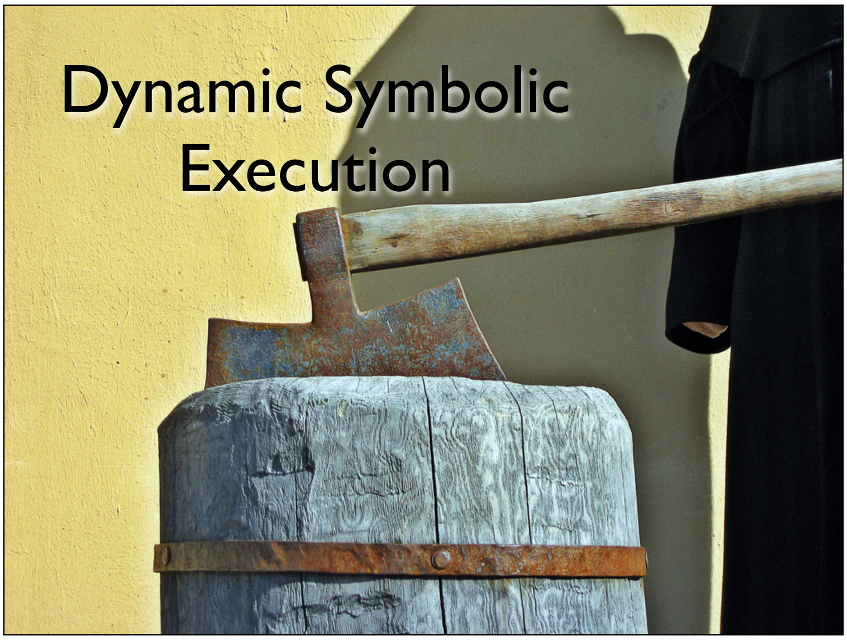
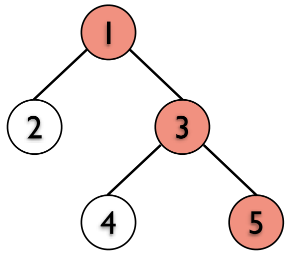


Dynamic Symbolic Execution

- Symbolic execution of a concrete execution
- Also called concolic execution (concrete +symbolic)
- By using input values, feasible paths only are (automatically) selected
- Implemented by instrumenting each statement of P

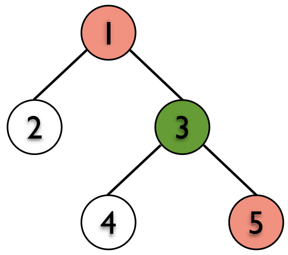
Dynamic symbolic execution combines symbolic execution with concrete execution: The program is instrumented for symbolic execution and then executed with concrete values. The instrumentation collects path conditions and symbolic states along the concrete execution.

Dynamic Symbolic Execution



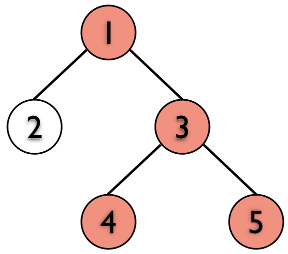
Generate a random input and execute the corresponding feasible path

Dynamic Symbolic Execution

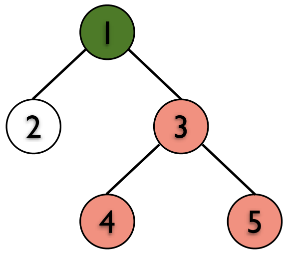


Try to solve the CS where the last constraint is refuted

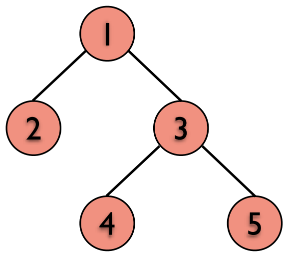
Dynamic Symbolic Execution



Dynamic Symbolic Execution

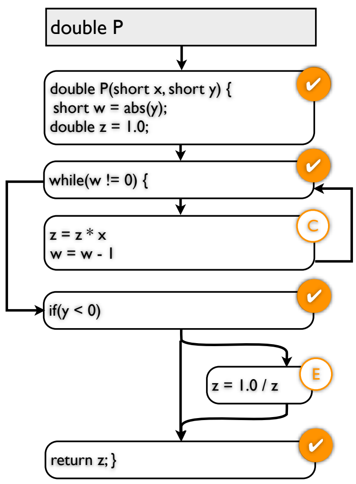


Dynamic Symbolic Execution



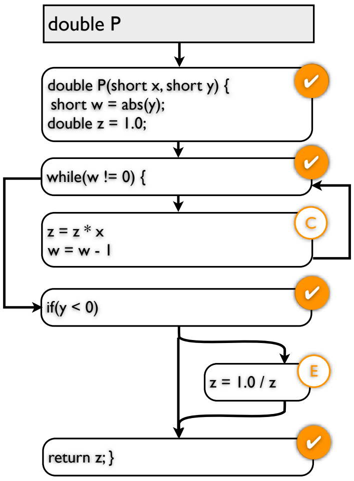
Generate a random input and execute the corresponding feasible path

(0,0)



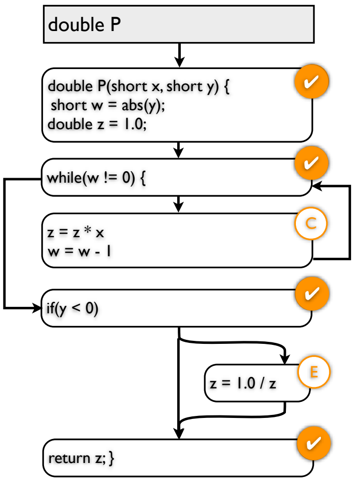
While executing, collect constraints along path taken

(0,0)   abs(y) = 0
         !(y < 0)



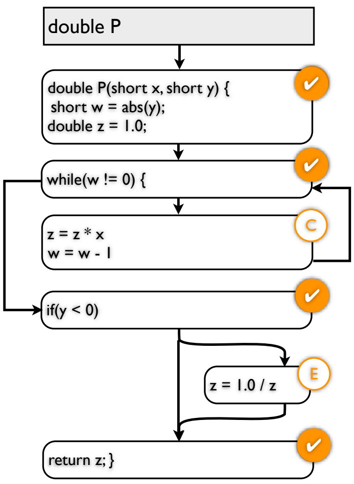
Try to solve the CS where the last constraint is refuted

(0,0)   abs(y) = 0
         (y < 0)



Try to solve the CS where the last constraint is refuted

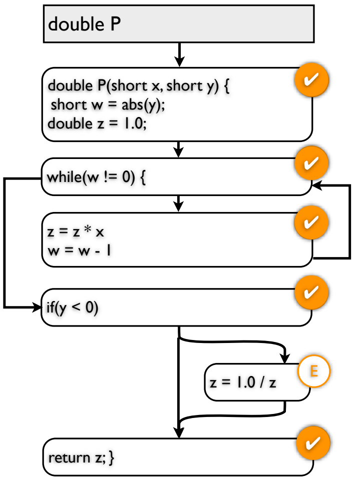
(0,0)   abs(y) != 0



Try to solve the CS where the last constraint is refuted

(0,1)

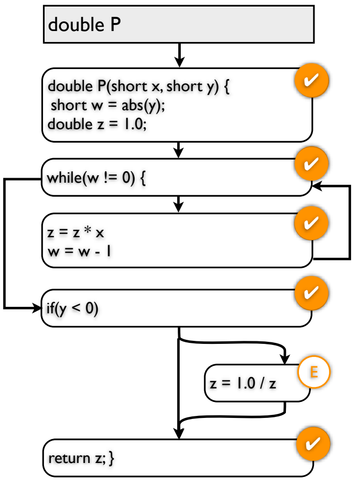
abs(y) != 0
abs(y) - 1 = 0
!(y < 0)



Try to solve the CS where the last constraint is refuted

(0,1)

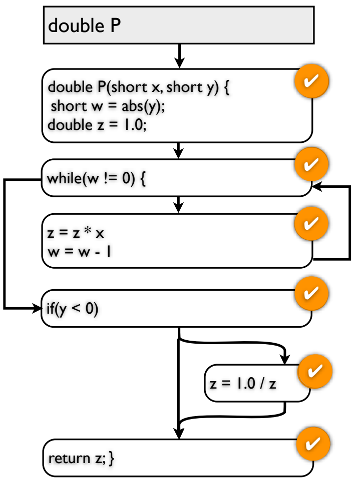
abs(y) != 0
abs(y) - 1 = 0
(y < 0)

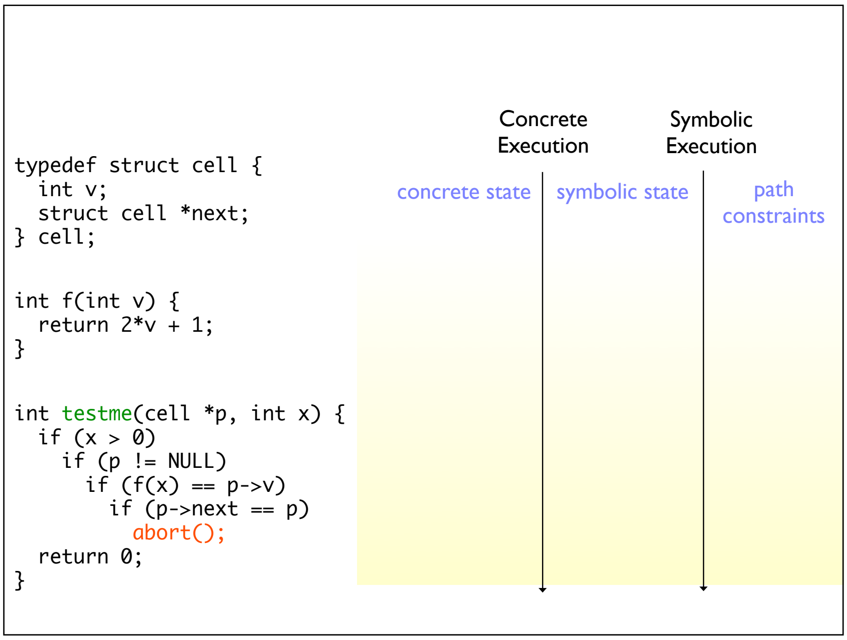


Try to solve the CS where the last constraint is refuted

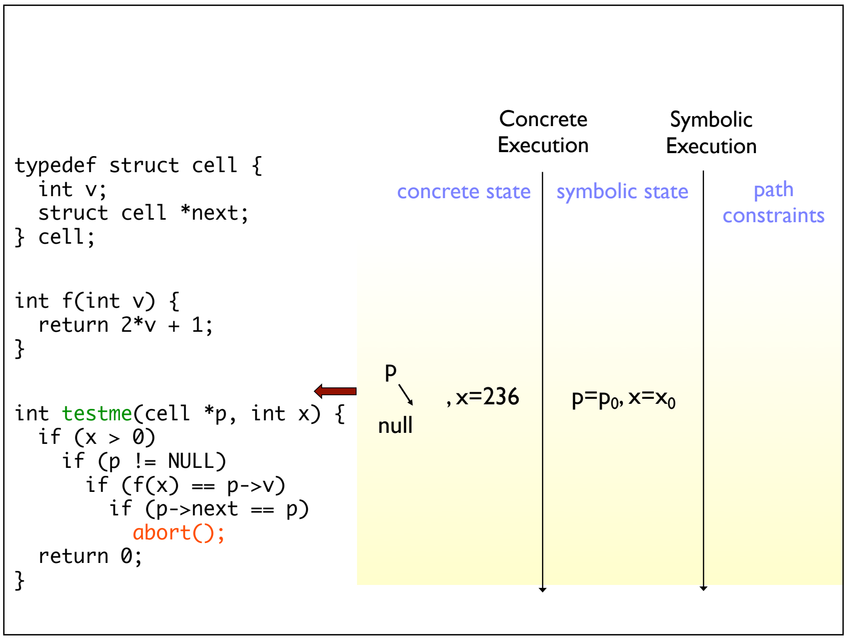
(0,-1)

abs(y) != 0
abs(y) - 1 = 0
(y < 0)

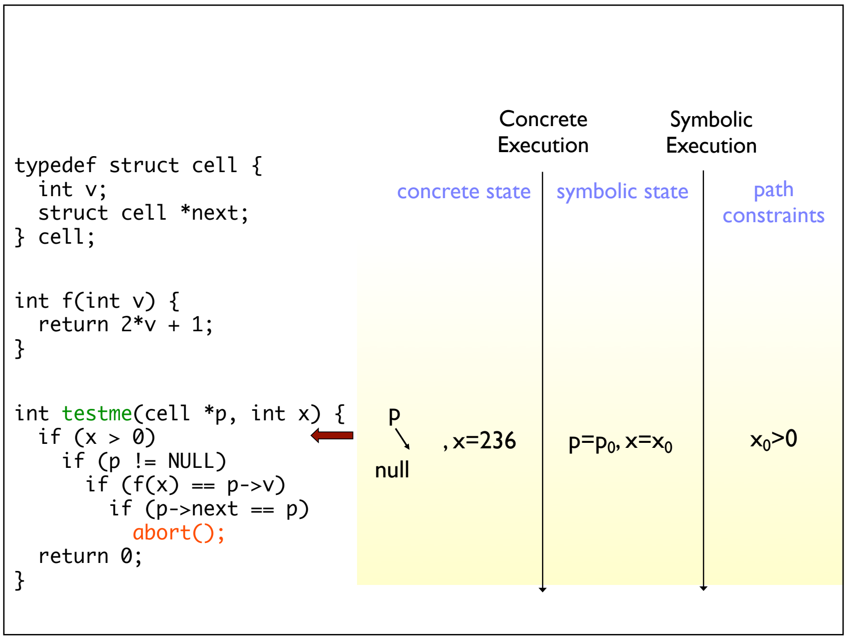




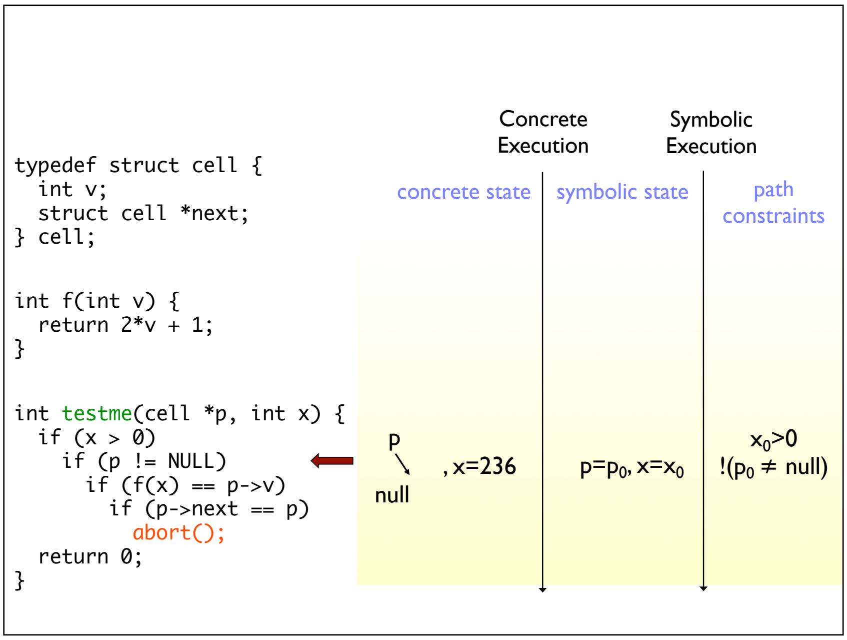
This example illustrates dynamic symbolic execution as done in the tool Cute (http://osl.cs.uiuc.edu/~ksen/cute/)
Assume we want to generate test data that reaches the abort statement.



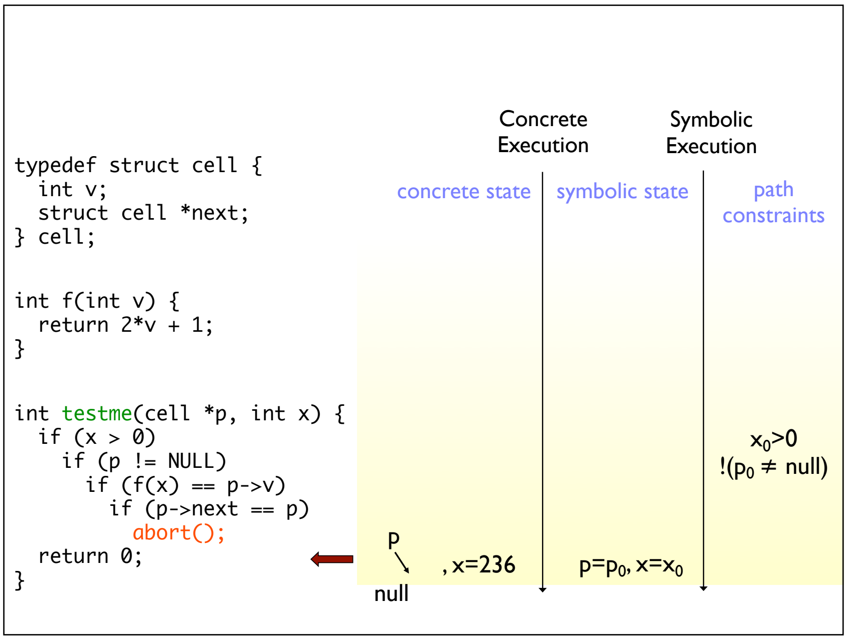
Before the first run: Pointers are set to null, ints assigned randomly; the symbolic state is initialized.



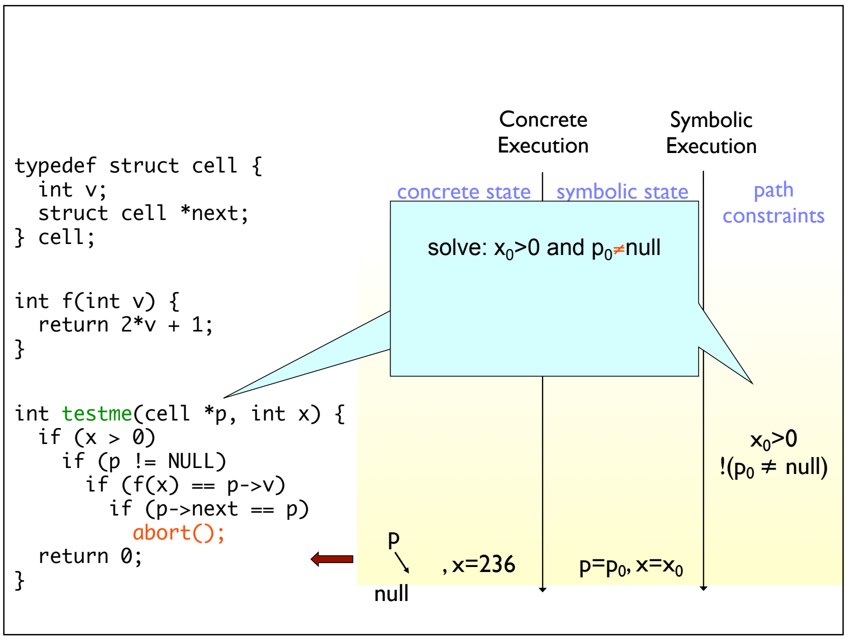
A new path condition is added in the first line of the function; the path condition is based on the symbolic state. If the concrete execution takes the true branch, then the path condition is added directly, else its negation is added.



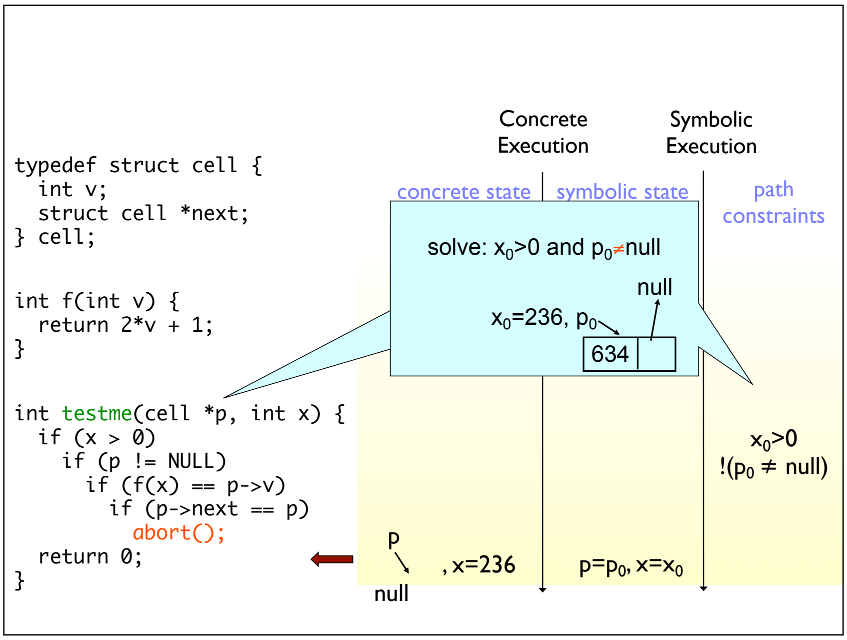
The second branch condition is false, so we add the negation to the path conditions.



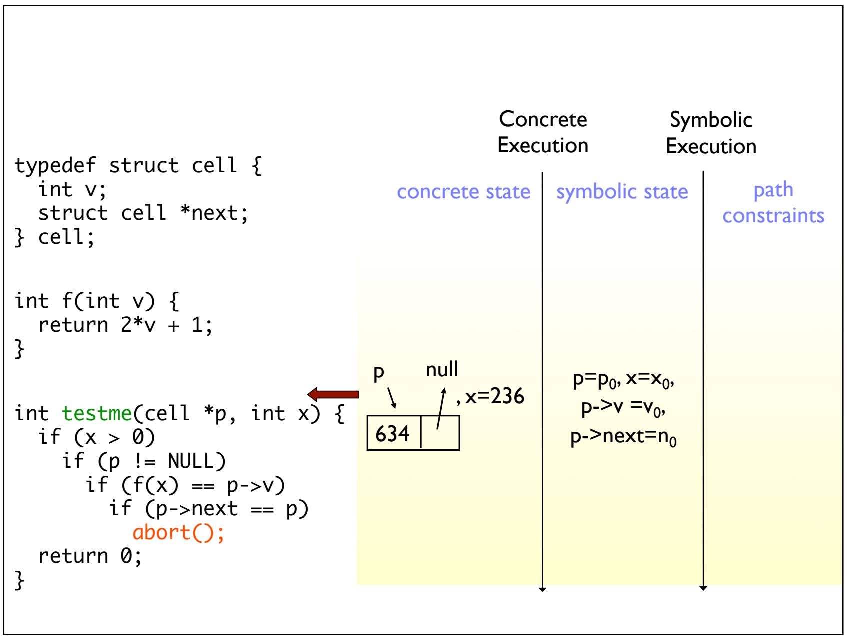
The else branch leads to the return statement.

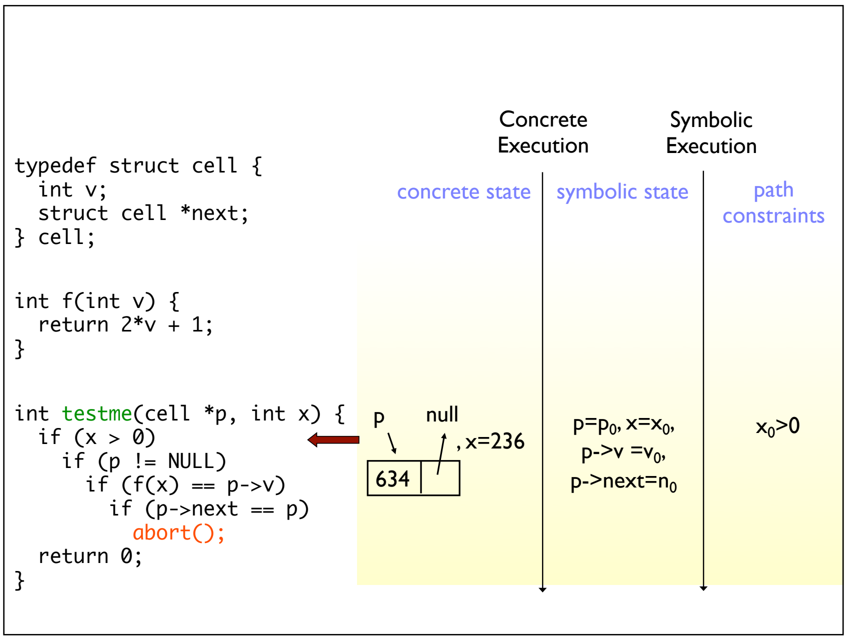


To explore a new path, we negate the last branch condition...

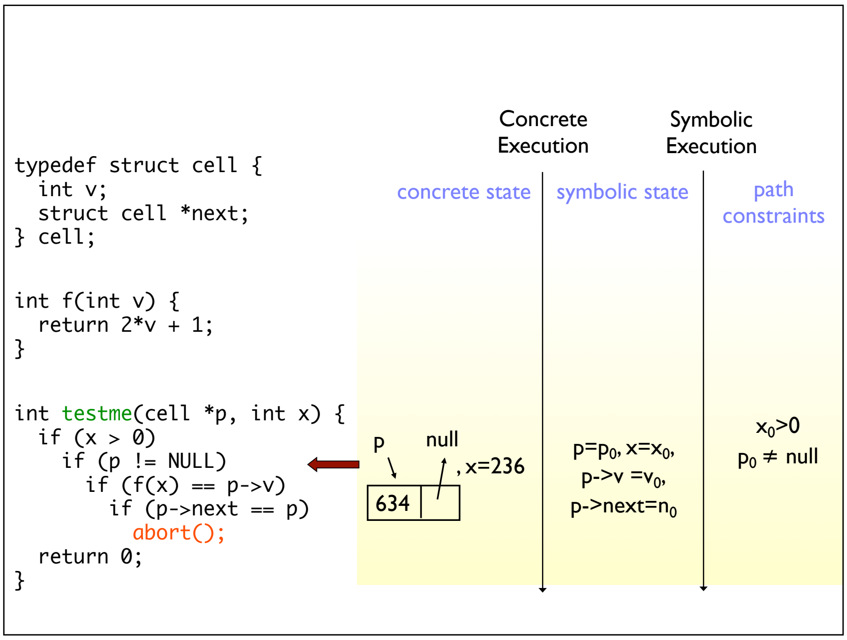


...and use a constraint solver to find a new solution. In this case we need to construct a new cell, for which the fields are initialized (pointers to null, ints randomly).

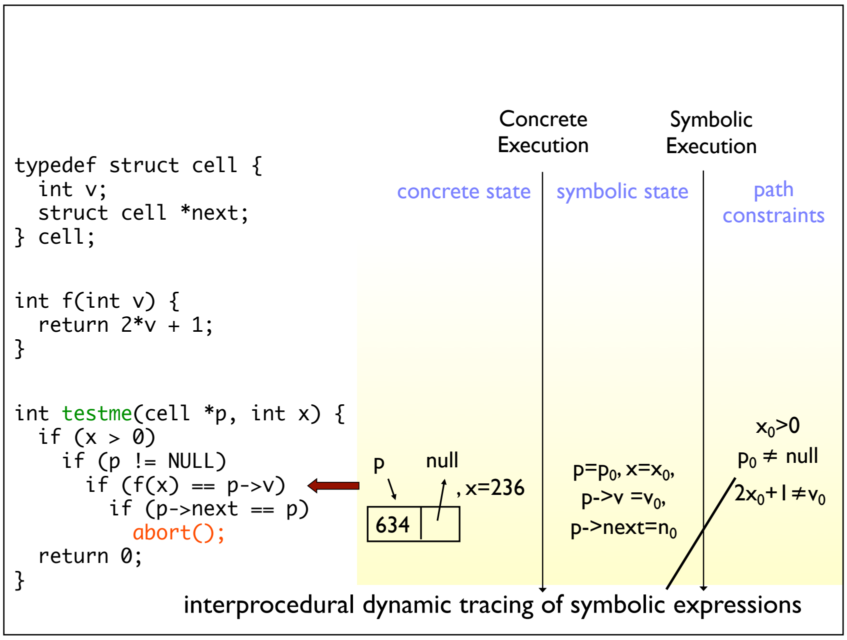




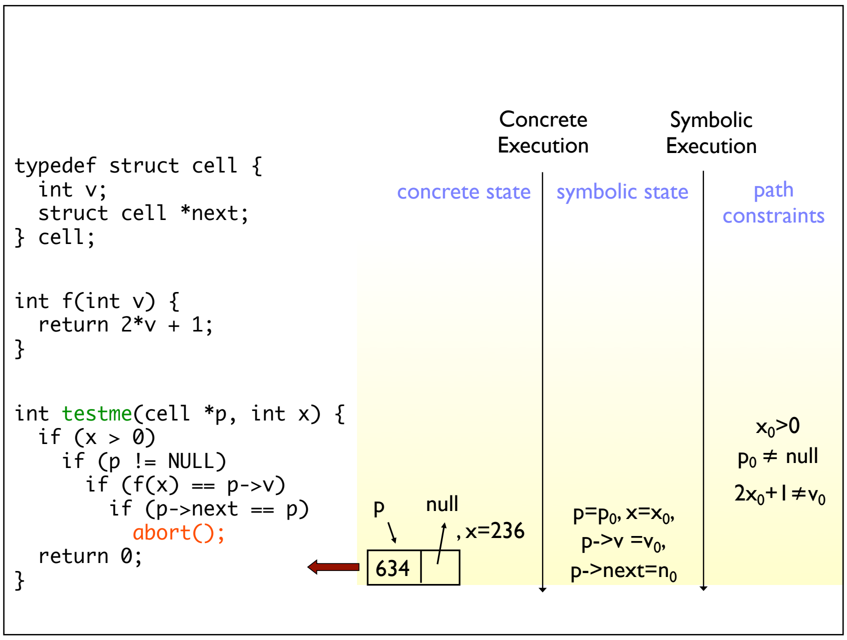
The first condition still holds with the new inputs (path condition is updated accordingly for this run).

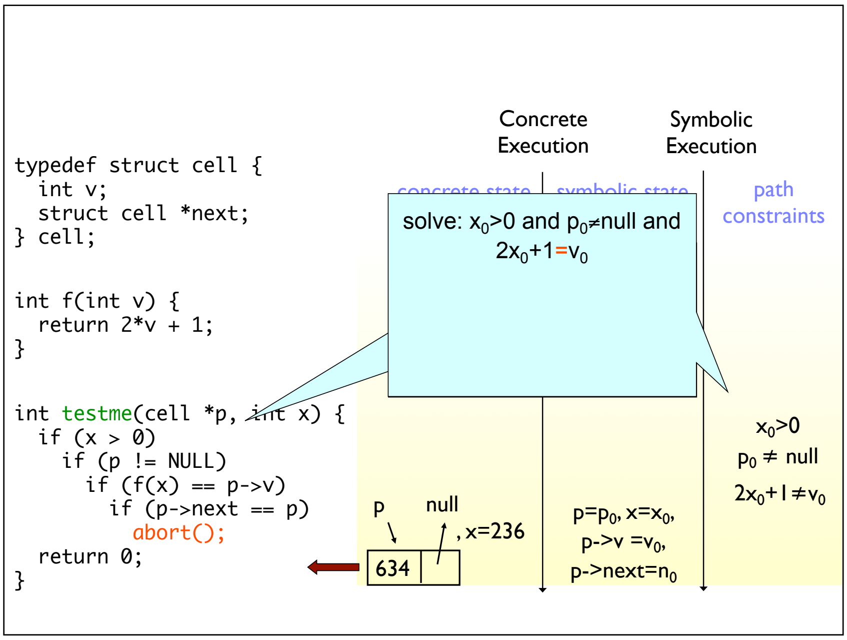


This time, the second branch condition also is true, and we update the path condition accordingly.

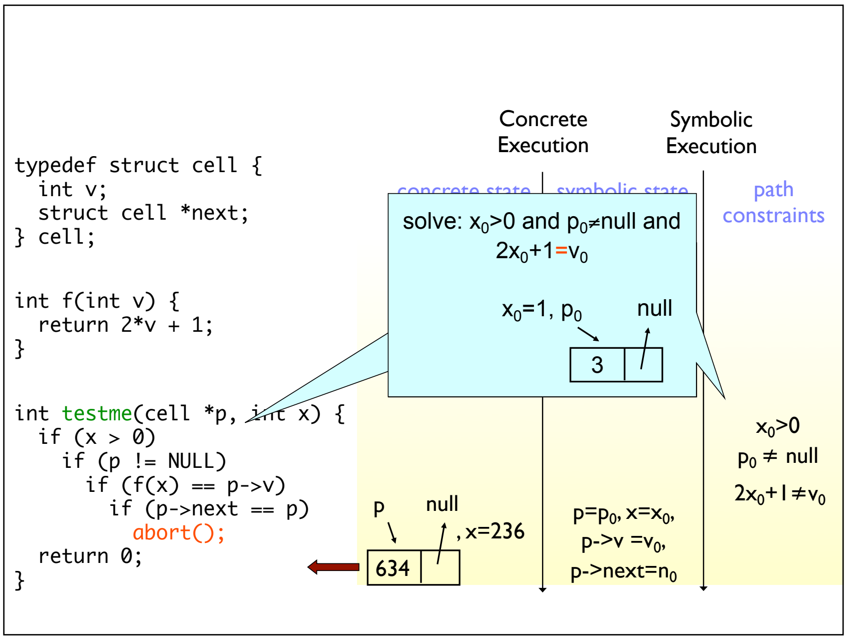


In the next line, the symbolic state is updated by the function call to f. The branch condition itself evaluates to false on the concrete run (p->v is 634, 2x+1 is 473), so we add the negation of the condition to the path constraints (note that v0 is short for p0->v).

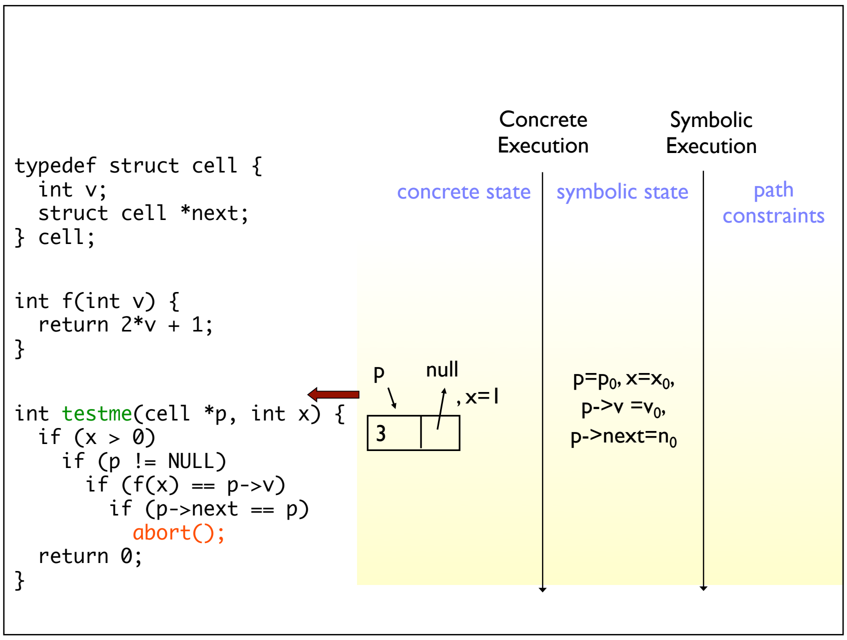




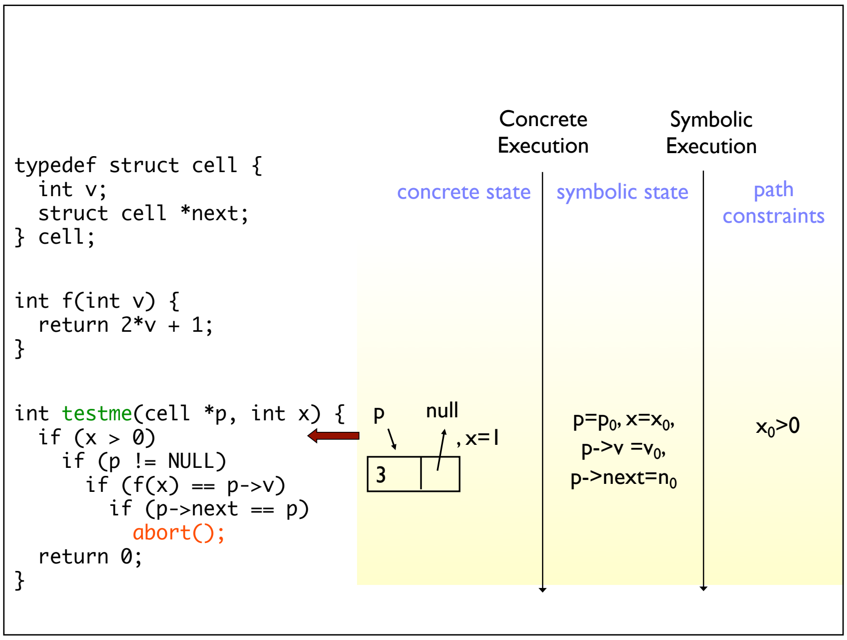
Again we negate the last branch condition...

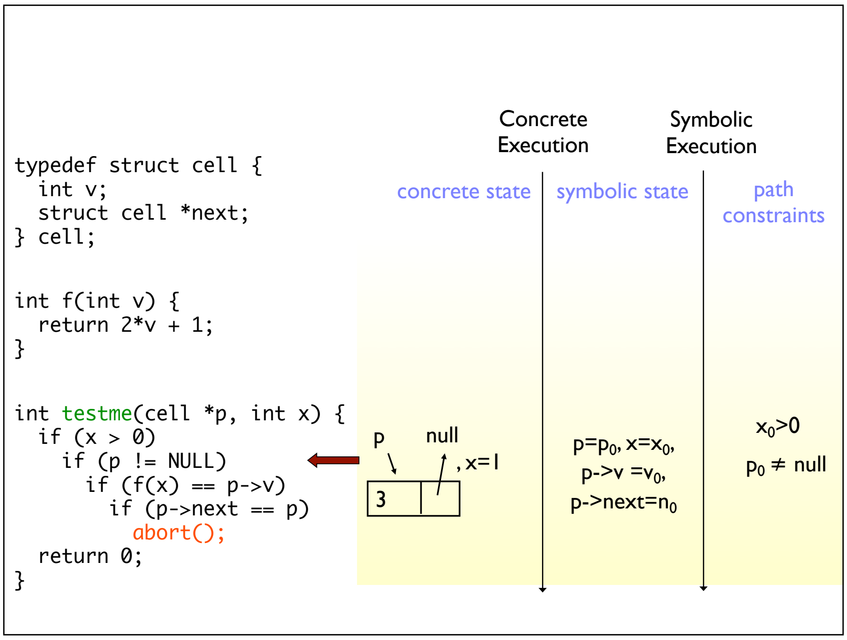


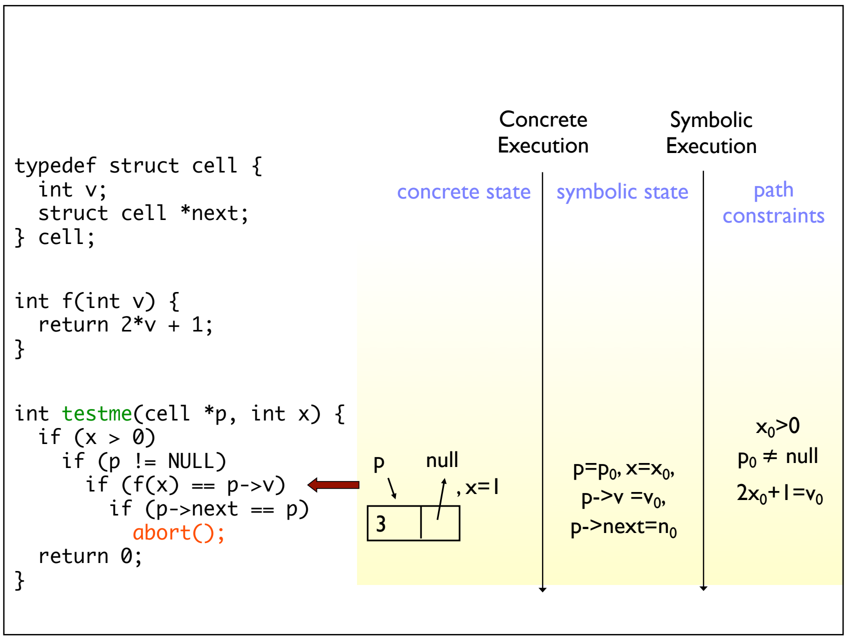
And solve it with a constraint solver, which tells us we need to set x to 1 and p->v to 3.



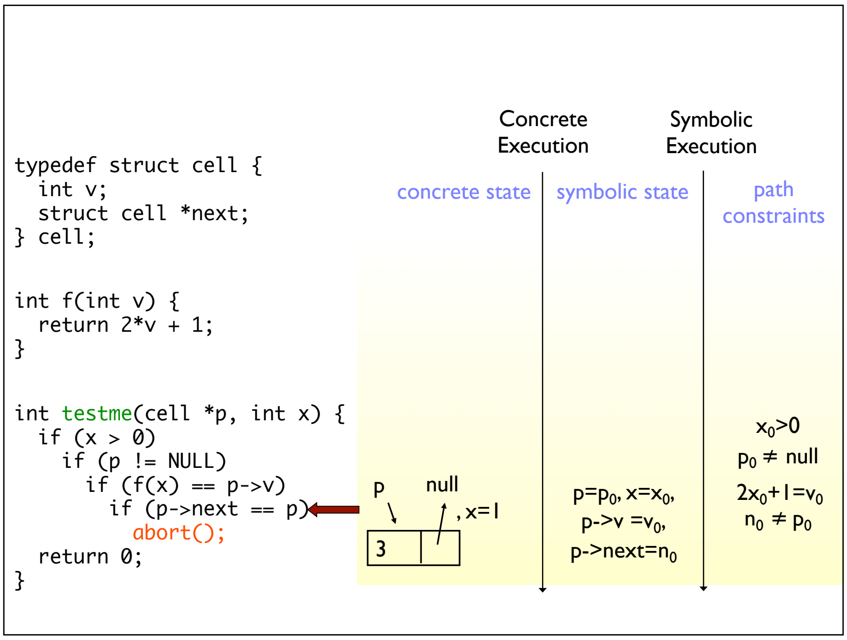
Test case is run with new values.



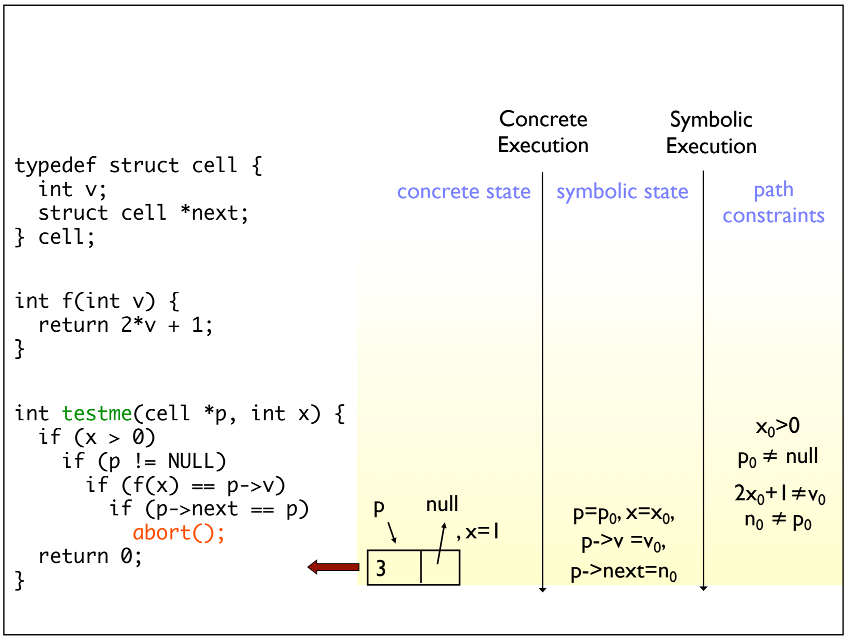


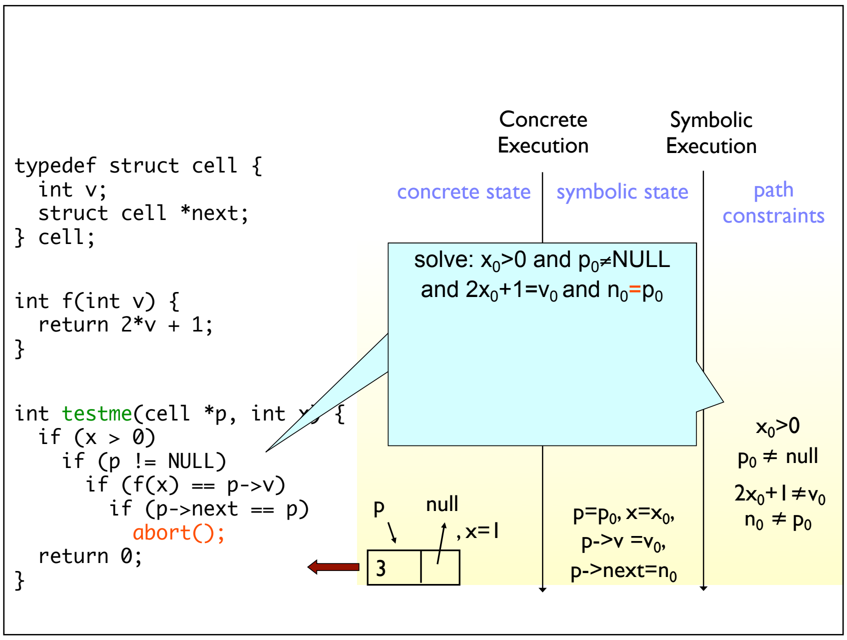


This time, the branch condition evaluates to true.

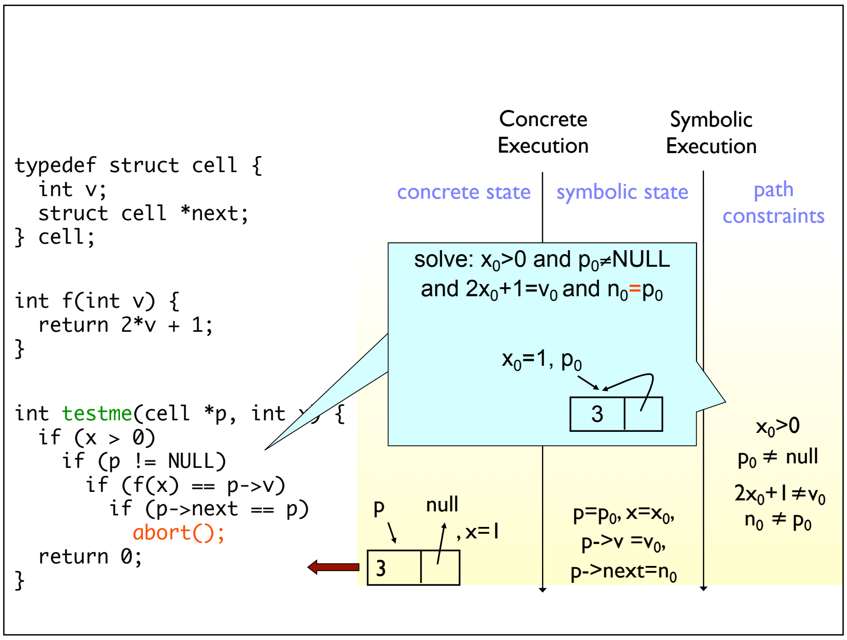


The last branch is false, as p->next is null, so we add the negated branch condition to the constraints.

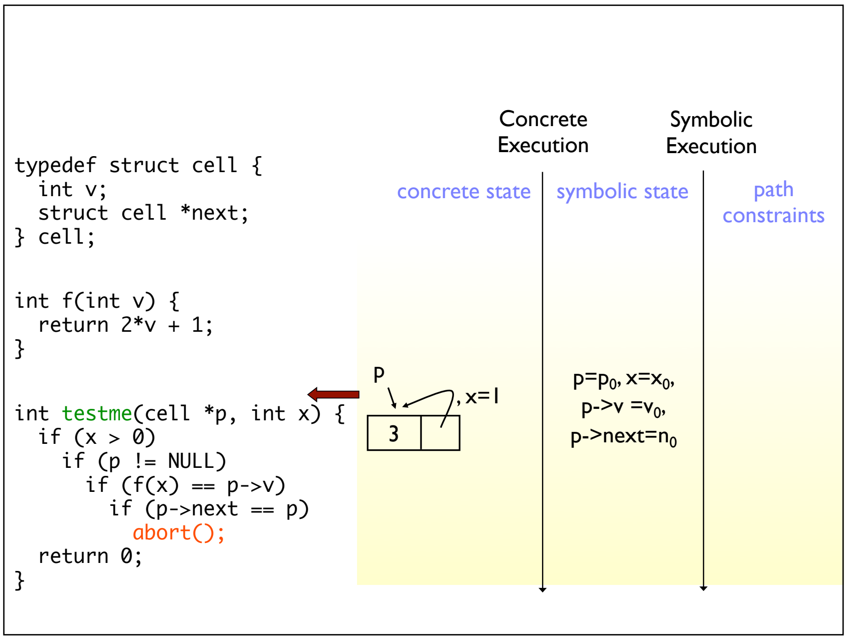




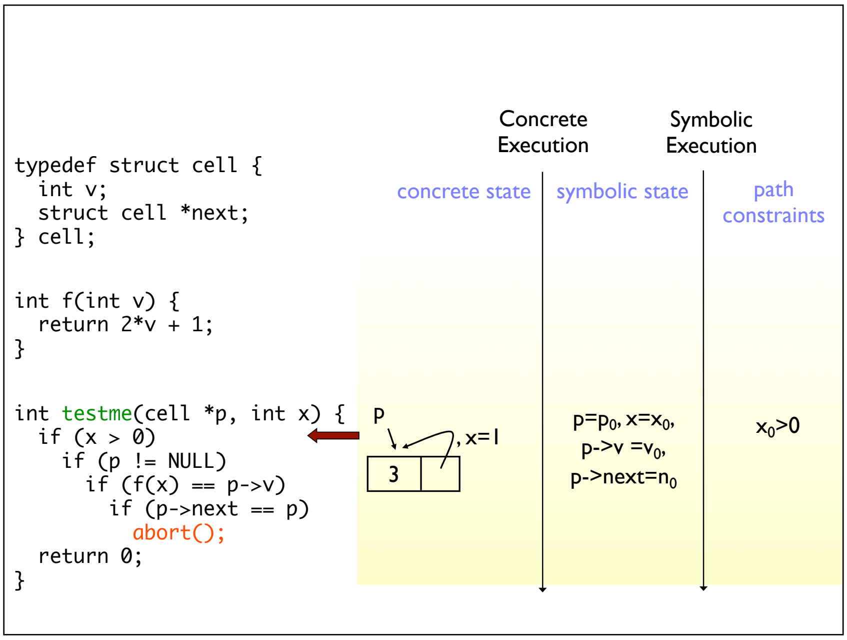
Once more, we negate and solve.



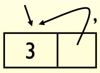
The constraint solver tells us p->next should point to p.



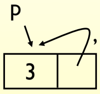
New run with new inputs...



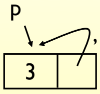
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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: concrete state	Symbolic Execution: symbolic state	Symbolic Execution: path constraints
 p, 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;
}
```

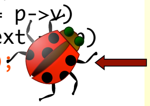
Concrete Execution: concrete state	Symbolic Execution: symbolic state	Symbolic Execution: path constraints
 p, x=1	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$ $p_0 \neq null$ $2x_0+1=v_0$

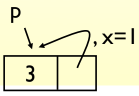
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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;
}
```

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

This time, the last branch evaluates to true.

```
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	path constraints
	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$ $p_0 \neq null$ $2x_0+1 \neq v_0$ $n_0 = p_0$

Finally, the desired statement is reached.

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y){
   z = x*x*x + 3*x*x + 9;
   if(z != y){
       printf("Good branch");
   } else {
       printf("Bad branch");
       abort();
   }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y){
   z = x*x*x + 3*x*x + 9;
   if(z != y){
       printf("Good branch");
   } else {
       printf("Bad branch");
       abort();
   }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
- take then branch with constraint x*x*x+ 3*x*x+9 != y

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y){
   z = x*x*x + 3*x*x + 9;
   if(z != y){
       printf("Good branch");
   } else {
       printf("Bad branch");
       abort();
   }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
- take then branch with constraint x*x*x+ 3*x*x+9 != y
- solve x*x*x+ 3*x*x+9 = y to take else branch
- Don't know how to solve !!
  - **Stuck ?**

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y){
   z = x*x*x + 3*x*x + 9;
   if(z != y){
       printf("Good branch");
   } else {
       printf("Bad branch");
       abort();
   }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
- take then branch with constraint x*x*x+ 3*x*x+9 != y
- solve x*x*x+ 3*x*x+9 = y to take else branch
- Don't know how to solve this
  - Stuck ?
  - Use concrete values!

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y){
   z = x*x*x + 3*x*x + 9;
   if(z != y){
       printf("Good branch");
   } else {
       printf("Bad branch");
       abort();
   }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y){
   z = x*x*x + 3*x*x + 9;
   if(z != y){
        printf("Good branch");
   } else {
        printf("Bad branch");
        abort();
   }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
  - cannot handle symbolic value of z

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y){
   z = x*x*x + 3*x*x + 9;
   if(z != y){
        printf("Good branch");
   } else {
        printf("Bad branch");
        abort();
   }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
  - cannot handle symbolic value of z
  - make symbolic z = 9 and proceed

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y){
   z = x*x*x + 3*x*x + 9;
   if(z != y){
        printf("Good branch");
   } else {
        printf("Bad branch");
        abort();
   }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
  - cannot handle symbolic value of z
  - make symbolic z = 9 and proceed
- take then branch with constraint 9 != y

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y){
   z = x*x*x + 3*x*x + 9;
   if(z != y){
        printf("Good branch");
   } else {
        printf("Bad branch");
        abort();
   }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
  - cannot handle symbolic value of z
  - make symbolic z = 9 and proceed
- take then branch with constraint 9 != y
- solve 9 = y to take else branch
- execute next run with x = -3 and y= 9
  - got error (reaches abort)

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y){
   z = x*x*x + 3*x*x + 9;
   if(z != y){
        printf("Good branch");
   } else {
        printf("Bad branch");

}
```

Replace symbolic expression by concrete value when symbolic expression becomes unmanageable (i.e. non-linear)

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
  - cannot handle symbolic value of z
  - make symbolic z = 9 and proceed
- take then branch with constraint 9 != y
- solve 9 = y to take else branch
- execute next run with x = -3 and y= 9
  - got error (reaches abort)

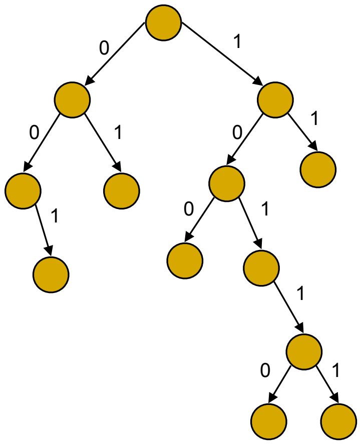
Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y){
   z = x*x*x + 3*x*x + 9;
   if(z != y){
        printf("Good branch");
   } else {
        printf("Bad branch");
        abort();
   }
}
```

```
void again_test_me(int x,int y){
   z = black_box_fun(x);
   if(z != y){
        printf("Good branch");
   } else {
        printf("Bad branch");
        abort();
   }
}
```

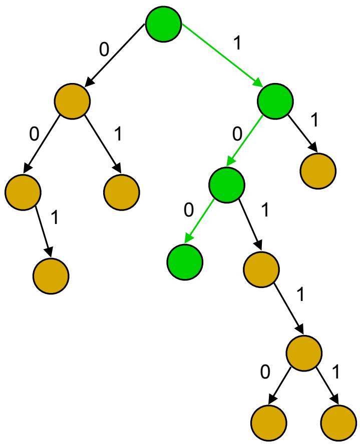
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants



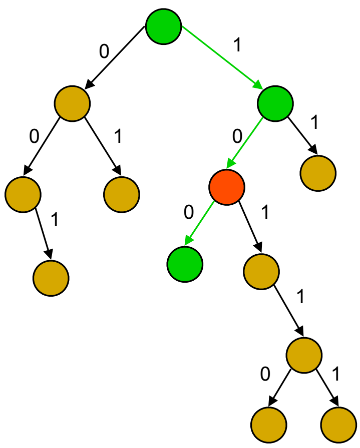
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants



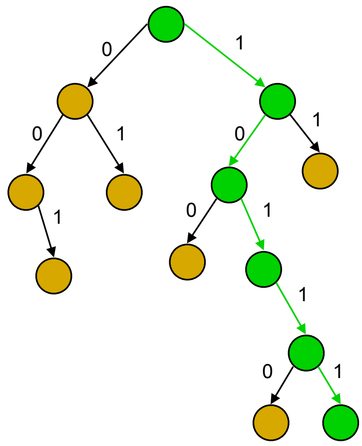
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants



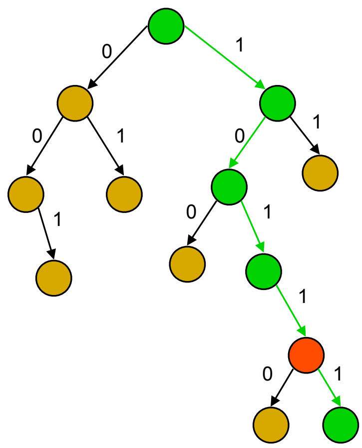
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants



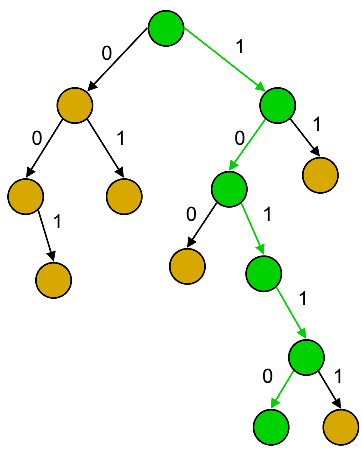
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants



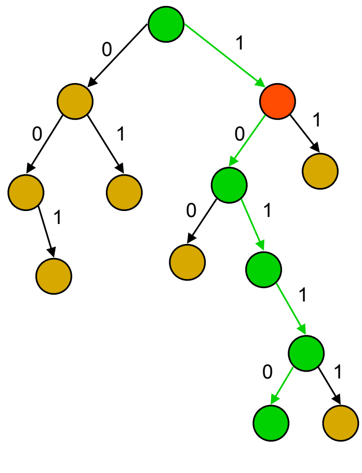
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants



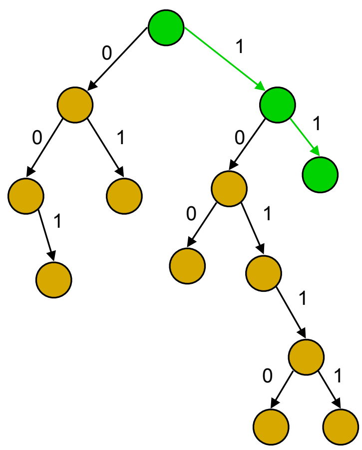
Explicit Path (not State) Model Checking

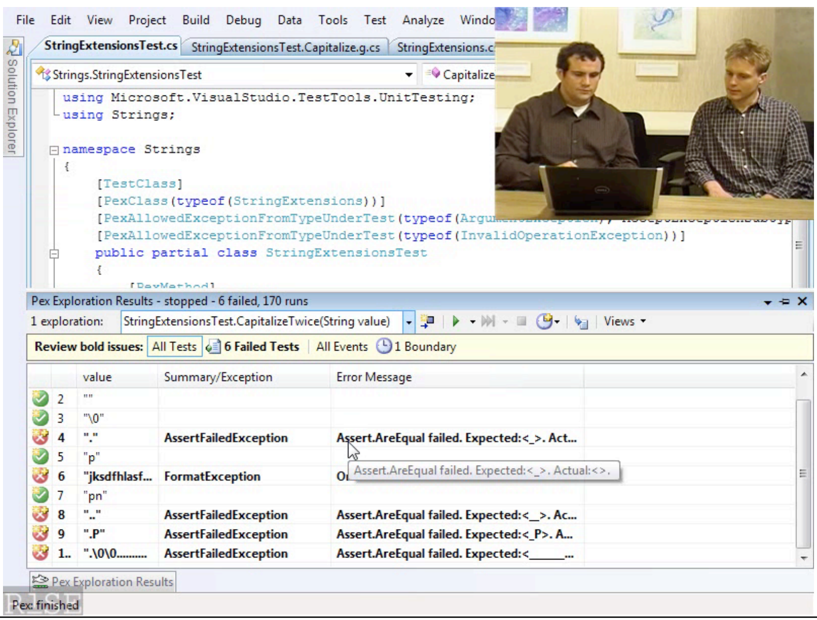
- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants



Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants





In this video, the Pex developers demonstrate Pex in action. The video is available at http://research.microsoft.com/en-us/projects/pex/ - the website also contains a video that explains dynamic symbolic execution.

Theory and Practice

Automated Test Data Generation for Coverage:

Have We Solved This Problem?

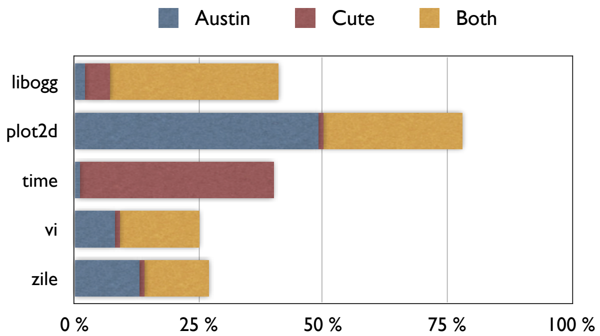
An Experiment

- 2009 study by Lakhotia, McMinn, Harman
- Cute
  Concolic testing tool
- Austin
  Search based testing tool

Subjects

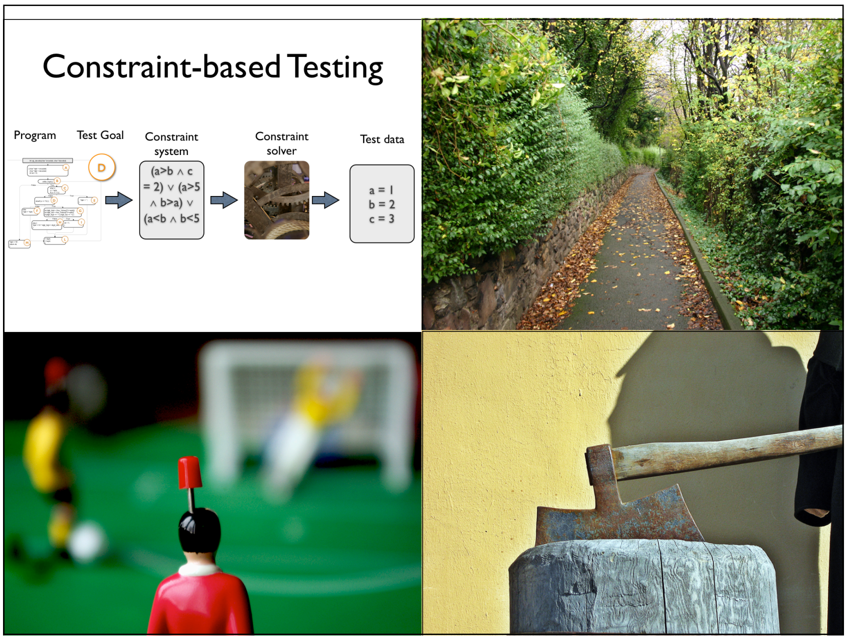
Subject	LOC	Branches
libogg	2,552	290
plot2d	6,062	1,524
time	5,503	202
vi	81,572	8,950
zile	73,472	3,630

Covered Branches



What's the problem?

- Search strategy
  Avoid getting stuck in loops
- Setting up complex data
- Strings
- Environment
- Complex constraints
- Fitness landscape
- Number of iterations in loops must be selected prior to any symbolic execution
- Arrays
- Symbolic execution constrains the shape of dynamically allocated objects
- Non-feasible paths and symbolic execution problems
- Handling loops (manuel vs automatic path selection)


Constraint-based Testing
Program
Test Goal
D
Constraint system
(a>b ∧ c = 2) ∨ (a>5 ∧ b>a) ∨ (a<b ∧ b<5
Constraint solver
Test data
a = 1
b = 2
c = 3