Programm- & Systemverifikation

Test-Case Generation

Georg Weissenbacher

184.741

What happened so far

- How bugs come into being:
  - **Fault** – cause of an error (e.g., mistake in coding)
  - **Error** – *incorrect* state that may lead to failure
  - **Failure** – deviation from *desired* behaviour
- We specified *intended* behaviour using **assertions**.
- We proved our programs correct (inductive invariants).
- We learned how to derive test-cases *by hand.*
- Coverage criteria. How “good” is our test-suite?

So far: Testing is a manual process

Driven by

- Requirements and specification
- Assumptions about program behaviour (equivalence classes!)

So far: Testing is a manual process

Driven by

- Requirements and specification
- Assumptions about program behaviour (equivalence classes!)

Can't we automate the generation of test cases?

Automating Test-Case Generation: A (Discouraging) Example

```
int power (int x,
           int y)
{
   int r = y * y;
   return r;
}
```

Automating Test-Case Generation: A (Discouraging) Example

```
int power (int x,
           int y)
{
   int r = y * y;
   return r;
}
```

→

Automating Test-Case Generation: A (Discouraging) Example

```
int power (int x,
           int y)
{
    int r = y * y;
    return r;
}
```

→

→

x	y	r
0	0	0
1	1	1
2	2	4
4	4	16
5	5	25
…		

Automating Test-Case Generation: A (Discouraging) Example

```
int power (int x,
           int y)
{
   int r = y * y;
   return r;
}
```

→

→

x	y	r
0	0	0
1	1	1
2	2	4
4	4	16
5	5	25
...		

Can you spot the problem?

- How are the return values generated?

Automating Test-Case Generation: A (Discouraging) Example

```
int power (int x,
           int y)
{
    int r = y * y;
    return r;
}
```

→

→

x	y	r
0	0	0
1	1	1
2	2	4
4	4	16
5	5	25
...		

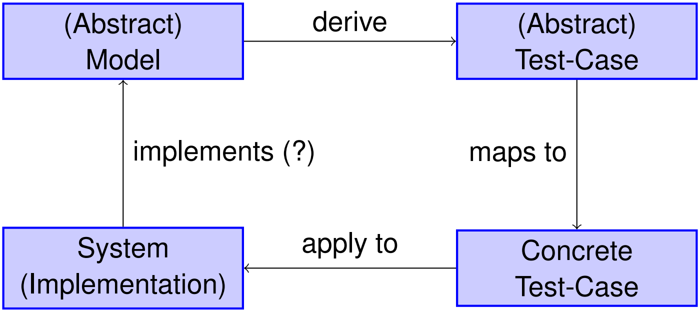
Can you spot the problem?

- How are the return values generated?
- Solution: let's get them from the specification!

Model-based Test Case Generation

- Idea: derive test-cases from a *model*
  - The model captures requirements at a more abstract level
  - The model is *not necessarily* executable
  - The model must be *easier to understand* (more abstract)

Model-based Test Case Generation



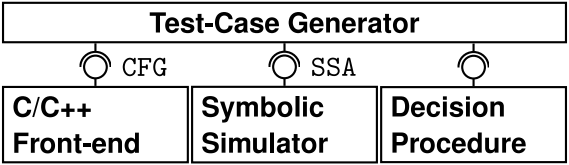
Model-based Test Case Generation

Common modelling languages:

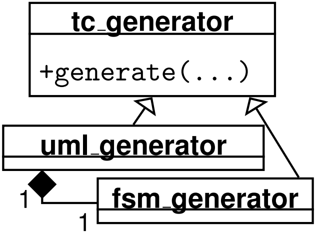
- Unified Modeling Language (UML)
  - + Object Constraint Language (OCL)
- Finite State Machines
- Matlab/Simulink
- SCADE/Esterel
- ...

Modelling Languages

- Component diagrams in UML
  - Illustrates architecture

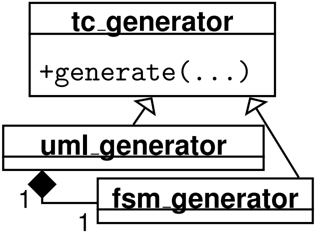


- Class diagrams in UML
  - Specifies class interfaces and relations



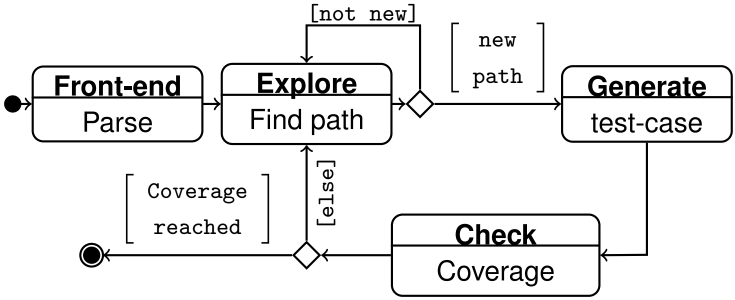
Modelling Languages

- Class diagrams in UML
  - Specifies class interfaces and relations

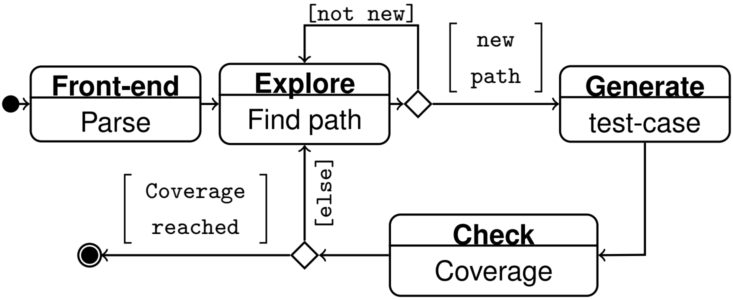


- Neither component nor class diagrams specify *behaviour!*
  - Needed for test-case generation!

Activity diagrams

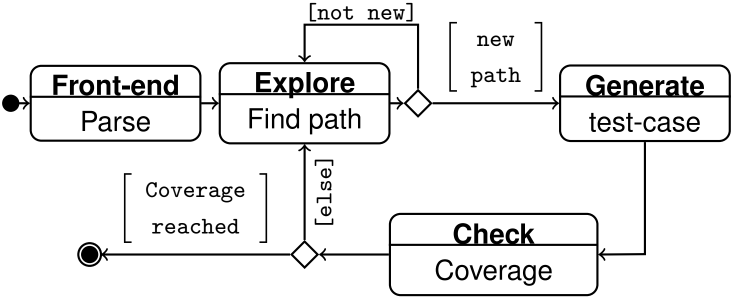


Activity diagrams



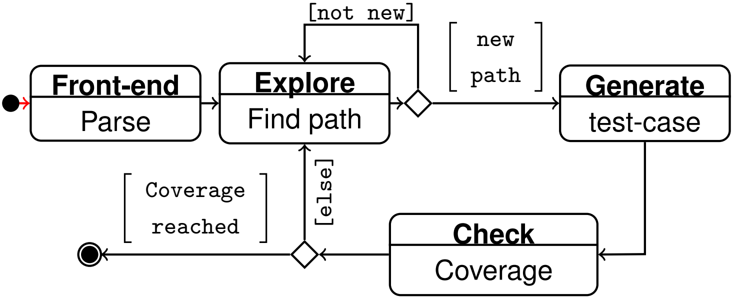
- Describes possible sequences of events
- Can be used to derive *abstract* test-cases

Model-based Test-Case Generation



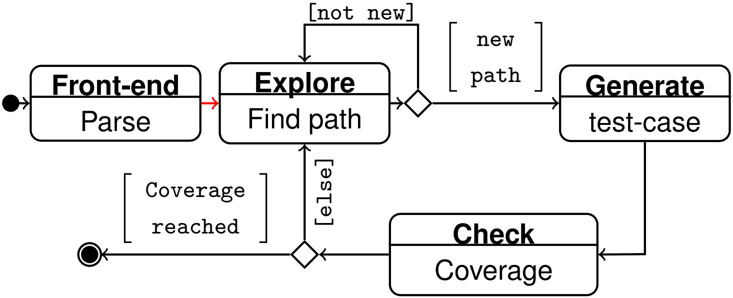
Test case:

Model-based Test-Case Generation



Test case:

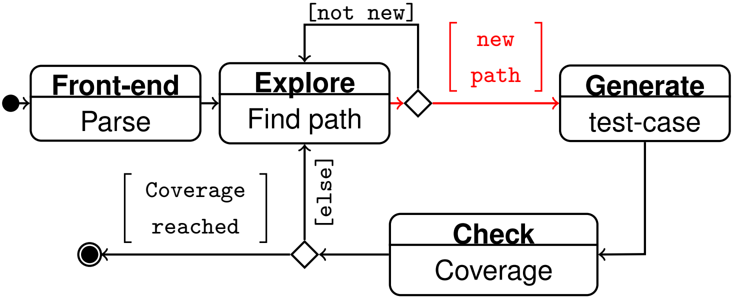
- Start,



Test case:

- Start, parse,

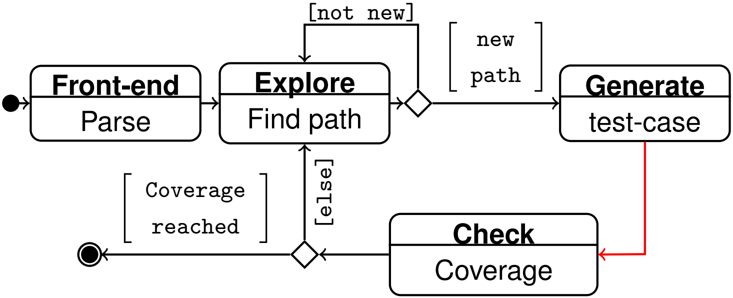
Model-based Test-Case Generation



Test case:

- Start, parse, find path,

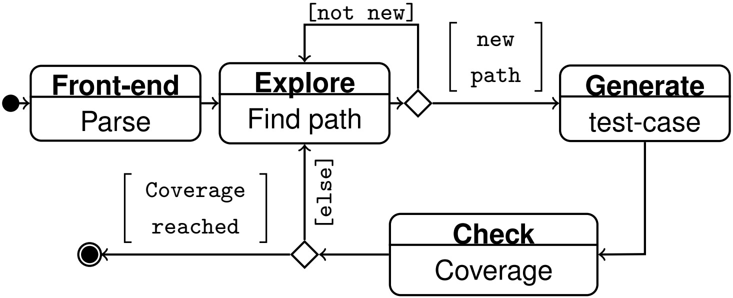
Model-based Test-Case Generation



Test case:

- Start, parse, find path, generate test-case,

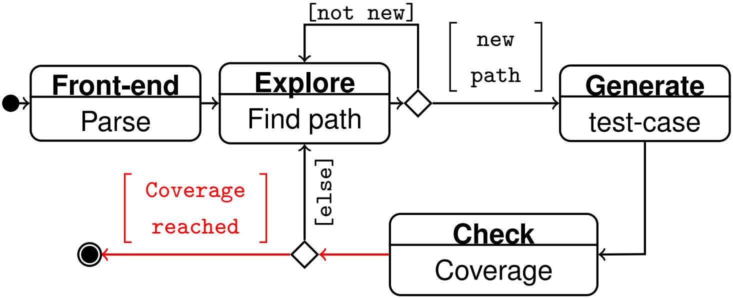
Model-based Test-Case Generation



Test case:

- Start, parse, find path, generate test-case, check coverage,

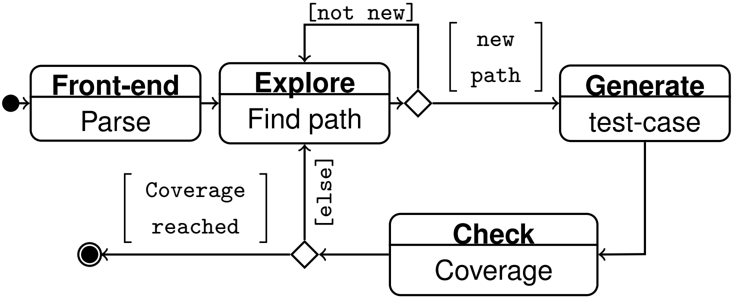
Model-based Test-Case Generation



Test case:

- Start, parse, find path, generate test-case, check coverage, coverage achieved,

Model-based Test-Case Generation



Test case:

- Start, parse, find path, generate test-case, check coverage, coverage achieved, done.

Challenge: Abstraction level

- We generated an *abstract* test-case
  - How can we map it to a concrete test-case?
    - Implementation may have *more details*
  - Is there even a corresponding concrete test-case? (feasibility)
  - How can we test the outcome?
  - Can we provide any *coverage* guarantees (for implementation)?

Challenge: Abstraction level

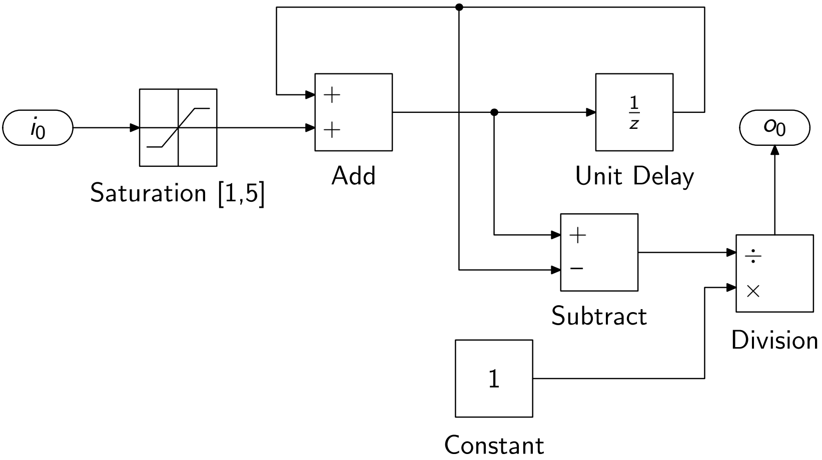
- We generated an *abstract* test-case
  - How can we map it to a concrete test-case?
    - Implementation may have *more details*
  - Is there even a corresponding concrete test-case? (feasibility)
  - How can we test the outcome?
  - Can we provide any *coverage* guarantees (for implementation)? Why not?

Challenge: Abstraction level

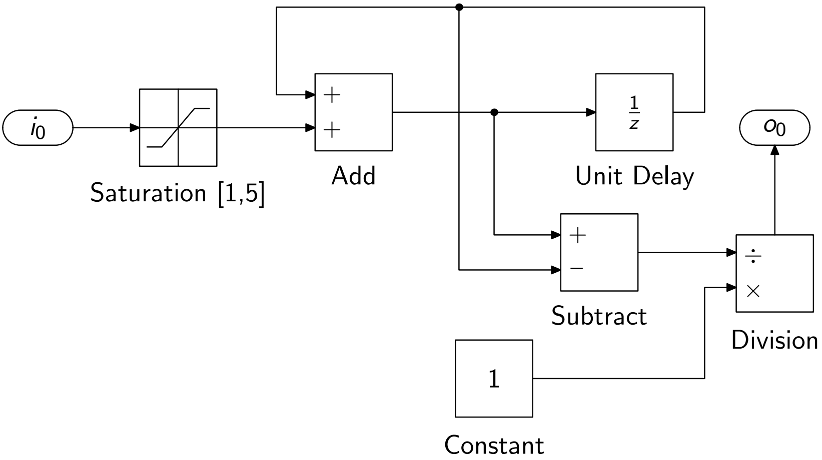
- We generated an *abstract* test-case
  - How can we map it to a concrete test-case?
    - Implementation may have *more details*
  - Is there even a corresponding concrete test-case? (feasibility)
  - How can we test the outcome?
  - Can we provide any *coverage* guarantees (for implementation)? Why not?

Maybe we can choose a *less abstract* modelling language?

Modelling Language: Simulink

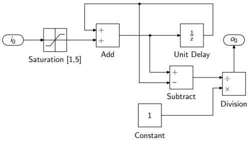


Modelling Language: Simulink

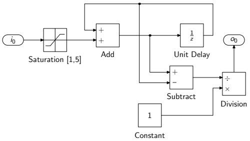


Sums up $t_n = \sum_{j=1}^{n} i_j$, computes $\frac{1}{t_n - t_{n-1}}$

Automating Test-Case Generation: A (Discouraging) Example

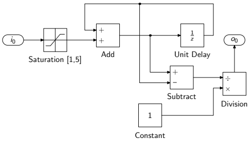


Automating Test-Case Generation: A (Discouraging) Example



$\longrightarrow$

Automating Test-Case Generation: A (Discouraging) Example

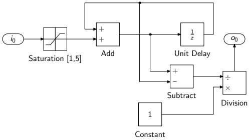


→

→

i_0	i_1	o_0	o_1
1	2	1	$\frac{1}{2}$
3	5	$\frac{1}{3}$	$\frac{1}{5}$
...			

Automating Test-Case Generation: A (Discouraging) Example



→

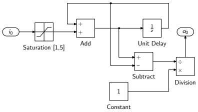
→

i_0	i_1	o_0	o_1
1	2	1	$\frac{1}{2}$
3	5	$\frac{1}{3}$	$\frac{1}{5}$
...			

- To which implementation will we apply the test-suite?

Automating Test-Case Generation: A (Discouraging) Example

- Simulink enables code generation

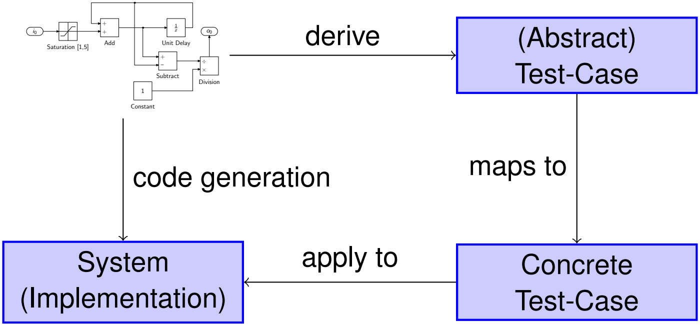


code generation

System
(Implementation)

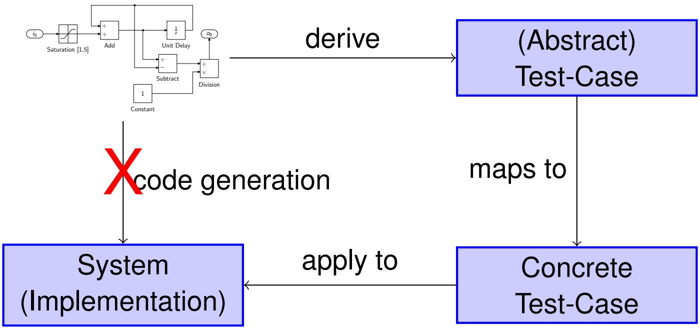
Automating Test-Case Generation: A (Discouraging) Example

- Simulink enables code generation



Automating Test-Case Generation: A (Discouraging) Example

- Simulink enables code generation
- Can you spot the problem?
- *What* are we testing here?



Automating Test-Case Generation: A Cautious Tale

Don't ...

- extract test-cases ($\stackrel{\text{def}}{=}$input+output) *from* implementation
- apply test-cases extracted from model to *generated code*
- let coverage criteria drive your test-case generation

Automating Test-Case Generation: A Cautious Tale

Don't . . .

- extract test-cases ($\stackrel{\text{def}}{=}$input+output) *from* implementation
- apply test-cases extracted from model to *generated code*
- let coverage criteria drive your test-case generation Why?
  - coverage becomes meaningless as a stopping criterion

A Case for Automated Test-Case Generation?

- Are there meaningful applications of TCG?

A Case for Automated Test-Case Generation?

- Are there meaningful applications of TCG?
- Can we “decouple” TCG from the specification?

A Case for Automated Test-Case Generation?

- Are there meaningful applications of TCG?
- Can we “decouple” TCG from the specification?
  - Assertions are partial *specifications*
  - Constrain *behaviour* of program

A Case for Automated Test-Case Generation?

- Are there meaningful applications of TCG?
- Can we “decouple” TCG from the specification?
  - Assertions are partial *specifications*
  - Constrain *behaviour* of program
- Bug hunt! (Assertion violations, crashes. . .)
  - Find inputs that crash the system
  - No outputs required

Generate Inputs that Make System Crash

Inputs that result in

- buffer overflows
- division by zero
- invalid pointer dereferences
- assertion violations
- ...

Generate Inputs that Make System Crash

Inputs that result in

- buffer overflows
- division by zero
- invalid pointer dereferences
- assertion violations
- ...

Assertions are the *most general* mechanism in this list

Generate Inputs that Make System Crash

Inputs that result in

- buffer overflows
  - `assert (i < len); ...   a[i]`
- division by zero
- invalid pointer dereferences
- assertion violations
- ...

Assertions are the *most general* mechanism in this list

Generate Inputs that Make System Crash

Inputs that result in

- buffer overflows
  - `assert (i < len); ...   a[i]`
- division by zero
  - `assert (y != 0); ...   x/y`
- invalid pointer dereferences
- assertion violations
- ...

Assertions are the *most general* mechanism in this list

Generate Inputs that Make System Crash

Inputs that result in

- buffer overflows
  - `assert (i < len); ...   a[i]`
- division by zero
  - `assert (y != 0); ...   x/y`
- invalid pointer dereferences
  - `assert (p != NULL); ...   *p`
- assertion violations
- ...

Assertions are the *most general* mechanism in this list

Assertion Violations

When does an assertion `assert(`*P*`);` fail?

- if *P* evaluates to *false*
- depends on *values of variables, heap, ...*

How do we evaluate *P*?

- As specified by language definition
- Remember lecture on assertions

Assertion Violations

When does an assertion `assert(`P`);` fail?

- if P evaluates to *false*
- depends on *values of variables, heap, . . .* (program state)

How do we evaluate P?

- As specified by language definition
- Remember lecture on assertions

Expressions (ISO/IEC 14882:2011, §5)

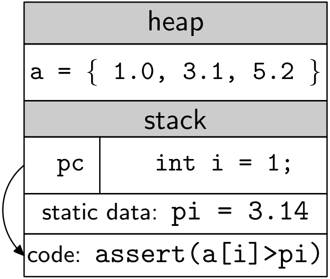
- e.g., syntax for *multiplicative expressions*:

  multiplicative-expression:
  - *pm-expression* (e.g., a variable)
  - *multiplicative-expression* * *pm-expression*
  - *multiplicative-expression* / *pm-expression*
  - *multiplicative-expression* % *pm-expression*

- semantics (meaning) of multiplicative operators:
  - "3 The binary $*$ operator indicates multiplication"
  - "4 The binary / operator yields the quotient, and the binary % operator yields the remainder from the division of the first expression by the second. If the second operand of / or % is zero the behavior is undefined. [...]"

Assertion Violations

What's going to happen next?



Assertion Violations

- There are *many* conceivable states violating that assertion
- We *only need to find one!*

assertion	pi	i	a
(a[i]>pi)	3.14	0	{ 0.1, 5.2, 3.14 }
(a[i]>pi)	3.14	2	{ 1.0, 3.1, 1.2 }
...			

Assertion Violations

A bit of terminology:

- An expression P is *satisfiable* if there *exists* a valuation of its variables that makes it true.
- An expression P is *unsatisfiable* if there *exists* no valuation of its variables that makes it true.

Assertion Violations

A bit of terminology:

- An expression *P* is *satisfiable* if there *exists* a valuation of its variables that makes it true.
- An expression *P* is *unsatisfiable* if there *exists* no valuation of its variables that makes it true.

A brief quiz: satisfiable or unsatisfiable?

1. `(a > b) && (y == a)`
2. `(a > b) && (a + b == 0)`
3. `((a + b) % 2 == 0) && (b & 1) && (a == 0)`
4. `(a != b) || (a == b)`

Assertion Violations

A bit of terminology:

- An expression *P* is *satisfiable* if there *exists* a valuation of its variables that makes it true.
- An expression *P* is *unsatisfiable* if there *exists* no valuation of its variables that makes it true.

A brief quiz: satisfiable or unsatisfiable?

1. `(a > b) && (y == a)`
2. `(a > b) && (a + b == 0)`
3. `((a + b) % 2 == 0) && (b & 1) && (a == 0)`
4. `(a != b) || (a == b)`
    - What's special about this one?

Assertion Violations

This can get trickier! (e.g., bit-vector arithmetic)

```
((x!=y)||(x&2)==2)&&(y==z+z)&&(x==(z<<1))&&((z&1)==0)
```

Assertion Violations

This can get trickier! (e.g., bit-vector arithmetic)

```
((x!=y)||(x&2)==2)&&(y==z+z)&&(x==(z<<1))&&((z&1)==0)
```

- `(z&1)==0`, therefore `(z<<1)&2==0`

Assertion Violations

This can get trickier! (e.g., bit-vector arithmetic)

```
((x!=y)||(x&2)==2)&&(y==z+z)&&(x==(z<<1))&&((z&1)==0)
```

- `(z&1)==0`, therefore `(z<<1)&2==0`
  - It follows that `x&2==0`

Assertion Violations

This can get trickier! (e.g., bit-vector arithmetic)

`((x!=y)||(x&2)==2)&&(y==z+z)&&(x==(z<<1))&&((z&1)==0)`

- `(z&1)==0`, therefore `(z<<1)&2==0`
  - It follows that `x&2==0`
- `y==z+z`, therefore `y==z<<1`

Assertion Violations

This can get trickier! (e.g., bit-vector arithmetic)

```
((x!=y)||(x&2)==2)&&(y==z+z)&&(x==(z<<1))&&((z&1)==0)
```

- `(z&1)==0`, therefore `(z<<1)&2==0`
  - It follows that `x&2==0`
- `y==z+z`, therefore `y==z<<1`
  - It follows that `x==y`

Assertion Violations

This can get trickier! (e.g., bit-vector arithmetic)

```
((x!=y)||(x&2)==2)&&(y==z+z)&&(x==(z<<1))&&((z&1)==0)
```

- `(z&1)==0`, therefore `(z<<1)&2==0`
  - It follows that `x&2==0`
- `y==z+z`, therefore `y==z<<1`
  - It follows that `x==y`
- Therefore, the disjunction `((x!=y)||(x&2)==2)` is false
  - Expression is *unsatisfiable*

Satisfiable or not?

- Manual analysis of these examples is tedious
- There are *automated decision procedures* for satisfiability
- e.g., the *Satisfiability Modulo Theory* (SMT) solver *Z3*
  - `https://github.com/Z3Prover/z3`
  - (there'll be a separate lecture on SMT solvers)

Satisfiable or not? – Z3

- Unfortunately, Z3 doesn't speak C++
  - Need to translate our input
  - Front end simplicity over "linguistic convenience"
  - Uses *polish notation*, i.e., `(+ 3 4)` instead of $3 + 4$
  - Tutorial on `https://github.com/Z3Prover/z3/wiki#background`

Z3 Crash Course

- Variables need to be declared and typed:
  - `(declare-const p Bool)`
  - `(declare-const q Bool)`

  ("variables" in Z3 are constants/null-ary functions)
- We can add "assertions" over declared variables
  - `(assert (or p q))`
- We can check satisfiability
  - `(check-sat)`
- We can ask for a *model*
  - `(get-model)`

Satisfiable or not? Let's ask Z3

```
(declare-const p Bool)
(declare-const q Bool)
(assert (or p q))
(check-sat)
(get-model)
```

And the answer is:

```
sat
(model
  (define-fun q () Bool
    false)
  (define-fun p () Bool
  true)
)
```

Satisfiable or not? Let's ask Z3

The answer is:

```
sat
(model
  (define-fun q () Bool
    false)
  (define-fun p () Bool
  true)
)
```

- Remember: Variables are constants/null-ary functions
- A null-ary function has *no* parameters
- Returns a value
- In this context, just like a variable

Satisfiable or not? Let's ask Z3

```
(declare-const p Bool)
(declare-const q Bool)
(assert (and (or p q) (and (not p) (not q))))
(check-sat)
```

- And the answer is ... `unsat`

Types and Theories supported by SMT-Solvers

- SMT-Solvers/Z3 can do more than just propositional logic
  - Arithmetic
  - “Uninterpreted” functions
  - Arrays
  - Bit-Vectors
  - ...

Bit-Vectors in SMT

- Binary and hexadecimal constants:
  - `#b0100`
  - `#x0a`
- Declare "variables" of type bit-vector:
  - `(declare-const x (_ BitVec 16))`
  - `(declare-const y (_ BitVec 16))`
- Bit-vector operations
  - `(bvadd x #x0001)` denotes $x + 1$
  - `(bvsub x y)` denotes $x - y$
  - `(bvneg x)` denotes $-x$
  - `(bvmul x y)` denotes $x * y$
  - `(bvshl x #x0001)` denotes $x << 1$ (shift-left)
  - ...

Bit-Vectors in SMT

- Binary and hexadecimal constants:
  - `#b0100` (this is decimal 4)
  - `#x0a`
- Declare "variables" of type bit-vector:
  - `(declare-const x (_ BitVec 16))`
  - `(declare-const y (_ BitVec 16))`
- Bit-vector operations
  - `(bvadd x #x0001)` denotes $x + 1$
  - `(bvsub x y)` denotes $x - y$
  - `(bvneg x)` denotes $-x$
  - `(bvmul x y)` denotes $x * y$
  - `(bvshl x #x0001)` denotes $x << 1$ (shift-left)
  - ...

Bit-Vectors in SMT

- Binary and hexadecimal constants:
  - `#b0100` (this is decimal 4)
  - `#x0a` (this is decimal 10)
- Declare "variables" of type bit-vector:
  - `(declare-const x (_ BitVec 16))`
  - `(declare-const y (_ BitVec 16))`
- Bit-vector operations
  - `(bvadd x #x0001)` denotes $x + 1$
  - `(bvsub x y)` denotes $x - y$
  - `(bvneg x)` denotes $-x$
  - `(bvmul x y)` denotes $x * y$
  - `(bvshl x #x0001)` denotes $x << 1$ (shift-left)
  - ...

Satisfiable or not?

```
(declare-const x (_ BitVec 16))
(declare-const y (_ BitVec 16))
(declare-const z (_ BitVec 16))

(assert
  (and
    (or
       (not (= x y))
       (= (bvand x #x0002) #x0002)
    )
    (= y (bvadd z z))
    (= x (bvshl z #x0001))
    (= (bvand z #x0001) #x0000)
  )
)
(check-sat)
```

Back to Assertion Violations

SMT solvers enable us to *guess* a state that violates assertion!

```
assert (a[i]>pi);
```

- `i`=0, `pi`=3.14, a={0.0} satisfies `!(a[i]>pi)`
- Can the program be in that state when assertion is reached?

```
const float pi = 3.14;
float a[] = {4.0, 4.0};
int i = 0;
assert (a[i]>pi);
```

Back to Assertion Violations

SMT solvers enable us to *guess* a state that violates assertion!

```
assert (a[i]>pi);
```

- `i`=0, `pi`=3.14, `a`={0.0} satisfies `!(a[i]>pi)`
- Can the program be in that state when assertion is reached?

```
const float pi = 3.14;      ↓(pi=3.14)
float a[] = {4.0, 4.0};
int i = 0;
assert (a[i]>pi);
```

Back to Assertion Violations

SMT solvers enable us to *guess* a state that violates assertion!

```
assert (a[i]>pi);
```

- `i`=0, `pi`=3.14, `a`={0.0} satisfies `!(a[i]>pi)`
- Can the program be in that state when assertion is reached?

```
const float pi = 3.14;       ↓(pi=3.14)
float a[] = {4.0, 4.0};      ↓(a[0]=4.0)&&(a[1]=4.0))
int i = 0;
assert (a[i]>pi);
```

Back to Assertion Violations

SMT solvers enable us to *guess* a state that violates assertion!

```
assert (a[i]>pi);
```

- `i`=0, `pi`=3.14, `a`={0.0} satisfies `!(a[i]>pi)`
- Can the program be in that state when assertion is reached?

```
const float pi = 3.14;        ↓(pi=3.14)
float a[] = {4.0, 4.0};       ↓(a[0]=4.0)&&(a[1]=4.0))
int i = 0;                    ↓(i=0)
assert (a[i]>pi);
```

Back to Assertion Violations

SMT solvers enable us to *guess* a state that violates assertion!

```
assert (a[i]>pi);
```

- `i`=0, `pi`=3.14, `a`={0.0} satisfies `!(a[i]>pi)`
- Can the program be in that state when assertion is reached?

```
const float pi = 3.14;        ↓(pi=3.14)
float a[] = {4.0, 4.0};       ↓(a[0]=4.0)&&(a[1]=4.0))
int i = 0;                    ↓(i=0)
assert (a[i]>pi);             ↓!(a[i]>pi)
```

Can the Assertion Be Violated?

- Is

  ```
  (pi==3.14)&&(a[0]==4.0)&&(a[1]==4.0)&&(i==0)
                     &&!(a[i]>pi)
  ```

  satisfiable?

Can the Assertion Be Violated?

- Is

```
(pi==3.14)&&(a[0]==4.0)&&(a[1]==4.0)&&(i==0)
          &&!(a[i]>pi)
```

  satisfiable?
- No!

Can the Assertion Be Violated?

- What about the following program?
  - Let `i` be an uninitialised variable (or user input)

```
int i;
const float pi = 3.14;
float a[] = {1.0, 5.0};
assert (a[i]>pi);
```

Can the Assertion Be Violated?

- What about the following program?
  - Let `i` be an uninitialised variable (or user input)

```
int i;                              ↓(i=?)
const float pi = 3.14;
float a[] = {1.0, 5.0};
assert (a[i]>pi);
```

Can the Assertion Be Violated?

- What about the following program?
  - Let `i` be an uninitialised variable (or user input)

```
int i;                             ↓(i=?)
const float pi = 3.14;             ↓(pi=3.14)
float a[] = {1.0, 5.0};
assert (a[i]>pi);
```

Can the Assertion Be Violated?

- What about the following program?
  - Let `i` be an uninitialised variable (or user input)

```
int i;                          ↓(i=?)
const float pi = 3.14;          ↓(pi=3.14)
float a[] = {1.0, 5.0};         ↓(a[0]=1.0)&&(a[1]=5.0))
assert (a[i]>pi);
```

Can the Assertion Be Violated?

- What about the following program?
  - Let `i` be an uninitialised variable (or user input)

```
int i;                          ↓(i=?)
const float pi = 3.14;          ↓(pi=3.14)
float a[] = {1.0, 5.0};         ↓(a[0]=1.0)&&(a[1]=5.0))
assert (a[i]>pi);               ↓!(a[i]>pi)
```

Can the Assertion Be Violated?

- What about the following program?
  - Let `i` be an uninitialised variable (or user input)

```
int i;
const float pi = 3.14;
float a[] = {1.0, 5.0};
assert (a[i]>pi);
```

↓ (i=?)
↓ (pi=3.14)
↓ (a[0]=1.0)&&(a[1]=5.0))
↓ !(a[i]>pi)

- `i`'s value is “undetermined”
  - Could be any `int` value

Can the Assertion Be Violated?

- What about the following program?
  - Let `i` be an uninitialised variable (or user input)

```
int i;                          ↓(i=?)
const float pi = 3.14;          ↓(pi=3.14)
float a[] = {1.0, 5.0};         ↓(a[0]=1.0)&&(a[1]=5.0))
assert (a[i]>pi);               ↓!(a[i]>pi)
```

- `i`'s value is "undetermined"
  - Could be any `int` value
- Assertion violated if we choose `i` to be 0

Symbolic vs. Concrete Values

- Concrete values:
  Actual values a variable or data-structure could take during execution, e.g., 1, 2, -3.14, `true`, “Hello world”, ...
- Symbolic values:
  Placeholder values (undetermined values), representing, for instance, user input

Symbolic vs. Concrete Values

- Let's use x_0 to denote *symbolic values* of `x`
- Which input value makes the following function fail?

```
int foo(int x)
{
  int y = x + 1;
  assert (y!=0);
  return (x/y);
}
```

Symbolic vs. Concrete Values

- Let's use x_0 to denote *symbolic values* of `x`
- Which input value makes the following function fail?

```
int foo(int x)
{
  int y = x + 1;
  assert (y!=0);
  return (x/y);
}
```

$\downarrow \texttt{x} \mapsto x_0$

Symbolic vs. Concrete Values

- Let's use x_0 to denote *symbolic values* of `x`
- Which input value makes the following function fail?

```
int foo(int x)
{
  int y = x + 1;
  assert (y!=0);
  return (x/y);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

Symbolic vs. Concrete Values

- Let's use x_0 to denote *symbolic values* of `x`
- Which input value makes the following function fail?

```
int foo(int x)
{
  int y = x + 1;
  assert (y!=0);
  return (x/y);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow \; (x_0 + 1 \neq 0)$

Symbolic vs. Concrete Values

- Let's use x_0 to denote *symbolic values* of `x`
- Which input value makes the following function fail?

```
int foo(int x)
{
  int y = x + 1;
  assert (y!=0);
  return (x/y);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow \ (x_0 + 1 \neq 0)$

- Representation of *an equivalence class of executions*
  - for *all possible values of* `x` (represented by x_0)

Symbolic vs. Concrete Values

- Let's use x_0 to denote *symbolic values* of `x`
- Which input value makes the following function fail?

```
int foo(int x)
{
  int y = x + 1;
  assert (y!=0);
  return (x/y);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow \; (x_0 + 1 \neq 0)$

- Representation of *an equivalence class of executions*
  - for *all possible values of* `x` (represented by x_0)
- Can we make this "symbolic" execution fail?

Symbolic vs. Concrete Values

- Let's use x_0 to denote *symbolic values* of `x`
- Which input value makes the following function fail?

```
int foo(int x)
{
  int y = x + 1;
  assert (y!=0);
  return (x/y);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$
$\downarrow \,!\,(x_0 + 1 \neq 0)$

- Representation of *an equivalence class of executions*
  - for *all possible values of* `x` (represented by x_0)
- Can we make this "symbolic" execution fail?

Symbolic vs. Concrete Values

- Let's use x_0 to denote *symbolic values* of `x`
- Which input value makes the following function fail?

```
int foo(int x)
{
  int y = x + 1;
  assert (y!=0);
  return (x/y);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow\, !\,(x_0 + 1 \neq 0)$

- Representation of *an equivalence class of executions*
  - for *all possible values of* `x` (represented by x_0)
- Can we make this "symbolic" execution fail?
  - Ask the SMT solver whether $!(x_0 + 1 \neq 0)$ is satisfiable

Symbolic Executions

- What happens if we encounter conditions?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

Symbolic Executions

- What happens if we encounter conditions?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \texttt{x} \mapsto x_0$

- What happens if we encounter conditions?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

Symbolic Executions

- What happens if we encounter conditions?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow (x_0 > -1)$

Symbolic Executions

- What happens if we encounter conditions?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow (x_0 > -1)$

$\downarrow y \mapsto x_0 + 2$

Symbolic Executions

- What happens if we encounter conditions?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow (x_0 > -1)$

$\downarrow y \mapsto x_0 + 2$

$\downarrow \ (x_0 + 2 \neq 0)$

- What happens if we encounter conditions?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow (x_0 > -1)$

$\downarrow y \mapsto x_0 + 2$

$\downarrow !(x_0 + 2 \neq 0)$

Symbolic Executions

- What happens if we encounter conditions?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow (x_0 > -1)$

$\downarrow y \mapsto x_0 + 2$

$\downarrow !(x_0 + 2 \neq 0)$

Symbolic Executions

- What happens if we encounter conditions?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow (x_0 > -1)$

$\downarrow y \mapsto x_0 + 2$

$\downarrow !(x_0 + 2 \neq 0)$

- All conditions along the path must be satisfied

Symbolic Executions

- What happens if we encounter conditions?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow (x_0 > -1)$

$\downarrow y \mapsto x_0 + 2$

$\downarrow !(x_0 + 2 \neq 0)$

- All conditions along the path must be satisfied
- Ask the SMT solver whether

$$(x_0 > -1)\&\&!(x_0 + 2 \neq 0)$$

is satisfiable

Symbolic Executions

- What happens if we encounter conditions?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow (x_0 > -1)$

$\downarrow y \mapsto x_0 + 2$

$\downarrow !(x_0 + 2 \neq 0)$

- All conditions along the path must be satisfied
- Ask the SMT solver whether

$$(x_0 > -1) \&\& !(x_0 + 2 \neq 0)$$

  is satisfiable
  - It is not! Path is *safe*

Symbolic Executions

- What if we take the `else`-branch?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

Symbolic Executions

- What if we take the `else`-branch?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

Symbolic Executions

- What if we take the `else`-branch?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

Symbolic Executions

- What if we take the `else`-branch?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow (x_0 \leq -1)$

- What if we take the `else`-branch?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow (x_0 \leq -1)$

$\downarrow \ (x_0 + 1 \neq 0)$

Symbolic Executions

- What if we take the `else`-branch?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \texttt{x} \mapsto x_0$

$\downarrow \texttt{y} \mapsto x_0 + 1$

$\downarrow (x_0 \leq -1)$

$\downarrow !(x_0 + 1 \neq 0)$

Symbolic Executions

- What if we take the `else`-branch?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow (x_0 \leq -1)$

$\downarrow !(x_0 + 1 \neq 0)$

- All conditions along the path must be satisfied

Symbolic Executions

- What if we take the `else`-branch?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \texttt{x} \mapsto x_0$

$\downarrow \texttt{y} \mapsto x_0 + 1$

$\downarrow (x_0 \leq -1)$

$\downarrow !(x_0 + 1 \neq 0)$

- All conditions along the path must be satisfied
- Ask the SMT solver whether

$$(x_0 \leq -1) \&\& !(x_0 + 1 \neq 0)$$

  is satisfiable

Symbolic Executions

- What if we take the `else`-branch?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow (x_0 \leq -1)$

$\downarrow !(x_0 + 1 \neq 0)$

- All conditions along the path must be satisfied
- Ask the SMT solver whether

$$(x_0 \leq -1)\&\&!(x_0 + 1 \neq 0)$$

is satisfiable

Symbolic Executions

- What if we take the `else`-branch?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \texttt{x} \mapsto x_0$

$\downarrow \texttt{y} \mapsto x_0 + 1$

$\downarrow (x_0 \leq -1)$

$\downarrow !(x_0 + 1 \neq 0)$

- All conditions along the path must be satisfied
- Ask the SMT solver whether

$$(x_0 \leq -1) \&\& !(x_0 + 1 \neq 0)$$

is satisfiable

Symbolic Executions

- What if we take the `else`-branch?

```
void bar(int x)
{
  int y = x + 1;
  if (x > -1)
    y = y + 1;
  assert (y!=0);
}
```

$\downarrow \mathtt{x} \mapsto x_0$

$\downarrow \mathtt{y} \mapsto x_0 + 1$

$\downarrow (x_0 \leq -1)$

$\downarrow !(x_0 + 1 \neq 0)$

- All conditions along the path must be satisfied
- Ask the SMT solver whether

$$(x_0 \leq -1)\&\&!(x_0 + 1 \neq 0)$$

  is satisfiable
  - It is $(x_0 = -1)$, therefore assertion can be violated

Symbolic Executions

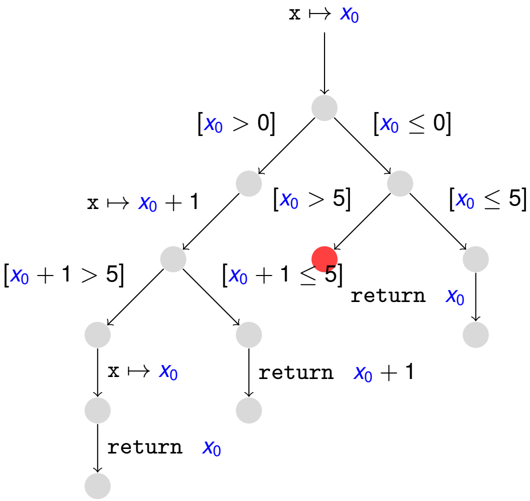
① Perform *symbolic* execution of path

② At any assertion:

- ask SMT solver for input assignment violating it

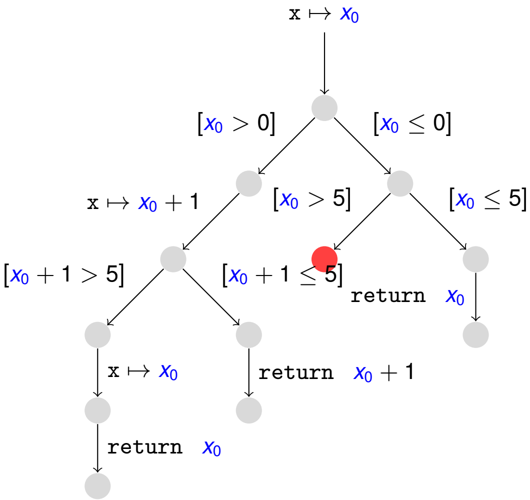
Symbolic Execution Trees

```
int baz(int x)
{
 if (x>0)
  x = x + 1;
 if (x>5)
  x = x - 1;
 return x;
}
```



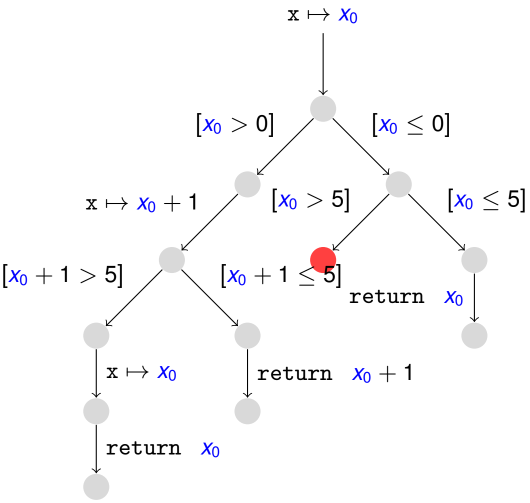
Symbolic Execution Trees

```
int baz(int x)
{
 if (x>0)
  x = x + 1;
 if (x>5)
  x = x - 1;
 return x;
}
```

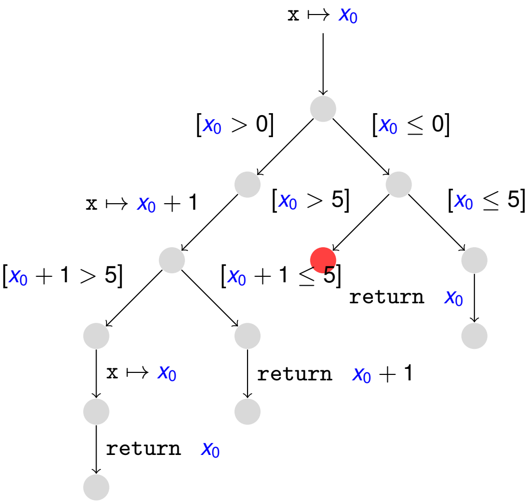


Explore paths; search for reachable assertions

Optimisations

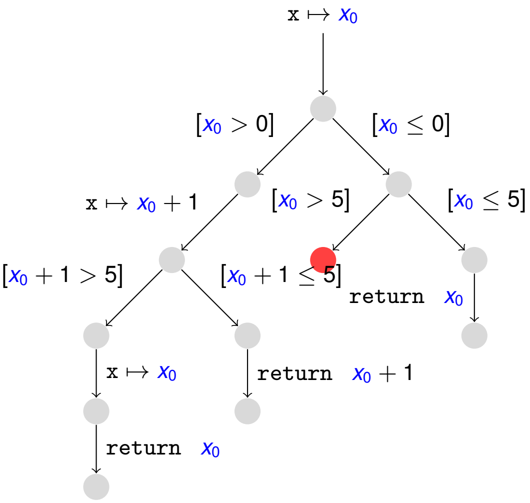


Optimisations



- Some paths are *infeasible*

Optimisations



- Some paths are *infeasible*
- Some conditions are *implied* (e.g., $(x_0 \leq 0) \Rightarrow (x_0 \leq 5)$)

Optimisations

- *Infeasible* paths don't need to be explored further
  - Reduces number of paths
- *Implied* conditions can be dropped
  - Makes problem for SMT solver easier

Optimisations

- *Infeasible* paths don't need to be explored further
  - Reduces number of paths
- *Implied* conditions can be dropped
  - Makes problem for SMT solver easier
- Two different concerns:

Path explosion	Constraint solving
(will address this now)	(see lectures end of April)

Optimisations

- How many *paths* in this function:

```
for (int i = 0; i < N; i++)
{
  char ch = getchar();
  if (ch == ' ')
      space++;
  else
      other++;
}
```

- Naïve exploration quickly becomes a problem!

Optimisations

- How many *paths* in this function:

```
for (int i = 0; i < N; i++)
{
  char ch = getchar();
  if (ch == ' ')
     space++;
  else
     other++;
}
```

- Naïve exploration quickly becomes a problem!
  - Solution: search heuristics!

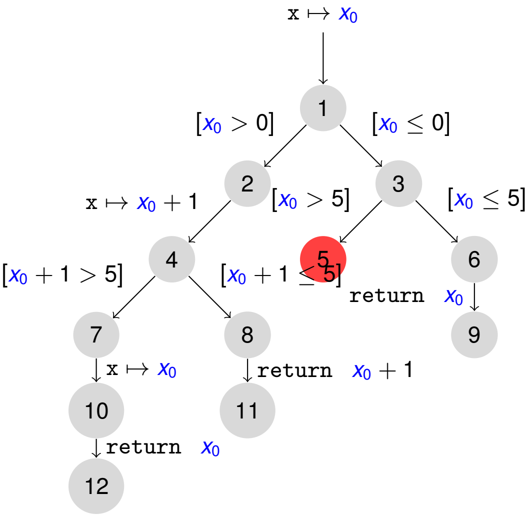
Optimisations: Search Heuristics

Search heuristics:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Coverage-optimised search (Best-First)
  - “best” paths increase coverage
- Random selection/expansion

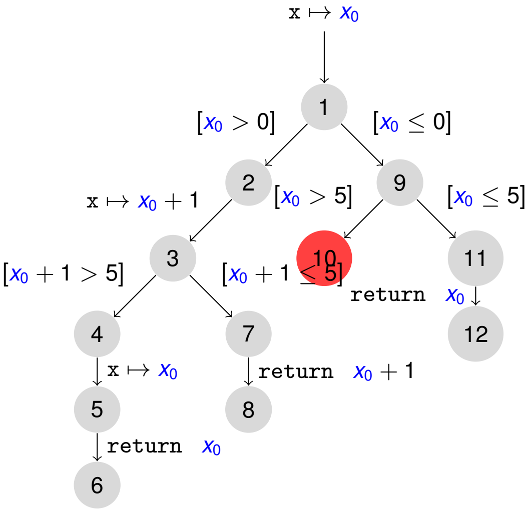
Search Heuristics: BFS

- Don't explore paths of length $k + 1$ before all paths of length k are explored



Search Heuristics: DFS

- Follow path to the end before you explore a new one



Search Heuristics: Coverage-based

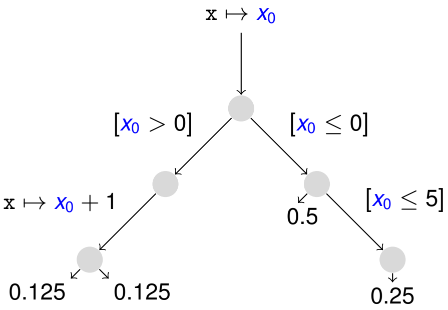
Which (incomplete) path in the search tree do we expand next?

- Expand path “close” to an uncovered instruction
- Favour paths that recently visited *new* code

Search Heuristics: Random

Which (incomplete) path in the search tree do we expand next?

- Randomly choose one
- "Shorter paths" have higher probability
  - Avoids starvation (e.g., symbolic loop)



Search Heuristics

- Can also apply a combination of search heuristics
  - e.g., multiple heuristics in round-robin fashion
  - implemented by KLEE (`http://klee.llvm.org`)

Other Optimisations

- Eliminate *redundant paths*
  - paths that reach same program location with same constraints

Other Optimisations

- Merge paths
  - *merge* paths that reach same program location
  - covered in more detail towards the end of the course

Applications of TCG: Checking Contracts

- We used TCG to detect assertion violations
- Therefore, can also be used to check contract!

```
float sqrt (float x);
```

pre: $\mathtt{x} \geq 0$

post: $|\mathtt{result}^2 - \mathtt{x}| < \varepsilon$

① Generate new test-case *from implementation*

② Check whether input satisfies pre-condition

③ If yes, check whether output satisfies post-condition

Applications of TCG

- Alternatively, we can as an *oracle* for correct output
- The *oracle* could be
  - a less efficient (but correct) implementation
  - an executable specification
  - …

① Generate new test-case *from implementation*

② Run *oracle* with the generated input

③ Compare output of oracle and implementation

Summary

- Automated test case generation is feasible
- But dangerous if applied *naïvely*
  - Outputs *must* be derived from specification
  - Should not be driven by coverage!
  - However, can be applied if outputs are not needed (e.g., crash detection, assertion violations)