Symbolic execution

Software Analysis
Topic 7

Carlo A. Furia
USI – Università della Svizzera Italiana

Today's menu

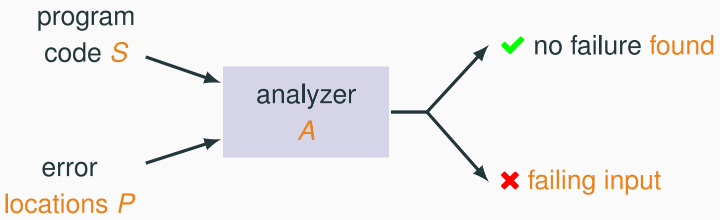
Symbolic execution

- Classic symbolic execution
- Dynamic symbolic execution

Challenges, tools, and applications

- A brief demo of Klee

Symbolic execution: the very idea



Symbolic execution:

- analyzes real program code
- analyzes reachability properties (equivalent to assertions), which offers a good flexibility
- is automatic
- is unsound because it may not analyze all possible inputs
- is complete because every failure comes with a concrete input that triggers it

Static vs. dynamic, reloaded

Static:

- without executing the software
- on generic/abstract inputs
- based on symbolic constraints
- typically sound and incomplete

Dynamic:

- while executing the software
- on specific/concrete inputs
- based on concrete states
- typically unsound and complete

Static vs. dynamic, reloaded

Static:

- without executing the software
- on generic/abstract inputs
- based on symbolic constraints
- typically sound and incomplete

Dynamic:

- while executing the software
- on specific/concrete inputs
- based on concrete states
- typically unsound and complete

symbolic execution

Static vs. dynamic, reloaded

Static:

- without executing the software
- on generic/abstract inputs
- based on symbolic constraints
- typically sound and incomplete

Dynamic:

- while executing the software
- on specific/concrete inputs
- based on concrete states
- typically unsound and complete

symbolic execution

"Symbolic execution" is a technique that executes programs on symbolic values, so that the output is expressed as a function of symbolic input.

Static vs. dynamic, reloaded

Static:

- without executing the software
- on generic/abstract inputs
- based on symbolic constraints
- typically sound and incomplete

Dynamic:

- while executing the software
- on specific/concrete inputs
- based on concrete states
- typically unsound and complete

symbolic execution

"Symbolic execution" is a technique that executes programs on symbolic values, so that the output is expressed as a <u>function of symbolic input</u>.

model checking

STATIC → DYNAMIC

static analysis | deductive verification | software model checking | symbolic execution | dynamic analysis

Reachability

Reachability properties are of the form:
does some execution of program S reach program point p in S?

Often the program points are location of error, and hence reachability tries to verify that the error locations cannot be reached.

Reachability properties have the same expressiveness as assertions:

execution reaches p iff assertion fails

```
⋮
{ /* error */ }^p
⋮
{ assert x > 0 }^q
⋮
```

```
⋮
{ assert false }^p
⋮
if x ≤ 0
  { /* error */ }^q
⋮
```

execution reaches q iff assertion fails

Symbolic execution

Symbolic execution

Classic symbolic execution

Symbolic state and path condition

Symbolic execution executes programs on symbolic values.

Symbolic state and path condition

Symbolic execution executes programs on symbolic values.

The state keeps track of the (symbolic) value of every variable:

```
procedure max(x, y: Integer):
                     (z: Integer)
```

State:

x	y	z

Symbolic state and path condition

Symbolic execution executes programs on symbolic values.

The state keeps track of the (symbolic) value of every variable:

- inputs are initialized with symbolic (generic) values

```
procedure max(x, y: Integer):
                    (z: Integer)
```

State:

x	y	z
x_0	y_0	z_0

Symbolic state and path condition

Symbolic execution executes programs on symbolic values.

The state keeps track of the (symbolic) value of every variable:

- inputs are initialized with symbolic (generic) values
- executing a state-changing statement updates the symbolic state

```
procedure max(x, y: Integer):
                          (z: Integer)
  z := x
```

State:

x	y	z
x_0	y_0	x_0

Symbolic state and path condition

Symbolic execution executes programs on symbolic values.

The state keeps track of the (symbolic) value of every variable:

- inputs are initialized with symbolic (generic) values
- executing a state-changing statement updates the symbolic state
- executing a branch splits the symbolic state

```
procedure max(x, y: Integer):
                        (z: Integer)

  z := x

  if (z < y)
```

State:

then:

x	y	z
x_0	y_0	x_0

else:

x	y	z
x_0	y_0	x_0

Symbolic state and path condition

Symbolic execution executes programs on symbolic values.

The state keeps track of the (symbolic) value of every variable:

- inputs are initialized with symbolic (generic) values
- executing a state-changing statement updates the symbolic state
- executing a branch splits the symbolic state
- a path condition π records the conditions of each path

```
procedure max(x, y: Integer):
                          (z: Integer)

  z := x
  if (z < y)
```

State:

then:

x	y	z	π
x_0	y_0	x_0	$x_0 < y_0$

else:

x	y	z	π
x_0	y_0	x_0	$x_0 \geq y_0$

Symbolic state and path condition

Symbolic execution executes programs on symbolic values.

The state keeps track of the (symbolic) value of every variable:

- inputs are initialized with symbolic (generic) values
- executing a state-changing statement updates the symbolic state
- executing a branch splits the symbolic state
- a path condition π records the conditions of each path
- execution continues along every path that is feasible: (the conjunction of) its path conditions is satisfiable

```
procedure max(x, y: Integer):
                          (z: Integer)

  z := x

  if (z < y)
    z := y
```

State:

then:

x	y	z	π
x_0	y_0	x_0	$x_0 < y_0$

else:

x	y	z	π
x_0	y_0	x_0	$x_0 \geq y_0$

Symbolic state and path condition

Symbolic execution executes programs on symbolic values.

The state keeps track of the (symbolic) value of every variable:

- inputs are initialized with symbolic (generic) values
- executing a state-changing statement updates the symbolic state
- executing a branch splits the symbolic state
- a path condition π records the conditions of each path
- execution continues along every path that is feasible: (the conjunction of) its path conditions is satisfiable

```
procedure max(x, y: Integer):
                              (z: Integer)

  z := x

  if (z < y)
    z := y

  assert z ≥ x ∧ z ≥ y
```

State:

x	y	z	π
x_0	y_0	y_0	$x_0 < y_0$

x	y	z	π
x_0	y_0	x_0	$x_0 \geq y_0$

Final states

When symbolic execution hits an exit point p:

```
procedure max(x, y: Integer):
                    (z: Integer)
  z := x
  if (z < y)
    z := y
  assert z ≥ x ∧ z ≥ y
```

Final states

When symbolic execution hits an exit point p:

normal exit point: π is satisfiable.
Any satisfying assignment to π's variables gives a concrete input that reaches p in a concrete execution.

```
procedure max(x, y: Integer):
                          (z: Integer)
  z := x
  if (z < y)
    z := y
  assert z ≥ x ∧ z ≥ y
```

Normal exit:

x	y	z	π
x_0	y_0	y_0	$x_0 < y_0$, $y_0 \geq x_0 \land y_0 \geq y_0$

satisfying assignment: $x_0 = 1, y_0 = 3$

Final states

When symbolic execution hits an exit point p:

normal exit point: π is satisfiable.
Any satisfying assignment to π's variables gives a concrete input that reaches p in a concrete execution.

error exit point : π should be unsatisfiable.
Any satisfying assignment to π's variables gives a concrete input that triggers a failure at p.

```
procedure max(x, y: Integer):
                   (z: Integer)
  z := x
  if (z < y)
    z := y
  assert z ≥ x ∧ z ≥ y
```

Error exit:

x	y	z	π
x_0	y_0	x_0	$x_0 \geq y_0$, $x_0 < x_0 \vee x_0 < y_0$

there is no satisfying assignment

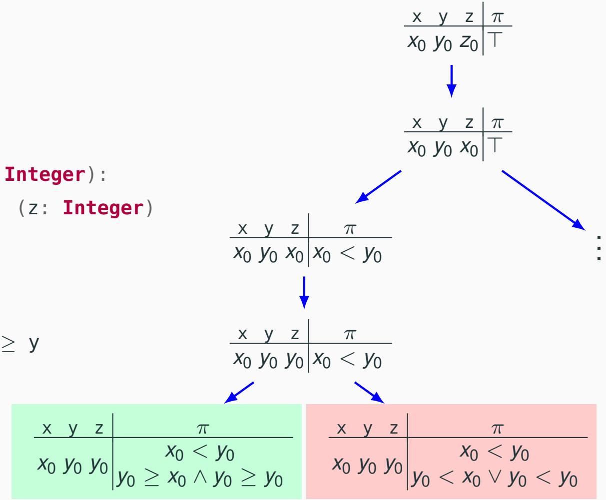
Execution trees

All execution paths are collected in an execution tree, where final nodes are marked as normal ✔ or error ✖.

Execution trees

All execution paths are collected in an execution tree, where final nodes are marked as normal ✔ or error ✖.

```
procedure max(x, y: Integer):
                    (z: Integer)
  z := x
  if (z < y)
    z := y
  assert z ≥ x ∧ z ≥ y
```



What does this program do?

```
procedure swap(x, y: Integer):
              (X, Y: Integer)
ensure X = old(y) ∧ Y = old(x)
  X := x + y
  Y := X - y
  X := X - Y
  assert (X = y ∧ Y = x)
```

What does this program do?

x	y	X	Y	π
x_0	y_0			$\top$

```
procedure swap(x, y: Integer):
              (X, Y: Integer)
ensure X = old(y) ∧ Y = old(x)
  X := x + y
  Y := X - y
  X := X - Y
  assert (X = y ∧ Y = x)
```

What does this program do?

x	y	X	Y	π
x_0	y_0			$\top$

$\downarrow$

x	y	X	Y	π
x_0	y_0	$x_0 + y_0$		$\top$

```
procedure swap(x, y: Integer):
              (X, Y: Integer)
ensure X = old(y) ∧ Y = old(x)
  X := x + y
  Y := X - y
  X := X - Y
  assert (X = y ∧ Y = x)
```

What does this program do?

```
procedure swap(x, y: Integer):
              (X, Y: Integer)
ensure X = old(y) ∧ Y = old(x)
  X := x + y
  Y := X - y
  X := X - Y
  assert (X = y ∧ Y = x)
```

x	y	X	Y	π
x_0	y_0			$\top$

x	y	X	Y	π
x_0	y_0	$x_0 + y_0$		$\top$

x	y	X	Y	π
x_0	y_0	$x_0 + y_0$	x_0	$\top$

What does this program do?

```
procedure swap(x, y: Integer):
              (X, Y: Integer)
ensure X = old(y) ∧ Y = old(x)
  X := x + y
  Y := X - y
  X := X - Y
  assert (X = y ∧ Y = x)
```

x	y	X	Y	π
x_0	y_0			$\top$

x	y	X	Y	π
x_0	y_0	$x_0 + y_0$		$\top$

x	y	X	Y	π
x_0	y_0	$x_0 + y_0$	x_0	$\top$

x	y	X	Y	π
x_0	y_0	y_0	x_0	$\top$

What does this program do?

```
procedure swap(x, y: Integer):
              (X, Y: Integer)
ensure X = old(y) ∧ Y = old(x)
  X := x + y
  Y := X - y
  X := X - Y
  assert (X = y ∧ Y = x)
```

x	y	X	Y	π
x_0	y_0			$\top$

x	y	X	Y	π
x_0	y_0	$x_0 + y_0$		$\top$

x	y	X	Y	π
x_0	y_0	$x_0 + y_0$	x_0	$\top$

x	y	X	Y	π
x_0	y_0	y_0	x_0	$\top$

x	y	X	Y	π
x_0	y_0	y_0	x_0	$y_0 = y_0 \wedge x_0 = x_0$

What does this program do?

```
procedure swap(x, y: Integer):
              (X, Y: Integer)
ensure X = old(y) ∧ Y = old(x)
  X := x + y
  Y := X - y
  X := X - Y
  assert (X = y ∧ Y = x)
```

x	y	X	Y	π
x_0	y_0			$\top$

x	y	X	Y	π
x_0	y_0	$x_0 + y_0$		$\top$

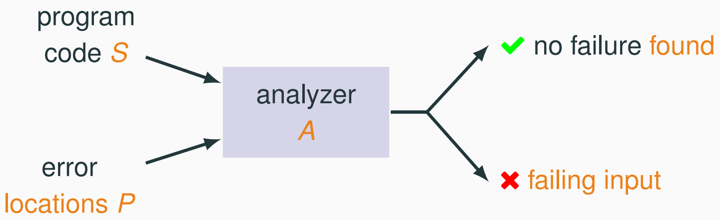
x	y	X	Y	π
x_0	y_0	$x_0 + y_0$	x_0	$\top$

x	y	X	Y	π
x_0	y_0	y_0	x_0	$\top$

x	y	X	Y	π
x_0	y_0	y_0	x_0	$y_0 = y_0$ $\wedge x_0 = x_0$

x	y	X	Y	π
x_0	y_0	y_0	x_0	$y_0 \neq y_0$ $\vee x_0 \neq x_0$

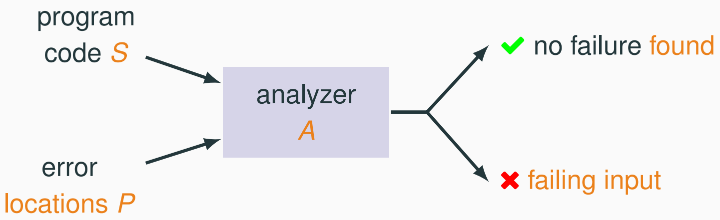
Completeness of symbolic execution



Whenever symbolic execution reaches an assertion it tries to solve a constraint. This is equivalent to solving a reachability problem:

A program is correct iff all its error locations are unreachable

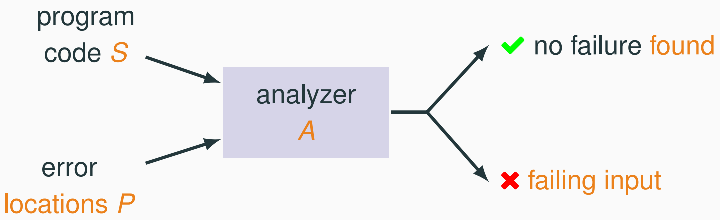
Completeness of symbolic execution



Whenever symbolic execution reaches an assertion it tries to solve a constraint. This is equivalent to solving a reachability problem:

A program is correct iff all its error locations are unreachable
that is:
A program is correct iff all path constraints of error are unsatisfiable

Completeness of symbolic execution



Whenever symbolic execution reaches an assertion it tries to solve a constraint. This is equivalent to solving a reachability problem:

A program is correct iff all its error locations are unreachable
that is:
A program is correct iff all path constraints of error are unsatisfiable

complete: error location e is reachable
$\Longrightarrow$ path constraint π_e is satisfiable
$\Longrightarrow$ assignment to π_e is failure-inducing input

Soundness of symbolic execution

A program is correct iff all its error locations are unreachable.
that is,
A program is correct iff all path constraints of error are unsatisfiable.

What happens when none of the failing locations have satisfiable path constraints?

Soundness of symbolic execution

A program is correct iff all its error locations are unreachable.
that is,
A program is correct iff all path constraints of error are unsatisfiable.

What happens when none of the failing locations have satisfiable path constraints?

- There may be other paths that the symbolic execution could not explore in reasonable time.
- A path constraints may be too complex for the constraint solver, which cannot find a satisfying assignment even if one exists.

Since the state-space exploration performed by symbolic execution is incomplete (it has limitations), symbolic execution is unsound as a verification technique.

Sources of unsoundness

Unsoundness of symbolic execution may originate in language features or state-space size:

complex constraints involving nonlinear arithmetic (`v*v % 50`), bit-wise operations (`v << 1`), or strings (`"foo" + "bar"`), about which the constraint solver cannot reason

external code whose source is not available (for example, calls to a pre-compiled library), whose behavior the constraint solver does not know

large or infinite state spaces (for example, involving recursion and loops)

If we can guarantee that all execution paths have been enumerated, and the constraint solver has a complete search procedure for the kinds of path constraints that were generated, then symbolic execution's search becomes exhaustive (and sound).

In all other cases, it is just a thorough test-case generation process.

Unsoundness: external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

Unsoundness: external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\top$

z = x

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\mathtt{bb}(y_0) = x_0$

¬ (z = x)

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\mathtt{bb}(y_0) \neq x_0$

x	y	z	π
x_0	y_0	$y_0 + 10$	$\mathtt{bb}(y_0) = x_0$

x ≤ z

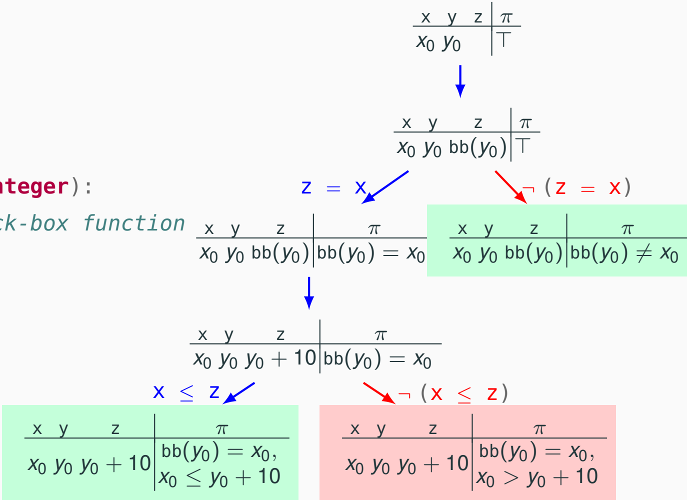
x	y	z	π
x_0	y_0	$y_0 + 10$	$\mathtt{bb}(y_0) = x_0$, $x_0 \leq y_0 + 10$

¬ (x ≤ z)

x	y	z	π
x_0	y_0	$y_0 + 10$	$\mathtt{bb}(y_0) = x_0$, $x_0 > y_0 + 10$

Unsoundness: external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```



The constraint solver cannot find a satisfying assignment to path conditions involving $\mathtt{bb}(y_0)$ because it does not know anything about how function bb behaves. Thus, those paths remain unexplored.

Symbolic execution

Dynamic symbolic execution

Symbolic execution's coming of age

Symbolic execution was first presented in two 1976 papers:

A PROGRAM TESTING SYSTEM*

Lori A. Clarke
Computer and Information Science Dept.
University of Massachusetts
Amherst, Massachusetts 01002

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

Symbolic execution's coming of age

Symbolic execution was first presented in two 1976 papers:

A PROGRAM TESTING SYSTEM*

Lori A. Clarke
Computer and Information Science Dept.
University of Massachusetts
Amherst, Massachusetts 01002

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

However, it hasn't become used in practice until about 30 years later:

Execution Generated Test Cases: How to Make Systems Code Crash Itself

Cristian Cadar and Dawson Engler*

DART: Directed Automated Random Testing

Patrice Godefroid Nils Klarlund
Bell Laboratories, Lucent Technologies
{god,klarlund}@bell-labs.com

Koushik Sen
Computer Science Department
University of Illinois at Urbana-Champaign
ksen@cs.uiuc.edu

Symbolic execution's coming of age

Symbolic execution was first presented in two 1976 papers:

A PROGRAM TESTING SYSTEM*

Lori A. Clarke
Computer and Information Science Dept.
University of Massachusetts
Amherst, Massachusetts 01002

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

However, it hasn't become used in practice until about 30 years later:

Execution Generated Test Cases: How to Make Systems Code Crash Itself

Cristian Cadar and Dawson Engler*

DART: Directed Automated Random Testing

Patrice Godefroid Nils Klarlund
Bell Laboratories, Lucent Technologies
{god,klarlund}@bell-labs.com

Koushik Sen
Computer Science Department
University of Illinois at Urbana-Champaign
ksen@cs.uiuc.edu

Symbolic execution has finally become practical thanks to:

- spectacular progress in constraint solving – the same SMT solvers that power much of deductive verification
- the combination of dynamic and symbolic execution – which can alleviate several of the shortcomings of classical symbolic execution

Dynamic symbolic execution

The basic idea of dynamic symbolic execution is performing symbolic execution alongside a normal dynamic execution (on concrete inputs).

The state of dynamic symbolic execution combines a symbolic part and a concrete part, which are used as needed to make progress in the state-space exploration.

Dynamic symbolic execution

The basic idea of dynamic symbolic execution is performing symbolic execution alongside a normal dynamic execution (on concrete inputs).

The state of dynamic symbolic execution combines a symbolic part and a concrete part, which are used as needed to make progress in the state-space exploration.

symbolic to concrete: whenever a computation cannot be executed symbolically – for example, a path constraint that is too complex: replace some symbolic state component with concrete values, thus simplifying the constraints

Dynamic symbolic execution

The basic idea of dynamic symbolic execution is performing symbolic execution alongside a normal dynamic execution (on concrete inputs).

The state of dynamic symbolic execution combines a symbolic part and a concrete part, which are used as needed to make progress in the state-space exploration.

symbolic to concrete: whenever a computation cannot be executed symbolically – for example, a path constraint that is too complex: replace some symbolic state component with concrete values, thus simplifying the constraints

concrete to symbolic: whenever a concrete computation terminates: negate one component of the symbolic path constraint, and solve it to get inputs exploring a new path

Dynamic symbolic execution

The basic idea of dynamic symbolic execution is performing symbolic execution alongside a normal dynamic execution (on concrete inputs).

The state of dynamic symbolic execution combines a symbolic part and a concrete part, which are used as needed to make progress in the state-space exploration.

symbolic to concrete: whenever a computation cannot be executed symbolically – for example, a path constraint that is too complex: replace some symbolic state component with concrete values, thus simplifying the constraints

concrete to symbolic: whenever a concrete computation terminates: negate one component of the symbolic path constraint, and solve it to get inputs exploring a new path

In the following example of procedure `m`, we start from a (random) concrete input, and then solve path constraints to explore new paths.

Dynamic symbolic execution: example

```
procedure m(x, y: Integer):
  z := 2*y
  if z = x
    z := y + 10
    assert x ≤ z
```

Dynamic symbolic execution: example

x	y	z	π
x_0	y_0		$\top$
22	7		

```
procedure m(x, y: Integer):
  z := 2∗y
  if z = x
    z := y + 10
    assert x ≤ z
```

Dynamic symbolic execution: example

x	y	z	π
x_0	y_0		$\top$
22	7		

$\downarrow$

x	y	z	π
x_0	y_0	$2y_0$	$\top$
22	7	14	

```
procedure m(x, y: Integer):
  z := 2*y
  if z = x
    z := y + 10
    assert x ≤ z
```

Dynamic symbolic execution: example

```
procedure m(x, y: Integer):
  z := 2*y
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
22	7		

x	y	z	π
x_0	y_0	$2y_0$	$\top$
22	7	14	

$\neg\ (z = x)$

x	y	z	π
x_0	y_0	$2y_0$	$2y_0 \neq x_0$
22	7	14	

Dynamic symbolic execution: example

x	y	z	π
x_0	y_0		$\top$
22	7		

x	y	z	π
x_0	y_0	$2y_0$	$\top$
22	7	14	

$\neg(z = x)$

x	y	z	π
x_0	y_0	$2y_0$	$2y_0 \neq x_0$
22	7	14	

```
procedure m(x, y: Integer):
  z := 2∗y
  if z = x
    z := y + 10
    assert x ≤ z
```

solve: $\neg(2y_0 \neq x_0)$

Dynamic symbolic execution: example

x	y	z	π
x_0	y_0		$\top$
2	1		

```
procedure m(x, y: Integer):
  z := 2*y
  if z = x
    z := y + 10
    assert x ≤ z
```

solve: $\neg(2y_0 \neq x_0)$

Dynamic symbolic execution: example

x	y	z	π
x_0	y_0		$\top$
2	1		

$\downarrow$

x	y	z	π
x_0	y_0	$2y_0$	$\top$
2	1	2	

```
procedure m(x, y: Integer):
  z := 2*y
  if z = x
    z := y + 10
    assert x ≤ z
```

Dynamic symbolic execution: example

```
procedure m(x, y: Integer):
  z := 2*y
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
2	1		

$\downarrow$

x	y	z	π
x_0	y_0	$2y_0$	$\top$
2	1	2	

z = x $\swarrow$

x	y	z	π
x_0	y_0	$2y_0$	$2y_0 = x_0$
2	1	2	

Dynamic symbolic execution: example

```
procedure m(x, y: Integer):
  z := 2∗y
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
2	1		

$\downarrow$

x	y	z	π
x_0	y_0	$2y_0$	$\top$
2	1	2	

z = x $\swarrow$

x	y	z	π
x_0	y_0	$2y_0$	$2y_0 = x_0$
2	1	2	

$\downarrow$

x	y	z	π
x_0	y_0	$y_0 + 10$	$2y_0 = x_0$
2	1	11	

Dynamic symbolic execution: example

```
procedure m(x, y: Integer):
  z := 2*y
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
2	1		

$\downarrow$

x	y	z	π
x_0	y_0	$2y_0$	$\top$
2	1	2	

z = x $\downarrow$

x	y	z	π
x_0	y_0	$2y_0$	$2y_0 = x_0$
2	1	2	

$\downarrow$

x	y	z	π
x_0	y_0	$y_0 + 10$	$2y_0 = x_0$
2	1	11	

x ≤ z $\downarrow$

x	y	z	π
x_0	y_0	$y_0 + 10$	$2y_0 = x_0$,
2	1	11	$x_0 \leq y_0 + 10$

Dynamic symbolic execution: example

```
procedure m(x, y: Integer):
  z := 2*y
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
2	1		

$\downarrow$

x	y	z	π
x_0	y_0	$2y_0$	$\top$
2	1	2	

`z = x` $\swarrow$

x	y	z	π
x_0	y_0	$2y_0$	$2y_0 = x_0$
2	1	2	

$\downarrow$

x	y	z	π
x_0	y_0	$y_0 + 10$	$2y_0 = x_0$
2	1	11	

`x ≤ z` $\swarrow$

x	y	z	π
x_0	y_0	$y_0 + 10$	$2y_0 = x_0$,
2	1	11	$x_0 \leq y_0 + 10$

solve: $2y_0 = x_0 \wedge \neg(x_0 \leq y_0 + 10)$

Dynamic symbolic execution: example

x	y	z	π
x_0	y_0		$\top$
30	15		

```
procedure m(x, y: Integer):
  z := 2*y
  if z = x
    z := y + 10
    assert x ≤ z
```

solve: $2y_0 = x_0 \wedge \neg(x_0 \leq y_0 + 10)$

Dynamic symbolic execution: example

x	y	z	π
x_0	y_0		$\top$
30	15		

$\downarrow$

x	y	z	π
x_0	y_0	$2y_0$	$\top$
30	15	30	

```
procedure m(x, y: Integer):
  z := 2*y
  if z = x
    z := y + 10
    assert x ≤ z
```

Dynamic symbolic execution: example

x	y	z	π
x_0	y_0		$\top$
30	15		

$\downarrow$

x	y	z	π
x_0	y_0	$2y_0$	$\top$
30	15	30	

z = x $\swarrow$

x	y	z	π
x_0	y_0	$2y_0$	$2y_0 = x_0$
30	15	30	

```
procedure m(x, y: Integer):
  z := 2*y
  if z = x
    z := y + 10
    assert x ≤ z
```

Dynamic symbolic execution: example

```
procedure m(x, y: Integer):
  z := 2*y
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
30	15		

↓

x	y	z	π
x_0	y_0	$2y_0$	$\top$
30	15	30	

z = x ↙

x	y	z	π
x_0	y_0	$2y_0$	$2y_0 = x_0$
30	15	30	

↓

x	y	z	π
x_0	y_0	$y_0 + 10$	$2y_0 = x_0$
30	15	25	

Dynamic symbolic execution: example

```
procedure m(x, y: Integer):
  z := 2*y
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
30	15		

↓

x	y	z	π
x_0	y_0	$2y_0$	$\top$
30	15	30	

z = x ↙

x	y	z	π
x_0	y_0	$2y_0$	$2y_0 = x_0$
30	15	30	

↓

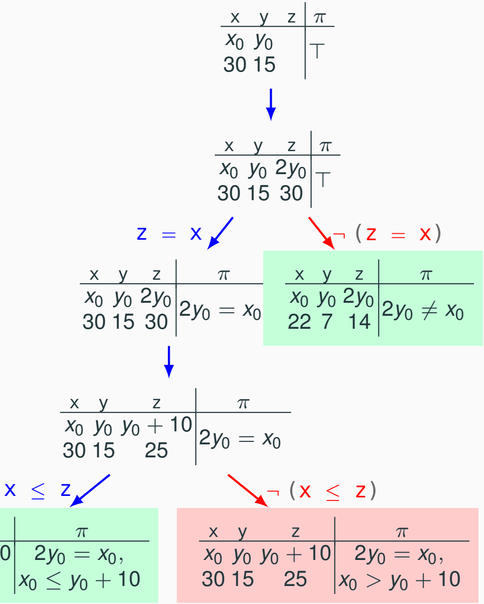
x	y	z	π
x_0	y_0	$y_0 + 10$	$2y_0 = x_0$
30	15	25	

↘ ¬(x ≤ z)

x	y	z	π
x_0	y_0	$y_0 + 10$	$2y_0 = x_0$,
30	15	25	$x_0 > y_0 + 10$

Dynamic symbolic execution: example

```
procedure m(x, y: Integer):
  z := 2*y
  if z = x
    z := y + 10
    assert x ≤ z
```



Concolic vs. execution generated

The same fundamental ideas of dynamic symbolic execution are implemented in two slightly different ways:

concolic (concrete + symbolic) execution starts from (concrete) test-cases, and uses symbolic execution to extend the test-case generation capabilities

execution-generated testing starts from symbolic exploration, and concretizes parts of the state selectively when a symbolic representation is not available (for example, when executing pre-compile code)

Concolic vs. execution generated

The same fundamental ideas of dynamic symbolic execution are implemented in two slightly different ways:

concolic (concrete + symbolic) execution starts from (concrete) test-cases, and uses symbolic execution to extend the test-case generation capabilities

execution-generated testing starts from symbolic exploration, and concretizes parts of the state selectively when a symbolic representation is not available (for example, when executing pre-compile code)

In practice, "concolic execution" and "dynamic symbolic execution" are often used as synonyms.

In our presentation, we need not emphasize these (fuzzy) differences, and hence will also use the two terms as synonyms.

Dynamic symbolic execution: explore external code

Let's see how concretization can explore exhaustively code that includes an external function.

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
22	7		

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
22	7		

↓

x	y	z	π
x_0	y_0	$\mathrm{bb}(y_0)$	$\top$
22	7	14	

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
22	7		

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\top$
22	7	14	

$\neg$ (z = x)

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\mathtt{bb}(y_0) \neq x_0$
22	7	14	

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
22	7		

$\downarrow$

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\top$
22	7	14	

$\neg\,(z = x)$

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\mathtt{bb}(y_0) \neq x_0$
22	7	14	

concretize: $\neg(\mathtt{bb}(y_0) \neq x_0)$

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
14	7		

concretize: $\neg(\texttt{bb}(y_0) \neq x_0)$

solve: $\neg(14 \neq x_0)$

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
14	7		

↓

x	y	z	π
x_0	y_0	$\mathrm{bb}(y_0)$	$\top$
14	7	14	

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
14	7		

$\downarrow$

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\top$
14	7	14	

z = x $\swarrow$

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\mathtt{bb}(y_0) = x_0$
14	7	14	

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
14	7		

x	y	z	π
x_0	y_0	$\mathrm{bb}(y_0)$	$\top$
14	7	14	

z = x

x	y	z	π
x_0	y_0	$\mathrm{bb}(y_0)$	$\mathrm{bb}(y_0) = x_0$
14	7	14	

x	y	z	π
x_0	y_0	$y_0 + 10$	$\mathrm{bb}(y_0) = x_0$
14	7	17	

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
14	7		

$\downarrow$

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\top$
14	7	14	

$\xrightarrow{\texttt{z = x}}$

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\mathtt{bb}(y_0) = x_0$
14	7	14	

$\downarrow$

x	y	z	π
x_0	y_0	$y_0 + 10$	$\mathtt{bb}(y_0) = x_0$
14	7	17	

$\xrightarrow{\texttt{x} \leq \texttt{z}}$

x	y	z	π
x_0	y_0	$y_0 + 10$	$\mathtt{bb}(y_0) = x_0,$
14	7	17	$x_0 \leq y_0 + 10$

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	π
x_0	y_0	$\top$
14	7	

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\top$
14	7	14	

z = x

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\mathtt{bb}(y_0) = x_0$
14	7	14	

x	y	z	π
x_0	y_0	$y_0 + 10$	$\mathtt{bb}(y_0) = x_0$
14	7	17	

x ≤ z

x	y	z	π
x_0	y_0	$y_0 + 10$	$\mathtt{bb}(y_0) = x_0,$
14	7	17	$x_0 \le y_0 + 10$

concretize: $\mathtt{bb}(y_0) = x_0 \land \neg(x_0 \le y_0 + 10)$

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
28	14		

solve: $\neg(x_0 \leq 17 + 10)$

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
28	14		

$\downarrow$

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\top$
28	14	28	

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
28	14		

$\downarrow$

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\top$
28	14	28	

$\swarrow$ z = x

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\mathtt{bb}(y_0) = x_0$
28	14	28	

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
28	14		

$\downarrow$

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\top$
28	14	28	

z = x $\swarrow$

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\mathtt{bb}(y_0) = x_0$
28	14	28	

$\downarrow$

x	y	z	π
x_0	y_0	$y_0 + 10$	$\mathtt{bb}(y_0) = x_0$
28	14	24	

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
28	14		

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\top$
28	14	28	

z = x

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\mathtt{bb}(y_0) = x_0$
28	14	28	

x	y	z	π
x_0	y_0	$y_0 + 10$	$\mathtt{bb}(y_0) = x_0$
28	14	24	

¬ (x ≤ z)

x	y	z	π
x_0	y_0	$y_0 + 10$	$\mathtt{bb}(y_0) = x_0$,
28	14	24	$x_0 > y_0 + 10$

Dynamic symbolic execution: explore external code

```
procedure m2(x, y: Integer):
  z := bb(y)  // black-box function
  if z = x
    z := y + 10
    assert x ≤ z
```

x	y	z	π
x_0	y_0		$\top$
28	14		

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\top$
28	14	28	

z = x

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\mathtt{bb}(y_0) = x_0$
28	14	28	

¬ (z = x)

x	y	z	π
x_0	y_0	$\mathtt{bb}(y_0)$	$\mathtt{bb}(y_0) \neq x_0$
22	7	14	

x	y	z	π
x_0	y_0	$y_0 + 10$	$\mathtt{bb}(y_0) = x_0$
28	14	24	

x ≤ z

x	y	z	π
x_0	y_0	$y_0 + 10$	$\mathtt{bb}(y_0) = x_0$,
14	7	17	$x_0 \leq y_0 + 10$

¬ (x ≤ z)

x	y	z	π
x_0	y_0	$y_0 + 10$	$\mathtt{bb}(y_0) = x_0$,
28	14	24	$x_0 > y_0 + 10$

Challenges, tools, and applications

Path explosion

The main challenge to symbolic execution is path explosion, which limits the technique's scalability.

Path explosion has two different forms:

- large but finite search space – generated by complex nested conditionals
- infinite search space – generated by loops or recursion with symbolic bounds

Solutions to address path explosion depend on whether we are interested in trying to achieve a complete exploration (verification), or we just want to find as many bugs as possible (testing).

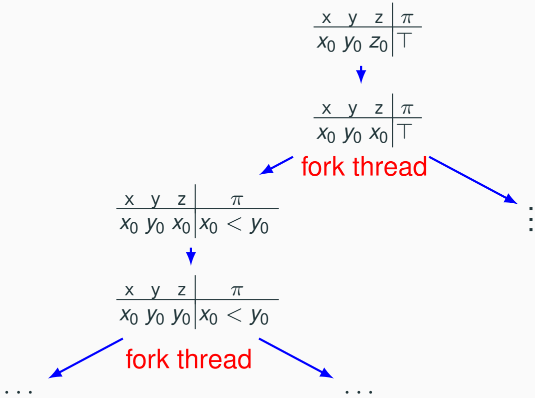
testing solutions are typically best effort

verification solutions try not to give up soundness when possible

Parallelization

Using parallelization is a straightforward way to search larger state spaces.

Parallelizing symbolic execution is particularly easy because no coordination is necessary between different branches in an execution tree. The challenge is deciding when to fork new threads in a way that work is balanced between threads.



Search heuristics

If we do not insist that all paths be explored, it becomes important to prioritize more interesting paths – such as those that are more likely to show an error.

To this end, we can use heuristics to decide how to explore the execution tree:

coverage metrics favor exploring branches or statements that have not been explored yet

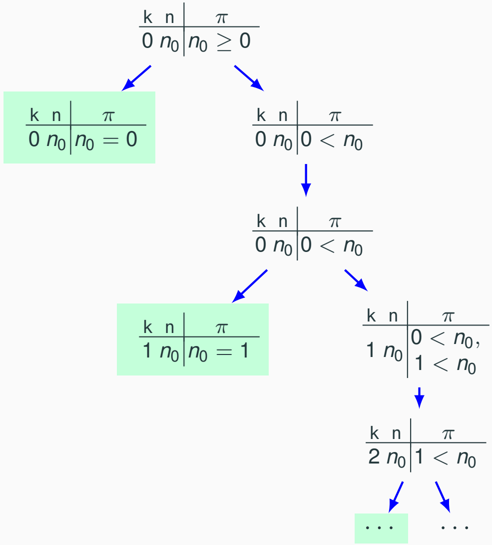
random search reduces bias in a search, thus helping reach special cases that systematic search may neglect

evolutionary search uses a fitness function to direct the search towards trying to maximize fitness

These heuristics are all unsound but can be effective to find bugs more quickly or in more complex programs.

Loops

Loops with symbolic bounds determine infinite execution trees.




```
procedure to_n(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while k < n
    k := k + 1
  assert k = n
```

Loop summarization

If we can infer a loop invariant, we can summarize the input/output behavior of a loop symbolically.

```
procedure to_n(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while k < n
    k := k + 1
  assert k = n
```

Since $\mathtt{k} \leq n_0 = \mathtt{n}$ is a loop invariant, the path condition after the loop is always $\mathtt{k} \leq n_0 = \mathtt{n} \land \neg(\mathtt{k} < \mathtt{n})$, that is $\mathtt{k} = n_0$.

Loop summarization

If we can infer a loop invariant, we can summarize the input/output behavior of a loop symbolically.

k	n	π
0	n_0	$n_0 \geq 0$

```
procedure to_n(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while k < n
    k := k + 1
  assert k = n
```

Since $\texttt{k} \leq n_0 = \texttt{n}$ is a loop invariant, the path condition after the loop is always $\texttt{k} \leq n_0 = \texttt{n} \wedge \neg(\texttt{k} < \texttt{n})$, that is $\texttt{k} = n_0$.

Loop summarization

If we can infer a loop invariant, we can summarize the input/output behavior of a loop symbolically.

k	n	π
0	n_0	$n_0 \geq 0$

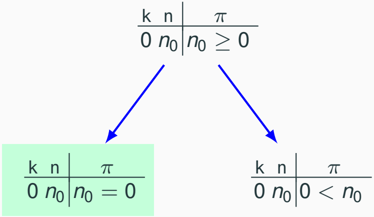
k	n	π
0	n_0	$n_0 = 0$

```
procedure to_n(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while k < n
    k := k + 1
  assert k = n
```

Since $\mathtt{k} \leq n_0 = \mathtt{n}$ is a loop invariant, the path condition after the loop is always $\mathtt{k} \leq n_0 = \mathtt{n} \land \neg(\mathtt{k} < \mathtt{n})$, that is $\mathtt{k} = n_0$.

Loop summarization

If we can infer a loop invariant, we can summarize the input/output behavior of a loop symbolically.

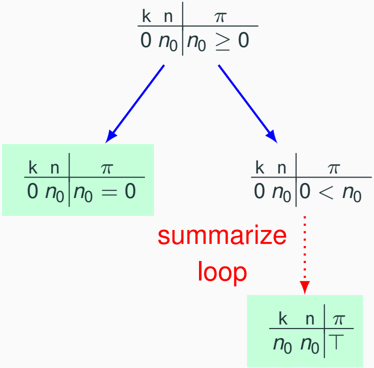



```
procedure to_n(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while k < n
    k := k + 1
  assert k = n
```

Since $\mathsf{k} \leq n_0 = \mathsf{n}$ is a loop invariant, the path condition after the loop is always $\mathsf{k} \leq n_0 = \mathsf{n} \wedge \neg(\mathsf{k} < \mathsf{n})$, that is $\mathsf{k} = n_0$.

Loop summarization

If we can infer a loop invariant, we can summarize the input/output behavior of a loop symbolically.




```
procedure to_n(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while k < n
    k := k + 1
  assert k = n
```

Since $k \leq n_0 = n$ is a loop invariant, the path condition after the loop is always $k \leq n_0 = n \land \neg(k < n)$, that is $k = n_0$.

Loop summarization: example

```
procedure loop(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while n > 0
    assert k ≠ 200
    k := k + 1
    n := n - 1
  assert k ≠ 100
```

Using the loop invariant:

$$\mathtt{n} + \mathtt{k} = n_0 \land \mathtt{n} \geq 0$$

we can summarize all possible termination conditions.

Loop summarization: example

k	n	π
0	n_0	$n_0 \geq 0$

```
procedure loop(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while n > 0
    assert k ≠ 200
    k := k + 1
    n := n - 1
  assert k ≠ 100
```

Using the loop invariant:

$$\mathtt{n} + \mathtt{k} = n_0 \wedge \mathtt{n} \geq 0$$

we can summarize all possible termination conditions.

Loop summarization: example

k	n	π
0	n_0	$n_0 \geq 0$

k	n	π
200	$n_0 - 200$	$n_0 > 200$

```
procedure loop(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while n > 0
    assert k ≠ 200
    k := k + 1
    n := n - 1
  assert k ≠ 100
```

Using the loop invariant:

$$\mathtt{n} + \mathtt{k} = n_0 \wedge \mathtt{n} \geq 0$$

we can summarize all possible termination conditions.

Loop summarization: example

k	n	π
0	n_0	$n_0 \geq 0$

loop summary

k	n	π
200	$n_0 - 200$	$n_0 > 200$

k	n	π
n_0	0	$n_0 \leq 200$

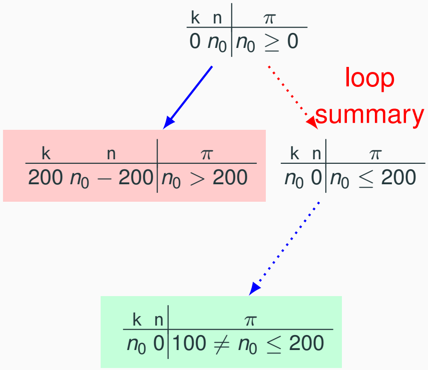
```
procedure loop(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while n > 0
    assert k ≠ 200
    k := k + 1
    n := n - 1
  assert k ≠ 100
```

Using the loop invariant:

$$\mathtt{n} + \mathtt{k} = n_0 \land \mathtt{n} \geq 0$$

we can summarize all possible termination conditions.

Loop summarization: example



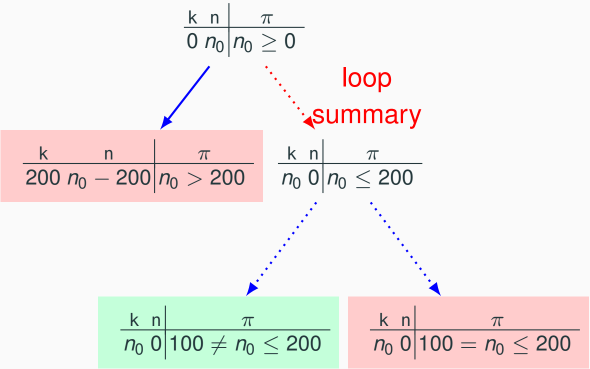

```
procedure loop(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while n > 0
    assert k ≠ 200
    k := k + 1
    n := n - 1
  assert k ≠ 100
```

Using the loop invariant:

$$\mathrm{n} + \mathrm{k} = n_0 \wedge \mathrm{n} \geq 0$$

we can summarize all possible termination conditions.

Loop summarization: example




```
procedure loop(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while n > 0
    assert k ≠ 200
    k := k + 1
    n := n - 1
  assert k ≠ 100
```

Using the loop invariant:

$$n + k = n_0 \land n \geq 0$$

we can summarize all possible <u>termination conditions</u>.

State abstraction

An unsound way of handling loops, recursion, or very large state spaces is to abstract the symbolic state in a way that we effectively merge different states together.

This generally loses precision in the search, and hence we may miss errors, but helps scalability.

State abstraction

An unsound way of handling loops, recursion, or very large state spaces is to abstract the symbolic state in a way that we effectively merge different states together.

This generally loses precision in the search, and hence we may miss errors, but helps scalability.

Abstraction is often used to represent data structures such as arrays and lists in a manageable way.

- a list may be abstracted by keeping track of whether it is `null`, empty, or non-empty.
- an array may be abstracted by keeping track of its content ignoring the order

State abstraction

An unsound way of handling loops, recursion, or very large state spaces is to abstract the symbolic state in a way that we effectively merge different states together.

This generally loses precision in the search, and hence we may miss errors, but helps scalability.

Abstraction is often used to represent data structures such as arrays and lists in a manageable way.

- a list may be abstracted by keeping track of whether it is `null`, empty, or non-empty.
- an array may be abstracted by keeping track of its content ignoring the order

Let's see abstraction on a simpler example where we abstract an integer variable by only keeping track of whether it is zero or positive.

Integer abstraction

```
procedure to_n(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while k < n
    k := k + 1
  assert k = n
```

In symbolic execution, abstracting the state generally leads to an under approximation because it coalesces paths that are distinct in the concrete program.

Integer abstraction

$\mathsf{k} = 0 \land \mathsf{n} = n_0 \geq 0$

```
procedure to_n(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while k < n
    k := k + 1
  assert k = n
```

In symbolic execution, abstracting the state generally leads to an under approximation because it coalesces paths that are distinct in the concrete program.

Integer abstraction

$$\begin{tikzcd} \mathtt{k} = 0 \wedge \mathtt{n} = n_0 \geq 0 \arrow[d] \\ \boxed{\mathtt{k} = 0 = \mathtt{n} = n_0} \end{tikzcd}$$

```
procedure to_n(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while k < n
    k := k + 1
  assert k = n
```

In symbolic execution, abstracting the state generally leads to an under approximation because it coalesces paths that are distinct in the concrete program.

Integer abstraction

$$\begin{tikzcd}[column sep=tiny]
 & \mathtt{k} = 0 \land \mathtt{n} = n_0 \geq 0 \arrow[dl] \arrow[dr] & \\
\mathtt{k} = 0 = \mathtt{n} = n_0 & & \mathtt{k} = 0 \land \mathtt{n} = n_0 > 0
\end{tikzcd}$$

```
procedure to_n(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while k < n
    k := k + 1
  assert k = n
```

In symbolic execution, abstracting the state generally leads to an under approximation because it coalesces paths that are distinct in the concrete program.

Integer abstraction

$$
\begin{tikzcd}
 & \mathrm{k} = 0 \land \mathrm{n} = n_0 \geq 0 \arrow[dl] \arrow[dr] & \\
\mathrm{k} = 0 = \mathrm{n} = n_0 & & \mathrm{k} = 0 \land \mathrm{n} = n_0 > 0 \arrow[d] \\
 & & \mathrm{k} > 0 \land \mathrm{n} = n_0 > 0
\end{tikzcd}
$$

```
procedure to_n(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while k < n
    k := k + 1
  assert k = n
```

In symbolic execution, abstracting the state generally leads to an under approximation because it coalesces paths that are distinct in the concrete program.

Integer abstraction

$$k = 0 \land n = n_0 \geq 0$$

$$\boxed{k = 0 = n = n_0} \qquad k = 0 \land n = n_0 > 0$$

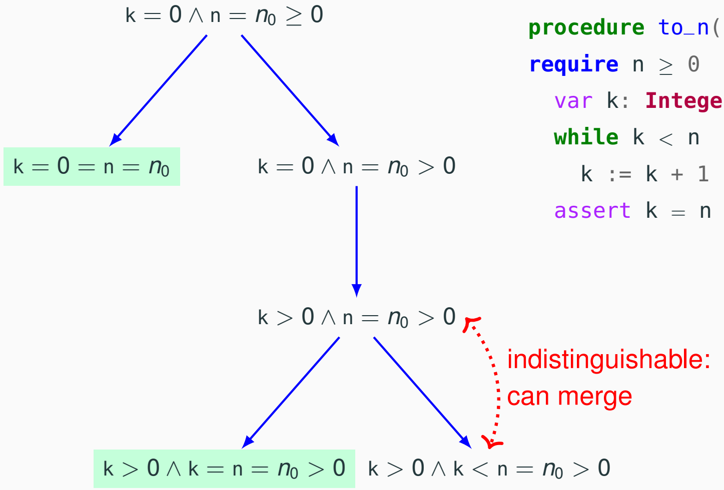
$$k > 0 \land n = n_0 > 0$$

$$\boxed{k > 0 \land k = n = n_0 > 0} \qquad k > 0 \land k < n = n_0 > 0$$

```
procedure to_n(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while k < n
    k := k + 1
  assert k = n
```

In symbolic execution, abstracting the state generally leads to an under approximation because it coalesces paths that are distinct in the concrete program.

Integer abstraction




```
procedure to_n(n: Integer):
require n ≥ 0
  var k: Integer := 0
  while k < n
    k := k + 1
  assert k = n
```

In symbolic execution, abstracting the state generally leads to an under approximation because it coalesces paths that are distinct in the concrete program.

Memory modeling

Many aspects of symbolic memory representation affect precision and scalability of symbolic execution's state space exploration:

- numeric representation: machine vs. mathematical integers, floating point vs. reals
- string representation: tracking string variable content vs. only checking for equality
- arrays, lists, and other common data structures: detailed representation, content but no ordering, null vs. non-null
- references, pointers: memory footprint modeling, aliasing information, or only equality checking

The precision vs. scalability trade-off impacts on bug-finding effectiveness in subtle ways: a very precise exploration may find more kinds of bugs, but it may also be inapplicable in practice if it doesn't scale

Constraint solving

Constraint solving remains a major bottleneck of symbolic execution.

While constraint solvers' performance keep getting better, optimizations in the way constraints are built, checked, and modified are key to improving the scalability of symbolic execution.

elimination of constraints that are not relevant to the current branch

incremental checking relies on caching of constraint checking results, so that the constraint solver is called only on the new parts of a constraint. Choosing a suitable representation of constraints helps detect similarities that support caching

Constraint solving

Example: suppose the state at an exit point is:

π	x	y	z
$(x + y) > 10 \wedge (z > 0) \wedge (y < 12)$	3	8	3

and we want to explore the branch with path condition

$$(x + y) > 10 \wedge (z > 0) \wedge \neg(y < 12)$$

The constraint involving z does not affect the satisfiability of the new $\neg(y < 12)$. Hence, we invoke the constraint solver on:

$$(x + y) > 10 \wedge \neg(y < 12)$$

and reuse the previous value of z to complete the concrete state:

π	x	y	z
$(x + y) > 10 \wedge (z > 0) \wedge \neg(y < 12)$	2	14	3

Symbolic execution tools

Some (mostly open source) symbolic executors (normally based on dynamic symbolic execution):

CUTE/jCUTE (successors to DART) combine random testing and symbolic execution techniques to generate test cases that explore a program's paths (including concurrent executions) as exhaustively as possible

CREST is an extensible version of CUTE, which supports combination of different path-selection heuristics

KLEE (successor to EXE) is a symbolic execution engine for LLVM code, geared towards bit-level modeling of systems C code (including environment models of system calls)

JPF (Java PathFinder) is a model checker for Java source code supporting concurrency and symbolic execution (Symbolic PathFinder) of structured data (such as linked lists and trees)

Case studies

Notable case studies have demonstrated symbolic execution tools' capabilities of finding complex critical bugs.

Case studies

KLEE has found several critical bugs in Unix systems software (GNU Coreutils, ext2/3 file systems, network servers, kernel code, system libraries, and so on).

With bit-level modeling of data and suitable abstractions, symbolic execution engines can be used as white-box fuzzers to generate structured input that triggers failures in communication protocols.

- KLEE has been used to generate packets that trigger failures in Apple's Bonjour communication protocol
- Microsoft's SAGE/SAGAN tools generated image and text file inputs that trigger many vulnerabilities in Windows applications. These tools run over the clock at Microsoft on new releases and existing software, constituting one of the largest-case deployment of software analysis tools

The symbolic-execution engine of Java PathFinder has been used extensively at NASA to test more effectively and more quickly control software.

Challenges, tools, and applications

A brief demo of Klee

Symbolic input

Klee is a powerful dynamic-symbolic execution engine for C, whose implementation is based on LLVM.

Symbolic input

Klee is a powerful dynamic-symbolic execution engine for C, whose implementation is based on LLVM.

To analyze a program with Klee, we explicitly choose which variables are to be treated as symbolic inputs; everything else takes on concrete values.

```c
#include <klee/klee.h>

int max(int x, int y)
{
  int max;
  if (x > y)
        max = x;
  else
        max = y;
  return max;
}
```

```c
// driver of `max', which defines symbolic input
int main()
{
  int x, y;
  klee_make_symbolic(&x, sizeof(x), "x");
  klee_make_symbolic(&y, sizeof(y), "y");
  return max(x, y);
}
```

Input generation

Running Klee on `max.c` simply generates inputs for each program path, which will be stored in subdirectory `klee-last`.

```
# Run shell with Klee Docker image distro
> cd <DIRECTORY WHERE max.c IS>
> docker run -it -v "$(pwd):/home/klee/examples" klee/klee:2.0
# Compile max.c to LLVM bytecode
>> cd examples
>> clang -I ../klee_src/include -emit-llvm -c -g -O0 \
            -Xclang -disable-O0-optnone max.c
# Run Klee
>> klee max.bc
```

Test cases

Klee generates two tests for `max.c`: one where $x \leq y$ and one where $x > y$.

In test 1: $x = y = 0$

```
>> ktest-tool test000001.ktest
num objects: 2
object 0: name: 'x'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x00'
object 0: hex : 0x00000000
object 0: int : 0
object 1: name: 'y'
object 1: size: 4
object 1: data: b'\x00\x00\x00\x00'
object 1: hex : 0x00000000
object 1: int : 0
```

In test 2: $x = 1 > y = 0$

```
>> ktest-tool test000002.ktest
num objects: 2
object 0: name: 'x'
object 0: size: 4
object 0: data: b'\x01\x00\x00\x00'
object 0: hex : 0x01000000
object 0: int : 1
object 1: name: 'y'
object 1: size: 4
object 1: data: b'\x00\x00\x00\x00'
object 1: hex : 0x00000000
object 1: int : 0
```

Symbolic input constraints

Klee finds an out-of-bound memory access in `maxa.c`, which happens when `n` takes a value that is larger than the actual size of array `a`.

```
KLEE: ERROR: maxa.c:25: memory error: out of bound pointer

>> ls klee-last/*.err
klee-last/test000002.ptr.err

>> ktest-tool klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['maxa.bc']
num objects: 2
object 0: name: 'a'
object 0: size: 12
object 0: data: b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
object 1: name: 'n'
object 1: size: 4
object 1: int : 2147483647
```

Symbolic input constraints

Klee finds an out-of-bound memory access in `maxa.c`, which happens when `n` takes a value that is larger than the actual size of array `a`.

To fix these kinds of spurious errors, we can constraint the symbolic input using `klee_assume`:

```
klee_assume(0 <= n && n <= N);
```

Assumptions, combined with the symbolic representation of memory as bit strings, make Klee suitable to analyze low-level code and complex, structured input.

Concrete error-triggering input

On the buggy example `negpow.c`, which we also analyzed using CPAchecker, Klee quickly finds a concrete input that triggers an error (even though generating all paths may not terminate because of the unbounded loop): $\mathtt{x} = -2147483648 < 0$ and $\mathtt{y} = 1$.

```
>> ktest-tool klee-last/test000001.ktest
num objects: 2
object 0: name: 'x'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x80'
object 0: hex : 0x00000080
object 0: int : -2147483648
object 1: name: 'y'
object 1: size: 4
object 1: data: b'\x01\x00\x00\x00'
object 1: hex : 0x01000000
object 1: int : 1
```

Solving a maze using Klee

A fun example of Klee's constraint solving capabilities (by Felipe Manzano).

```
>> gcc -o maze maze.original.c
>> ./maze
Program the player moves with a sequence of 'w', 's', 'a' and 'd'
Try to reach the price(#)!
+-+---+---+
|X|     |#|
| | --+ | |
| |   | | |
| +-- | | |
|     |   |
+-----+---+
```

The goal of the game is to give a sequence of letters `w` (up), `s` (down), `a` (left), and `d` (right) that, when applied, go from the initial `X` to the goal `#`. We lose if we give an incomplete sequence, or one that tries to cross walls.

Instrumenting the maze code

Make the input sequence (an array) symbolic:

```
// read(0,program,ITERS);
        // use symbolic input
klee_make_symbolic(program,
                   sizeof(char)*ITERS,
                   "program");
```

Add an `assert false` when winning:

```
printf ("You win!\n");
      // mark winning sequence as "error"
klee_assert(0);
```

```
>> klee maze.symbolic.bc
[...]
>> ls klee-last/*.err
klee-last/test000139.assert.err
>> ktest-tool klee-last/test000139.ktest
object 0: text: sddwddddsddw................
```

Instrumenting the maze code

Make the input sequence (an array) symbolic:

```
// read(0,program,ITERS);
      // use symbolic input
klee_make_symbolic(program,
                   sizeof(char)*ITERS,
                   "program");
```

Add an `assert false` when winning:

```
printf ("You win!\n");
      // mark winning sequence as "error"
klee_assert(0);
```

```
>> klee maze.symbolic.bc
[...]
>> ls klee-last/*.err
klee-last/test000139.assert.err
>> ktest-tool klee-last/test000139.ktest
object 0: text: sddwddddsddw................
```

By using this input sequence, we find out that the maze program does not forbid crossing walls everywhere!

Instrumenting the maze code

Make the input sequence (an array) symbolic:

```
// read(0,program,ITERS);
      // use symbolic input
klee_make_symbolic(program,
                   sizeof(char)*ITERS,
                   "program");
```

Add an `assert false` when winning:

```
printf ("You win!\n");
      // mark winning sequence as "error"
klee_assert(0);
```

```
>> klee maze.symbolic.bc
[...]
>> ls klee-last/*.err
klee-last/test000139.assert.err
>> ktest-tool klee-last/test000139.ktest
object 0: text: sddwddddsddw................
```

By using this input sequence, we find out that the maze program does not forbid crossing walls everywhere!

To find out all four input sequences that lead to the goal run:

```
klee --emit-all-errors maze.symbolic.bc
```

Summary

Symbolic execution: techniques

Symbolic execution is a systematic path-exploration technique based on executing a program with symbolic inputs.

Symbolic execution techniques are mostly best effort and rely on constraint solvers to determine which paths are feasible.

soundness/completeness: unsound and complete – symbolic execution primarily focuses on finding concrete inputs that trigger bugs

complexity: according to the kinds of constraints that are used, and hence by how accurately program features are modeled

automation: fully automated

expressiveness: any reachability properties that are expressible in code (assertions)

Symbolic execution: tools and practice

Symbolic execution tools focus on test-case generation with high coverage, and on (white-box) fuzzing to generate failure-inducing inputs. They can often handle complex code, including systems code and concurrency.

In notable case studies, symbolic execution was used to detect vulnerabilities in Unix systems and communication software, in image filters, and in document converters.

Main outstanding challenges:

- scalability by optimizing the usage of constraint solving
- thoroughness of the search, which has to be traded-off against scalability
- handling complex language features (loops and recursion, memory modeling, external code) in a practical way

Credits and further reading

Two surveys on dynamic symbolic execution describe the state of the art and include many references to tools and techniques:

- Cadar and Sen: Symbolic execution for software testing: three decades later, Communications of the ACM, 56(2), 2013
- Baldoni et al.: A survey of symbolic execution techniques, ACM Computing Surveys, 51(3), 2018

Some of the examples in this class are based on these two surveys, as well as on slides by Antonio Filieri – which in turn were based on Corina Păsăreanu's tutorial at Marktoberdorf 2012.

These slides' license