CS 453/698: Software and Systems Security

Module: Bug Finding Tools and Practices

Lecture: Symbolic execution

Meng Xu *(University of Waterloo)*

Winter 2025

Outline

Illustration

```
fn foo(x: u64): u64 {
    if (x * 3 == 42) {
        some_hidden_bug();
    }
    if (x * 5 == 42) {
        some_hidden_bug();
    }
    return 2 * x;
}
```

Illustration

Unit Test

```
foo(0);
foo(1);
```

```
fn foo(x: u64): u64 {
    if (x * 3 == 42) {
        some_hidden_bug();
    }
    if (x * 5 == 42) {
        some_hidden_bug();
    }
    return 2 * x;
}
```

Illustration

```
fn foo(x: u64): u64 {
    if (x * 3 == 42) {
        some_hidden_bug();
    }
    if (x * 5 == 42) {
        some_hidden_bug();
    }
    return 2 * x;
}
```

Unit Test

```
foo(0);
foo(1);
```

Fuzzing

```
foo(0);
foo(1);
foo(12);
foo(78);
......
foo(9,223,372,036,854,775,808);
```

Illustration

```
fn foo(x: u64): u64 {
    if (x * 3 == 42) {
        some_hidden_bug();
    }
    if (x * 5 == 42) {
        some_hidden_bug();
    }
    return 2 * x;
}
```

Unit Test

`foo(0);`

`foo(1);`

Fuzzing

`foo(0);`

`foo(1);`

`foo(12);`

`foo(78);`

......

`foo(9,223,372,036,854,775,808);`

Symbolic execution

foo(x)

aborts when $x = 14$

returns $2x$ otherwise

Satisfiability Modulo Theories (SMT)

Definition: A procedure that decides whether a mathematical formula is satisfiable.

Example:

- $3x = 42$
- $2x \geq 2^{64}$
- $5x = 42$

Satisfiability Modulo Theories (SMT)

Definition: A procedure that decides whether a mathematical formula is satisfiable.

Example:

- $3x = 42 \longrightarrow$ satisfiable with $x = 14$
- $2x \geq 2^{64} \longrightarrow$ satisfiable with $x \geq 2^{63}$
- $5x = 42 \longrightarrow$ unsatisfiable, cannot find an x

Ask two question whenever you see a symbolic execution work:

- How does it convert code into mathematical formula?
- What does it try to solve for?

Program Modeling Desiderata

- Control-flow graph exploration
- Loop handling
- Memory modeling
- Concurrency

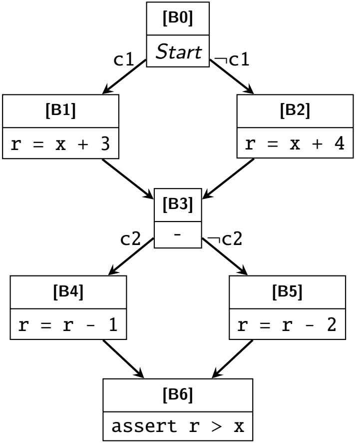
Outline

An example of a pure function

```
fn foo(
    c1: bool, c2: bool,
    x: u64
) -> u64 {
    let r = if (c1) {
        x + 3
    } else {
        x + 4
    };

    let r = if (c2) {
        r - 1
    } else {
        r - 2
    };

    r
}
spec foo {
    ensures r > x;
}
```

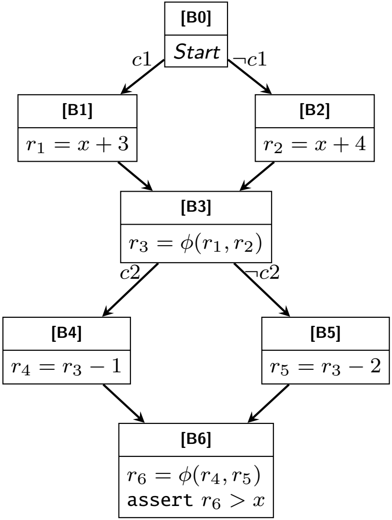
An example of a pure function

```
fn foo(
    c1: bool, c2: bool,
    x: u64
) -> u64 {
    let r = if (c1) {
        x + 3
    } else {
        x + 4
    };

    let r = if (c2) {
        r - 1
    } else {
        r - 2
    };

    r
}
spec foo {
    ensures r > x;
}
```



The example in SSA form

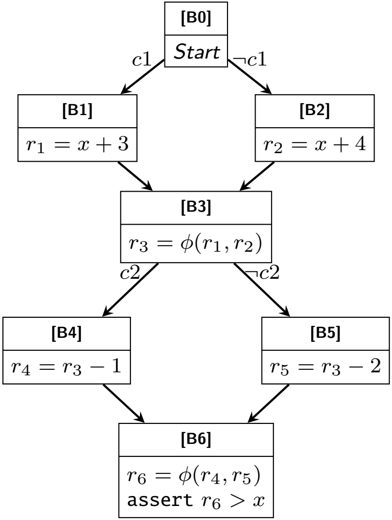
```
fn foo(
    c1: bool, c2: bool,
    x: u64
) -> u64 {
    let r = if (c1) {
        x + 3
    } else {
        x + 4
    };

    let r = if (c2) {
        r - 1
    } else {
        r - 2
    };

    r
}
spec foo {
    ensures r > x;
}
```



Path-based exploration

Vars: $c1$, $c2$, x, r_{1-6}

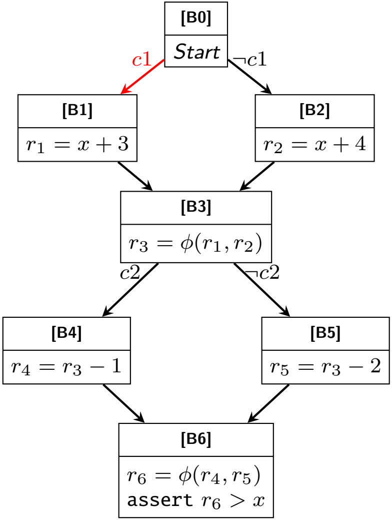
B0	Sym. repr. Path cond.	$\emptyset$ True



Path-based exploration

Vars: $c1$, $c2$, x, r_{1-6}

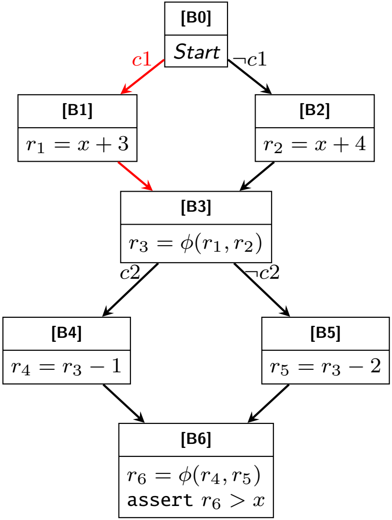
B0	Sym. repr. Path cond.	$\emptyset$ True
B1	Sym. repr. Path cond.	$r_1 = x + 3$ $c1$



Path-based exploration

Vars: $c1$, $c2$, x, r_{1-6}

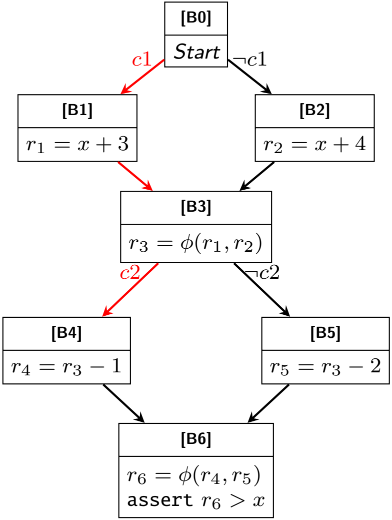
B0	Sym. repr. Path cond.	$\emptyset$ True
B1	Sym. repr. Path cond.	$r_1 = x + 3$ $c1$
B3	Sym. repr. Path cond.	$r_1 = x + 3$ $r_3 = r_1$ $c1$



Path-based exploration

Vars: $c1$, $c2$, x, r_{1-6}

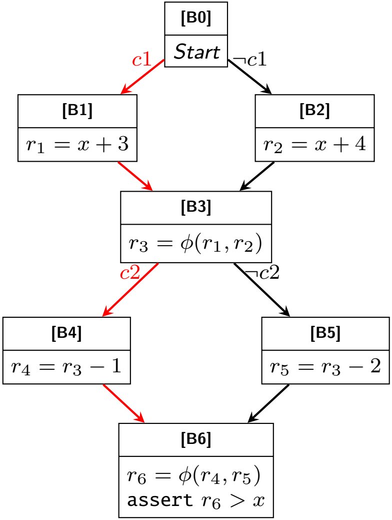
B0	Sym. repr. Path cond.	$\emptyset$ True
B1	Sym. repr. Path cond.	$r_1 = x + 3$ $c1$
B3	Sym. repr. Path cond.	$r_1 = x + 3$ $r_3 = r_1$ $c1$
B4	Sym. repr. Path cond.	$r_1 = x + 3$ $r_3 = r_1$ $r_4 = r_3 - 1$ $c_1 \wedge c_2$



Path-based exploration

Vars: $c1$, $c2$, x, r_{1-6}

B0	Sym. repr. Path cond.	$\emptyset$ True
B1	Sym. repr. Path cond.	$r_1 = x + 3$ $c1$
B3	Sym. repr. Path cond.	$r_1 = x + 3$ $r_3 = r_1$ $c1$
B4	Sym. repr. Path cond.	$r_1 = x + 3$ $r_3 = r_1$ $r_4 = r_3 - 1$ $c_1 \wedge c_2$
B6	Sym. repr. Path cond.	$r_1 = x + 3$ $r_3 = r_1$ $r_4 = r_3 - 1$ $r_6 = r_4$ $c_1 \wedge c_2$

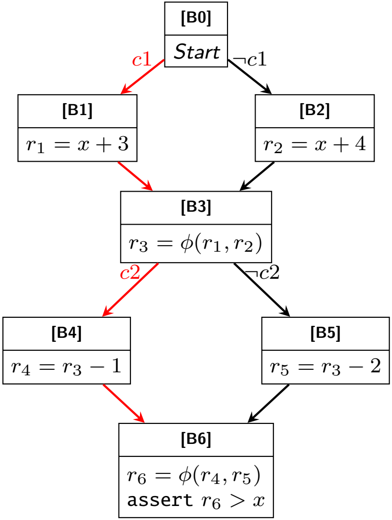


Proving procedure (per path)

Vars: $c1$, $c2$, x, r_{1-6}

B6	Sym. repr.	$r_1 = x + 3$
		$r_3 = r_1$
		$r_4 = r_3 - 1$
		$r_6 = r_4$
	Path cond.	$c_1 \wedge c_2$

$\rightsquigarrow$



Proving procedure (per path)

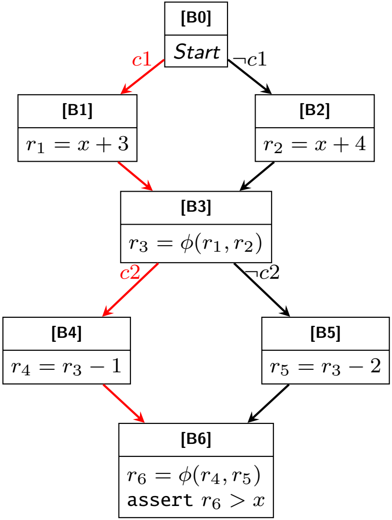
Vars: $c1$, $c2$, x, r_{1-6}

B6	Sym. repr.	$r_1 = x + 3$ $r_3 = r_1$ $r_4 = r_3 - 1$ $r_6 = r_4$
	Path cond.	$c_1 \wedge c_2$

$\rightsquigarrow$

Prove that $\forall\ c1, c2, x, r_{1-6}$:

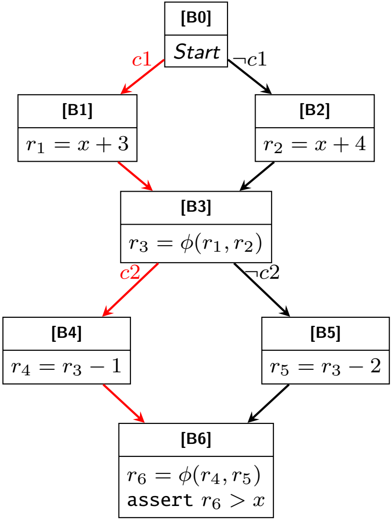
$$
\begin{aligned}
&((c1 \wedge c2) \wedge (\\
&\quad (r_1 = x + 3) \\
&\quad (r_3 = r_1) \\
&\quad (r_4 = r_3 - 1) \\
&\quad (r_6 = r_4) \\
&)) \Rightarrow (r_6 > x)
\end{aligned}
$$



Proving procedure (all paths)

Prove that
$\forall\ c1, c2, x, r_{1-6}$:

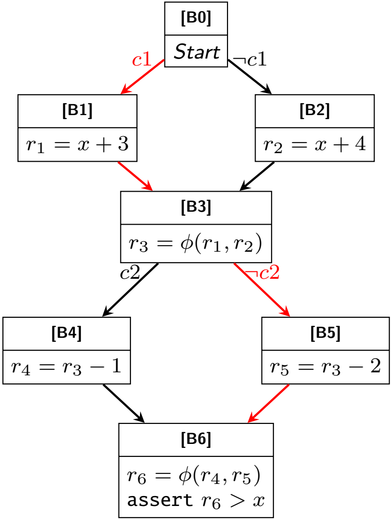
$$
\begin{aligned}
&((c1 \wedge c2) \wedge (\\
&\quad (r_1 = x + 3) \\
&\quad (r_3 = r_1) \\
&\quad (r_4 = r_3 - 1) \\
&\quad (r_6 = r_4) \\
&)) \Rightarrow (r_6 > x)
\end{aligned}
$$



Proving procedure (all paths)

Prove that
$\forall\, c1, c2, x, r_{1-6}$:

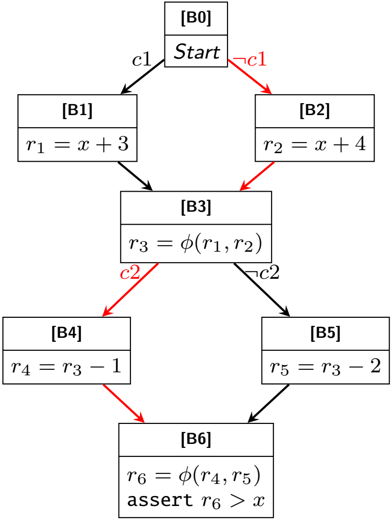
$$
\begin{aligned}
&((c1 \land \neg c2) \land (\\
&\quad (r_1 = x + 3)\\
&\quad (r_3 = r_1)\\
&\quad (r_5 = r_3 - 2)\\
&\quad (r_6 = r_5)\\
&)) \Rightarrow (r_6 > x)
\end{aligned}
$$



Proving procedure (all paths)

Prove that
$\forall\ c1, c2, x, r_{1-6}$:

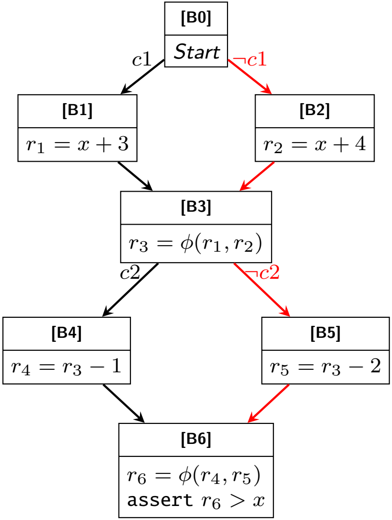
$$((\neg c1 \wedge c2) \wedge ($$
$$(r_2 = x + 4)$$
$$(r_3 = r_2)$$
$$(r_4 = r_3 - 1)$$
$$(r_6 = r_4)$$
$$)) \Rightarrow (r_6 > x)$$



Proving procedure (all paths)

Prove that
$\forall\, c1, c2, x, r_{1-6}$:

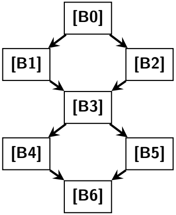
$$
\begin{aligned}
&((\neg c1 \wedge \neg c2) \wedge (\\
&\quad (r_2 = x + 4) \\
&\quad (r_3 = r_2) \\
&\quad (r_5 = r_3 - 2) \\
&\quad (r_6 = r_5) \\
&)) \Rightarrow (r_6 > x)
\end{aligned}
$$



Path explosion

Path explosion

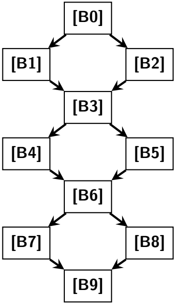
$2^{\color{red}2}$ paths



Path explosion

$2^{\color{red}{2}}$ paths

$2^{\color{red}{3}}$ paths



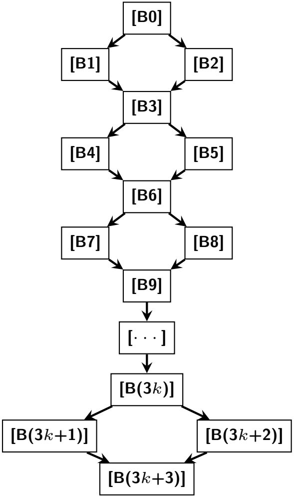
Path explosion

$2^{\color{red}2}$ paths

$2^{\color{red}3}$ paths

$\dots$

$2^{\color{red}k}$ paths



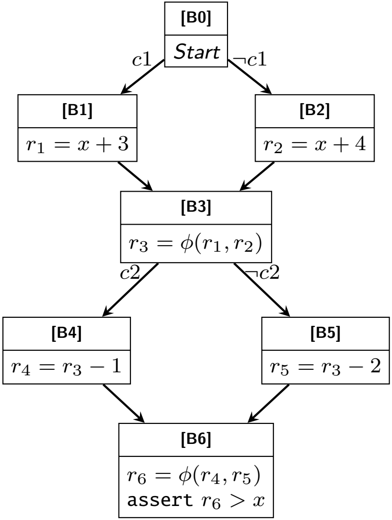
Outline

Weakest precondition calculus

When used in an automated formal verification context, most symbolic executors adopt a backward state exploration process, following the weakest precondition calculus.

The running example, once again

```
fn foo(
    c1: bool, c2: bool,
    x: u64
) -> u64 {
    let r = if (c1) {
        x + 3
    } else {
        x + 4
    };

    let r = if (c2) {
        r - 1
    } else {
        r - 2
    };

    r
}
spec foo {
    ensures r > x;
}
```



The passification process

Convert the program into a dynamic single assignment (DSA) form.

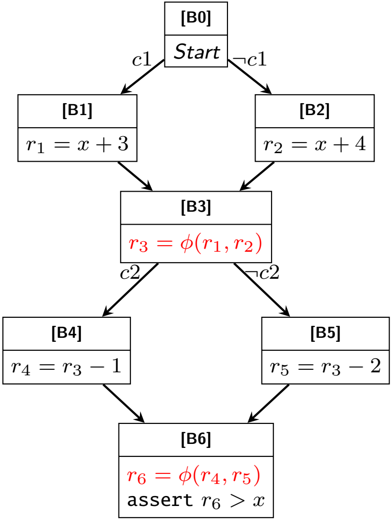
The passification process

Convert the program into a dynamic single assignment (DSA) form.

DSA is extremely similar to static single assignment (SSA) with the ϕ-node eagerly uplifted.

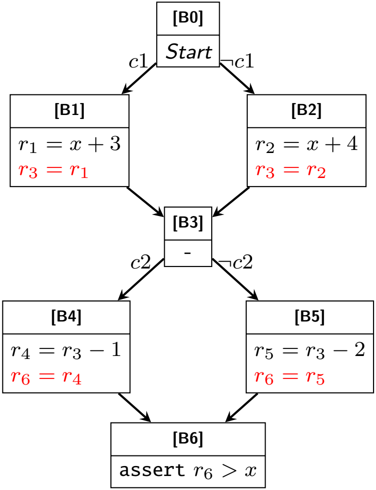
The passification process

```
fn foo(
    c1: bool, c2: bool,
    x: u64
) -> u64 {
    let r = if (c1) {
        x + 3
    } else {
        x + 4
    };

    let r = if (c2) {
        r - 1
    } else {
        r - 2
    };

    r
}
spec foo {
    ensures r > x;
}
```



The passification process

```
fn foo(
    c1: bool, c2: bool,
    x: u64
) -> u64 {
    let r = if (c1) {
        x + 3
    } else {
        x + 4
    };

    let r = if (c2) {
        r - 1
    } else {
        r - 2
    };

    r
}
spec foo {
    ensures r > x;
}
```



The walk-up process

Do a topological sort on the CFG and traverse backward.

The walk-up process

Do a topological sort on the CFG and traverse backward.

This ensures that for each block in the CFG, we visit it *once and only once* (assuming no loops).

The walk-up algorithm

Follow these rules for the intra-block walk-up process:

- $wp(\text{assert } c) = c$
- $wp(\text{assert } c, Q) = c \wedge Q$
- $wp(\text{assign } e, Q) = e \implies Q$
- $wp(s_1; s_2, Q) = wp(s_1, wp(s_2, Q))$

The walk-up algorithm

Follow these rules for the intra-block walk-up process:

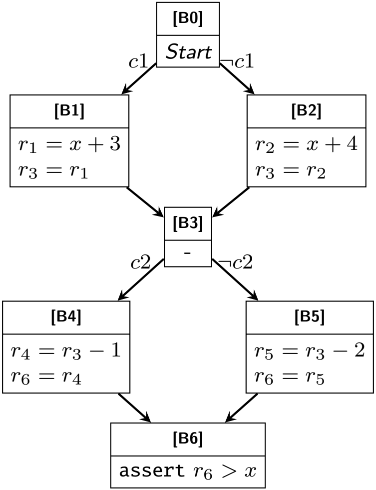
- $wp(\text{assert } c) = c$
- $wp(\text{assert } c, Q) = c \wedge Q$
- $wp(\text{assign } e, Q) = e \implies Q$
- $wp(s_1; s_2, Q) = wp(s_1, wp(s_2, Q))$

The rule for inter-block walk-up is:

$$A \leftarrow wp(s_1; s_2; ...; s_n, \bigwedge_{B \in \text{Succ}(A)} B)$$

The walk-up process with an example

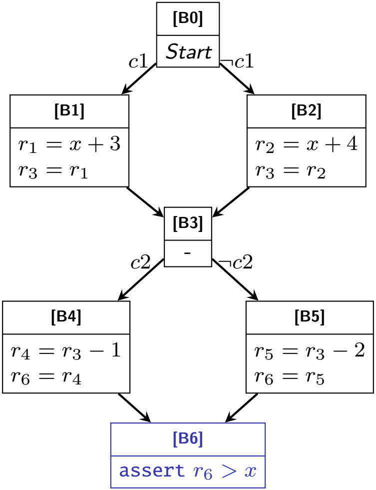
Vars: $c1$, $c2$, x, r_{1-6}, B_{0-6}



The walk-up process with an example

Vars: $c1$, $c2$, x, r_{1-6}, B_{0-6}

$B_6 \leftarrow r_6 > x$

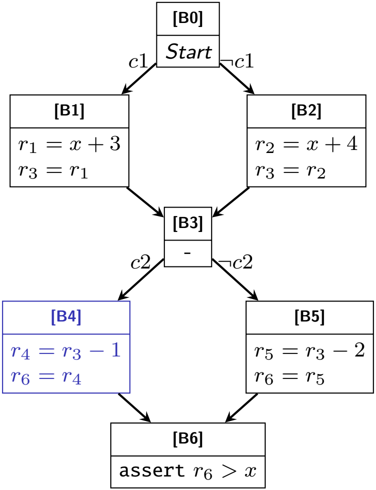


The walk-up process with an example

Vars: $c1$, $c2$, x, r_{1-6}, B_{0-6}

$B_6 \leftarrow r_6 > x$

$$B_4 \leftarrow (c2) \Rightarrow ((r_4 = r_3 - 1) \Rightarrow ((r_6 = r_4) \Rightarrow B_6))$$



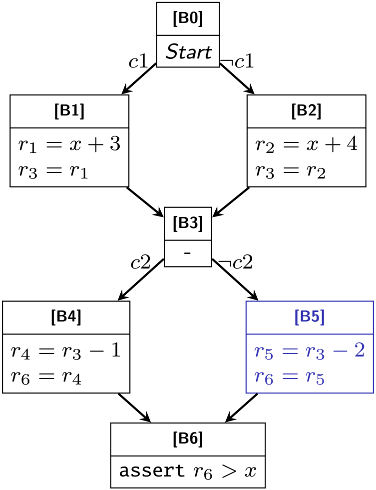
The walk-up process with an example

Vars: $c1,\ c2,\ x,\ r_{1-6},\ B_{0-6}$

$B_6 \leftarrow r_6 > x$

$$B_4 \leftarrow (c2) \Rightarrow (\\ (r_4 = r_3 - 1) \Rightarrow (\\ (r_6 = r_4) \Rightarrow B_6))$$

$$B_5 \leftarrow (\neg c2) \Rightarrow (\\ (r_5 = r_3 - 2) \Rightarrow (\\ (r_6 = r_5) \Rightarrow B_6))$$



The walk-up process with an example

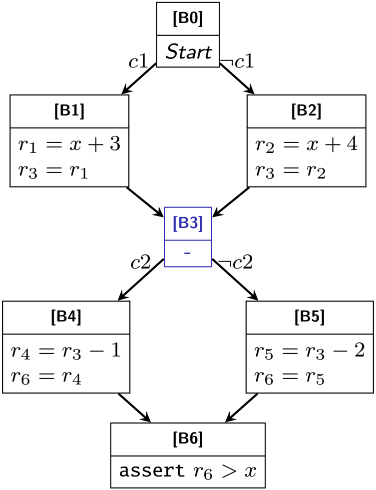
Vars: $c1,\ c2,\ x,\ r_{1-6},\ B_{0-6}$

$B_6 \leftarrow r_6 > x$

$$B_4 \leftarrow (c2) \Rightarrow ((r_4 = r_3 - 1) \Rightarrow ((r_6 = r_4) \Rightarrow B_6))$$

$$B_5 \leftarrow (\neg c2) \Rightarrow ((r_5 = r_3 - 2) \Rightarrow ((r_6 = r_5) \Rightarrow B_6))$$

$B_3 \leftarrow B_4 \wedge B_5$



The walk-up process with an example

Vars: $c1$, $c2$, x, r_{1-6}, B_{0-6}

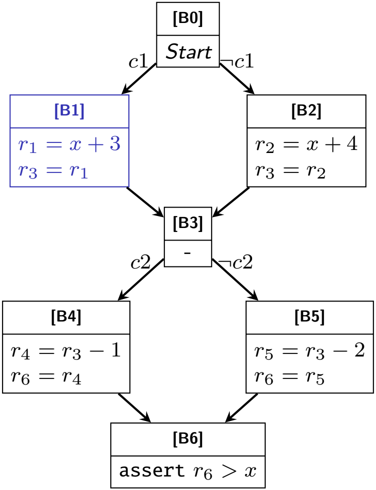
$B_6 \leftarrow r_6 > x$

$$B_4 \leftarrow (c2) \Rightarrow (\\ (r_4 = r_3 - 1) \Rightarrow (\\ (r_6 = r_4) \Rightarrow B_6))$$

$$B_5 \leftarrow (\neg c2) \Rightarrow (\\ (r_5 = r_3 - 2) \Rightarrow (\\ (r_6 = r_5) \Rightarrow B_6))$$

$B_3 \leftarrow B_4 \wedge B_5$

$$B_1 \leftarrow (c1) \Rightarrow (\\ (r_1 = x + 3) \Rightarrow (\\ (r_3 = r_1) \Rightarrow B_3))$$



The walk-up process with an example

Vars: $c1,\ c2,\ x,\ r_{1-6},\ B_{0-6}$

$B_6 \leftarrow r_6 > x$

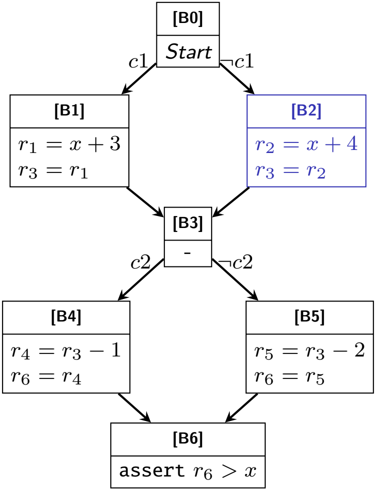
$$B_4 \leftarrow (c2) \Rightarrow ((r_4 = r_3 - 1) \Rightarrow ((r_6 = r_4) \Rightarrow B_6))$$

$$B_5 \leftarrow (\neg c2) \Rightarrow ((r_5 = r_3 - 2) \Rightarrow ((r_6 = r_5) \Rightarrow B_6))$$

$B_3 \leftarrow B_4 \wedge B_5$

$$B_1 \leftarrow (c1) \Rightarrow ((r_1 = x + 3) \Rightarrow ((r_3 = r_1) \Rightarrow B_3))$$

$$B_2 \leftarrow (\neg c1) \Rightarrow ((r_2 = x + 4) \Rightarrow ((r_3 = r_2) \Rightarrow B_3))$$



The walk-up process with an example

Vars: $c1$, $c2$, x, r_{1-6}, B_{0-6}

$B_6 \leftarrow r_6 > x$

$$B_4 \leftarrow (c2) \Rightarrow ((r_4 = r_3 - 1) \Rightarrow ((r_6 = r_4) \Rightarrow B_6))$$

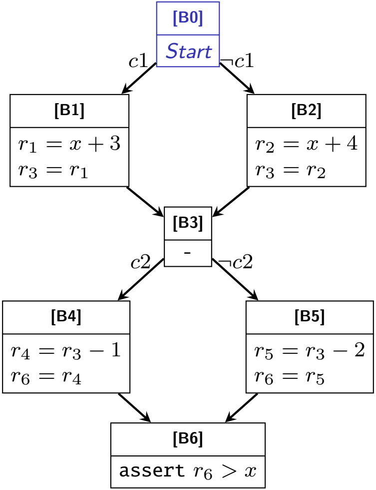
$$B_5 \leftarrow (\neg c2) \Rightarrow ((r_5 = r_3 - 2) \Rightarrow ((r_6 = r_5) \Rightarrow B_6))$$

$B_3 \leftarrow B_4 \wedge B_5$

$$B_1 \leftarrow (c1) \Rightarrow ((r_1 = x + 3) \Rightarrow ((r_3 = r_1) \Rightarrow B_3))$$

$$B_2 \leftarrow (\neg c1) \Rightarrow ((r_2 = x + 4) \Rightarrow ((r_3 = r_2) \Rightarrow B_3))$$

$B_0 \leftarrow B_1 \wedge B_2$

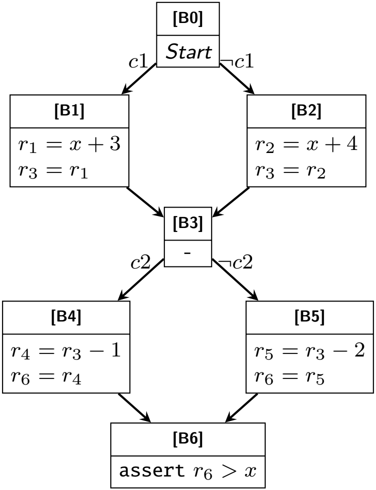


Proving procedure

Prove that
$\forall\ c1, c2, x, r_{1-6}, B_{0-6}$:

$$
\begin{aligned}
&B_6 \leftarrow r_6 > x\\
&B_4 \leftarrow (c2) \Rightarrow (\\
&\qquad (r_4 = r_3 - 1) \Rightarrow (\\
&\qquad\quad (r_6 = r_4) \Rightarrow B_6))\\
&B_5 \leftarrow (\neg c2) \Rightarrow (\\
&\qquad (r_5 = r_3 - 2) \Rightarrow (\\
&\qquad\quad (r_6 = r_5) \Rightarrow B_6))\\
&B_3 \leftarrow B_4 \wedge B_5\\
&B_1 \leftarrow (c1) \Rightarrow (\\
&\qquad (r_1 = x + 3) \Rightarrow (\\
&\qquad\quad (r_3 = r_1) \Rightarrow B_3))\\
&B_2 \leftarrow (\neg c1) \Rightarrow (\\
&\qquad (r_2 = x + 4) \Rightarrow (\\
&\qquad\quad (r_3 = r_2) \Rightarrow B_3))\\
&B_0 \leftarrow B_1 \wedge B_2
\end{aligned}
$$

$B_0 =$ True



Comparison of forward and backward symbolic execution

Prove that $\forall\, c1, c2, x, r_{1-6}$:

$$
\begin{array}{l}
((c1 \land c2) \land (\\
\quad (r_1 = x + 3) \\
\quad (r_3 = r_1) \\
\quad (r_4 = r_3 - 1) \\
\quad (r_6 = r_4) \\
)) \Rightarrow (r_6 > x)
\end{array}
$$

However, need to repeat this process multiple (worst case exponential) times.

Prove that
$\forall\, c1, c2, x, r_{1-6}, B_{0-6}$:

$$
\begin{array}{l}
B_6 \leftarrow r_6 > x \\
B_4 \leftarrow (c2) \Rightarrow (\\
\qquad\qquad (r_4 = r_3 - 1) \Rightarrow (\\
\qquad\qquad\quad (r_6 = r_4) \Rightarrow B_6)) \\
B_5 \leftarrow (\neg c2) \Rightarrow (\\
\qquad\qquad (r_5 = r_3 - 2) \Rightarrow (\\
\qquad\qquad\quad (r_6 = r_5) \Rightarrow B_6)) \\
B_3 \leftarrow B_4 \land B_5 \\
B_1 \leftarrow (c1) \Rightarrow (\\
\qquad\qquad (r_1 = x + 3) \Rightarrow (\\
\qquad\qquad\quad (r_3 = r_1) \Rightarrow B_3)) \\
B_2 \leftarrow (\neg c1) \Rightarrow (\\
\qquad\qquad (r_2 = x + 4) \Rightarrow (\\
\qquad\qquad\quad (r_3 = r_2) \Rightarrow B_3)) \\
B_0 \leftarrow B_1 \land B_2
\end{array}
$$

$B_0 = \text{True}$

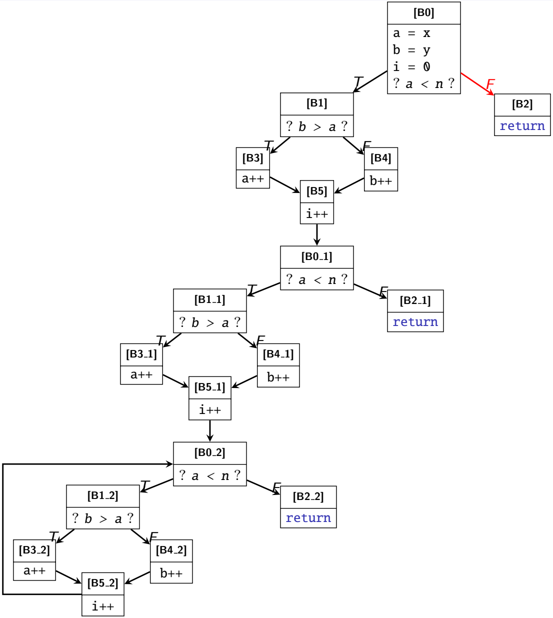
Outline

How about loops?

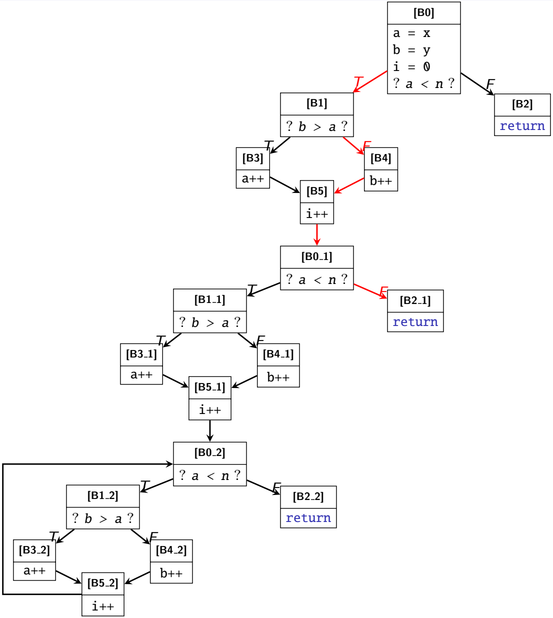
```
// a library function
fn sync(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```
// core application logic
pub fn main() {
  let (x, y, n) = input();
  let (a, b, i) = sync(x, y, n);
  assert!(i == 0 || i < 2*n);

}
```

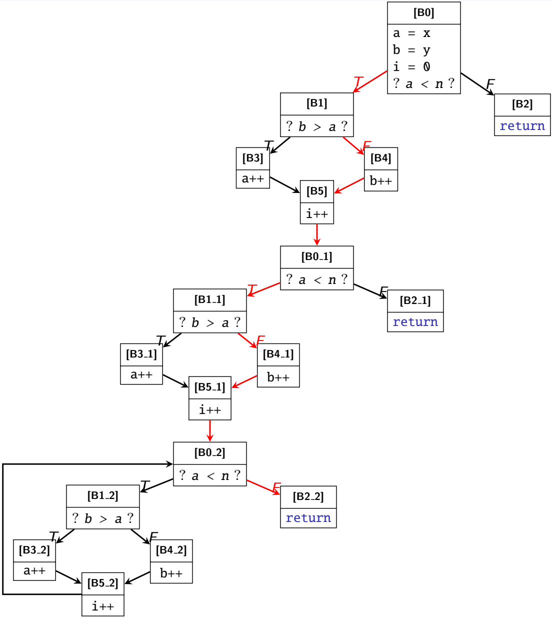
Conventional symbolic execution



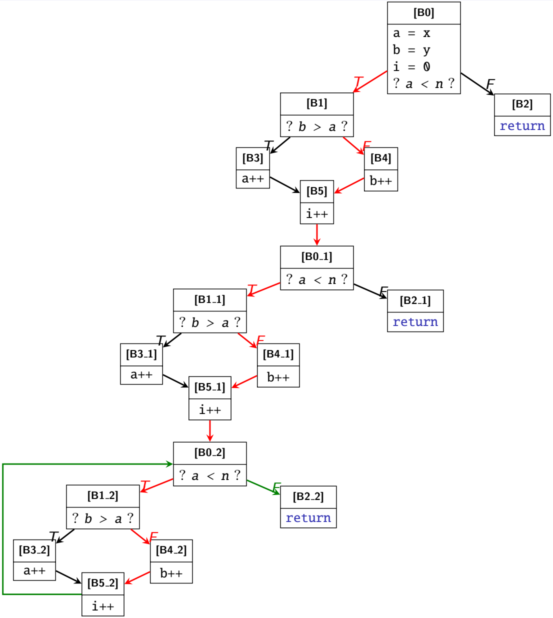
Conventional symbolic execution



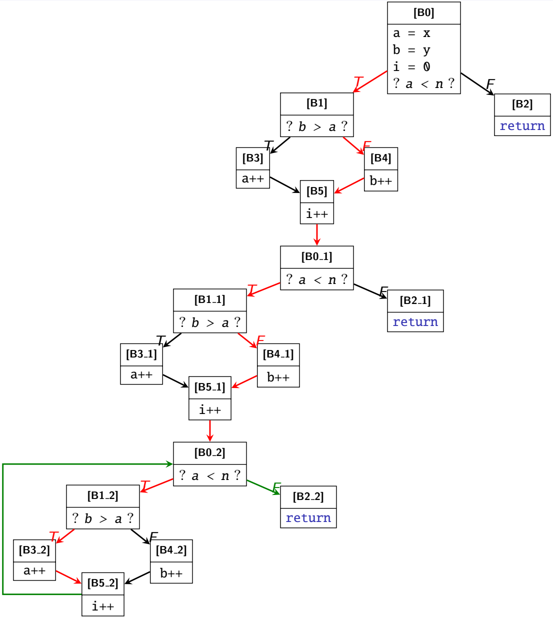
Conventional symbolic execution



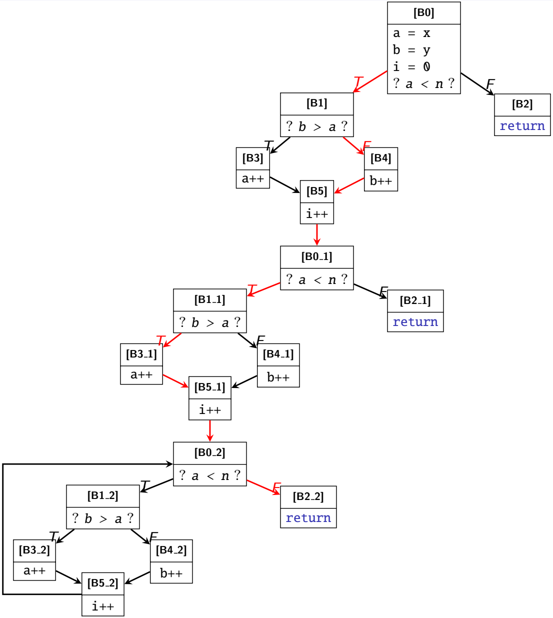
Conventional symbolic execution



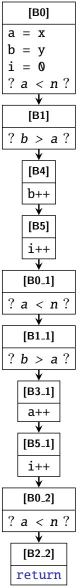
Conventional symbolic execution



Conventional symbolic execution



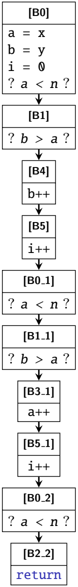
Encoding the path conditions



Find x, y, n such that

- $x < n$ (from [B0])
- $y \leq x$ (from [B1])
- $x < n$ (from [B0_1])
- $y + 1 > x$ (from [B1_1])
- $x + 1 \geq n$ (from [B0_2])
- $n \neq 0 \wedge i \geq 2n$ (from `assert!`)

Encoding the path conditions



Find x, y, n such that

- $x < n$ (from [B0])
- $y \leq x$ (from [B1])
- $x < n$ (from [B0_1])
- $y + 1 > x$ (from [B1_1])
- $x + 1 \geq n$ (from [B0_2])
- $n \neq 0 \wedge i \geq 2n$ (from `assert!`)

Solving the predicates yield:
$\{\ x = 0,\ y = 0,\ n = 1\ \}$

Symbolic execution for bug finding

```
// a library function
fn sync(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```
// core application logic
pub fn main() {
  let (x, y, n) = input();
  let (a, b, i) = sync(x, y, n);
  assert!(i == 0 || i < 2*n);

}
```

- x=0, y=0, n=1 $\rightarrow$ a=1, b=1, i=2
- x=0, y=0, n=2 $\rightarrow$ a=2, b=2, i=4
- ......
- x=0, y=0, n=k $\rightarrow$ a=k, b=k, i=$2k$

Path explosion in symbolic execution

```rust
// a library function
fn sync(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```rust
// core application logic
pub fn main() {
  let (x, y, n) = input();
  let (a, b, i) = sync(x, y, n);
  assert!(n-a-b+i != 42);

}
```

Q: What if a bug can only be triggered after exploring k branches?

Path explosion in symbolic execution

```
// a library function
fn sync(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```
// core application logic
pub fn main() {
  let (x, y, n) = input();
  let (a, b, i) = sync(x, y, n);
  assert!(n-a-b+i != 42);

}
```

Q: What if a bug can only be triggered after exploring k branches?

In fact, this bug can only be triggered after at least 42 levels of loop unrolling.

- x=0, y=0, n=42 $\rightarrow$ a=42, b=42, i=84
- x=9, y=5, n=56 $\rightarrow$ a=56, b=56, i=98

In the conventional way of symbolic execution, finding this bug requires an exhaustive search of 2^{42} paths.

Outline

Definition of concolic execution

Background: **concolic**, as the name suggests, is the combination of two English words: *concrete* and *symbolic*, and the order matters!

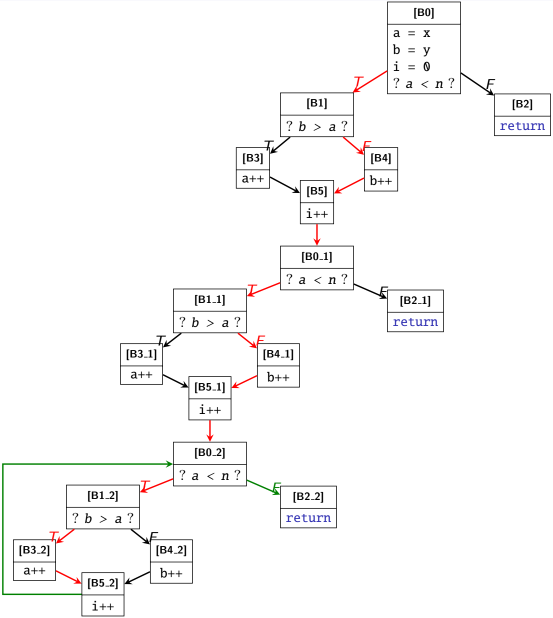
Definition of concolic execution

Background: **concolic**, as the name suggests, is the combination of two English words: *concrete* and *symbolic*, and the order matters!

The basic idea of concolic execution is:

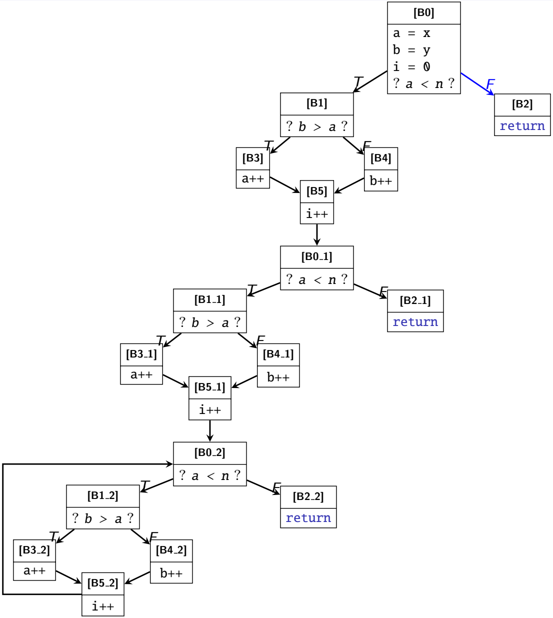
1. Execute a test case concretely
2. For each branch encountered in the test case,
   find another test case that toggles this branch symbolically.

Concolic execution with the running example



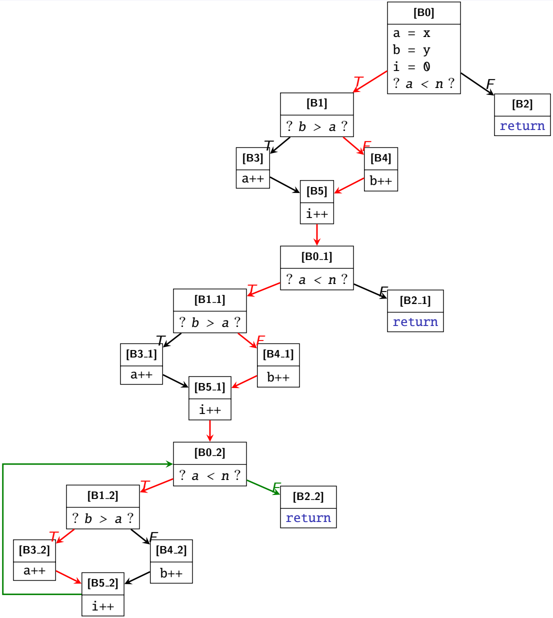
{x=1, y=0, n=2}

Concolic execution with the running example



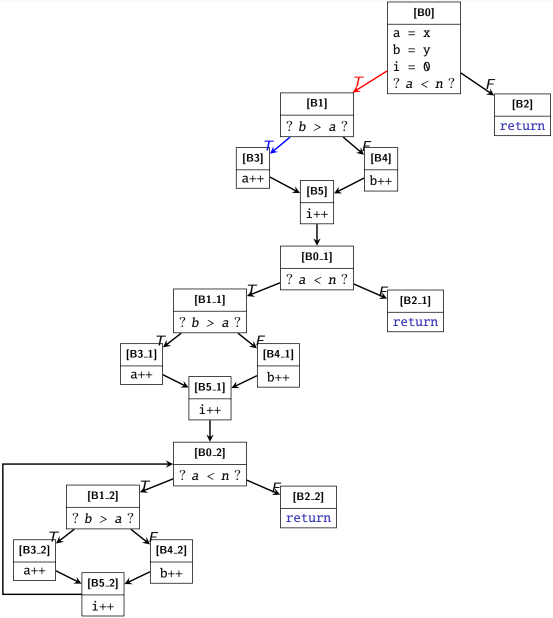
{x=9, y=2, n=6}

Concolic execution with the running example



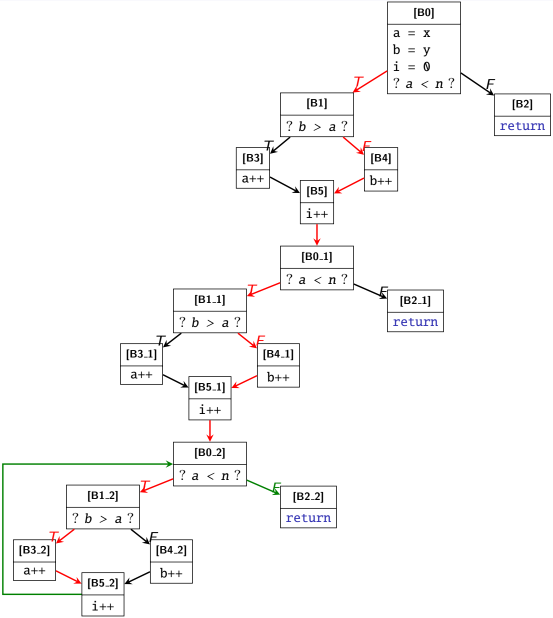
{x=1, y=0, n=2}

Concolic execution with the running example



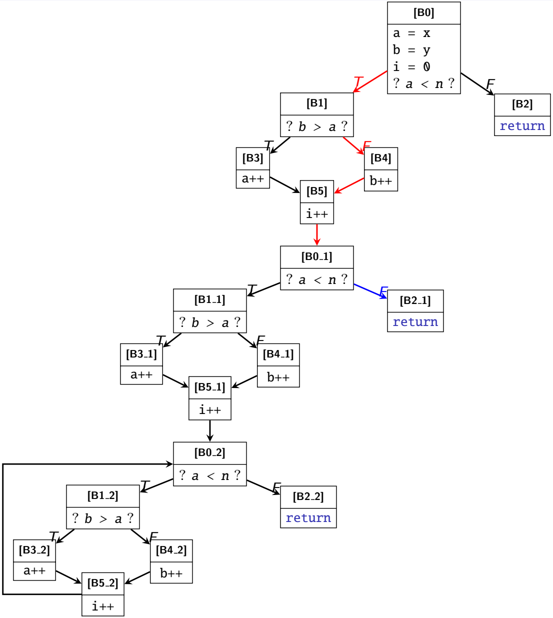
{x=3, y=4, n=5}

Concolic execution with the running example



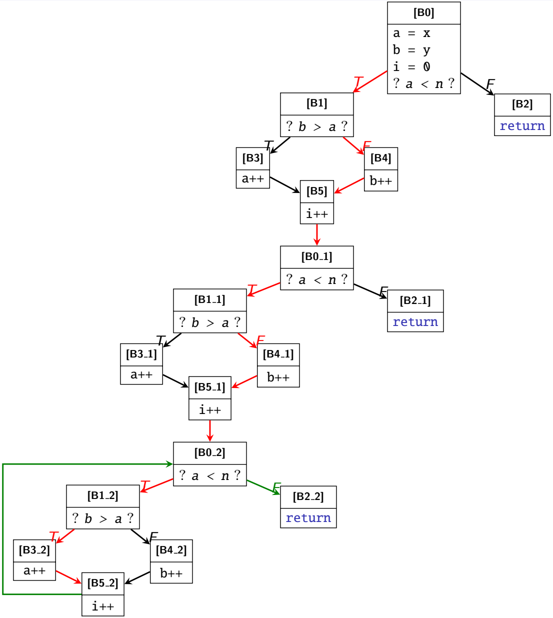
{x=1, y=0, n=2}

Concolic execution with the running example



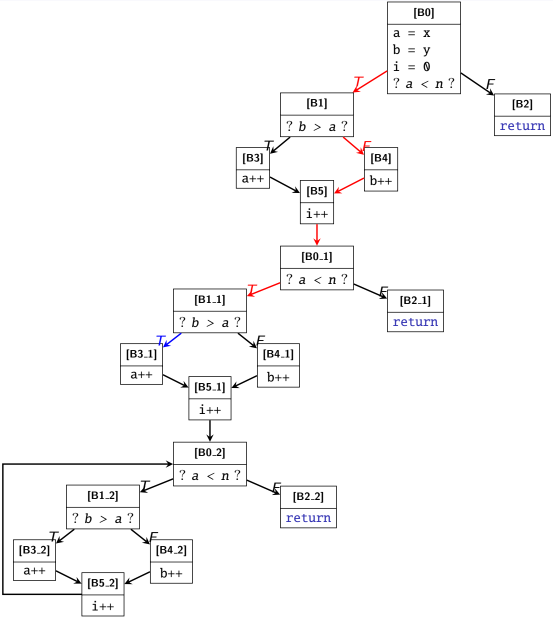
⟨*infeasible*⟩

Concolic execution with the running example



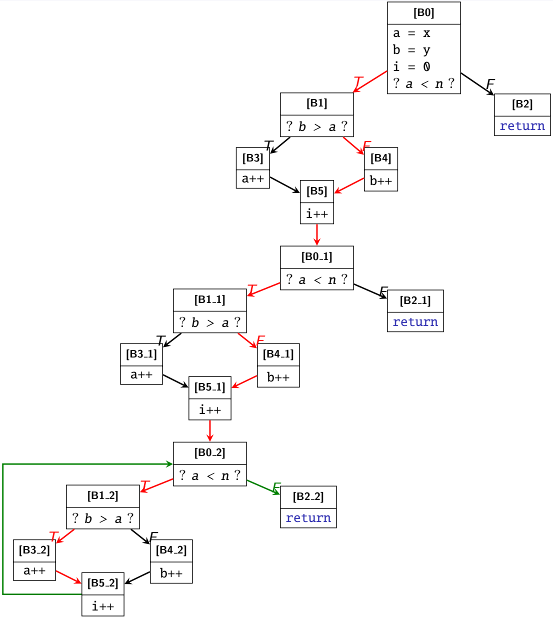
$\{x=1,\ y=0,\ n=2\}$

Concolic execution with the running example



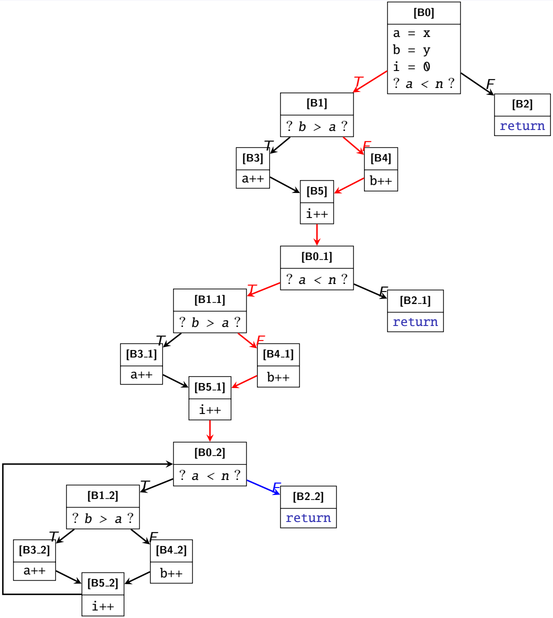
{x=5, y=5, n=8}

Concolic execution with the running example



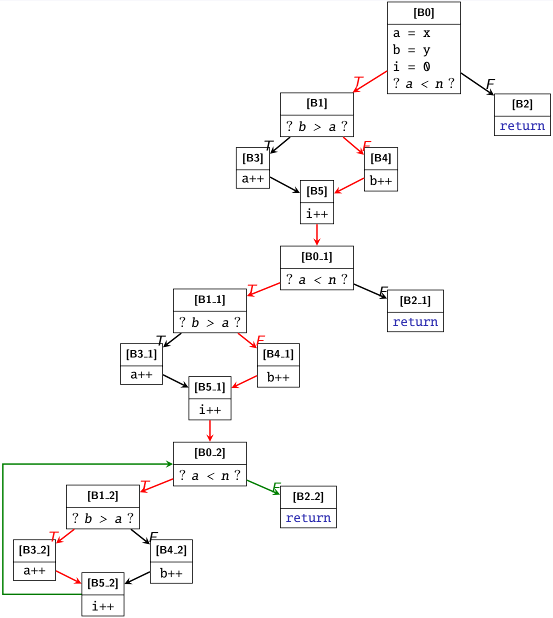
{x=1, y=0, n=2}

Concolic execution with the running example



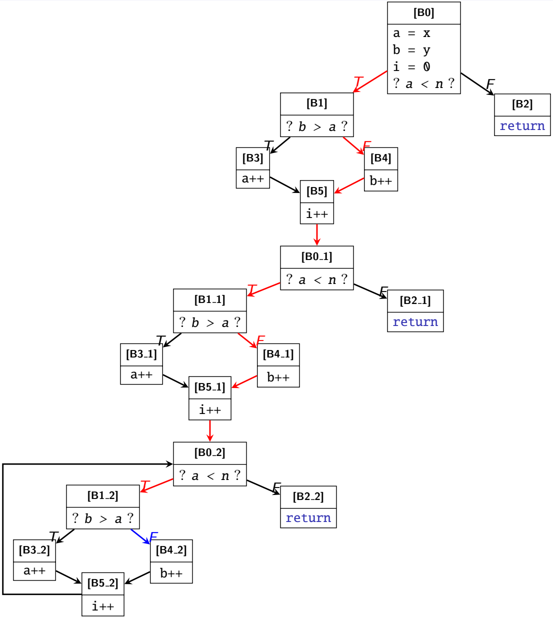
⟨*infeasible*⟩

Concolic execution with the running example



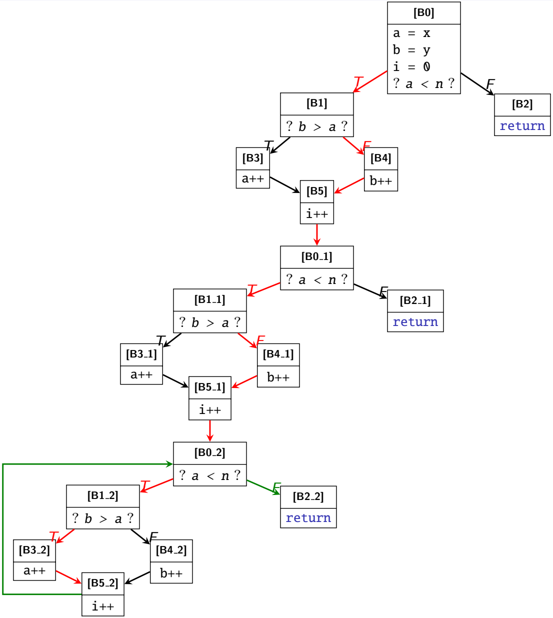
{x=1, y=0, n=2}

Concolic execution with the running example



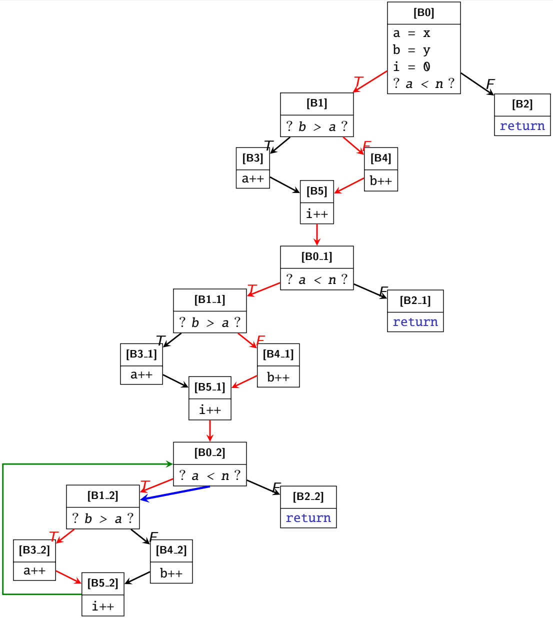
{x=7, y=3, n=9}

Concolic execution with the running example



{x=1, y=0, n=2}

Concolic execution with the running example



. . . endless loop . . .

$\langle$ **End** $\rangle$