Lecture 5 – Dynamic Symbolic Execution

AAA705: Software Testing and Quality Assurance

Jihyeok Park

PLRG

2026 Fall

Recall

- Search Based Software Engineering (SBSE)
  - Representation / fitness / operators; exploitation vs. exploration
- Fitness Landscape – plateau, needle, ruggedness; the landscape is ours to design
- Local Search – hill climbing, local optima, simulated annealing, tabu search
- Genetic Algorithms – selection / crossover / mutation; representation decides the operators
- Bio-inspired Algorithms – PSO / ACO; no free lunch
- Search Based Software Testing (SBST)
  - Fitness = approach level + branch distance; AVM
  - Flag problem; whole test suite (EvoSuite) / many-objective (MOSA)
  - Search beats random only where the fitness has a **slope**

From a Distance to a Constraint

```
int f(int a, int b) {
  if (a >= 10)
    if (b == 2 * a)
      if (a + b > 100) return 1;
  return 0;
}
```

From a Distance to a Constraint

PLRG

```
int f(int a, int b) {
  if (a >= 10)
    if (b == 2 * a)
      if (a + b > 100) return 1;
  return 0;
}
```

- **Last time**: a **distance** to the branch where the input left. The search moved the input step by step, and often got stuck.

From a Distance to a Constraint

```
int f(int a, int b) {
  if (a >= 10)
    if (b == 2 * a)
      if (a + b > 100) return 1;
  return 0;
}
```

- **Last time**: a **distance** to the branch where the input left. The search moved the input step by step, and often got stuck.
- The path to `return 1` is one formula:
  $a \geq 10 \wedge b = 2a \wedge a + b > 100$. A **solver** returns $(34, 68)$ at once.

From a Distance to a Constraint

PLRG

```
int f(int a, int b) {
  if (a >= 10)
    if (b == 2 * a)
      if (a + b > 100) return 1;
  return 0;
}
```

- **Last time**: a **distance** to the branch where the input left. The search moved the input step by step, and often got stuck.
- The path to `return 1` is one formula: $a \geq 10 \wedge b = 2a \wedge a + b > 100$. A **solver** returns $(34, 68)$ at once.
- **Today**: run the program on **symbols** to collect the formula, solve it to reach the other side of a branch, and use **concrete** values where the solver cannot.

Contents

Contents

Symbolic Execution

- **Random testing** needs about $2^{32} \approx 4 \times 10^9$ runs on average to hit this error:

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

[1] **[CACM'76]** J. C. King. "*Symbolic Execution and Program Testing.*" **[TSE'76]** L. A. Clarke. "*A System to Generate Test Data and Symbolically Execute Programs.*"

Symbolic Execution

- **Random testing** needs about $2^{32} \approx 4 \times 10^9$ runs on average to hit this error:

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

- **Symbolic execution** runs the program **once** on a **symbol** α in place of a number. The branch leaves the constraint $\alpha = 93589$, and a solver returns the input from it.[1]

[1] **[CACM'76]** J. C. King. "*Symbolic Execution and Program Testing.*" **[TSE'76]** L. A. Clarke. "*A System to Generate Test Data and Symbolically Execute Programs.*"

Symbolic Execution

- **Random testing** needs about $2^{32} \approx 4 \times 10^9$ runs on average to hit this error:

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

- **Symbolic execution** runs the program **once** on a **symbol** α in place of a number. The branch leaves the constraint $\alpha = 93589$, and a solver returns the input from it.[1]

- The idea is from 1976. It became practical around 2005 with **SMT solvers**, heuristics against **path explosion**, and models of the **heap** and the **environment** – the rest of today.

[1] **[CACM'76]** J. C. King. "*Symbolic Execution and Program Testing.*" **[TSE'76]** L. A. Clarke. "*A System to Generate Test Data and Symbolically Execute Programs.*"

Symbolic Execution – Basic Idea

1. Run the program on **symbolic inputs** $\alpha, \beta, \ldots$ instead of numbers – one symbol per input.

Symbolic Execution – Basic Idea

1. Run the program on **symbolic inputs** $\alpha, \beta, \ldots$ instead of numbers – one symbol per input.

2. A variable then holds a **symbolic expression** $\omega \in \Omega$, built from those symbols, constants, and the program's own operators:

$$\omega ::= \alpha \mid n \mid \omega \oplus \omega \mid f(\omega, \ldots) \qquad \text{e.g., } 2\alpha,\ \ \beta - \alpha,\ \ \mathit{hash}(\alpha)$$

Symbolic Execution – Basic Idea

PLRG

1. Run the program on **symbolic inputs** $\alpha, \beta, \ldots$ instead of numbers – one symbol per input.

2. A variable then holds a **symbolic expression** $\omega \in \Omega$, built from those symbols, constants, and the program's own operators:

$$\omega ::= \alpha \mid n \mid \omega \oplus \omega \mid f(\omega, \ldots) \qquad \text{e.g., } 2\alpha,\ \ \beta - \alpha,\ \ \mathit{hash}(\alpha)$$

3. A **symbolic state** $\sigma \in \mathbb{S} = \mathbb{X} \xrightarrow{\text{fin}} \Omega$ maps each variable to the expression it currently holds.

Symbolic Execution – Basic Idea

1. Run the program on **symbolic inputs** $\alpha, \beta, \ldots$ instead of numbers – one symbol per input.

2. A variable then holds a **symbolic expression** $\omega \in \Omega$, built from those symbols, constants, and the program's own operators:

$$\omega ::= \alpha \mid n \mid \omega \oplus \omega \mid f(\omega, \ldots) \qquad \text{e.g., } 2\alpha,\ \ \beta - \alpha,\ \ \mathit{hash}(\alpha)$$

3. A **symbolic state** $\sigma \in \mathbb{S} = \mathbb{X} \xrightarrow{\text{fin}} \Omega$ maps each variable to the expression it currently holds.

4. A **path condition** $\Phi = \phi_1 \land \ldots \land \phi_n$ collects one **boolean** symbolic expression ϕ_i per branch taken so far.

Symbolic Execution – Basic Idea

PLRG

1. Run the program on **symbolic inputs** $\alpha, \beta, \ldots$ instead of numbers – one symbol per input.

2. A variable then holds a **symbolic expression** $\omega \in \Omega$, built from those symbols, constants, and the program's own operators:

$$\omega ::= \alpha \mid n \mid \omega \oplus \omega \mid f(\omega, \ldots) \qquad \text{e.g., } 2\alpha, \;\; \beta - \alpha, \;\; \mathit{hash}(\alpha)$$

3. A **symbolic state** $\sigma \in \mathbb{S} = \mathbb{X} \xrightarrow{\text{fin}} \Omega$ maps each variable to the expression it currently holds.

4. A **path condition** $\Phi = \phi_1 \wedge \ldots \wedge \phi_n$ collects one **boolean** symbolic expression ϕ_i per branch taken so far.

5. All paths form the **execution tree**. A path is **feasible** when its Φ is satisfiable, and **infeasible** otherwise.

Symbolic Execution – Basic Idea

```
void f(int x, int y) {
  /* 1 */
  int z = 2 * x;
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

1: $\boxed{\begin{array}{l} \mathtt{x} \mapsto \alpha \\ \mathtt{y} \mapsto \beta \\ \Phi = \mathit{true} \end{array}}$

Symbolic Execution – Basic Idea

```c
void f(int x, int y) {
  /* 1 */
  int z = 2 * x;
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

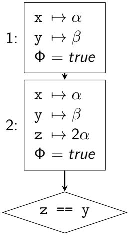
1:
$$\begin{array}{|lcl|}\hline \mathtt{x} & \mapsto & \alpha \\ \mathtt{y} & \mapsto & \beta \\ \Phi & = & \textit{true} \\ \hline \end{array}$$

$\downarrow$

2:
$$\begin{array}{|lcl|}\hline \mathtt{x} & \mapsto & \alpha \\ \mathtt{y} & \mapsto & \beta \\ \mathtt{z} & \mapsto & 2\alpha \\ \Phi & = & \textit{true} \\ \hline \end{array}$$

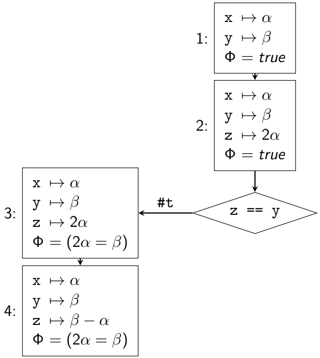
Symbolic Execution – Basic Idea

```
void f(int x, int y) {
  /* 1 */
  int z = 2 * x;
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```



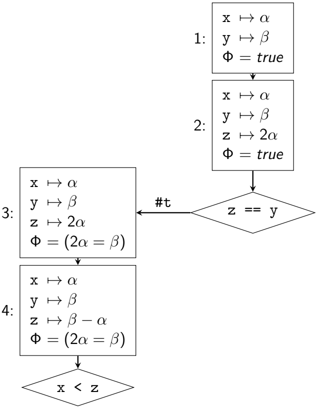
Symbolic Execution – Basic Idea

```
void f(int x, int y) {
  /* 1 */
  int z = 2 * x;
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```



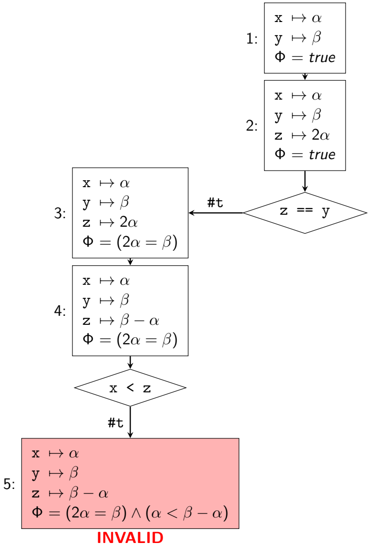
Symbolic Execution – Basic Idea

```
void f(int x, int y) {
  /* 1 */
  int z = 2 * x;
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```



Symbolic Execution – Basic Idea

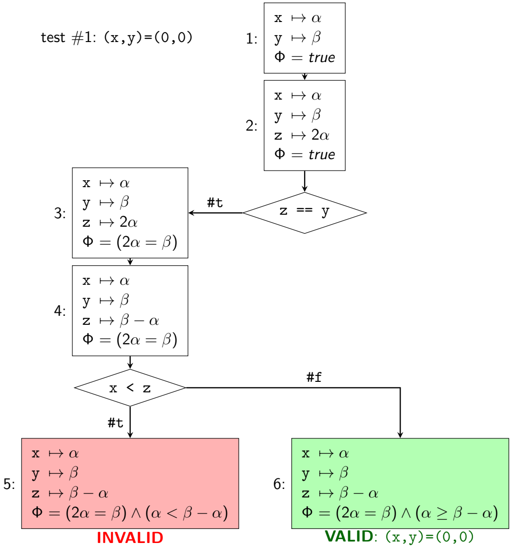
```
void f(int x, int y) {
  /* 1 */
  int z = 2 * x;
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```



Symbolic Execution – Basic Idea

```
void f(int x, int y) {
  /* 1 */
  int z = 2 * x;
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

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

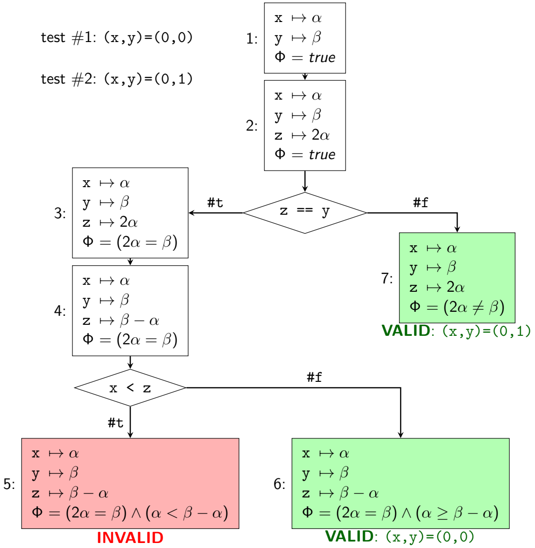


Symbolic Execution – Basic Idea

```
void f(int x, int y) {
  /* 1 */
  int z = 2 * x;
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

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?

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, cvc5) to check it.

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, cvc5) 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.

$$b + 2 = c \land f(read(write(a, b, 3), c - 2)) \neq f(c - b + 1)$$

Satisfiability Modulo Theories (SMT) – Example

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

$$b + 2 = c \land f(read(write(a, b, 3), c - 2)) \neq f(c - b + 1)$$

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

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

Satisfiability Modulo Theories (SMT) – Example

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

$$b + 2 = c \land f(read(write(a, b, 3), c - 2)) \neq f(c - b + 1)$$

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

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

- **Arithmetic simplification** of the formula.

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

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

$$b + 2 = c \land f(read(write(a, b, 3), c - 2)) \neq f(c - b + 1)$$

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

$$b + 2 = c \land f(read(write(a, b, b + 2 - 2), b)) \neq f(b + 2 - b + 1)$$

- **Arithmetic simplification** of the formula.

$$b + 2 = c \land f(read(write(a, b, 3), b)) \neq f(3)$$

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

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

Satisfiability Modulo Theories (SMT) – Example

Z3[2] is one of the most popular SMT solvers developed by Microsoft Research and used in many symbolic execution tools.

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

Satisfiability Modulo Theories (SMT) – Example

Z3[2] 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
```

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

Satisfiability Modulo Theories (SMT) – Example

Z3[2] 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)
```

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

Z3 From a Program

A DSE tool builds the path condition **as data** and asks Z3 in a loop. Node 5's condition, through the JavaScript binding:[3]

```javascript
import { init } from 'z3-solver';
const { Context } = await init();
const { Int, Solver } = new Context('main');

const x = Int.const('x'), y = Int.const('y');
const s = new Solver();
s.add(x.mul(2).eq(y), x.lt(y.sub(x)));   // 2x == y && x < y - x
console.log(await s.check());             // unsat
```

[3] https://github.com/Z3Prover/z3 `npm install z3-solver`

Z3 From a Program

A DSE tool builds the path condition **as data** and asks Z3 in a loop. Node 5's condition, through the JavaScript binding:[3]

```
import { init } from 'z3-solver';
const { Context } = await init();
const { Int, Solver } = new Context('main');

const x = Int.const('x'), y = Int.const('y');
const s = new Solver();
s.add(x.mul(2).eq(y), x.lt(y.sub(x)));   // 2x == y && x < y - x
console.log(await s.check());             // unsat
```

- Flip the last conjunct to `x.ge(y.sub(x))` – node 6, the script on the previous slide: `check()` says `sat` and `s.model()` gives $x = 0, y = 0$.

[3] https://github.com/Z3Prover/z3 `npm install z3-solver`

Limitations of Symbolic Execution

```
void f(int x, int y) {
  /* 1 */
  int z = hash(x);
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

No SMT solver can decide a path condition Φ that contains `hash`.

1: $\boxed{\begin{array}{l} \mathtt{x} \mapsto \alpha \\ \mathtt{y} \mapsto \beta \\ \Phi = \textit{true} \end{array}}$

Limitations of Symbolic Execution

```
void f(int x, int y) {
  /* 1 */
  int z = hash(x);
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

No SMT solver can decide a path condition Φ that contains `hash`.

1: $\mathtt{x} \mapsto \alpha$, $\mathtt{y} \mapsto \beta$, $\Phi = \mathit{true}$

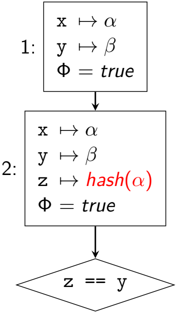
↓

2: $\mathtt{x} \mapsto \alpha$, $\mathtt{y} \mapsto \beta$, $\mathtt{z} \mapsto \mathit{hash}(\alpha)$, $\Phi = \mathit{true}$

Limitations of Symbolic Execution

```
void f(int x, int y) {
  /* 1 */
  int z = hash(x);
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

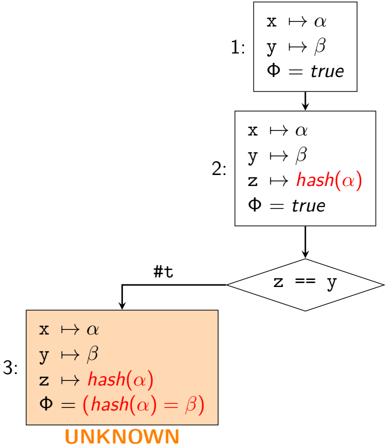
No SMT solver can decide a path condition Φ that contains `hash`.



Limitations of Symbolic Execution

```
void f(int x, int y) {
  /* 1 */
  int z = hash(x);
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

No SMT solver can decide a path condition Φ that contains `hash`.

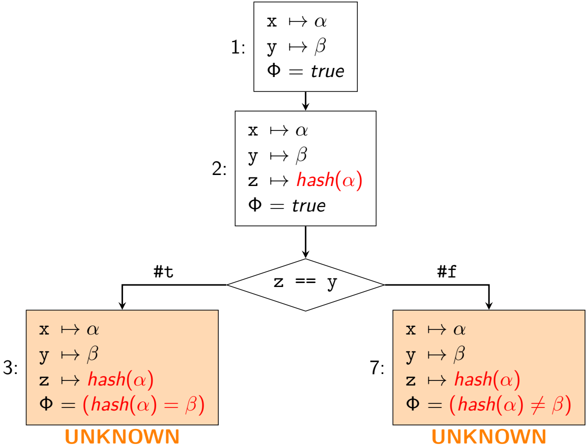


Limitations of Symbolic Execution

PLRG

```
void f(int x, int y) {
  /* 1 */
  int z = hash(x);
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

No SMT solver can decide a path condition Φ that contains hash.



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.

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**.

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.

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 = hash(x);
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

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

1: $\begin{array}{|lcl|} \hline \texttt{x} & \mapsto & 3 \mid \alpha \\ \texttt{y} & \mapsto & 7 \mid \beta \\ \Phi & = & \mathit{true} \\ \hline \end{array}$

Dynamic Symbolic Execution (DSE) – Example

```
void f(int x, int y) {
  /* 1 */
  int z = hash(x);
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

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

1:

x	$\mapsto$ 3	α
y	$\mapsto$ 7	β
Φ	= *true*	

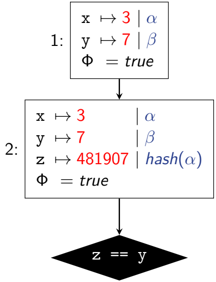
2:

x	$\mapsto$ 3	α
y	$\mapsto$ 7	β
z	$\mapsto$ 481907	$hash(\alpha)$
Φ	= *true*	

Dynamic Symbolic Execution (DSE) – Example

```
void f(int x, int y) {
  /* 1 */
  int z = hash(x);
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

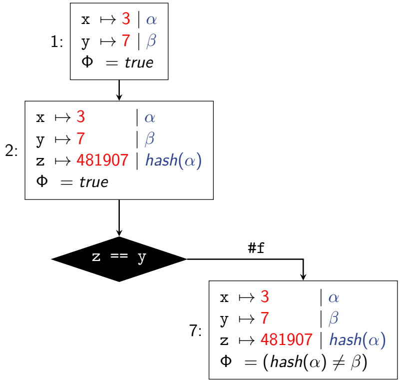
test #1: (x,y)=(3,7)



Dynamic Symbolic Execution (DSE) – Example

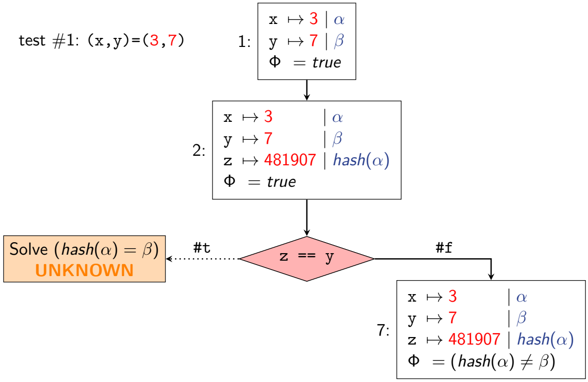
```
void f(int x, int y) {
  /* 1 */
  int z = hash(x);
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

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



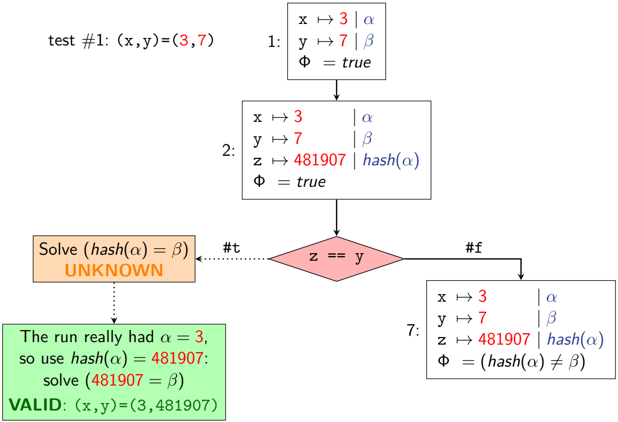
Dynamic Symbolic Execution (DSE) – Example

```
void f(int x, int y) {
  /* 1 */
  int z = hash(x);
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```



Dynamic Symbolic Execution (DSE) – Example

```
void f(int x, int y) {
  /* 1 */
  int z = hash(x);
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

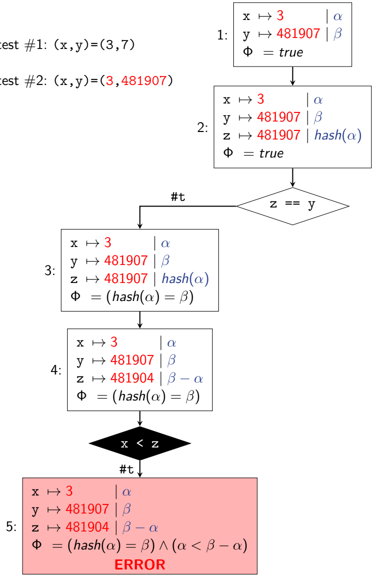


Dynamic Symbolic Execution (DSE) – Example

```
void f(int x, int y) {
  /* 1 */
  int z = hash(x);
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

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

test #2: (x,y)=(3,481907)

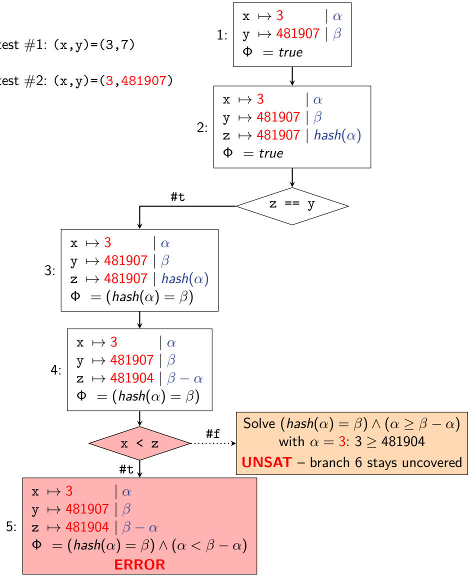


Dynamic Symbolic Execution (DSE) – Example

```
void f(int x, int y) {
  /* 1 */
  int z = hash(x);
  /* 2 */
  if (z == y) {
    /* 3 */
    z = y - x;
    /* 4 */
    if (x < z) {
      /* 5 */
      ERROR;
    } else {
      /* 6 */
    }
  } else {
    /* 7 */
  }
}
```

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

test #2: (x,y)=(3,481907)



Dynamic Symbolic Execution (DSE) – Algorithm

Input: program P, initial input v, budget N.
Output: test suite V.

$V \leftarrow \{v\}$; $\quad T \leftarrow \emptyset$ $\qquad$ ▷ paths run so far
for $m = 1$ **to** N **do**
    $\Phi \leftarrow \mathit{Execute}(P, v)$ $\qquad$ ▷ $\Phi = \phi_1 \wedge \cdots \wedge \phi_n$
    $T \leftarrow T \cup \{\Phi\}$
    repeat
        $(\Phi, i) \leftarrow \mathit{Choose}(T)$ $\qquad$ ▷ search heuristic
        $\Psi \leftarrow \phi_1 \wedge \cdots \wedge \phi_{i-1} \wedge \neg\phi_i$ $\qquad$ ▷ same prefix, other side
    until $\mathit{SAT}(\Psi)$
    $v \leftarrow \mathit{Model}(\Psi)$; $\quad V \leftarrow V \cup \{v\}$ $\qquad$ ▷ an input that gets there
return V

Dynamic Symbolic Execution (DSE) – Algorithm

Input: program P, initial input v, budget N.
Output: test suite V.

1: $V \leftarrow \{v\}$; $\quad T \leftarrow \emptyset$ $\qquad\triangleright$ paths run so far
2: **for** $m = 1$ **to** N **do**
3: $\quad \Phi \leftarrow \mathit{Execute}(P, v)$ $\qquad\triangleright$ $\Phi = \phi_1 \wedge \cdots \wedge \phi_n$
4: $\quad T \leftarrow T \cup \{\Phi\}$
5: $\quad$ **repeat**
6: $\qquad (\Phi, i) \leftarrow \mathit{Choose}(T)$ $\qquad\triangleright$ search heuristic
7: $\qquad \Psi \leftarrow \phi_1 \wedge \cdots \wedge \phi_{i-1} \wedge \neg\phi_i$ $\qquad\triangleright$ same prefix, other side
8: $\quad$ **until** $\mathit{SAT}(\Psi)$
9: $\quad v \leftarrow \mathit{Model}(\Psi)$; $\quad V \leftarrow V \cup \{v\}$ $\qquad\triangleright$ an input that gets there
10: **return** V

Line 3 is what makes it **dynamic**: Φ comes from a real run, so concrete values can replace what the solver cannot handle.

Dynamic Symbolic Execution (DSE)

Choose picks the branch to flip. That is the **search heuristic**.

[4] **[PLDI'05]** P. Godefroid, N. Klarlund, K. Sen. "*DART: Directed Automated Random Testing.*" **[FSE'05]** K. Sen, D. Marinov, G. Agha. "*CUTE: A Concolic Unit Testing Engine for C.*"

[5] **[ASE'08]** J. Burnim, K. Sen. "*Heuristics for Scalable Dynamic Test Generation.*"

[6] **[FSE'14]** H. Seo, S. Kim. "*How We Get There: A Context-Guided Search Strategy in Concolic Testing.*"

Dynamic Symbolic Execution (DSE)

Choose picks the branch to flip. That is the **search heuristic**.

- **Random Search** – any branch of the last path.

[4] **[PLDI'05]** P. Godefroid, N. Klarlund, K. Sen. "*DART: Directed Automated Random Testing.*" **[FSE'05]** K. Sen, D. Marinov, G. Agha. "*CUTE: A Concolic Unit Testing Engine for C.*"

[5] **[ASE'08]** J. Burnim, K. Sen. "*Heuristics for Scalable Dynamic Test Generation.*"

[6] **[FSE'14]** H. Seo, S. Kim. "*How We Get There: A Context-Guided Search Strategy in Concolic Testing.*"

Dynamic Symbolic Execution (DSE)

Choose picks the branch to flip. That is the **search heuristic**.

- **Random Search** – any branch of the last path.
- **Depth-First Search (DFS)** – the deepest branch not flipped yet. It chases **paths** (DART, CUTE).[4]

[4] **[PLDI'05]** P. Godefroid, N. Klarlund, K. Sen. "*DART: Directed Automated Random Testing.*" **[FSE'05]** K. Sen, D. Marinov, G. Agha. "*CUTE: A Concolic Unit Testing Engine for C.*"

[5] **[ASE'08]** J. Burnim, K. Sen. "*Heuristics for Scalable Dynamic Test Generation.*"

[6] **[FSE'14]** H. Seo, S. Kim. "*How We Get There: A Context-Guided Search Strategy in Concolic Testing.*"

Dynamic Symbolic Execution (DSE)

Choose picks the branch to flip. That is the **search heuristic**.

- **Random Search** – any branch of the last path.
- **Depth-First Search (DFS)** – the deepest branch not flipped yet. It chases **paths** (DART, CUTE).[4]
- **Control-Flow Directed Search (CFDS)** – the nearest branch whose other side is **uncovered**. It chases **branches**.[5]

[4] **[PLDI'05]** P. Godefroid, N. Klarlund, K. Sen. "*DART: Directed Automated Random Testing.*" **[FSE'05]** K. Sen, D. Marinov, G. Agha. "*CUTE: A Concolic Unit Testing Engine for C.*"

[5] **[ASE'08]** J. Burnim, K. Sen. "*Heuristics for Scalable Dynamic Test Generation.*"

[6] **[FSE'14]** H. Seo, S. Kim. "*How We Get There: A Context-Guided Search Strategy in Concolic Testing.*"

Dynamic Symbolic Execution (DSE)

Choose picks the branch to flip. That is the **search heuristic**.

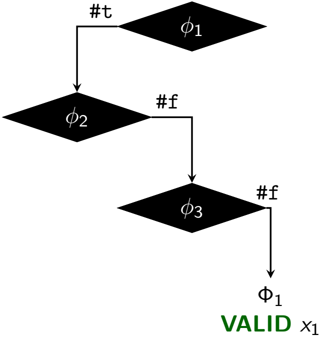
- **Random Search** – any branch of the last path.
- **Depth-First Search (DFS)** – the deepest branch not flipped yet. It chases **paths** (DART, CUTE).[4]
- **Control-Flow Directed Search (CFDS)** – the nearest branch whose other side is **uncovered**. It chases **branches**.[5]
- **Context-Guided Search (CGS)** – from the root outward, and only a **new** (branch, context) pair.[6]

[4] **[PLDI'05]** P. Godefroid, N. Klarlund, K. Sen. "*DART: Directed Automated Random Testing.*" **[FSE'05]** K. Sen, D. Marinov, G. Agha. "*CUTE: A Concolic Unit Testing Engine for C.*"

[5] **[ASE'08]** J. Burnim, K. Sen. "*Heuristics for Scalable Dynamic Test Generation.*"

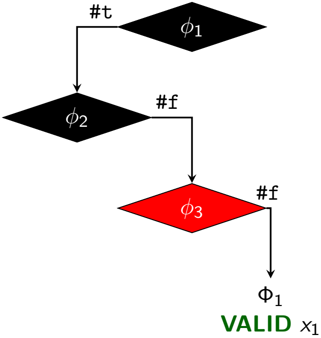
[6] **[FSE'14]** H. Seo, S. Kim. "*How We Get There: A Context-Guided Search Strategy in Concolic Testing.*"

CFDS negates the uncovered branch **closest to the end** of the last path. Black = one side still uncovered, white = both sides covered, **red** = the branch being negated.



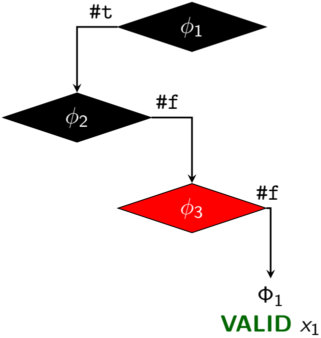
Search Heuristics in DSE

CFDS negates the uncovered branch **closest to the end** of the last path. Black = one side still uncovered, white = both sides covered, **red** = the branch being negated.



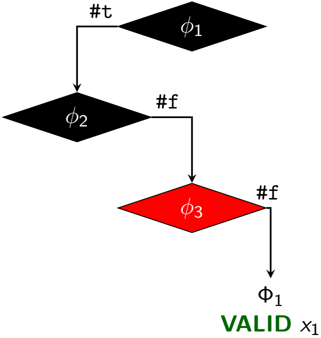
Search Heuristics in DSE

CFDS negates the uncovered branch **closest to the end** of the last path. Black = one side still uncovered, white = both sides covered, **red** = the branch being negated.



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

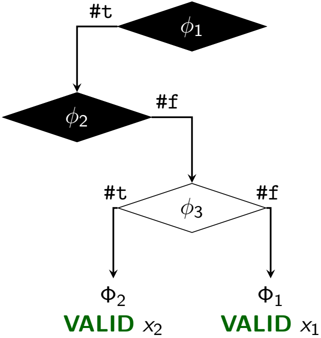
CFDS negates the uncovered branch **closest to the end** of the last path. Black = one side still uncovered, white = both sides covered, **red** = the branch being negated.



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

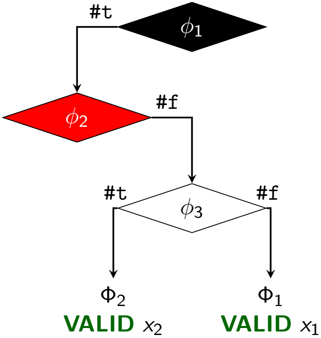
VALID x_2

CFDS negates the uncovered branch **closest to the end** of the last path. Black = one side still uncovered, white = both sides covered, **red** = the branch being negated.



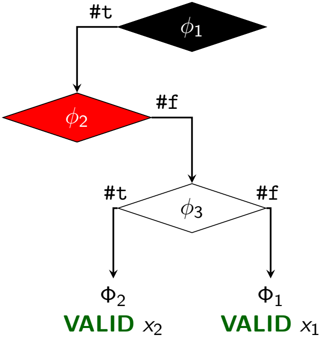
Search Heuristics in DSE

CFDS negates the uncovered branch **closest to the end** of the last path. Black = one side still uncovered, white = both sides covered, **red** = the branch being negated.



Search Heuristics in DSE

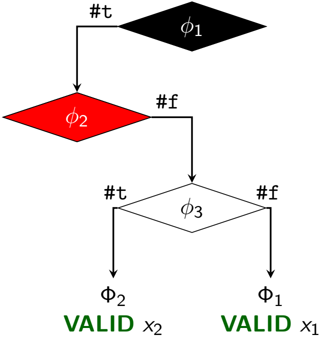
CFDS negates the uncovered branch **closest to the end** of the last path. Black = one side still uncovered, white = both sides covered, red = the branch being negated.



Solve $\phi_1 \wedge \phi_2$

Search Heuristics in DSE

CFDS negates the uncovered branch **closest to the end** of the last path. Black = one side still uncovered, white = both sides covered, **red** = the branch being negated.

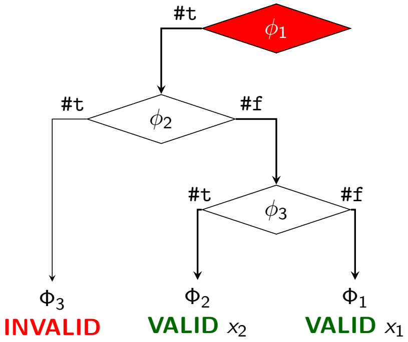


Solve $\phi_1 \wedge \phi_2$

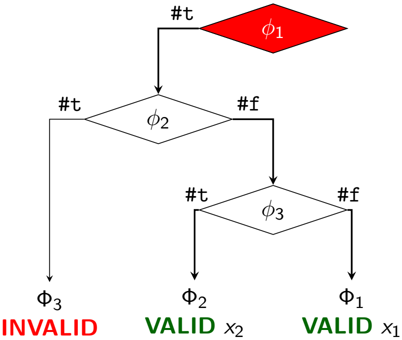
INVALID

Search Heuristics in DSE

CFDS negates the uncovered branch **closest to the end** of the last path. Black = one side still uncovered, white = both sides covered, **red** = the branch being negated.



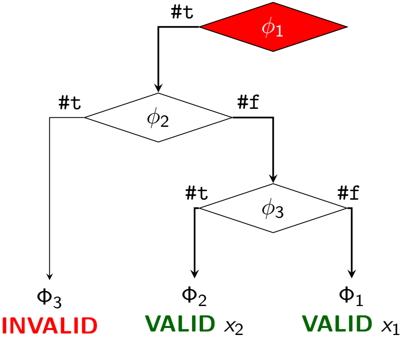
CFDS negates the uncovered branch **closest to the end** of the last path. Black = one side still uncovered, white = both sides covered, **red** = the branch being negated.



Solve $\neg\phi_1$

Search Heuristics in DSE

CFDS negates the uncovered branch **closest to the end** of the last path. Black = one side still uncovered, white = both sides covered, **red** = the branch being negated.

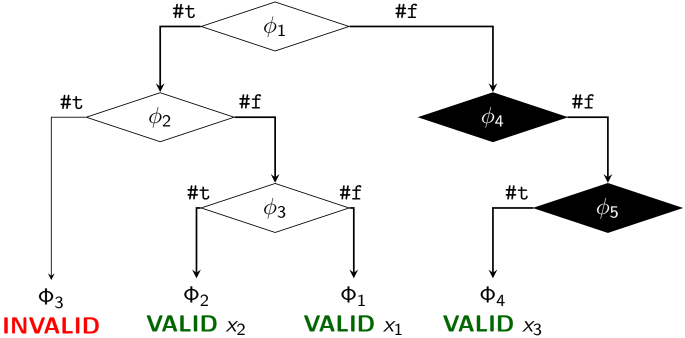


Solve $\neg\phi_1$

VALID x_3

Search Heuristics in DSE

CFDS negates the uncovered branch **closest to the end** of the last path. Black = one side still uncovered, white = both sides covered, **red** = the branch being negated.



DFS and CFDS Are Not the Same

```
void f(int x, int y) {
  if (x > 0) a(); // b1
  if (y > 0) b(); // b2
}
```

run	input	path	next: DFS	next: CFDS
1	$(1, 1)$	b1 #t b2 #t	flip b2	flip b2
2	$(1, 0)$	b1 #t b2 #f	flip b1	flip b1
3	$(0, 0)$	b1 #f b2 #f	flip b2	both sides covered
4	$(0, 1)$	b1 #f b2 #t	–	**never run**

DFS and CFDS Are Not the Same

PLRG

```
void f(int x, int y) {
  if (x > 0) a(); // b1
  if (y > 0) b(); // b2
}
```

run	input	path	next: DFS	next: CFDS
1	$(1, 1)$	`b1 #t  b2 #t`	flip `b2`	flip `b2`
2	$(1, 0)$	`b1 #t  b2 #f`	flip `b1`	flip `b1`
3	$(0, 0)$	`b1 #f  b2 #f`	flip `b2`	both sides covered
4	$(0, 1)$	`b1 #f  b2 #t`	–	**never run**

- Four leaves, but only four branch sides. Run **3** covers them all. DFS runs a fourth test anyway, because that `b2` was never flipped **on this path**.

DFS and CFDS Are Not the Same

```
void f(int x, int y) {
  if (x > 0) a(); // b1
  if (y > 0) b(); // b2
}
```

run	input	path	next: DFS	next: CFDS
1	$(1, 1)$	`b1#t b2#t`	flip `b2`	flip `b2`
2	$(1, 0)$	`b1#t b2#f`	flip `b1`	flip `b1`
3	$(0, 0)$	`b1#f b2#f`	flip `b2`	both sides covered
4	$(0, 1)$	`b1#f b2#t`	–	**never run**

- Four leaves, but only four branch sides. Run **3** covers them all. DFS runs a fourth test anyway, because that `b2` was never flipped **on this path**.
- n independent `if`s in sequence – DFS walks all 2^n paths, CFDS stops at $n + 1$ tests.

Search Heuristics in DSE

CGS scans **from the root outward**, not back from the end, and flips only a **new** (branch, context) pair. Context = the last d branches; here $d = 1$.

```
i = 0;
while (i < 2) {         // b1
  if (arr[i] > 0) s++; // b2
  i++;
}
if (x > 0) t++;         // b3
```

The first run takes the loop twice:

$$\Phi = \phi_1 \wedge \cdots \wedge \phi_6$$

Search Heuristics in DSE

CGS scans **from the root outward**, not back from the end, and flips only a **new** (branch, context) pair. Context = the last d branches; here $d = 1$.

```
i = 0;
while (i < 2) {          // b1
  if (arr[i] > 0) s++; // b2
  i++;
}
if (x > 0) t++;          // b3
```

The first run takes the loop twice:

$$\Phi = \phi_1 \wedge \cdots \wedge \phi_6$$

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6
branch	b1	b2	b1	b2	b1	b3
context	$\langle\,\rangle$	$\langle$b1$\rangle$	$\langle$b2$\rangle$	$\langle$b1$\rangle$	$\langle$b2$\rangle$	$\langle$b1$\rangle$
pair	new	new	new	$= \phi_2$	$= \phi_3$	new
CGS	negate	negate	negate	**skip**	**skip**	negate

Search Heuristics in DSE

CGS scans **from the root outward**, not back from the end, and flips only a **new** (branch, context) pair. Context = the last d branches; here $d = 1$.

```
i = 0;
while (i < 2) {          // b1
  if (arr[i] > 0) s++; // b2
  i++;
}
if (x > 0) t++;          // b3
```

The first run takes the loop twice:

$$\Phi = \phi_1 \wedge \cdots \wedge \phi_6$$

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6
branch	b1	b2	b1	b2	b1	b3
context	$\langle\,\rangle$	$\langle$b1$\rangle$	$\langle$b2$\rangle$	$\langle$b1$\rangle$	$\langle$b2$\rangle$	$\langle$b1$\rangle$
pair	new	new	new	$= \phi_2$	$= \phi_3$	new
CGS	negate	negate	negate	**skip**	**skip**	negate

ϕ_4, ϕ_5 are the loop's **second turn** – the same pair as ϕ_2, ϕ_3. d grows by one when a round finds nothing new.

Learning Search Heuristics in DSE

Do not design *Choose* – **learn** it. A branch ϕ becomes a vector of **40 boolean features** $\pi(\phi)$, scored by a weight vector θ learned per program.[7]

$$\mathit{score}_\theta(\phi) = \pi(\phi) \cdot \theta \qquad \mathit{Choose}_\theta(\langle \Phi_1 \cdots \Phi_m \rangle) = (\Phi_m, \operatorname*{argmax}_{\phi_j \in \Phi_m} \mathit{score}_\theta(\phi_j))$$

[7] **[ICSE'18]** S. Cha, S. Hong, J. Lee, H. Oh. "*Automatically Generating Search Heuristics for Concolic Testing.*"

Learning Search Heuristics in DSE

Do not design *Choose* – **learn** it. A branch ϕ becomes a vector of **40 boolean features** $\pi(\phi)$, scored by a weight vector θ learned per program.[7]

$$score_\theta(\phi) = \pi(\phi) \cdot \theta \qquad Choose_\theta(\langle \Phi_1 \cdots \Phi_m \rangle) = (\Phi_m, \operatorname{argmax}_{\phi_j \in \Phi_m} score_\theta(\phi_j))$$

12 static – read off the code

- branch in the `main` function
- true branch of a loop
- nested branch
- branch with external calls
- branch inside a loop body

⋮

28 dynamic – from the runs so far

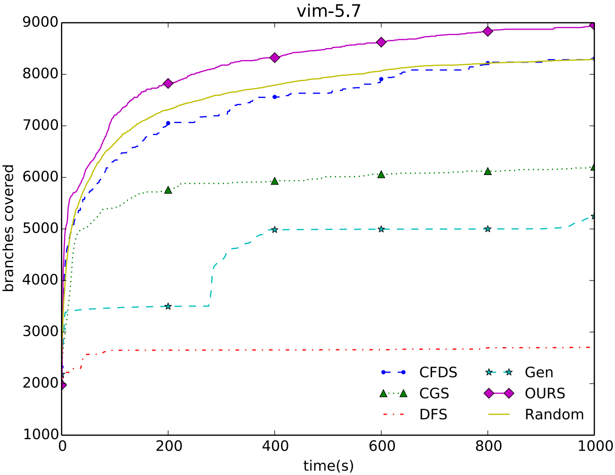
- in the first 10% of the path
- newly covered in the last run
- context ($d = 1$) already visited
- negated more than 10 times
- opposite branch still uncovered

⋮

[7] **[ICSE'18]** S. Cha, S. Hong, J. Lee, H. Oh. "*Automatically Generating Search Heuristics for Concolic Testing.*"

Learning Search Heuristics in DSE

θ is searched **on the program itself**: try weight vectors, keep the ones that cover more branches. Below, branches covered in `vim-5.7`.



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		
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

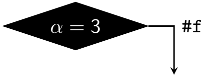
```
void f(int x) {

  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;
}
```

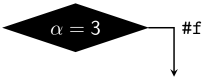
$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr	$\{3, 7, 2\}$	
i	1	
Φ	$(\alpha \neq 3)$	



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;
}
```

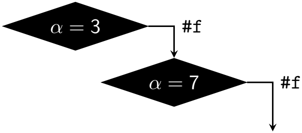
$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr	$\{3, 7, 2\}$	
i	1	
Φ	$(\alpha \neq 3)$	



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;
}
```

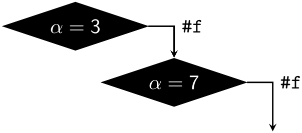
$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr	$\{3, 7, 2\}$	
i	2	
Φ	$(\alpha \neq 3) \wedge (\alpha \neq 7)$	



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;
}
```

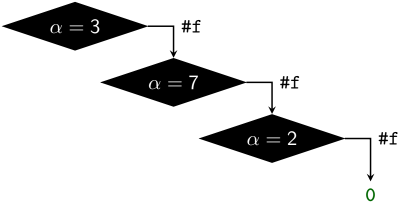
$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr	$\{3, 7, 2\}$	
i	2	
Φ	$(\alpha \neq 3) \wedge (\alpha \neq 7)$	



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;
}
```

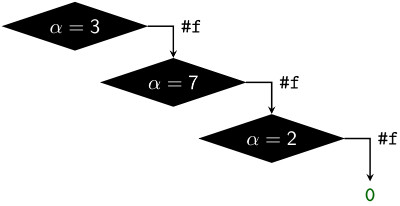
$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr	$\{3, 7, 2\}$	
i	3	
Φ	$(\alpha \neq 3) \land (\alpha \neq 7) \land (\alpha \neq 2)$	



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;
}
```

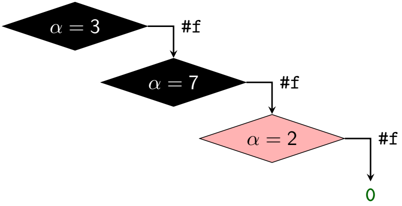
$\mathbb{X}$	$\mathbb{V}$	Ω
x	0	α
arr	$\{3, 7, 2\}$	
i	3	
Φ	$(\alpha \neq 3) \wedge (\alpha \neq 7) \wedge (\alpha \neq 2)$	



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	$(\alpha \neq 3) \wedge (\alpha \neq 7) \wedge (\alpha \neq 2)$	



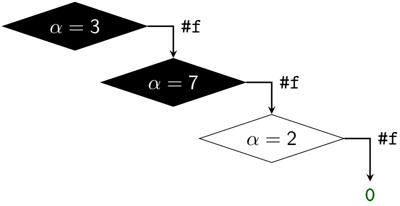
$(\alpha \neq 3) \wedge (\alpha \neq 7) \wedge (\alpha = 2)$ is **SAT**

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;
}
```

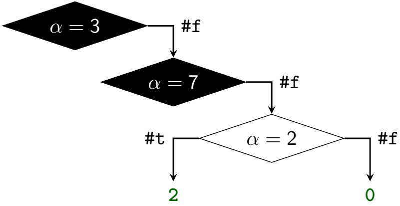
$\mathbb{X}$	$\mathbb{V}$	Ω
x	2	α
arr		
i		
Φ	*true*	



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;
}
```

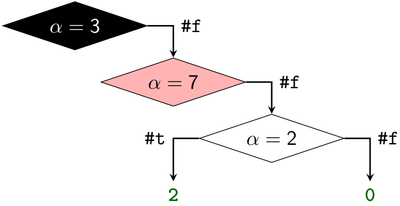
$\mathbb{X}$	$\mathbb{V}$	Ω
x	2	α
arr	$\{3, 7, 2\}$	
i	2	
Φ	$(\alpha \neq 3) \wedge (\alpha \neq 7) \wedge (\alpha = 2)$	



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	$(\alpha \neq 3) \wedge (\alpha \neq 7) \wedge (\alpha = 2)$	



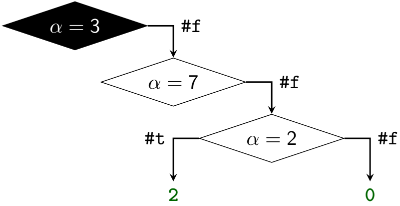
$(\alpha \neq 3) \wedge (\alpha = 7)$ is **SAT**

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;
}
```

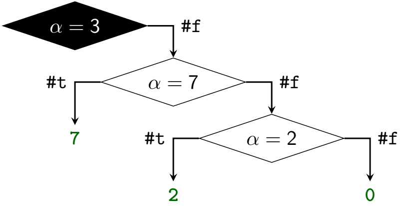
$\mathbb{X}$	$\mathbb{V}$	Ω
x	7	α
arr		
i		
Φ	*true*	



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;
}
```

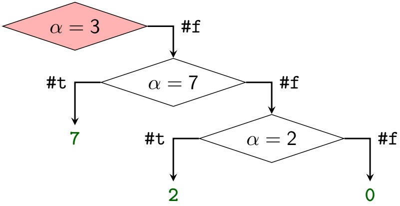
$\mathbb{X}$	$\mathbb{V}$	Ω
x	7	α
arr	$\{3,7,2\}$	
i	1	
Φ	$(\alpha \neq 3) \wedge (\alpha = 7)$	



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	$(\alpha \neq 3) \wedge (\alpha = 7)$	



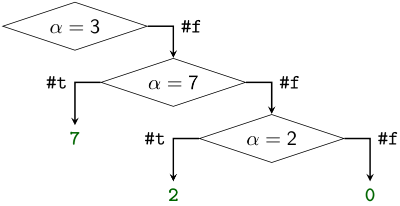
$(\alpha = 3)$ is **SAT**

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;
}
```

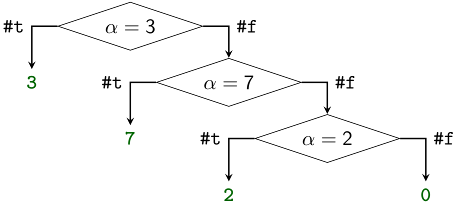
$\mathbb{X}$	$\mathbb{V}$	Ω
x	3	α
arr		
i		
Φ	$true$	



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

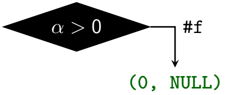
```
class Node {
  int data;
  Node* next;
};

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	$(\alpha \leq 0)$	

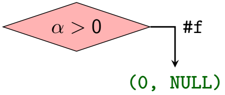


Example – Data Structures

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	0	α
p	$NULL$	β
Φ	$(\alpha \leq 0)$	



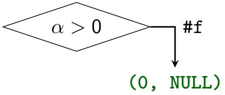
$(\alpha > 0)$ is **SAT**

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;
}
```

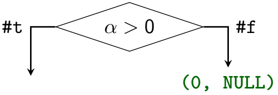
$\mathbb{X}$	$\mathbb{V}$	Ω
x	42	α
p	*NULL*	β
Φ	*true*	



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;
}
```

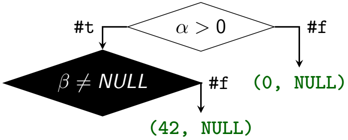
$\mathbb{X}$	$\mathbb{V}$	Ω
x	42	α
p	$NULL$	β
Φ	$(\alpha > 0)$	



Example – Data Structures

```
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;
}
```

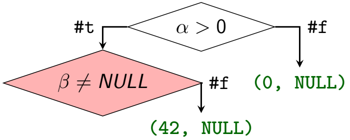
$\mathbb{X}$	$\mathbb{V}$	Ω
x	42	α
p	$NULL$	β
Φ	$(\alpha > 0) \wedge (\beta = NULL)$	



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	$(\alpha > 0) \wedge (\beta = NULL)$	



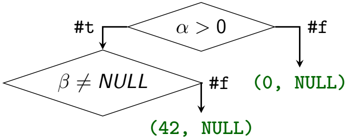
$(\alpha > 0) \wedge (\beta \neq NULL)$ is **SAT**

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;
}
```

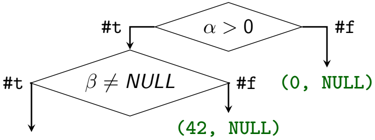
$\mathbb{X}$	$\mathbb{V}$	Ω
x	42	α
p	0xA0BF : | 0 | *NULL* |	β
Φ	*true*	



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;
}
```

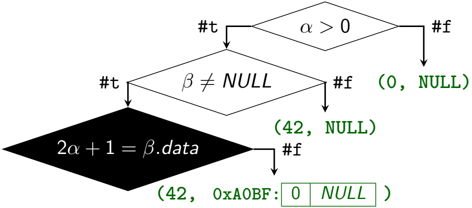
$\mathbb{X}$	$\mathbb{V}$	Ω
x	42	α
p	0xA0BF : $\boxed{0 \mid \mathit{NULL}}$	β
Φ	$(\alpha > 0) \wedge (\beta \neq \mathit{NULL})$	



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;
}
```

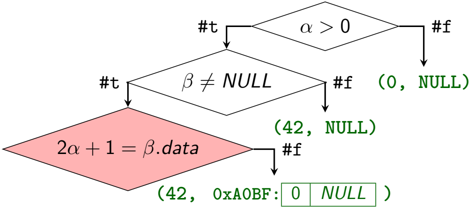
$\mathbb{X}$	$\mathbb{V}$	Ω
x	42	α
p	0xA0BF : $\boxed{0 \mid NULL}$	β
Φ	$(\alpha > 0) \land (\beta \neq NULL) \land (2\alpha + 1 \neq \beta.data)$	



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 : $\boxed{0 \mid \mathit{NULL}}$	β
Φ	$(\alpha > 0) \wedge (\beta \neq \mathit{NULL}) \wedge (2\alpha + 1 \neq \beta.\mathit{data})$	



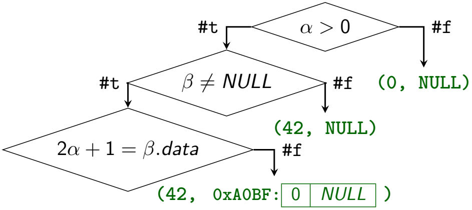
$(\alpha > 0) \wedge (\beta \neq \mathit{NULL})$
$\wedge (2\alpha + 1 = \beta.\mathit{data})$ is **SAT**

when $(x, p) = (1, 0\text{xA0BF} : \boxed{3 \mid \mathit{NULL}})$

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;
}
```

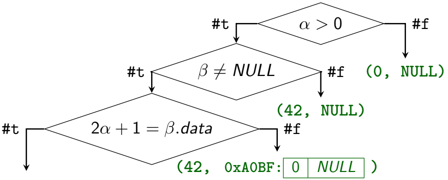
$\mathbb{X}$	$\mathbb{V}$	Ω
x	1	α
p	0xA0BF : | 3 | *NULL* |	β
Φ	*true*	



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;
}
```

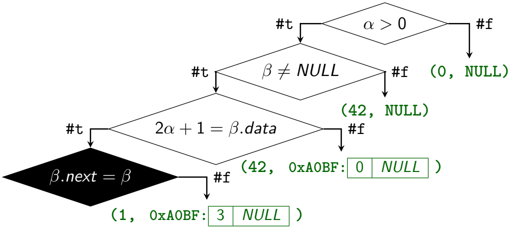
$\mathbb{X}$	$\mathbb{V}$	Ω
x	1	α
p	0xA0BF : $\boxed{3 \mid \mathit{NULL}}$	β
Φ	$(\alpha > 0) \land (\beta \neq \mathit{NULL}) \land (2\alpha + 1 = \beta.data)$	



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;
}
```

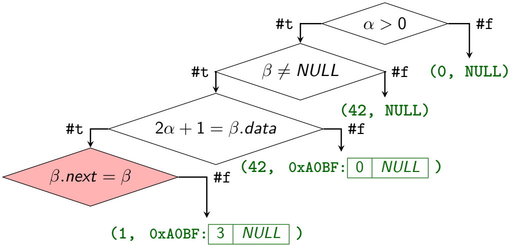
$\mathbb{X}$	$\mathbb{V}$	Ω
x	1	α
p	0xA0BF : | 3 | *NULL* |	β
Φ	$(\alpha > 0) \land (\beta \neq \mathit{NULL}) \land$ $(2\alpha + 1 = \beta.\mathit{data}) \land (\beta.\mathit{next} \neq \beta)$	



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	$(\alpha > 0) \wedge (\beta \neq NULL) \wedge$ $(2\alpha + 1 = \beta.data) \wedge (\beta.next \neq \beta)$	

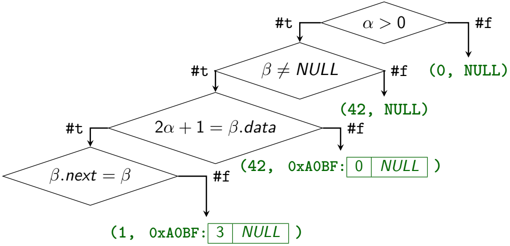


$$(\alpha > 0) \wedge (\beta \neq NULL)$$
$$\wedge (2\alpha + 1 = \beta.data) \wedge (\beta.next = \beta) \text{ is } \textbf{SAT}$$
when $(x, p) = (1, \text{0xA0BF} : \boxed{3 \mid 0xA0BF})$

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*	

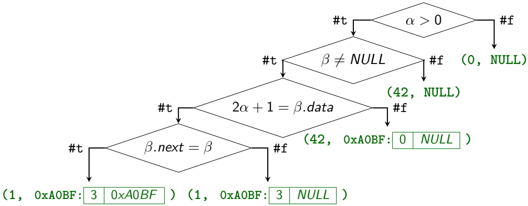


Example – Data Structures

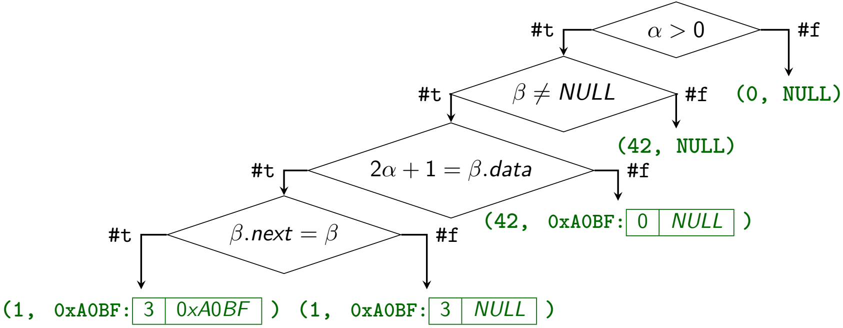
PLRG

```
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	1	α
p	0xA0BF : [3 | $0xA0BF$]	β
Φ	$(\alpha > 0) \wedge (\beta \neq NULL) \wedge (2\alpha + 1 = \beta.data) \wedge (\beta.next = \beta)$	



Example – Data Structures



Realistic Implementation

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

Hybrid Analysis

Symbolic execution is **static**; DSE mixed in **concrete** execution.
Combining the two is **hybrid analysis**.

Hybrid Analysis

Symbolic execution is **static**; DSE mixed in **concrete** execution.
Combining the two is **hybrid analysis**.

	what runs concretely	**what it buys**
DSE	one input, for real	a way past what the solver cannot model
blended analysis	the program, for its call graph	precision; drops paths no run took
heap snapshots	the program, for its heap	facts static analysis misses, e.g., reflection
Concerto, dynamic shortcuts	the part with nothing unknown	precision, and still **sound**

Hybrid Analysis

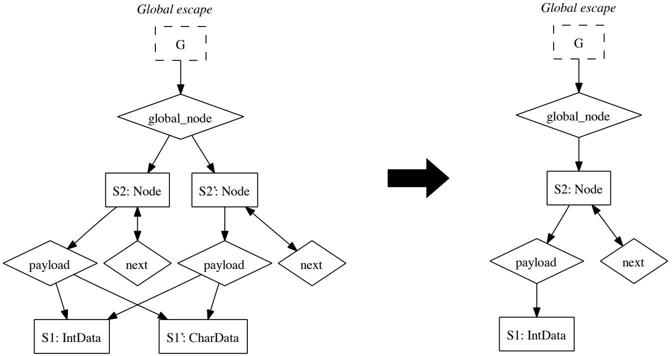
Symbolic execution is **static**; DSE mixed in **concrete** execution.
Combining the two is **hybrid analysis**.

	what runs concretely	**what it buys**
DSE	one input, for real	a way past what the solver cannot model
blended analysis	the program, for its call graph	precision; drops paths no run took
heap snapshots	the program, for its heap	facts static analysis misses, e.g., reflection
Concerto, dynamic shortcuts	the part with nothing unknown	precision, and still **sound**

Recorded runs inherit the runs' **blind spots**. Interleaving keeps **soundness**.

Hybrid Analysis – Blended Analysis

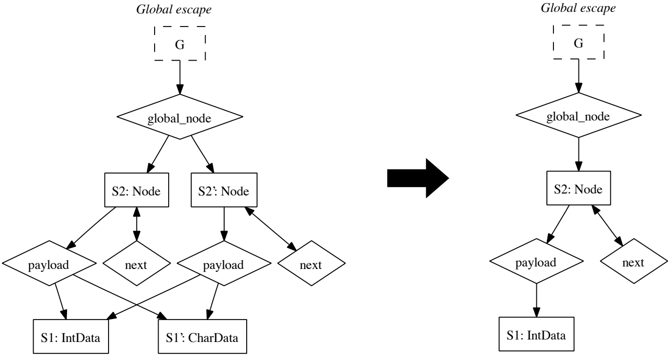
Run the program, keep the call graph it **actually used**, analyze that.[8]



[8] **[ISSTA'07]** B. Dufour, B. G. Ryder, G. Sevitsky. "*Blended Analysis for Performance Understanding of Framework-based Applications.*"

Hybrid Analysis – Blended Analysis

Run the program, keep the call graph it **actually used**, analyze that.[8]

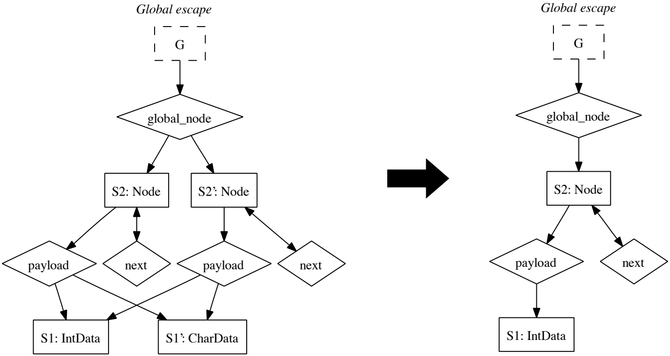


A framework's full call graph is huge and mostly unreachable. **Escape analysis** on the observed part finishes, and is far sharper.

[8] **[ISSTA'07]** B. Dufour, B. G. Ryder, G. Sevitsky. "*Blended Analysis for Performance Understanding of Framework-based Applications.*"

Hybrid Analysis – Blended Analysis

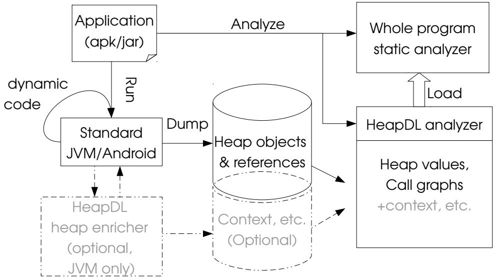
Run the program, keep the call graph it **actually used**, analyze that.[8]



A framework's full call graph is huge and mostly unreachable. **Escape analysis** on the observed part finishes, and is far sharper. **Unsound**: a path no run took is simply not there.

[8] **[ISSTA'07]** B. Dufour, B. G. Ryder, G. Sevitsky. "*Blended Analysis for Performance Understanding of Framework-based Applications.*"

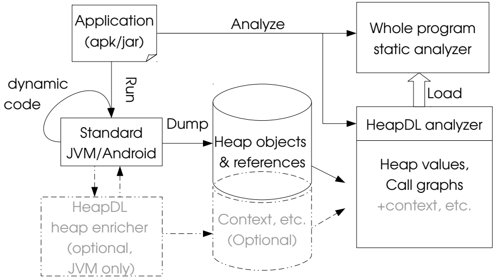
Record the **whole heap** of a run; the static analysis starts from it.[9]



[9] **[OOPSLA'17]** N. Grech, G. Fourtounis, A. Francalanza, Y. Smaragdakis. "*Heaps Don't Lie: Countering Unsoundness with Heap Snapshots.*"

Hybrid Analysis – Heap Snapshots

Record the **whole heap** of a run; the static analysis starts from it.[9]

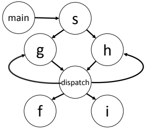


Reflection, native calls and dynamic class loading are what a Java or Android analysis cannot follow. A snapshot has them **already resolved**.

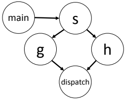
[9] **[OOPSLA'17]** N. Grech, G. Fourtounis, A. Francalanza, Y. Smaragdakis. "*Heaps Don't Lie: Countering Unsoundness with Heap Snapshots.*"

Hybrid Analysis – Combined Interpretation

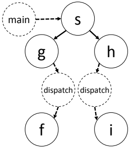
Concerto runs part of the program **concretely** and the rest **abstractly**, in one interpreter, and stays **sound**.[10]



(a) Sound but Imprecise Call-Graph



(b) Unsound Call-Graph

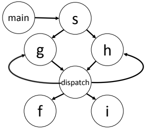


(c) Call-Graph with Concerto

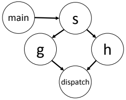
[10] **[POPL'19]** J. Toman, D. Grossman. "*Concerto: A Framework for Combined Concrete and Abstract Interpretation.*"

Hybrid Analysis – Combined Interpretation

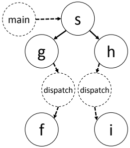
Concerto runs part of the program **concretely** and the rest **abstractly**, in one interpreter, and stays **sound**.[10]



(a) Sound but Imprecise Call-Graph



(b) Unsound Call-Graph



(c) Call-Graph with Concerto

Framework code runs on the **real** configuration; application code stays **abstract**.

[10] **[POPL'19]** J. Toman, D. Grossman. "*Concerto: A Framework for Combined Concrete and Abstract Interpretation.*"

Hybrid Analysis – Dynamic Shortcuts

Run **concretely** while nothing unknown is touched; otherwise **seal** and go back to abstract values. g is native: **no model**, but it **runs**.[11]

```
// JavaScript: analyzed
function f(x) {
  /* 0 */
  if (x >= 0) return x;
  /* 1 */ x = g(-1, x);
  /* 2 */ return -x;
}
```

```
// C: native, can only run
int g(int a, int b) {
  if (a < 0) return b;
  /* complex computation
     on a, b */
}
```

abstract only

0	$\top$

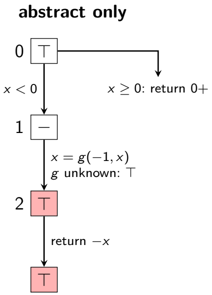
[11] **[ESEC/FSE'21]** J. Park, J. Park, D. Youn, S. Ryu. "*Accelerating JavaScript Static Analysis via Dynamic Shortcuts.*"

Hybrid Analysis – Dynamic Shortcuts

Run **concretely** while nothing unknown is touched; otherwise **seal** and go back to abstract values. g is native: **no model**, but it **runs**.[11]

```
// JavaScript: analyzed
function f(x) {
  /* 0 */
  if (x >= 0) return x;
  /* 1 */ x = g(-1, x);
  /* 2 */ return -x;
}
```

```
// C: native, can only run
int g(int a, int b) {
  if (a < 0) return b;
  /* complex computation
     on a, b */
}
```



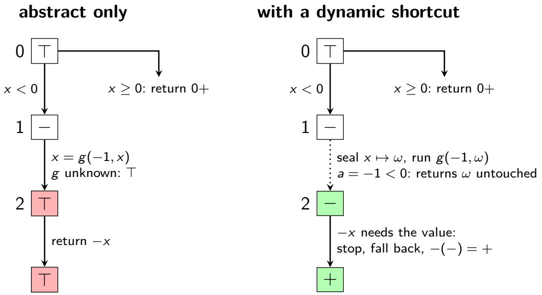
[11] **[ESEC/FSE'21]** J. Park, J. Park, D. Youn, S. Ryu. "*Accelerating JavaScript Static Analysis via Dynamic Shortcuts.*"

Hybrid Analysis – Dynamic Shortcuts

Run **concretely** while nothing unknown is touched; otherwise **seal** and go back to abstract values. g is native: **no model**, but it **runs**.[11]

```
// JavaScript: analyzed
function f(x) {
  /* 0 */
  if (x >= 0) return x;
  /* 1 */ x = g(-1, x);
  /* 2 */ return -x;
}
```

```
// C: native, can only run
int g(int a, int b) {
  if (a < 0) return b;
  /* complex computation
     on a, b */
}
```



-1 picks the branch in g; x passes through **untouched**. $-x$ needs x: the run **stops**, abstract analysis continues. $+$, not $\top$.

[11] **[ESEC/FSE'21]** J. Park, J. Park, D. Youn, S. Ryu. "*Accelerating JavaScript Static Analysis via Dynamic Shortcuts.*"

Summary

1. Symbolic Execution
   - Basic Idea
   - Satisfiability Modulo Theories (SMT)
   - Limitations of Symbolic Execution

2. Dynamic Symbolic Execution (DSE)
   - Search Heuristics
   - Example – Loops
   - Example – Data Structures
   - Realistic Implementation
   - Hybrid Analysis

Homework #2 – Dynamic Symbolic Execution

PLRG

- Implement a **simple DSE** for a small subset of **JavaScript**, using Z3 to solve path conditions.

- Please see this document on GitHub:

  https://github.com/ku-plrg-classroom/js-dse

- The due date is **23:59 on September 28 (Mon.)**, and please submit it to **LMS**.

Next Lecture

- Regression Testing

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