CS 575
Software Testing and Analysis

Ozyegin University
Graduate School of Engineering

Automated Test Case Generation
Concolic Testing and
Feedback-directed Random Test Generation

Hasan Sözer
hasan.sozer@ozyegin.edu.tr

Concolic Testing

- Combining **conc**rete execution with symb**olic** execution

- Concrete Execution
  - Based on a specification or random values

- Symbolic Execution
  - Use symbolic values for inputs and variables
  - Calculate path constraints
  - Use a theorem prover to check if a code block is reachable

Example: CUTE
(slides by D. Marinov and G. Agha)

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Probability of reaching abort( ) is extremely low by testing with random x values

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints
p → NULL	, x=236	$p=p_0$, $x=x_0$	

```c
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete Execution

Symbolic Execution

concrete state	symbolic state	constraints
p → NULL, x=236	$p=p_0$, $x=x_0$	

```c
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
p → NULL, x=236	$p=p_0$, $x=x_0$	$x_0>0$

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete Execution | Symbolic Execution

concrete state	symbolic state	constraints
		$x_0>0$
		$!(p_0!=NULL)$
p → NULL, x=236	$p=p_0$, $x=x_0$	

```c
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete Execution

Symbolic Execution

concrete

symbolic

constraints

solve: $x_0>0$ and $p_0 \neq NULL$

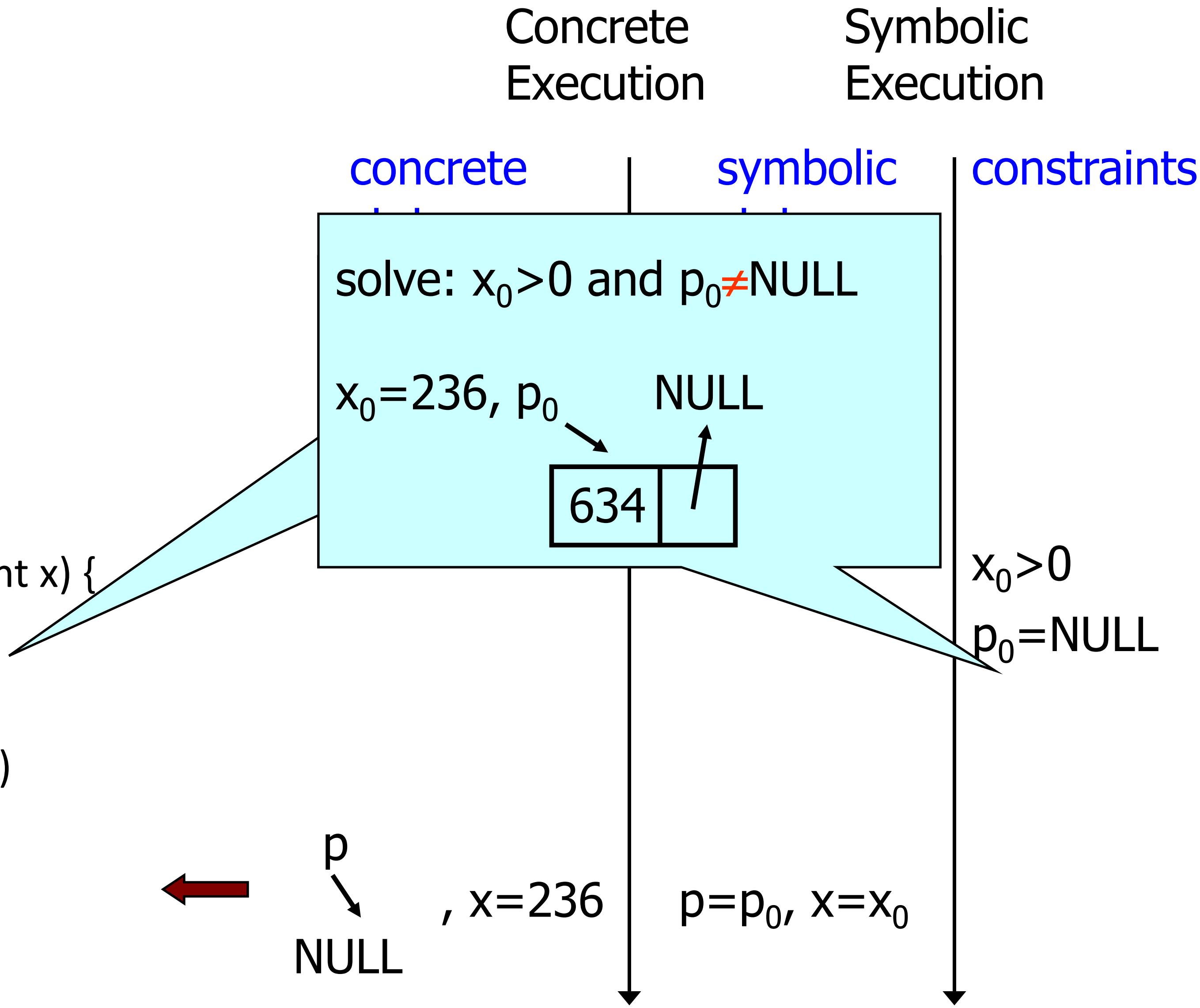
$x_0>0$

$p_0=NULL$

p → NULL, x=236

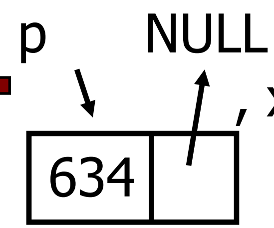
$p=p_0$, $x=x_0$

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



```c
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete Execution | Symbolic Execution

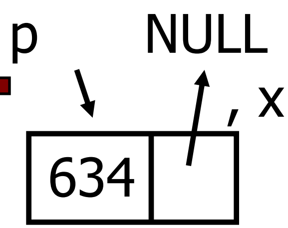
concrete state	symbolic state	constraints
 , x=236	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	

Concrete Execution

Symbolic Execution

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

concrete state



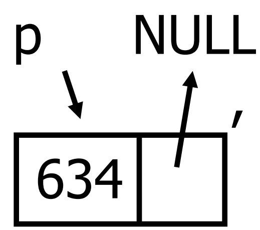
, x=236

symbolic state

$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next=n_0$

constraints

$x_0>0$

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
 , x=236	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next = n_0$	$x_0>0$ $p_0 \neq NULL$

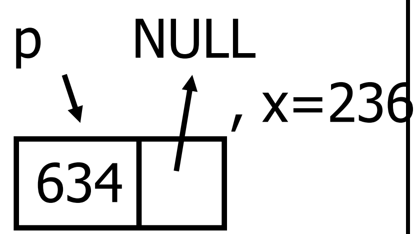
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete Execution

Symbolic Execution

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

concrete state



symbolic state

$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next=n_0$

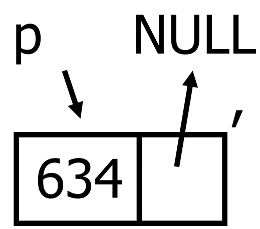
constraints

$x_0>0$

$p_0 \neq NULL$

$2x_0+1 \neq v_0$

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		constraints
 , x=236	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$		$x_0>0$ $p_0 \neq NULL$ $2x_0+1 \neq v_0$

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

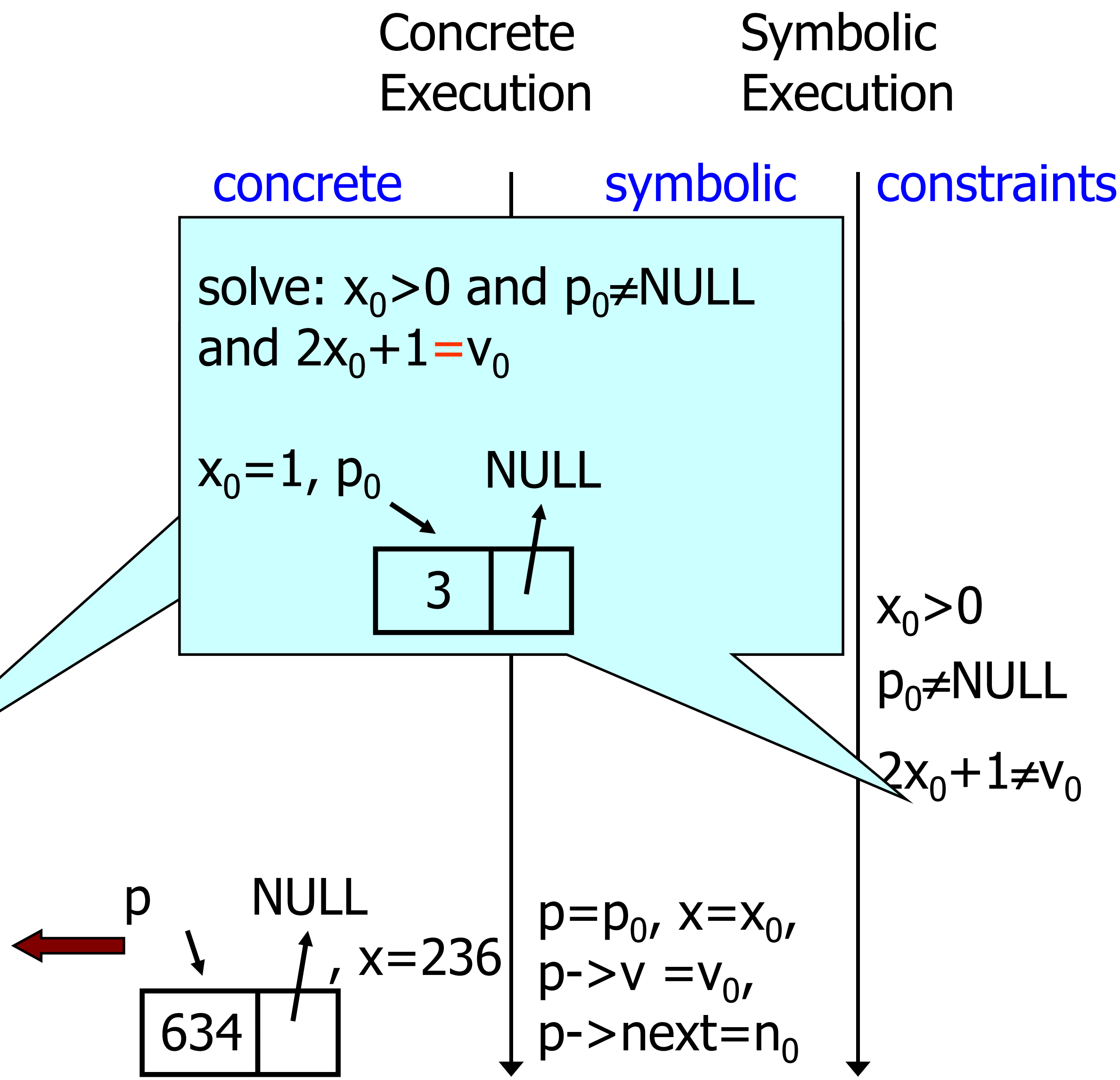
Concrete Execution

Symbolic Execution

concrete	symbolic	constraints
p, NULL, 634, x=236	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$ $p_0 \neq NULL$ $2x_0+1 \neq v_0$

solve: $x_0>0$ and $p_0 \neq NULL$ and $2x_0+1 = v_0$

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

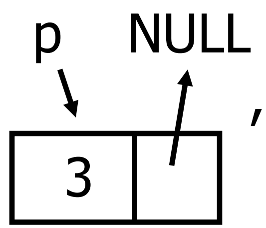


Concrete Execution

Symbolic Execution

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

concrete state



, x=1

symbolic state

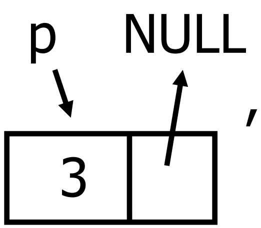
$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next=n_0$

constraints

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

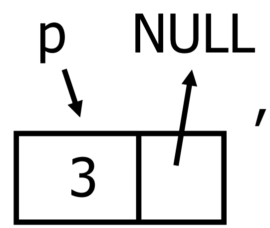
Concrete Execution

Symbolic Execution

concrete state	symbolic state	constraints
 , x=1	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

	Concrete Execution		Symbolic Execution	
	concrete state		**symbolic state**	**constraints**



, x=1

symbolic state:

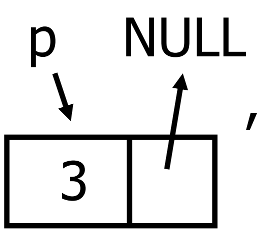
$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next=n_0$

constraints:

$x_0>0$

$p_0 \neq NULL$

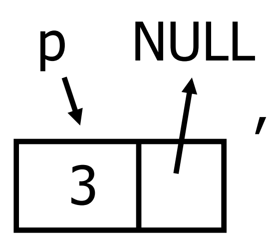
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
 , x=1	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$ $p_0 \neq NULL$ $2x_0+1=v_0$

Concrete Execution | Symbolic Execution

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

concrete state



, x=1

symbolic state

$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

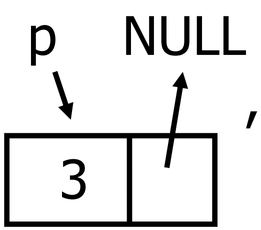
constraints

$x_0 > 0$
$p_0 \neq NULL$
$2x_0 + 1 = v_0$
$n_0 \neq p_0$

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete Execution

Symbolic Execution

concrete state	symbolic state	constraints
 , x=1	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next = n_0$	$x_0>0$ $p_0 \neq NULL$ $2x_0+1=v_0$ $n_0 \neq p_0$

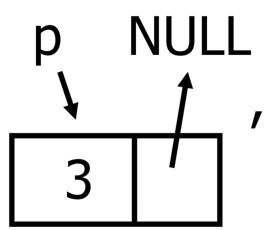
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state	constraints	

solve: $x_0>0$ and $p_0 \neq NULL$ and $2x_0+1=v_0$ and $n_0=p_0$

.

concrete state:



, x=1

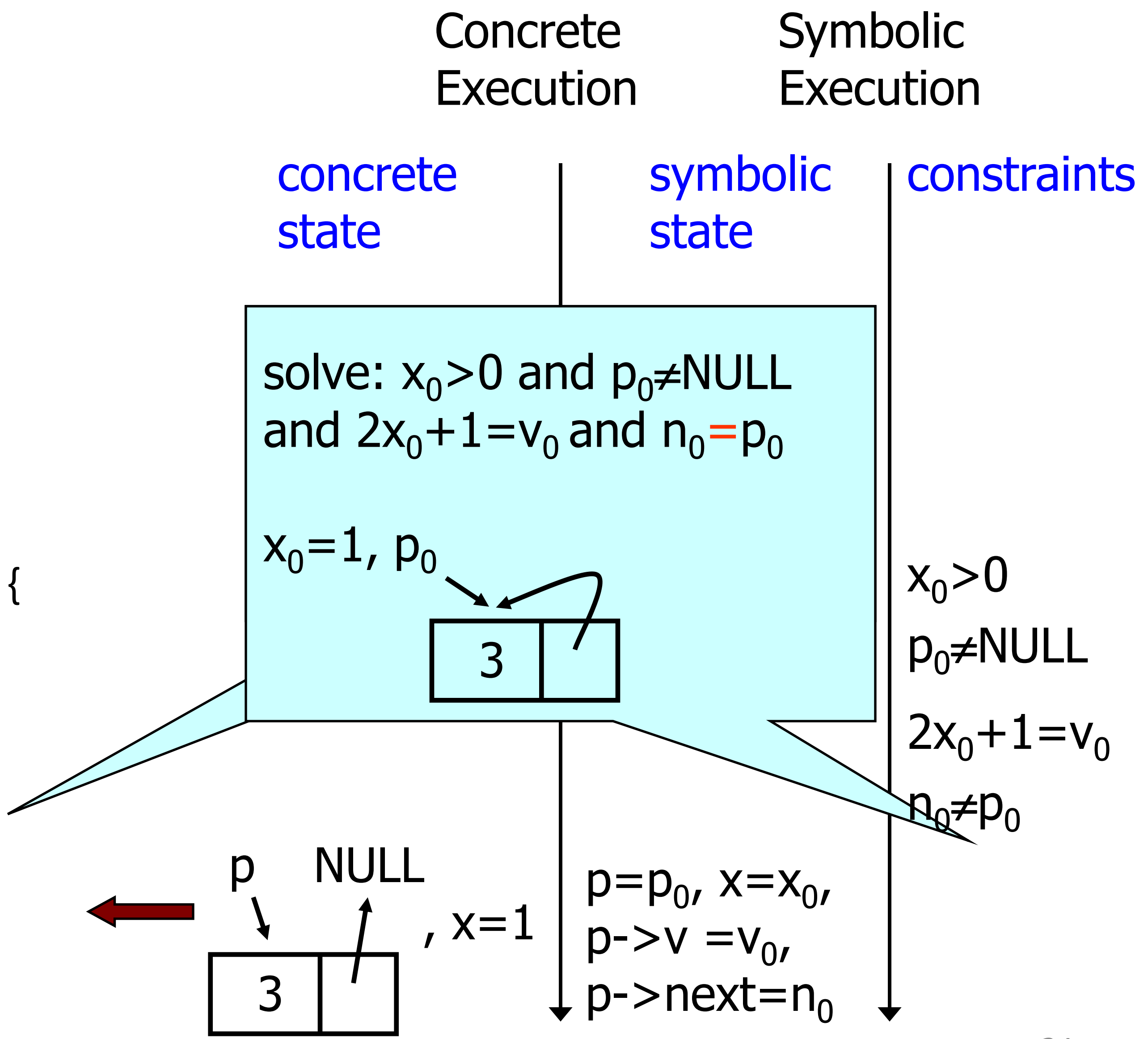
symbolic state:

$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

constraints:

$x_0>0$
$p_0 \neq NULL$
$2x_0+1=v_0$
$n_0 \neq p_0$

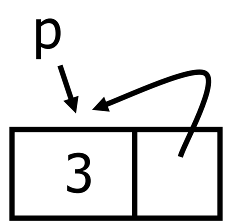
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

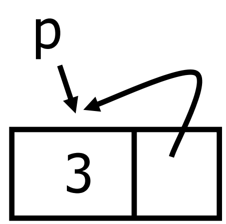
concrete state	symbolic state	constraints
 , x=1	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next = n_0$	

Concrete Execution

Symbolic Execution

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

concrete state



, x=1

symbolic state

$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

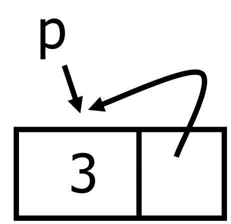
constraints

$x_0>0$

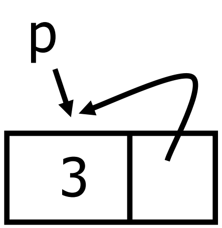
Concrete Execution

Symbolic Execution

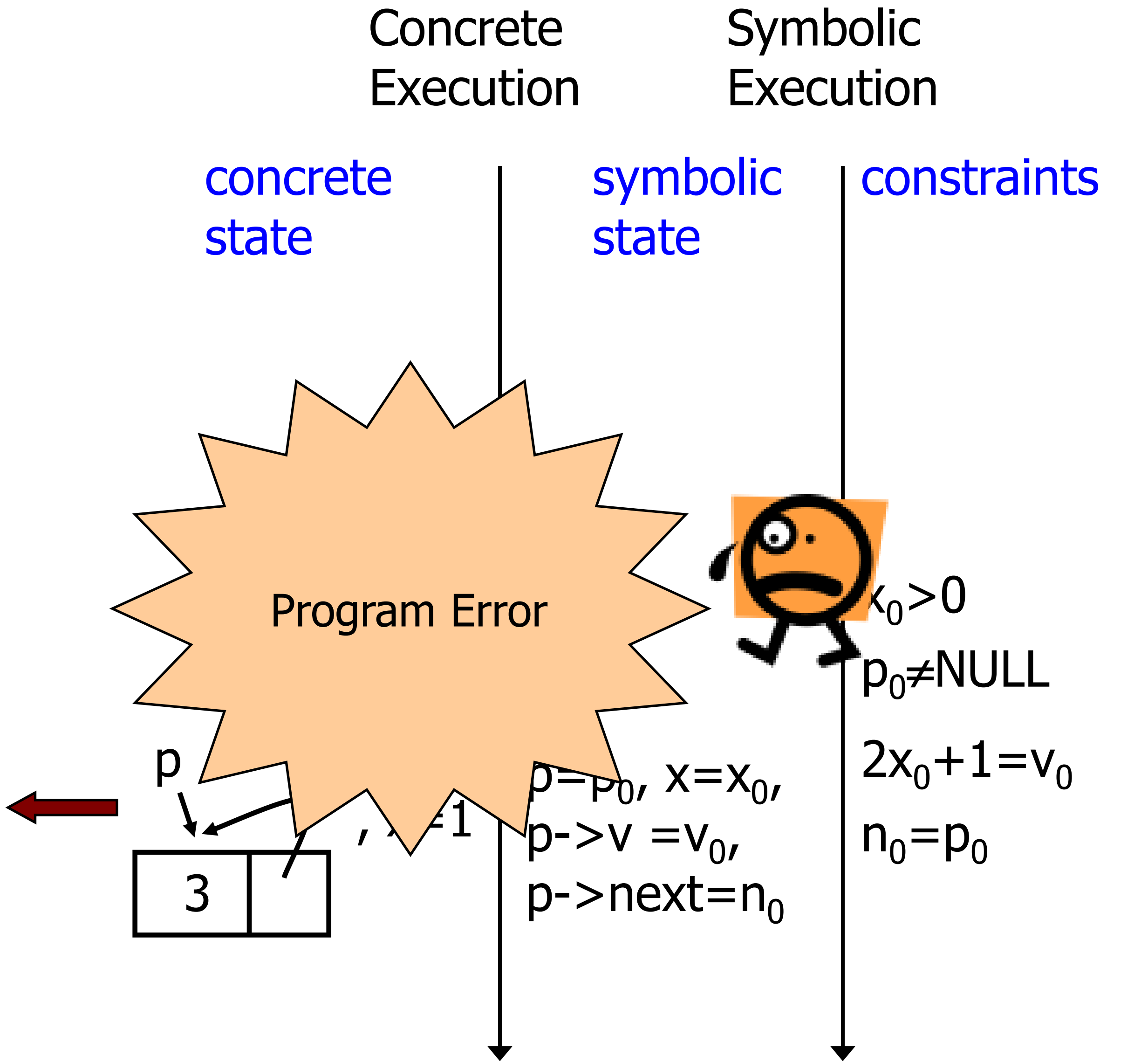
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

concrete state	symbolic state	constraints
 , x=1	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$, $p_0 \neq NULL$

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

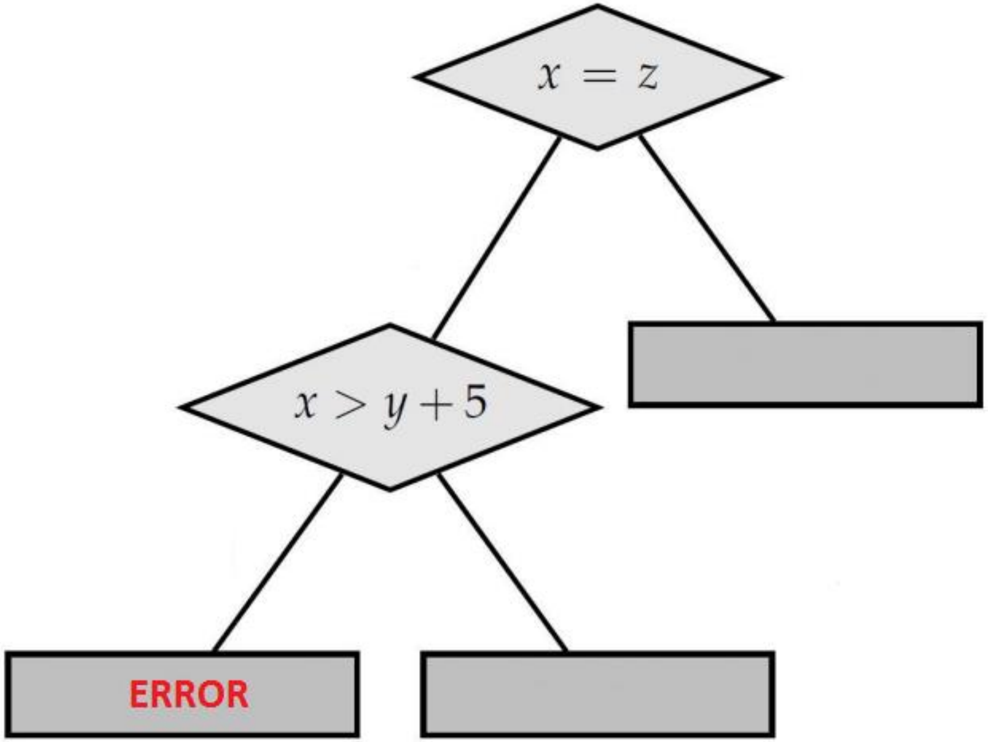
Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints
 , x=1		$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$ $p_0 \neq NULL$ $2x_0+1=v_0$

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



Exercise (credits: Daniel Paqué)

```
foo(int x,int y){
    z = 2*y;
    if (x == z){
        if (x > y + 5){
            //some error
        }
    }
}
```



Exercise (credits: Daniel Paqué)

```
foo(int x,int y){
    z = 2*y;
    if (x == z){
        if (x > y + 5){
            //some error
        }
    }
}
```

Iteration	Concrete Inputs		Accumulated Constraints to Solve
	x	y	
1	29	4	
2			
3			

Exercise (credits: Daniel Paqué)

Iteration	Concrete Inputs		Accumulated Constraints to Solve
	x	y	
1	29	4	x == 2y
2	8	4	(x == 2y) & (x > y + 5)
3	12	6	-

```
foo(int x,int y){
    z = 2*y;
    if (x == z){
        if (x > y + 5){
            //some error
        }
    }
}
```

Concolic Testing Tools

- KLEE for C
- CUTE for C
- DART for C
- Jcute for Java

  - Paper: CUTE: A Concolic Unit Testing Engine for C, Koushik Sen, Darko Marinov, and Gul Agha. In CAV, volume 4144 of Lecture Notes in Computer Science, 419–423. Springer, 2006.

Random Test Case Generation

- Scalable
- Easy to implement
- Can provide high number of test case

- Not systematic
- Can lead to useless test cases

Randoop
(slides by C. Pacheco, M. Ernst, S. Lahiri, T. Ball)

- **Rand**om Tester for **O**bject-**O**riented **P**rograms
- Randomized generation of test inputs
- Guided by feedback from the execution of previous inputs
- Automated generation of unit tests for Java and .NET

Random testing: pitfalls

1. Useful test

```
Set s = new HashSet();
s.add("hi");
assertTrue(s.equals(s));
```

2. Redundant test

```
Set s = new HashSet();
s.add("hi");
s.isEmpty();
assertTrue(s.equals(s));
```

3. Useful test

```
Date d = new Date(2006, 2, 14);
assertTrue(d.equals(d));
```

4. Illegal test

```
Date d = new Date(2006, 2, 14);
d.setMonth(-1); // pre: argument >= 0
assertTrue(d.equals(d));
```

5. Illegal test

```
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
d.setDay(5);
assertTrue(d.equals(d));
```

do not output

do not even create

Feedback-directed random test generation

- Build test inputs **incrementally**
  - New test inputs extend previous ones
  - In our context, a test input is a method sequence
- As soon as a test input is created, execute it
- Use execution results to **guide** generation
  - away from redundant or illegal method sequences
  - towards sequences that create new object states

Technique input/output

- Input: classes under test, time limit, set of contracts;
  - Method contracts (e.g. “o.hashCode() throws no exception”)
  - Object invariants (e.g. “o.equals(o) == true”)
- Output: contract-violating test cases

no contracts violated up to last method call

```
HashMap h = new HashMap();
Collection c = h.values();
Object[] a = c.toArray();
LinkedList l = new LinkedList();
l.addFirst(a);
TreeSet t = new TreeSet(l);
Set u = Collections.unmodifiableSet(t);
assertTrue(u.equals(u));
```

fails when executed

Technique

1. Seed components

   components = { `int i = 0;` `boolean b = false;` ... }

2. Do until time limit expires:

   a. Create a new sequence

      i. Randomly pick a method call $m(T_1 \ldots T_k)/T_{ret}$

      ii. For each input parameter of type T_i, randomly pick a sequence S_i from the components that constructs an object v_i of type Ti

      iii. Create new sequence S_{new} = S_1; ... ; S_k ; T_{ret} v_{new} = m(v1...vk);

      iv. if S_{new} was previously created (lexically), go to i

   b. Classify the new sequence S_{new}

      a. May discard, output as test case, or add to components

Example from the Paper

sequence s_1	sequence s_2	sequence s_3
`B b1 = new B(0);`	`B b2 = new B(0);`	`A a1 = new A();` `B b3 = a1.m1(a1);`

seqs	*vals*	*extend*(m2, *seqs*, *vals*)

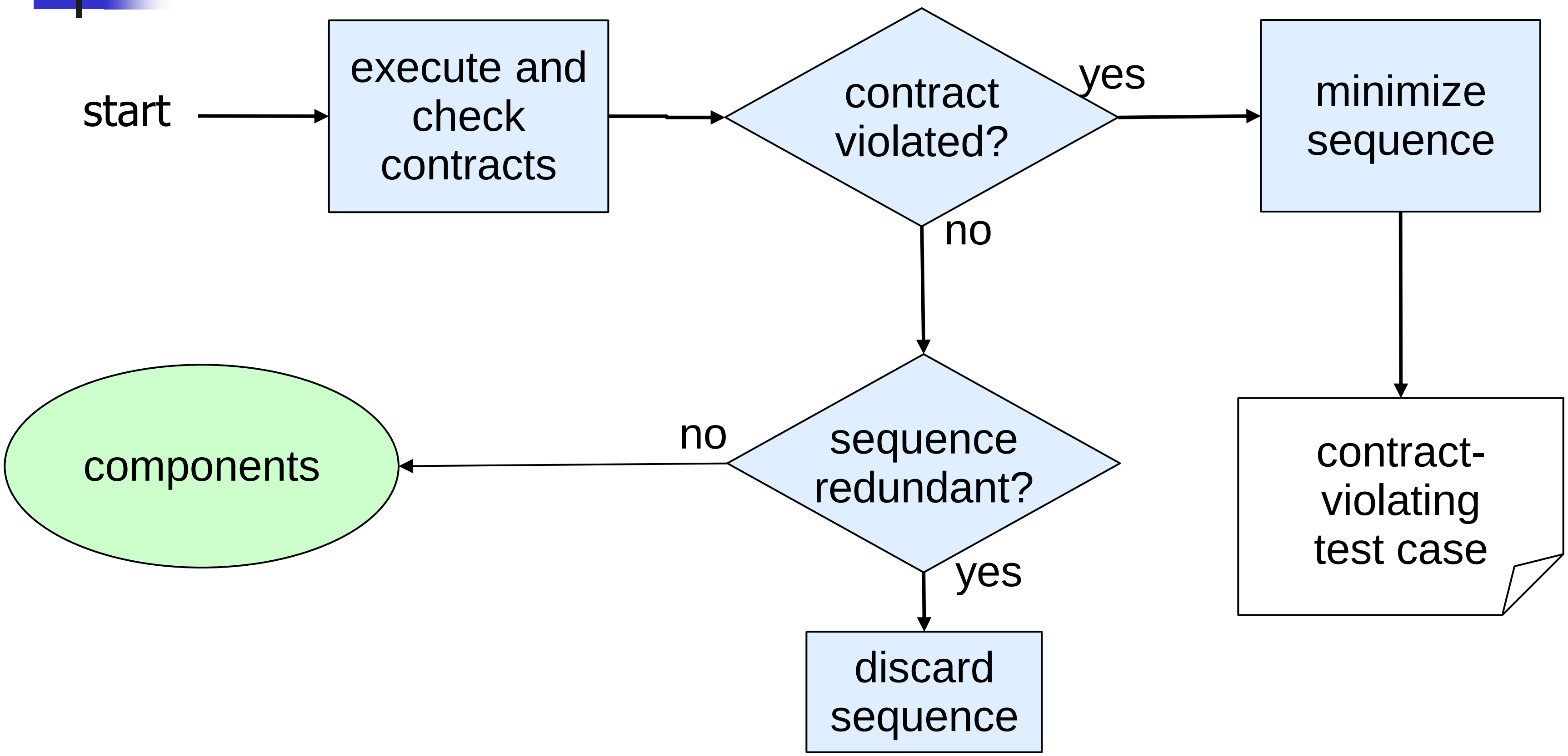
$\langle s_1, s_3 \rangle$	$\langle\ s_1.1, s_1.1, s_3.1\ \rangle$
	(i.e.: `b1`, `b1`, `a1`)
		`B b1 = new B(0);`
		`A a1 = new A();`
		`B b3 = a1.m1(a1);`
		`b1.m2(b1,a1);`

$\langle s_3, s_1 \rangle$	$\langle\ s_1.1, s_1.1, s_3.1\ \rangle$
	(i.e.: `b1`, `b1`, `a1`)
		`A a1 = new A();`
		`B b3 = a1.m1(a1);`
		`B b1 = new B(0);`
		`b1.m2(b1,a1);`

$\langle s_1, s_2 \rangle$	$\langle\ s_1.1, s_2.1,$ `null` $\rangle$
	(i.e.: `b1`, `b2`, `null`)
		`B b1 = new B(0);`
		`B b2 = new B(0);`
		`b1.m2(b2,null);`

```
public class A {
  public A() {...}
  public B m1(A a1) {...}
}

public class B {
  public B(int i) {...}
  public void m2(B b, A a) {...}
}
```

Classifying a sequence



Redundant sequences

- During generation, maintain a set of all objects created
- A sequence is redundant if all the objects created during its execution are members of the above set (using `equals` to compare)
- Could also use more sophisticated state equivalence methods
  - E.g. heap canonicalization

Exercise 1

S_1	Integer i1 = new Integer(1); List list1 = new LinkedList(); list1.add(i1);
S_2	Integer i2 = new Integer(2); List list2 = null; list2.add(i2); **illegal**
S_3	Integer i3 = new Integer(3);

Sequence	Useful/ Redundant/ Illegal	Would be generated by Randoop (Yes/No)?

Integer i2 = new Integer(2);
List list2 = null;
list2.add(i2);
Integer i3 = new Integer(3);
list2.add(i3);		

Integer i1 = new Integer(1);
List list1 = new LinkedList();
list1.add(i1);
list1.size();		

Integer i3 = new Integer(3);
Integer i1 = new Integer(1);
List list1 = new LinkedList();
list1.add(i1);
list1.contains(i3);		

Exercise 1

S_1	Integer i1 = new Integer(1); List list1 = new LinkedList(); list1.add(i1);
S_2	Integer i2 = new Integer(2); List list2 = null; list2.add(i2); **illegal**
S_3	Integer i3 = new Integer(3);

Sequence	Useful/ Redundant/ Illegal	Would be generated by Randoop (Yes/No)?

Integer i2 = new Integer(2);
List list2 = null;
list2.add(i2);
Integer i3 = new Integer(3);
list2.add(i3);
	Illegal	No

Integer i1 = new Integer(1);
List list1 = new LinkedList();
list1.add(i1);
list1.size();
	Redundant	Yes

Integer i3 = new Integer(3);
Integer i1 = new Integer(1);
List list1 = new LinkedList();
list1.add(i1);
list1.contains(i3);
	Redundant	Yes

Exercise 2

S_1	Integer i1 = new Integer(1); List list1 = new LinkedList(); list1.add(i1);
S_2	Integer i2 = new Integer(2); List list2 = null; list2.add(i2);
S_3	Integer i3 = new Integer(3);

illegal

Sequence	Useful/ Redundant/ Illegal	Would be generated by Randoop (Yes/No)?
Integer i1 = new Integer(1); List list1 = new LinkedList(); list1.add(i1); Integer i3 = new Integer(3); list1.add(i1);		
List list1 = new LinkedList(); list1.size();		
List list2 = null; list2.size();		

Integer i1 = new Integer(1);
List list1 = new LinkedList();
list1.add(i1);
Integer i4 = new Integer(4);
list1.add(i4);		

Exercise 2

S_1	Integer i1 = new Integer(1); List list1 = new LinkedList(); list1.add(i1);
S_2	Integer i2 = new Integer(2); List list2 = null; list2.add(i2); **illegal**
S_3	Integer i3 = new Integer(3);

Sequence	Useful/ Redundant/ Illegal	Would be generated by Randoop (Yes/No)?
Integer i1 = new Integer(1); List list1 = new LinkedList(); list1.add(i1); Integer i3 = new Integer(3); list1.add(i1);	Useful	Yes
List list1 = new LinkedList(); list1.size();	Redundant	No
List list2 = null; list2.size();	Illegal	No

Integer i1 = new Integer(1);
List list1 = new LinkedList();
list1.add(i1);
Integer i4 = new Integer(4);
list1.add(i4);
	Useful	No

Randoop outputs oracles

- Oracle for contract-violating test case:

```
Object o = new Object();
LinkedList l = new LinkedList();
l.addFirst(o);
TreeSet t = new TreeSet(l);
Set u = Collections.unmodifiableSet(t);
assertTrue(u.equals(u));            // expected to fail
```

- Oracle for normal-behavior test case:

```
Object o = new Object();
LinkedList l = new LinkedList();
l.addFirst(o);
l.add(o);
assertEquals(2, l.size());          // expected to pass
assertEquals(false, l.isEmpty()); // expected to pass
```

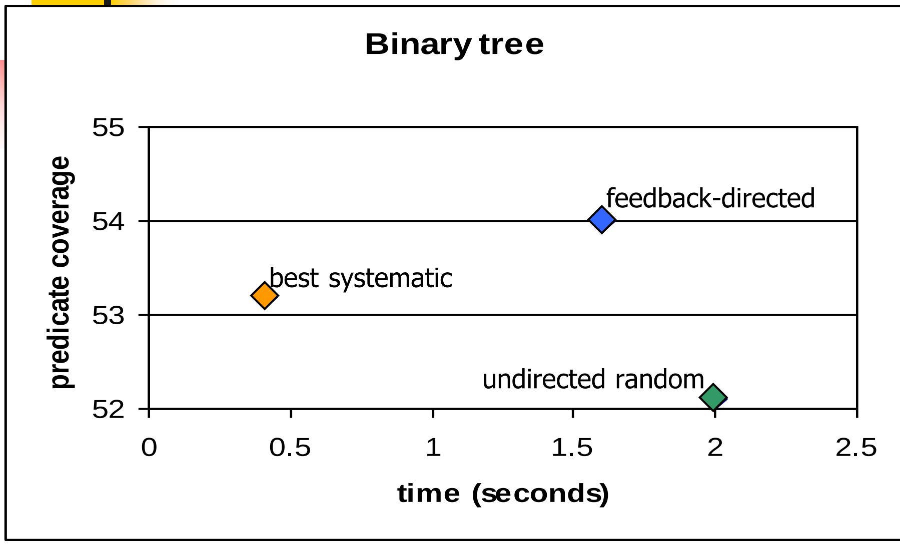
Randoop uses **observer methods** to capture object state

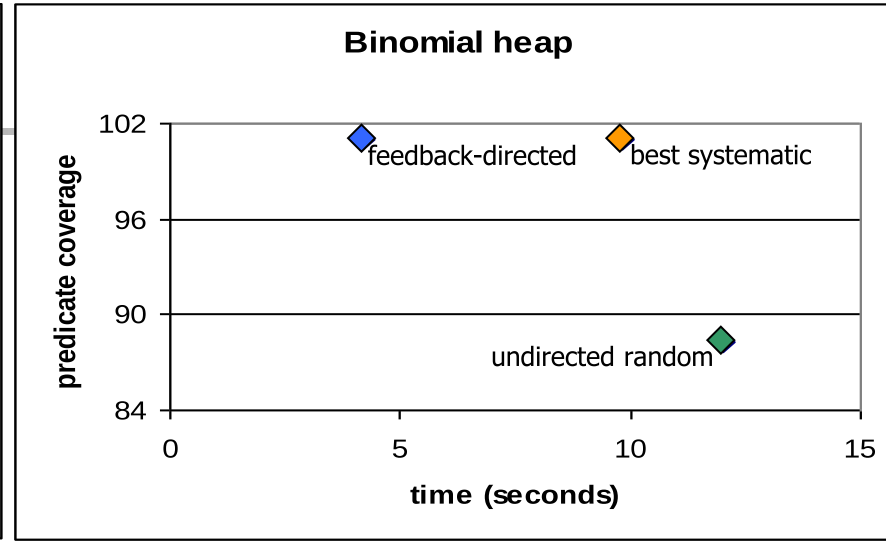
Evaluation of Randoop I

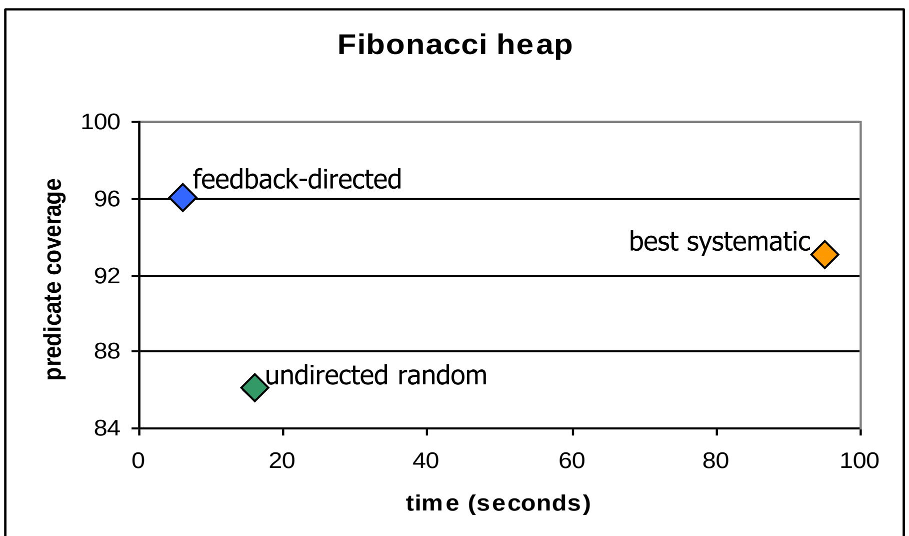
- In terms of **coverage** achieved

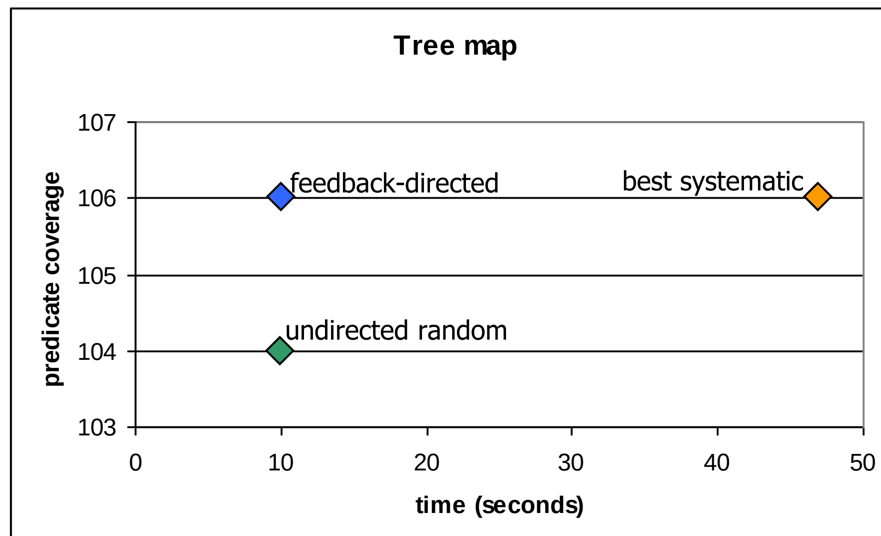
data structure	time (s)	branch cov.
Bounded stack (30 LOC)	1	100%
Unbounded stack (59 LOC)	1	100%
BS Tree (91 LOC)	1	96%
Binomial heap (309 LOC)	1	84%
Linked list (253 LOC)	1	100%
Tree map (370 LOC)	1	81%
Heap array (71 LOC)	1	100%

Predicate coverage









Evaluation of Randoop II

- In terms of **error detection** performance

Subjects:	**LOC**	**Classes**
JDK (2 libraries) (java.util, javax.xml)	53K	272
Apache commons (5 libraries) (logging, primitives, chain jelly, math, collections)	114K	974
.Net framework (5 libraries)	582K	3330

Methodology

- Ran Randoop on each library
  - Used default time limit (2 minutes)

- Contracts:
  - o.equals(o)==true
  - o.equals(o) throws no exception
  - o.hashCode() throws no exception
  - o.toString() throw no exception
  - No null inputs and:
    - Java: No NPEs
    - .NET: No NPEs, out-of-bounds, of illegal state exceptions

Results

	test cases output	error-revealing tests cases	distinct errors
JDK	32	29	8
Apache commons	187	29	6
.Net framework	192	192	192
Total	***411***	***250***	***206***

Errors found: examples

- JDK Collections classes have 4 methods that create objects violating o.equals(o) contract
- Javax.xml creates objects that cause hashCode and toString to crash, even though objects are well-formed XML constructs
- Apache libraries have constructors that leave fields unset, leading to NPE on calls of equals, hashCode and toString (this only counts as one bug)
- Many Apache classes require a call of an *init()* method before object is legal—led to many false positives
- .Net framework has at least 175 methods that throw an exception forbidden by the library specification (NPE, out-of-bounds, of illegal state exception)
- .Net framework has 8 methods that violate o.equals(o)
- .Net framework loops forever on a legal but unexpected input

```
Object o = new Object();
LinkedList l = new LinkedList();
l.addFirst(o);
l.add(o);
assertEquals(2, l.size());          // expected to pass
assertEquals(false, l.isEmpty()); // expected to pass
```

Regression testing

- Randoop can create *regression oracles*
- Generated test cases using JDK 1.5
  - Randoop generated 41K regression test cases
- Ran resulting test cases on
  - JDK 1.6 Beta
    - 25 test cases failed
  - Sun's implementation of the JDK
    - 73 test cases failed
  - Failing test cases pointed to 12 distinct errors
  - These errors were not found by the extensive compliance test suite that Sun provides to JDK developers

Conclusion

- Feedback-directed random test generation
  - Finds errors in widely-used, well-tested libraries
  - Can outperform systematic test generation
  - Can outperform undirected test generation
- Randoop:
  - Easy to use—just point at a set of classes
  - Has real clients: used by product groups at Microsoft
- A mid-point in the systematic-random space of input generation techniques

https://randoop.github.io/randoop/manual/index.html