	42	α
p	0xA0BF : $\boxed{0 \mid NULL}$	β
Φ	$(\alpha > 0) \land (\beta \neq NULL) \land (2\alpha + 1 \neq \beta.data)$	



Example – Data Structures

```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {

  if (x > 0)

    if (p != NULL)

      if (x*2+1 == p->data)

        if (p->next == p)

          ERROR;
  *
  return 0;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	42	α
p	0xA0BF : $\boxed{0 \mid NULL}$	β
Φ	$(\alpha > 0) \wedge (\beta \neq NULL) \wedge$ $(2\alpha + 1 \neq \beta.data)$	

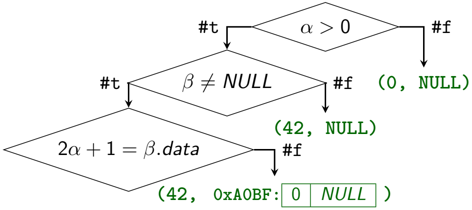


$$(\alpha > 0) \wedge (\beta \neq NULL)$$
$$\wedge (2\alpha + 1 = \beta.data) \text{ is } \textbf{SAT}$$
when $(x, p) = (1, \text{0xA0BF} : \boxed{3 \mid NULL})$

Example – Data Structures

```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {
  *
  if (x > 0)

    if (p != NULL)

      if (x*2+1 == p->data)

        if (p->next == p)

          ERROR;

  return 0;
}
```

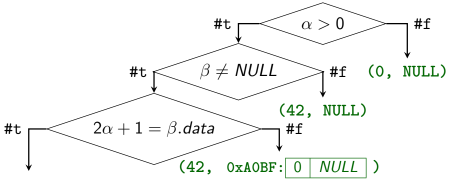
$\mathbb{X}$	$\mathbb{V}$	Ω
x	1	α
p	0xA0BF : | 3 | *NULL* |	β
Φ	*true*	



Example – Data Structures

```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {

  if (x > 0)

    if (p != NULL)

      if (x*2+1 == p->data)
       *
        if (p->next == p)

          ERROR;

  return 0;
}
```

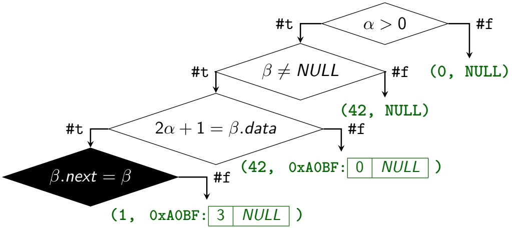
$\mathbb{X}$	$\mathbb{V}$	Ω
x	1	α
p	0xA0BF : | 3 | *NULL* |	β
Φ	$(\alpha > 0) \wedge (\beta \neq NULL) \wedge (2\alpha + 1 = \beta.data)$	



Example – Data Structures

```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {

  if (x > 0)

    if (p != NULL)

      if (x*2+1 == p->data)

        if (p->next == p)

          ERROR;
  *
  return 0;
}
```

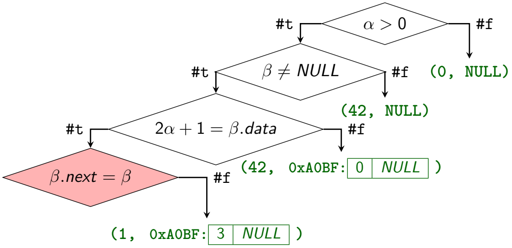
$\mathbb{X}$	$\mathbb{V}$	Ω
x	1	α
p	0xA0BF : $\boxed{3 \mid \mathit{NULL}}$	β
Φ	$(\alpha > 0) \wedge (\beta \neq \mathit{NULL}) \wedge$ $(2\alpha + 1 = \beta.data) \wedge (\beta.next \neq \beta)$	



Example – Data Structures

```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {

  if (x > 0)

    if (p != NULL)

      if (x*2+1 == p->data)

        if (p->next == p)

          ERROR;
  *
  return 0;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	1	α
p	0xA0BF : $\boxed{3 \mid NULL}$	β
Φ	$(\alpha > 0) \land (\beta \neq NULL) \land$ $(2\alpha + 1 = \beta.data) \land (\beta.next \neq \beta)$	

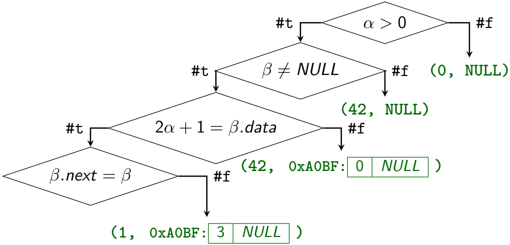


$(\alpha > 0) \land (\beta \neq NULL)$
$\land(2\alpha + 1 = \beta.data)$ is **SAT**
when $(x, p) = (1, 0\text{xA0BF} : \boxed{3 \mid 0xA0BF})$

Example – Data Structures

```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {
  *
  if (x > 0)

    if (p != NULL)

      if (x*2+1 == p->data)

        if (p->next == p)

          ERROR;

  return 0;
}
```

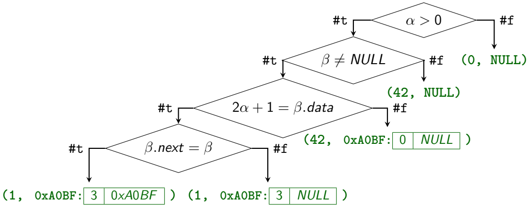
$\mathbb{X}$	$\mathbb{V}$	Ω
x	1	α
p	0xA0BF : [3 | $0xA0BF$]	β
Φ	*true*	



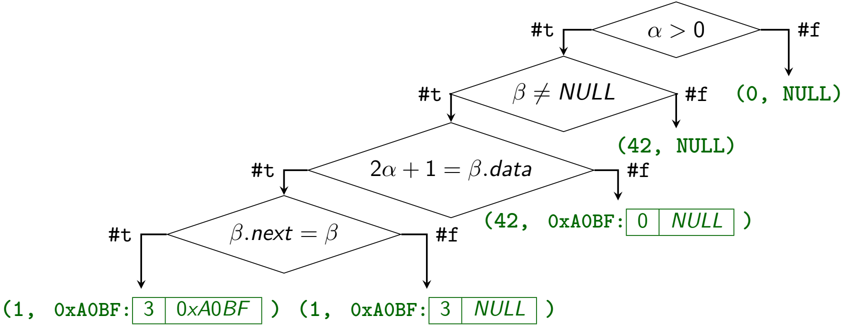
Example – Data Structures

```
class Node {
  int data;
  Node* next;
};

void f(int x, Node *p) {

  if (x > 0)

    if (p != NULL)

      if (x*2+1 == p->data)

        if (p->next == p)
          *
          ERROR;

  return 0;
}
```

$\mathbb{X}$	$\mathbb{V}$	Ω
x	1	α
p	0xA0BF : $\boxed{3 \mid 0xA0BF}$	β
Φ	$(\alpha > 0) \wedge (\beta \neq NULL) \wedge (2\alpha + 1 = \beta.data) \wedge (\beta.next = \beta)$	



Example – Data Structures



Realistic Implementation

- **KLEE** – LLVM based DSE engine.
- **Jalangi2** – JavaScript dynamic analysis framework by Samsung.
- **S2E** – Platform for symbolic execution of binary code (x86, ARM).
- **CutEr** – Concolic unit testing tool for Erlang.

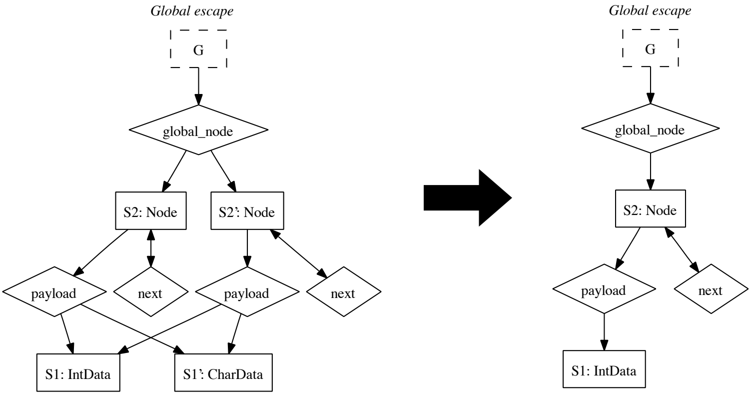
Other Hybrid Analysis Techniques

- Many efforts have combined dynamic and static analysis, yielding **blended** or **hybrid** analysis techniques.
- Some hybrid analyses use information recorded during dynamic executions of the program to improve the static-analysis precision. However, they are **generally unsound** because they rely on incomplete information.
- Some recent hybrid analyses have been proposed a **combined interpretation** to address the unsoundness issue, and it interchanges between **concrete** and **abstract interpretations** to improve the precision.

Blended Analysis

[ISSTA'07] Dufour et al., "Blended analysis for performance understanding of framework-based applications"

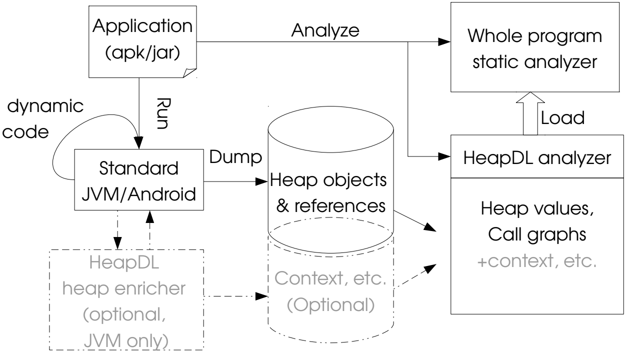
- **Escape Analysis** – A static analysis technique that determines whether an object can escape the scope of the method or a thread.



Heap Snapshots

[OOPSLA'17] Grech et al., "Heaps Don't Lie: Countering Unsoundness with Heap Snapshots"

- **Heap Snapshots** – A technique that records the whole-heap snapshots during program execution and then uses them to guide the static analysis. (especially for Android and JVM applications)



Just-in-time Static Type Checking

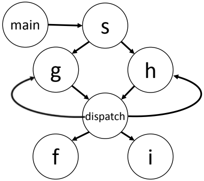
[PLDI'16] Briana M. Ren and Jeffrey S. Foster, "Just-in-time Static Type Checking for Dynamic Languages"

- **Just-in-time Static Type Checking** – A technique that type checks Ruby code even in the presence of meta-programming.
- It **dynamically gathers method type signatures** and utilizes them in **static type checking** of their body when they are invoked.

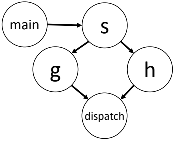
```
class Talk < ActiveRecord::Base
  belongs_to :owner, :class_name ⇒ "User"
  ....
  type :owner?, "(User) → %bool"
  def owner?(user)
    return owner == user
end end
```

Combined Interpretation

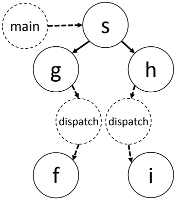
[POPL'18] J. Toman and D. Grossman, "Concerto: a framework for combined concrete and abstract interpretation"



(a) Sound but Imprecise Call-Graph



(b) Unsound Call-Graph



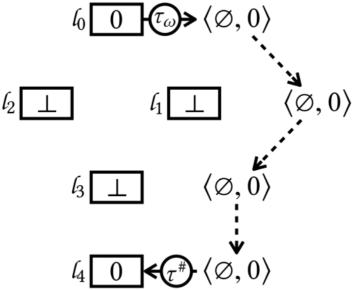
(c) Call-Graph with Concerto

Dynamic Shortcuts

[ESEC/FSE'21] J. Park et al., "Accelerating JavaScript Static Analysis via Dynamic Shortcuts"

- **Dynamic Shortcuts** – A technique to perform **sealed execution** (dynamic analysis) during the static analysis to accelerate and increase the precision of the analysis without losing soundness.

$\bullet_{\ell_0}$ `if ( x ≥ 0 )` $\bullet_{\ell_1}$ `x = x;`
`else` $\bullet_{\ell_2}$ `x = −x;`
$\bullet_{\ell_3}$ `x = −x;` $\bullet_{\ell_4}$



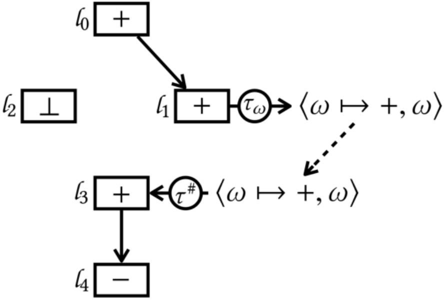
(b) $\mathbf{x} = 0$

Dynamic Shortcuts

[ESEC/FSE'21] J. Park et al., "Accelerating JavaScript Static Analysis via Dynamic Shortcuts"

- **Dynamic Shortcuts** – A technique to perform **sealed execution** (dynamic analysis) during the static analysis to accelerate and increase the precision of the analysis without losing soundness.

- $\bullet_{\ell_0}$ `if` (`x` ≥ 0) $\bullet_{\ell_1}$ `x = x;`
  `else` $\bullet_{\ell_2}$ `x = -x;`
- $\bullet_{\ell_3}$ `x = -x;` $\bullet_{\ell_4}$



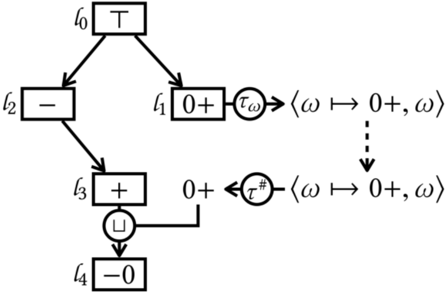
(c) $\mathbf{x} > 0$

Dynamic Shortcuts

[ESEC/FSE'21] J. Park et al., "Accelerating JavaScript Static Analysis via Dynamic Shortcuts"

- **Dynamic Shortcuts** – A technique to perform **sealed execution** (dynamic analysis) during the static analysis to accelerate and increase the precision of the analysis without losing soundness.

$\bullet_{\ell_0}$ `if ( x ≥ 0 )` $\bullet_{\ell_1}$ `x = x;`
`else` $\bullet_{\ell_2}$ `x = −x;`
$\bullet_{\ell_3}$ `x = −x;` $\bullet_{\ell_4}$



(d) $\mathbf{x} \in \mathbb{N}$

Summary

References

- Lecture for Dynamic Symbolic Execution by Prof. Mayur Naik at UPenn CIS.

  https://software-analysis-class.org/lectures/lecture14

- [CSUR'18] Baldoni et al., "A Survey of Symbolic Execution Techniques"

  https://dl.acm.org/doi/abs/10.1145/3182657

Next Lecture

- Mutation Testing

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr