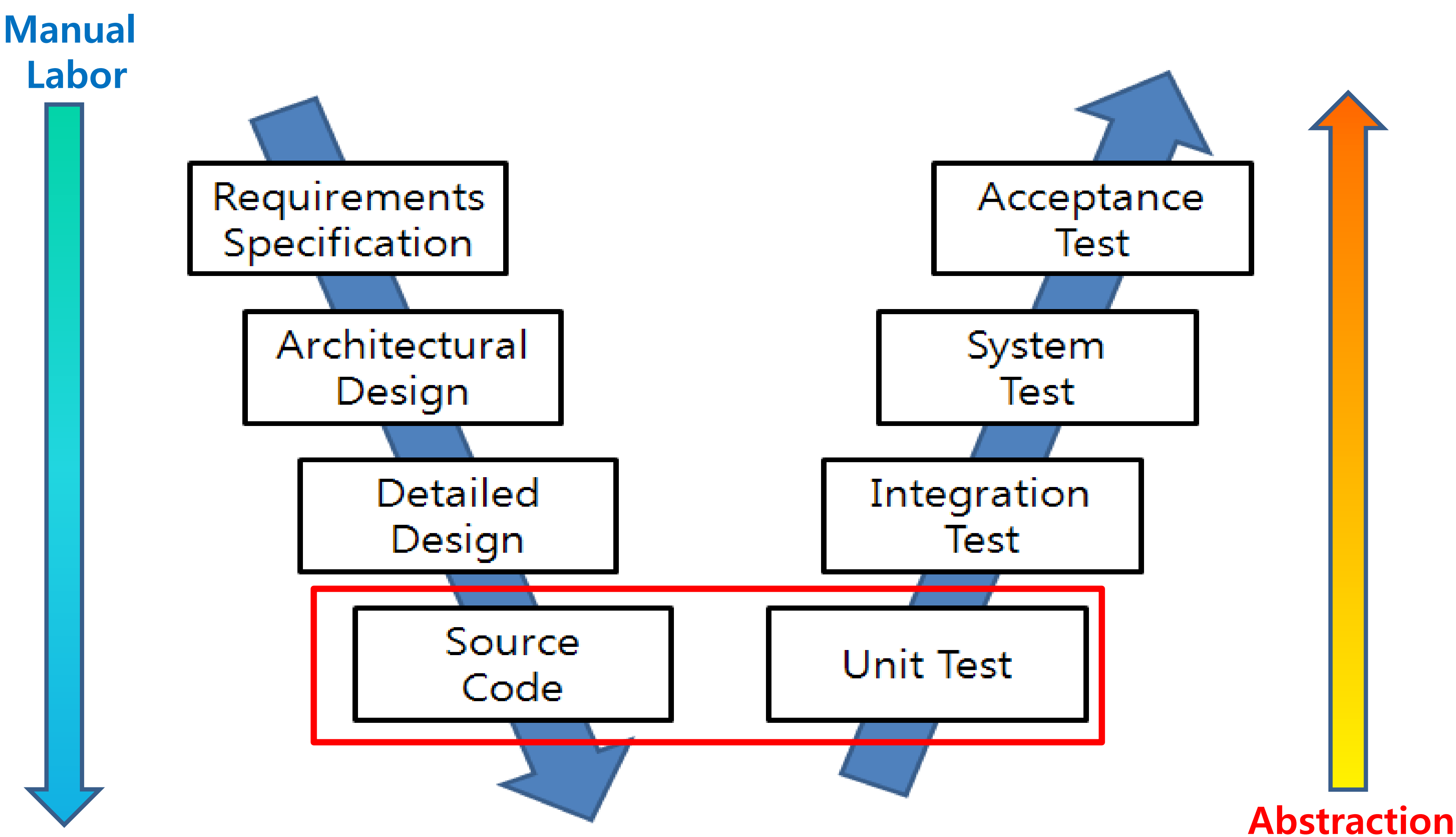
Automated Software Analysis Techniques For High Reliability: A Concolic Testing Approach

Moonzoo Kim

KAIST

Contents

Main Target of Automated SW Analysis



Automated Software Analysis Techniques

- Aims to explore possible behaviors of target systems in an exhaustive manner
- Key methods:
  – Represents a target program/or executions as a "logical formula"
  – Then, analyze the logical formula (a target program) by using logic analysis techniques

Weakness of conventional testing

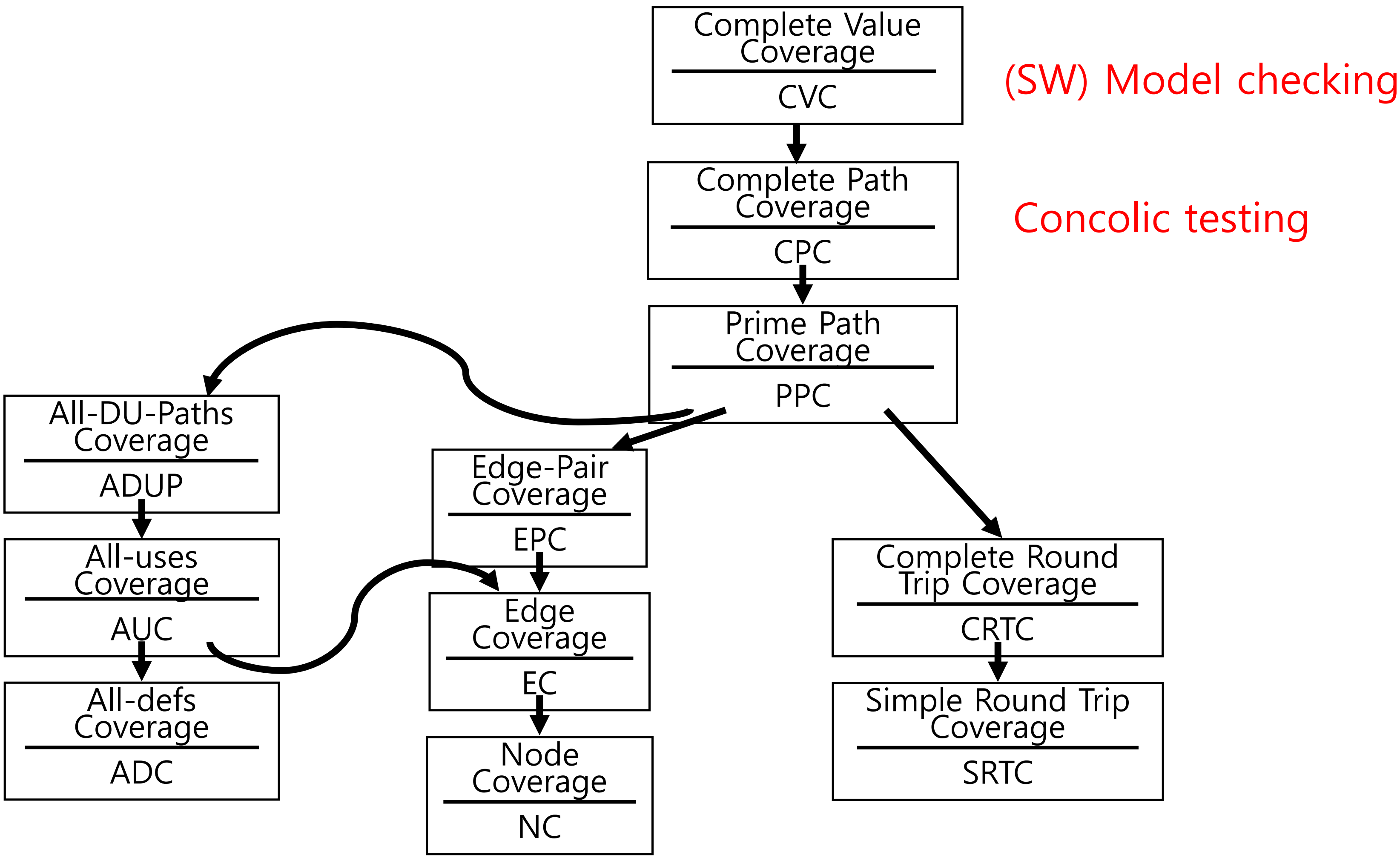
Symbolic execution (1970)

Model checking (1980)

SW model checking (2000)

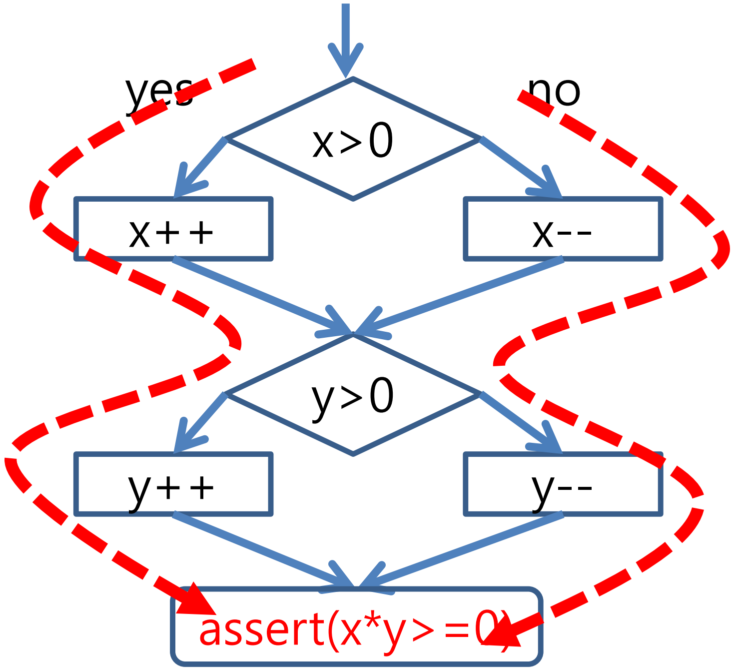
Concolic testing (2005 ~)

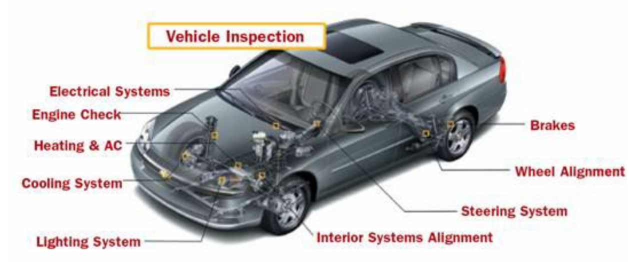
Hierarchy of SW Coverages



Weaknesses of the Branch Coverage

```
/* TC1: x= 1, y= 1;
   TC2: x=-1, y=-1;*/
  void foo(int x, int y) {
    if ( x > 0)
        x++;
    else
        x--;
    if(y >0)
        y++;
    else
        y--;
    assert (x * y >= 0);
}
```





Systematic testing techniques are necessary for quality software!
-> Integration testing is not enough
-> Unit testing with automated test case generation is desirable for both productivity and quality

KAIST

Dynamic v.s. Static Analysis

	Dynamic Analysis (i.e., testing)	Static Analysis (i.e. model checking)
Pros	•Real result •No environmental limitation •Binary library is ok	•Complete analysis result •Fully automatic •Concrete counter example
Cons	•Incomplete analysis result •Test case selection	•Consumed huge memory space •Takes huge time for verification •False alarms

-> Concolic testing

Concolic Approach

- Combine concrete and symbolic execution
  - **Conc**rete + Symb**olic** = Concolic
- In a nutshell, concrete execution over a concrete input guides symbolic execution
  - No false positives
- Automated testing of real-world C Programs
  - Execute target program on automatically generated test inputs
  - All possible execution paths are to be explored
  - Higher branch coverage than random testing

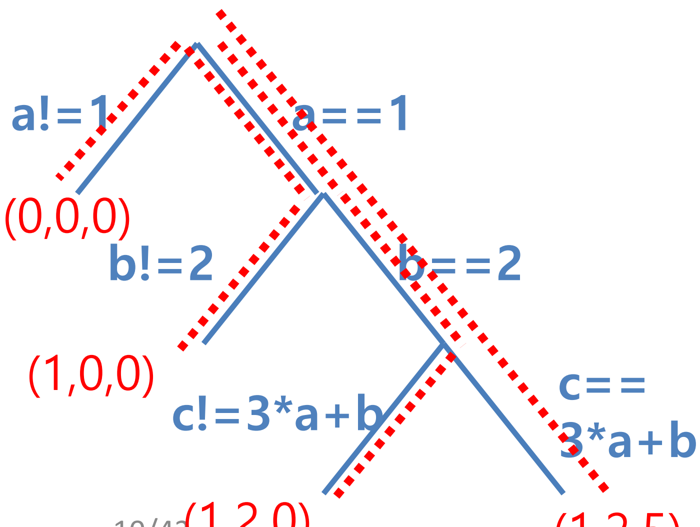
Overview of Concolic Testing Process

1. Select input variables to be handled symbolically
2. A target C program is statically instrumented with probes, which record symbolic path conditions
3. The instrumented C program is executed with given input values
   - Initial input values are assigned randomly
4. Obtain a symbolic path formula φ_i from a concrete execution over a concrete input
5. One branch condition of φ_i is negated to generate a modified symbolic path formula φ_i'
6. A constraint solver solves φ_i' to get next concrete input values
   - Ex. φ_i : (x < 2) && (x + y >= 2) and φ_i' : (x < 2) && (x + y < 2).
     One solution is x=1 and y=0
7. Repeat step 3 until all feasible execution paths are explored

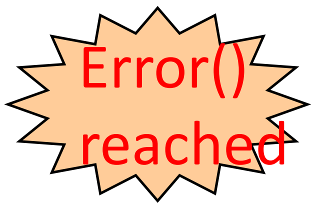
Itera-
tions

Concolic Testing Example

```
// Test input a, b, c
void f(int a, int b, int c) {
   if (a == 1) {
     if (b == 2) {
        if (c == 3*a + b) {
           Error();
} } } }
```

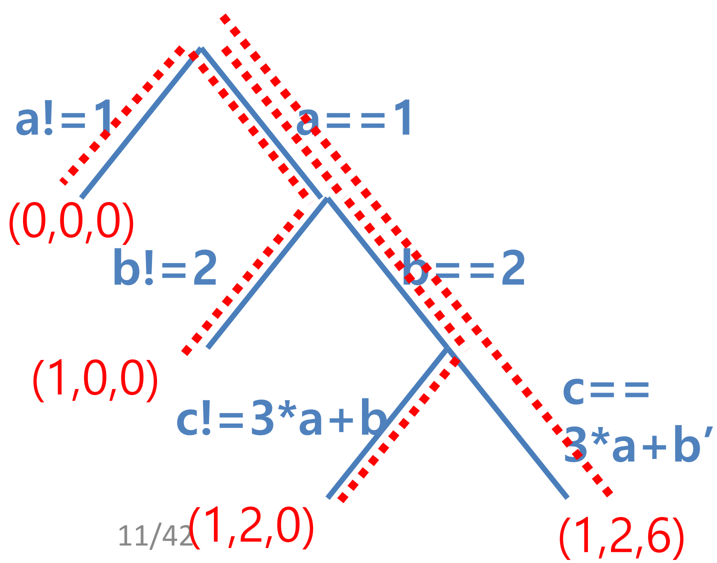


- Random testing
  - Probability of reaching Error( ) is extremely low
- Concolic testing generates the following 4 test inputs
  - (0,0,0): initial random input
    - Obtained symbolic path formula (SPF) φ: a!=1
    - A modified SPF φ' generated from φ : !(a!=1)
  - (1,0,0): a solution of φ' (i.e. !(a!=1))
    - SPF φ : a==1 && b!=2
    - A modified SPF φ' : a==1 && !(b!=2)
  - (1,2,0)
    - SPF φ : a==1 && (b==2) && (c!=3*a +b)
    - A modified SPF φ' : a==1 && (b==2) && !(c!=3*a +b)
  - (1,2,5)
    - Covered all paths and



Concolic Testing Example’

```
// Test input a, b, c
void f(int a, int b, int c) {
   if (a == 1) {
     if (b == 2) {
        b= b+a;
        if (c == 3*a + b) {
           Error();
} } } }
```



- Random testing
  - Probability of reaching Error( ) is extremely low
- Concolic testing generates the following 4 test inputs
  - (0,0,0): initial random input
    - Obtained symbolic path formula (SPF) φ: a!=1
    - A modified SPF φ' generated from φ : !(a!=1)
  - (1,0,0): a solution of φ' (i.e. !(a!=1))
    - SPF φ : a==1 && b!=2
    - A modified SPF φ' : a==1 && !(b!=2)
  - (1,2,0)
    - φ : a==1 && (b==2) && **b’== b +a** && (c!=3*a +**b’**)

    => φ : a==1 && (b==2) && (c!=3*a +(b+a))
    - φ' : a==1 && (b==2) && !(c!=3*a +(b+a))
  - (1,2,6)
    - Covered all paths and Error() reached

Example

```
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)
          Error();
  return 0;
}
```

- Random Test Driver:
  - random memory graph reachable from p
  - random value for x
- Probability of reaching Error( ) is extremely low

Example from the slides "CUTE: A Concolic Unit Testing Engine for C" by K.Sen 2005

Concolic Testing

```
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)
          Error();
  return 0;
}
```

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

Concolic Testing

```
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)
          Error();
  return 0;
}
```

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

Concolic Testing

```
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)
      Error();
  return 0;
}
```

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

Concolic Testing

```
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)
          Error();
  return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints
			$x_0>0$
			$!(p_0!=\text{NULL})$
$p \rightarrow$ NULL, $x=236$		$p=p_0,\ x=x_0$	

Concolic Testing

```
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)
      Error();
  return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete		symbolic	constraints

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

$x_0>0$

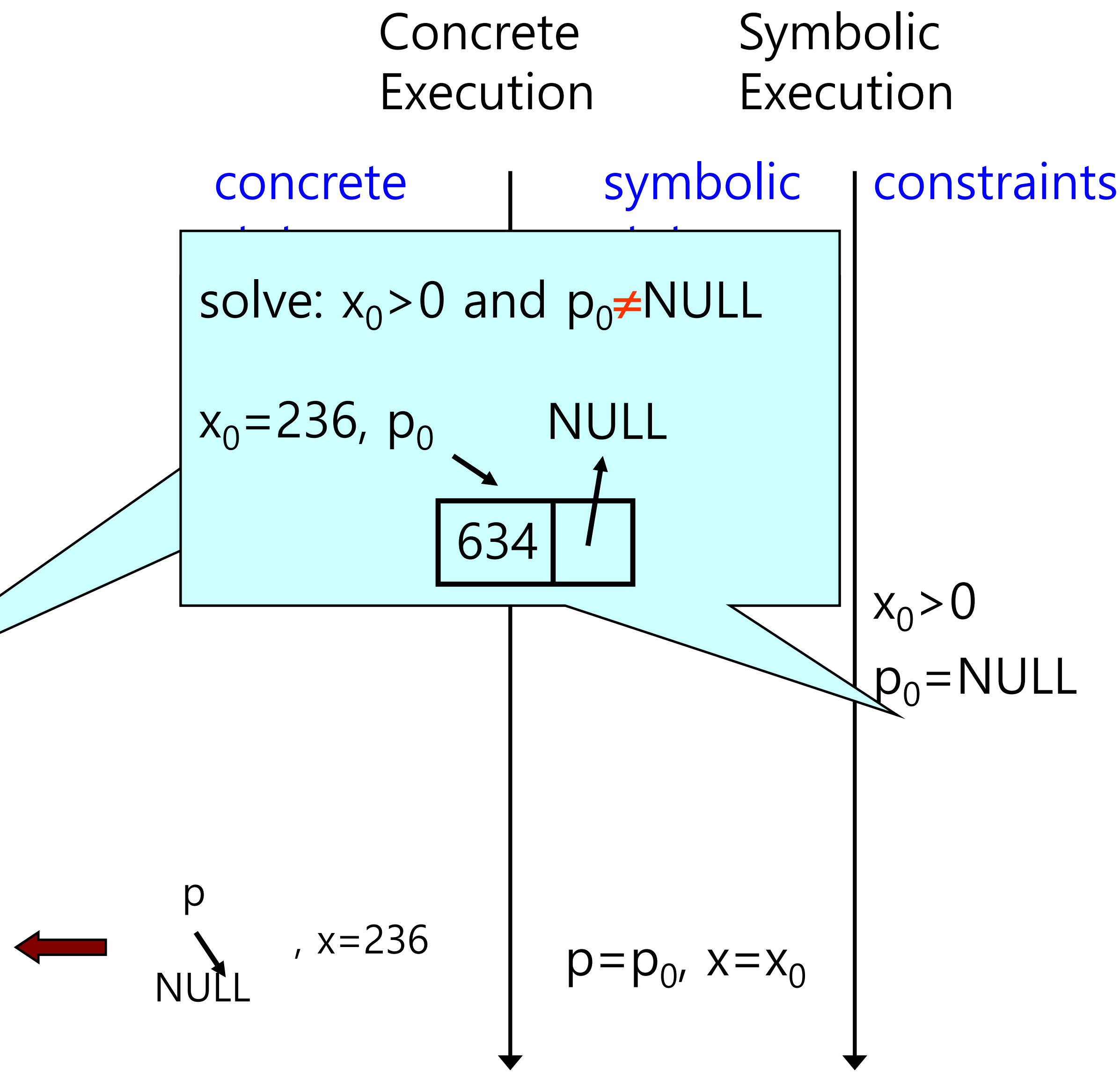
$p_0=NULL$

p → NULL , x=236

$p=p_0$, $x=x_0$

Concolic Testing

```
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)
          Error();
  return 0;
}
```



Concolic Testing

```
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)
          Error();
  return 0;
}
```

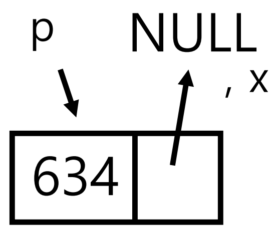
Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

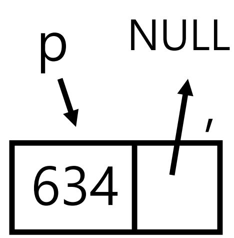


, x=236

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

Concolic Testing

```
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)
          Error();
  return 0;
}
```

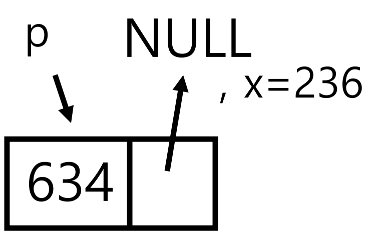
Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
 , x=236	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$

Concolic Testing

```
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)
          Error();
  return 0;
}
```

Concrete Execution | Symbolic Execution

concrete state



symbolic state

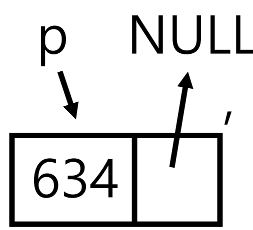
$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

constraints

$x_0 > 0$
$p_0 \neq NULL$

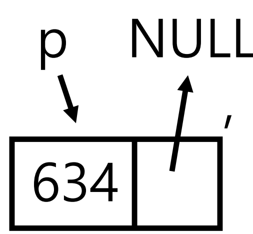
Concolic Testing

```
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)
          Error();
  return 0;
}
```

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

Concolic Testing

```
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)
          Error();
  return 0;
}
```

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

Concolic Testing

```
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)
          Error();
  return 0;
}
```

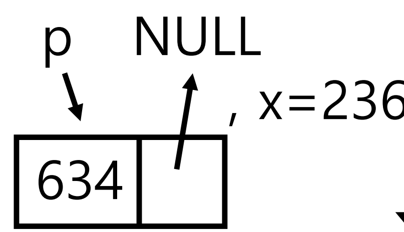
Concrete Execution	Symbolic Execution	
concrete	symbolic	constraints

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

$x_0>0$

$p_0 \neq NULL$

$2x_0+1 \neq v_0$

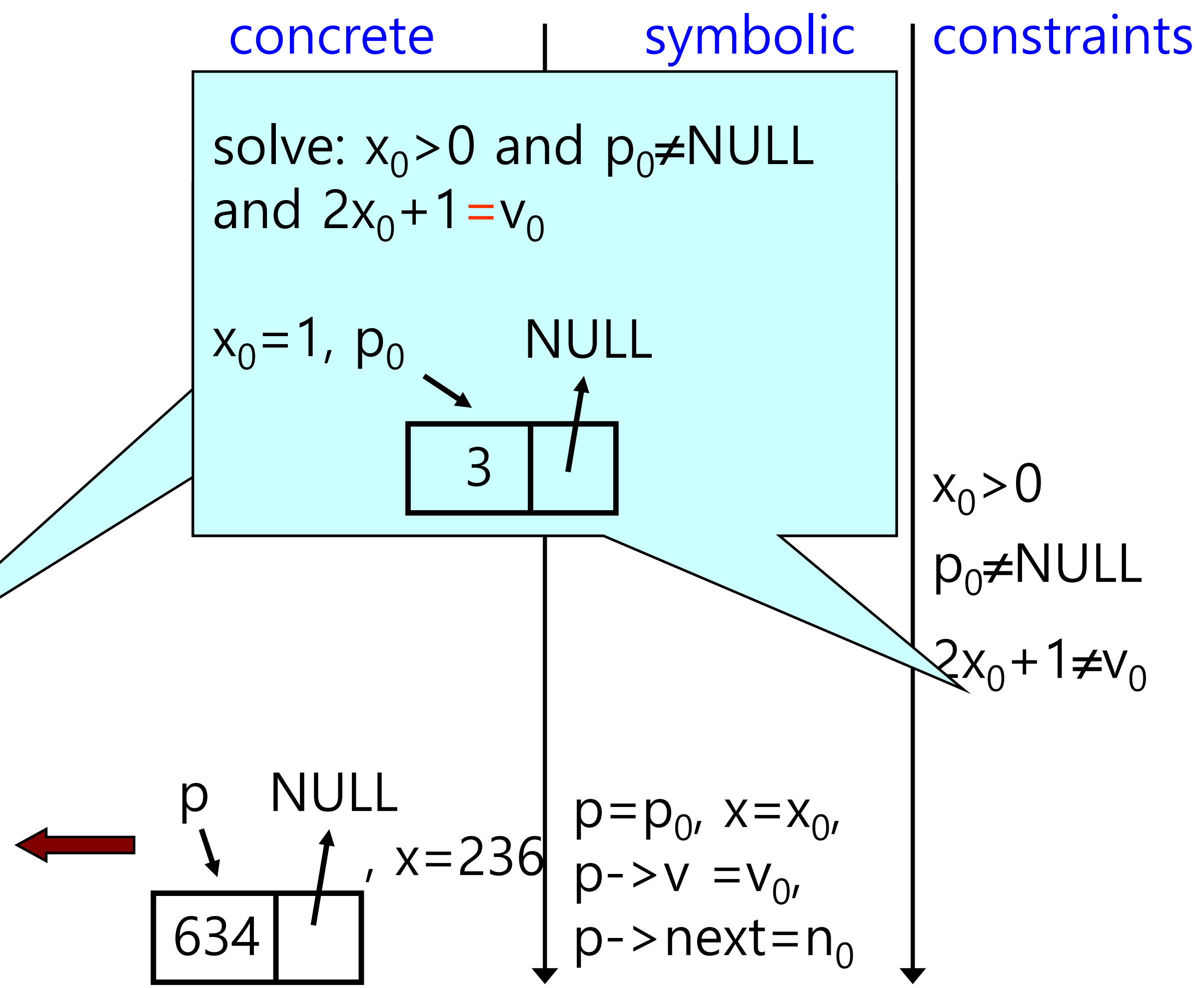


$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

Concolic Testing

```
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)
          Error();
  return 0;
}
```

Concrete Execution | Symbolic Execution



Concolic Testing

```
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)
          Error();
  return 0;
}
```

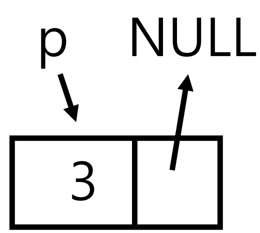
Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints



, x=1

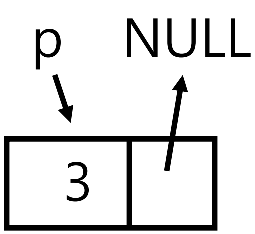
$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

Concolic Testing

```
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)
          Error();
  return 0;
}
```

Concrete Execution

Symbolic Execution

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

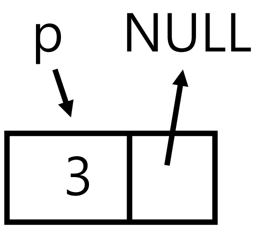
Concolic Testing

```
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)
      Error();
  return 0;
}
```

Concrete Execution

Symbolic Execution

concrete state



, x=1

symbolic state

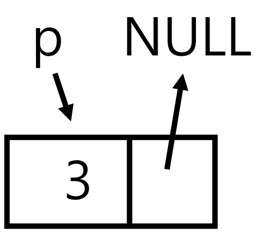
$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

constraints

$x_0>0$
$p_0 \neq NULL$

Concolic Testing

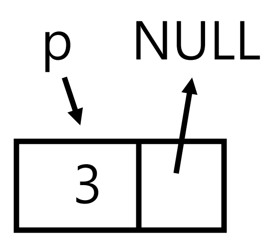
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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)
      Error();
  return 0;
}
```

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

Concolic Testing

```
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)
      Error();
  return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints



, $x=1$

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

$x_0>0$

$p_0 \neq NULL$

$2x_0+1=v_0$

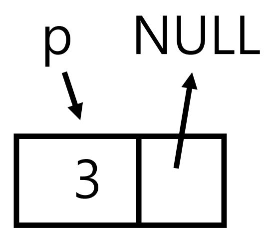
$n_0 \neq p_0$

Concolic Testing

```
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)
      Error();
  return 0;
}
```

Concrete Execution

Symbolic Execution

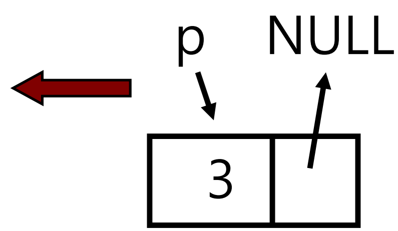
concrete state	symbolic state	constraints
		$x_0>0$
		$p_0 \neq NULL$
		$2x_0+1=v_0$
		$n_0 \neq p_0$
 , x=1	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	

Concolic Testing

```
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)
      Error();
  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 $n_0=p_0$



, $x=1$

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

$x_0>0$

$p_0 \neq NULL$

$2x_0+1=v_0$

$n_0 \neq p_0$

Concolic Testing

```
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)
      Error();
  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 $n_0=p_0$

$x_0=1$, p_0

3

$x_0>0$

$p_0 \neq NULL$

$2x_0+1=v_0$

$n_0 \neq p_0$

p NULL

3

, x=1

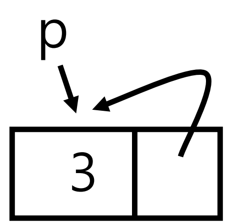
$p=p_0$, $x=x_0$,

$p\text{->}v = v_0$,

$p\text{->}next = n_0$

Concolic Testing

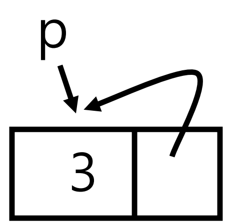
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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)
          Error();
  return 0;
}
```

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

Concolic Testing

```
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)
          Error();
  return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints



, x=1

$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

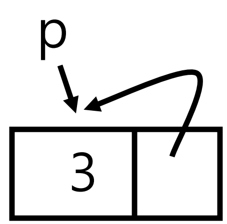
$x_0>0$

Concolic Testing

```
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)
          Error();
  return 0;
}
```

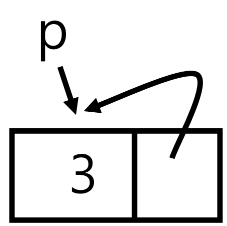
Concrete Execution

Symbolic Execution

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

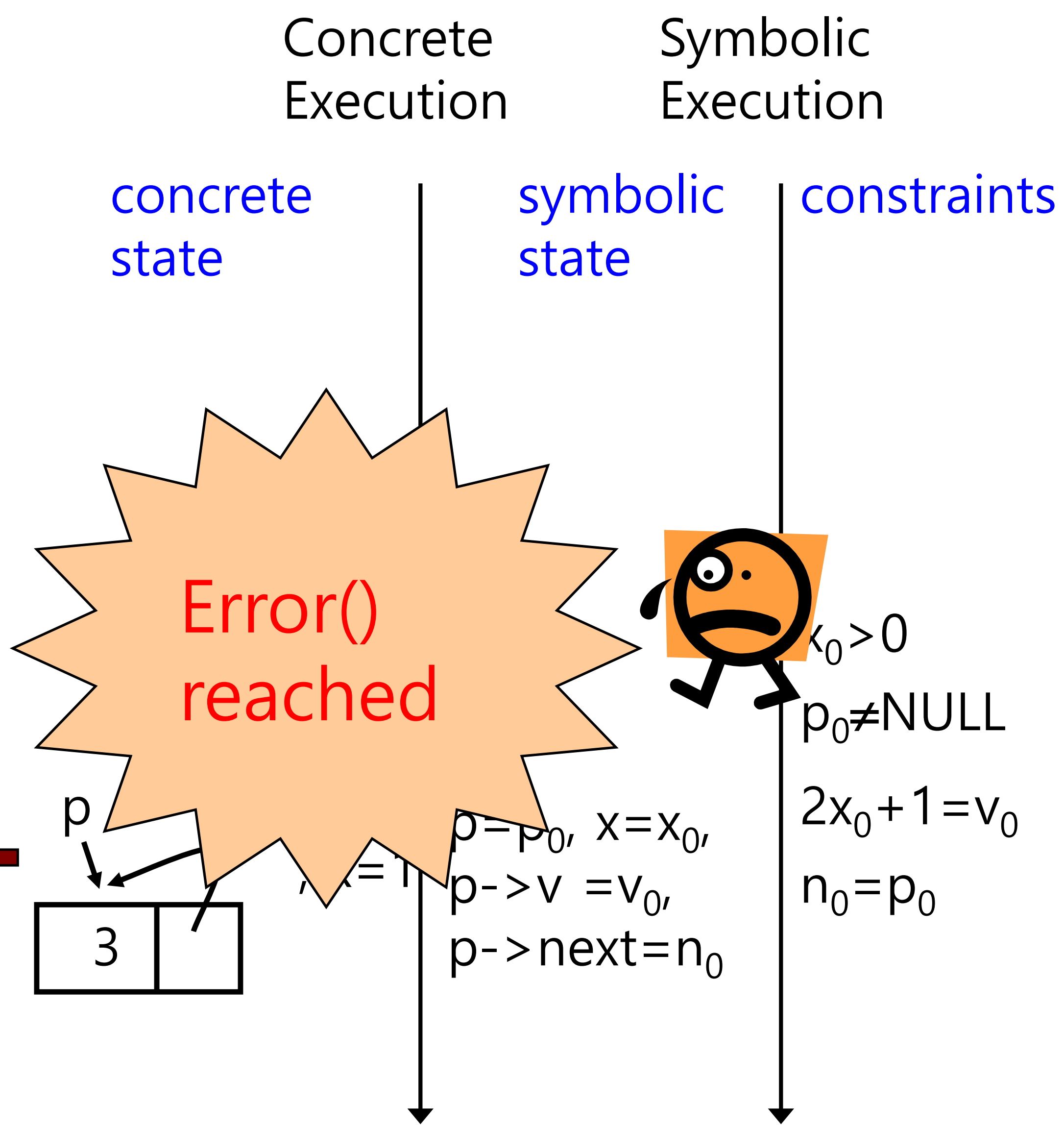
Concolic Testing

```
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)
      Error();
  return 0;
}
```

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

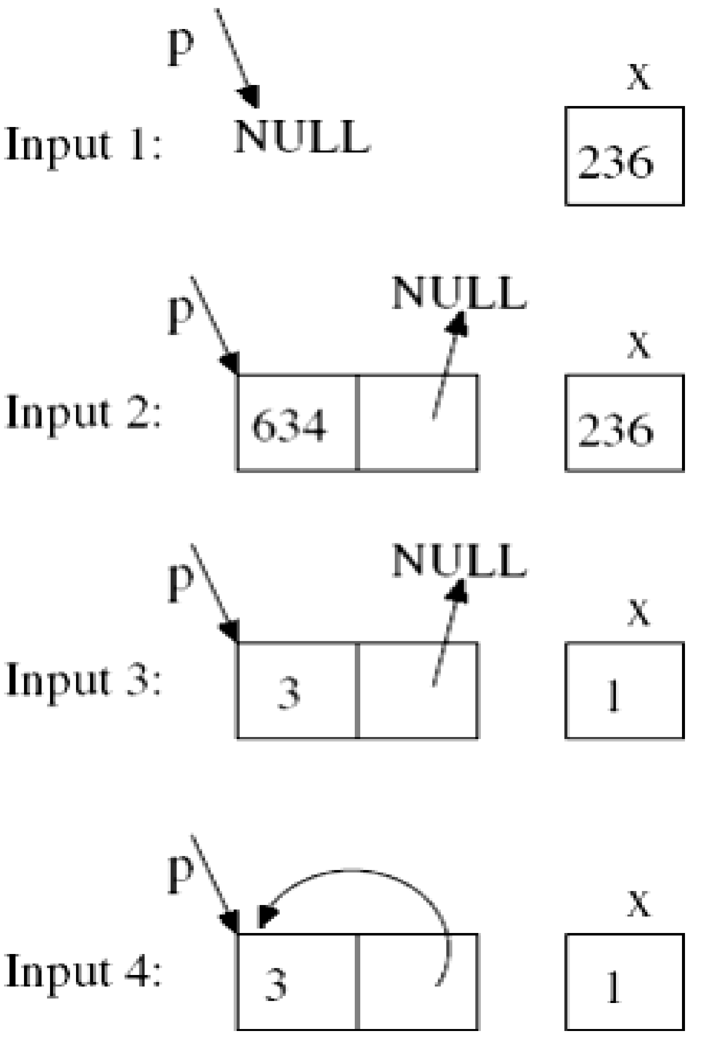
Concolic Testing

```
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)
      Error();
  return 0;
}
```



4 Test Inputs Generated

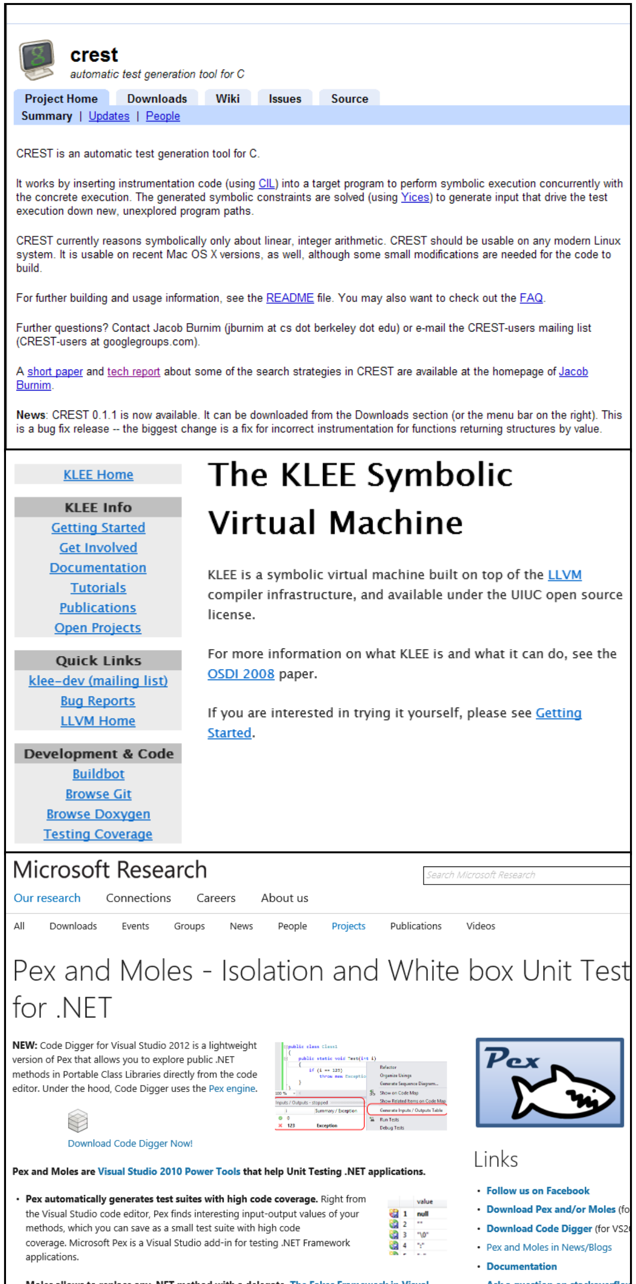
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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)
          Error();
  return 0;
}
```



Summary: Concolic Testing

- Pros
  - Automated test case generation
  - High coverage
  - High applicability (no restriction on target programs)
- Cons
  - If a target program has external binrary function calls, coverage might not be complete
    - Ex. if(sin(x) + query(y) == 0.3) { error(); }
  - Current limitation on pointer and array
  - Slow analysis speed due to a large # of TCs

Concolic Testing Tools



- CROWN
  - Target: C
  - Instrumentation based extraction
  - BV supported
  - https://github.com/swtv-kaist/CROWN
- KLEE (open source)
  - Target: LLVM
  - VM based symbolic formula extraction
  - BV supported
  - Symbolic POSIX library supported
  - http://ccadar.github.io/klee/
- PEX (IntelliTest incorporated in Visual Studio Enterprise)
  - Target: C#
  - VM based symbolic formula extraction
  - BV supported
  - Integrated with Visual Studio
  - http://research.microsoft.com/en-us/projects/pex/
- CATG (open source)
  - Target: Java
  - Trace/log based symbolic formula extraction
  - LIA supported

Case Study: Busybox

- We test a busybox by using CREST.
  - BusyBox is a one-in-all command-line utilities providing a fairly complete programming/debugging environment
  - It combines tiny versions of ~300 UNIX utilities into a sin gle small executable program suite.
  - Among those 300 utilities, we focused to test the followi ng 10 utilities
    - **`grep`, `vi`, `cut`, `expr`, `od` , `printf`, `tr`, `cp`, `ls`, `mv`.**
    - We selected these 10 utilities, because their behavior is easy to understand so that it is clear what variables should be declared as symbolic
    - Each utility generated 40,000 test cases for 4 different search strategies

Busybox Testing Result

Utility	LOC	# of branches	DFS #of covered branch/time	CFG #of covered branch/time	Random #of covered branch/time	Random_input #of covered branch/time	Merge of all 4 strategies #of covered branch/time
grep	914	178	105(59.0%)/2785s	85(47.8%)/56s	136(76.4%)/85s	50(28.1%)/45s	136(76.4%)
vi	4000	1498	855(57.1%)/1495s	965(64.4%)/1036s	1142(76.2)/723s	1019(68.0%)/463s	1238(82.6%)
cut	209	112	67(59.8%)/42s	60(53.6%)/45s	84(75.0%)/53s	48(42.9%)/45s	90(80.4%)
expr	501	154	104(67.5%)/58s	101(65.6%)/44s	105(68.1%)/50s	48(31.2%)/31s	108(70.1%)
od	222	74	59(79.7%)/35s	72(97.3%)/41s	66(89.2%)/42s	44(59.5%)/30s	72(97.3%)
printf	406	144	93(64.6%)/84s	109(75.7%)/41s	102(70.8%)/40s	77(53.5%)/30s	115(79.9%)
tr	328	140	67(47.9%)/58s	72(51.4%)/50s	72(51.4%)/50s	63(45%)/42s	73(52.1%)
cp	191	32	20(62.5%)/38s	20(62.5%)/38s	20(62.5%)/38s	17(53.1%)/30s	20(62.5%)
ls	1123	270	179(71.6%)/87s	162(64.8%)/111s	191(76.4%)/86s	131(52.4%)/105s	191(76.4%)
mv	135	56	24(42.9%)/0s	24(42.9%)/0s	24(42.9%)/0s	17(30.3%)/0s	24(47.9%)
AVG	**803**	**264**	**157.3(59.6%)/809s**	**167(63.3%)/146s**	**194.2(73.5%)/117s**	**151.4(57.4%)/83s**	**206.7(78.4%)/1155s**

Result of grep

Experiment 1:

Iterations: 10, 000

branches in grep.c : 178

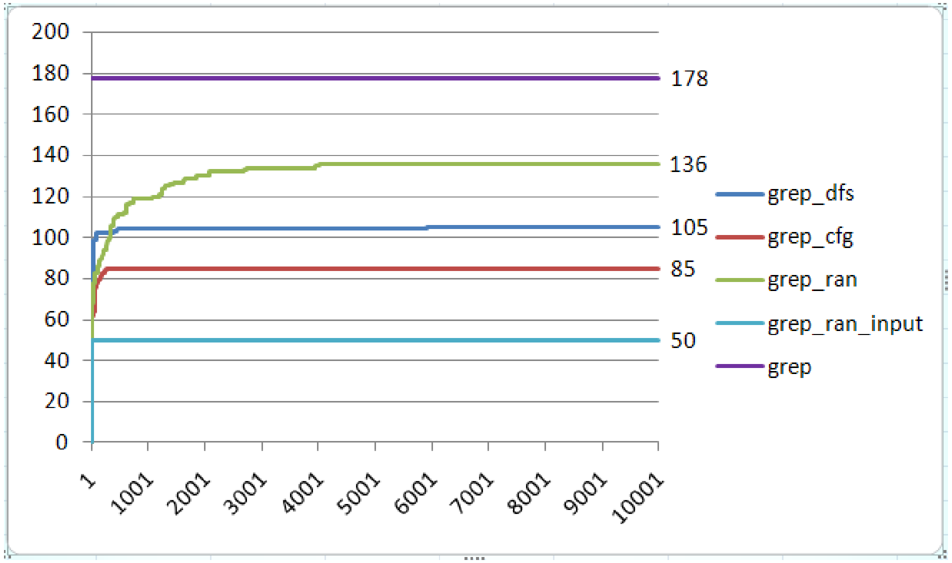
Execution Command:

run_crest './busybox grep "define" test_grep.dat' 10000 -**dfs**

run_crest './busybox grep "define" test_grep.dat' 10000 -**cfg**

run_crest './busybox grep "define" test_grep.dat' 10000 -**random**

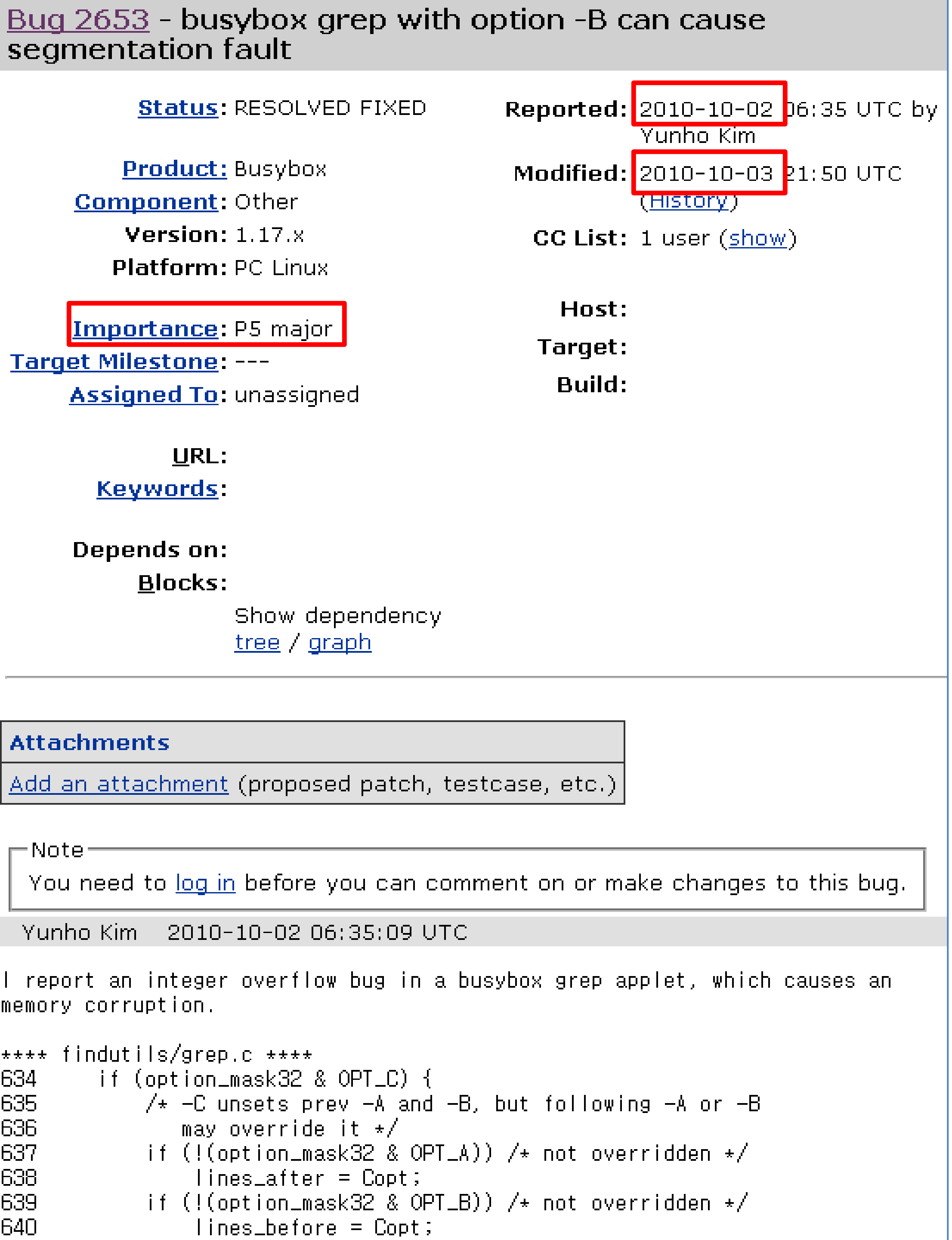
run_crest './busybox grep "define" test_grep.dat' 10000 -**random_input**



Strategy	Time cost (s)
Dfs	2758
Cfg	56
Random	85
Pure_random	45

Test Oracles

- In the busybox testing, we do not use any explicit test oracles
  - Test oracle is an orthogonal issue to test case generation
  - However, still violation of runtime conformance (i.e., no segmentation fault, no divide-by-zero, etc) can be checked
- Segmentation fault due to integer overflow detected at busybox grep
  - This bug was detected by test cases generated using DFS
  - The bug causes segmentation fault when
    - -B 1073741824 (i.e. 2^32/4)
    - PATTERN should match line(s) after the 1^{st} line
    - Text file should contain at least two lines
  - Bug scenario
    - Grep tries to dynamically allocate memory for buffering matched lines (-B option).
    - But due to integer overflow (# of line to buffer * sizeof(pointer)), memory is allocated in much less amount
    - Finally grep finally accesses illegal memory area


Bug 2653 - busybox grep with option -B can cause segmentation fault

Status: RESOLVED FIXED
Product: Busybox
Component: Other
Version: 1.17.x
Platform: PC Linux
Importance: P5 major
Target Milestone: ---
Assigned To: unassigned
URL:
Keywords:
Depends on:
Blocks:
Show dependency tree / graph

Reported: 2010-10-02 06:35 UTC by Yunho Kim
Modified: 2010-10-03 21:50 UTC (History)
CC List: 1 user (show)
Host:
Target:
Build:

Attachments
Add an attachment (proposed patch, testcase, etc.)

Note
You need to log in before you can comment on or make changes to this bug.

Yunho Kim 2010-10-02 06:35:09 UTC

I report an integer overflow bug in a busybox grep applet, which causes an memory corruption.

```
**** findutils/grep.c ****
    if (option_mask32 & OPT_C) {
        /* -C unsets prev -A and -B, but following -A or -B
           may override it */
        if (!(option_mask32 & OPT_A)) /* not overridden */
            lines_after = Copt;
        if (!(option_mask32 & OPT_B)) /* not overridden */
            lines_before = Copt;
```

- Bug patch was immediately made in 1 day, since this bug is critical one
  - Importance: P5 major
    - major loss of function
  - Busybox 1.18.x will have fix for this bug

SAGE: Whitebox Fuzzing for Security Testing @ Microsoft

- X86 binary concolic testing tool to generate millions of test files automatically targeting large applications
  - used daily in Windows, Office, etc.
- Mainly targets crash bugs in various windows file parsers (>hundreds)
- Impact: since 2007 to 2013
  - 500+ machine years
  - 3.4 Billion+ constraints
  - 100s of apps, 100s of bugs
    - 1/3 of all security bugs detected by Win7 WEX team were found by SAGE
  - Millions of dollars saved

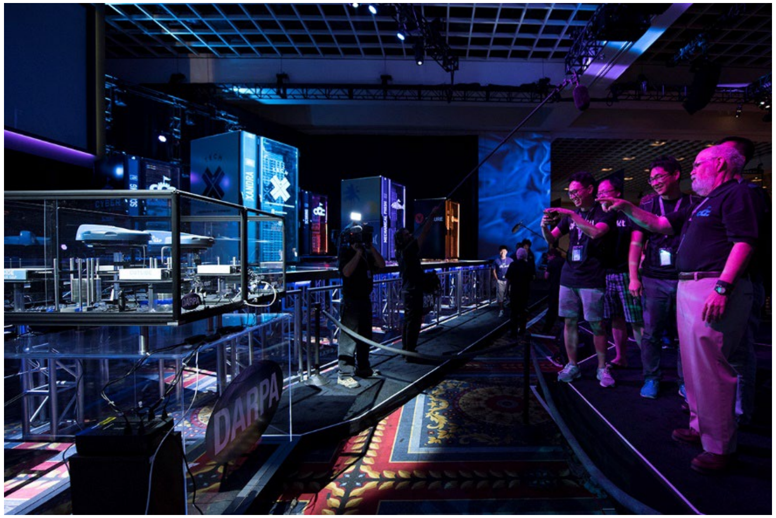
This slide quotes PLDI 2013 MSR Open House Event poster "SAGE: WhiteboxFuzzing for Security Testing" by E.Bounimova, P.Godefroid, and D.Molnar

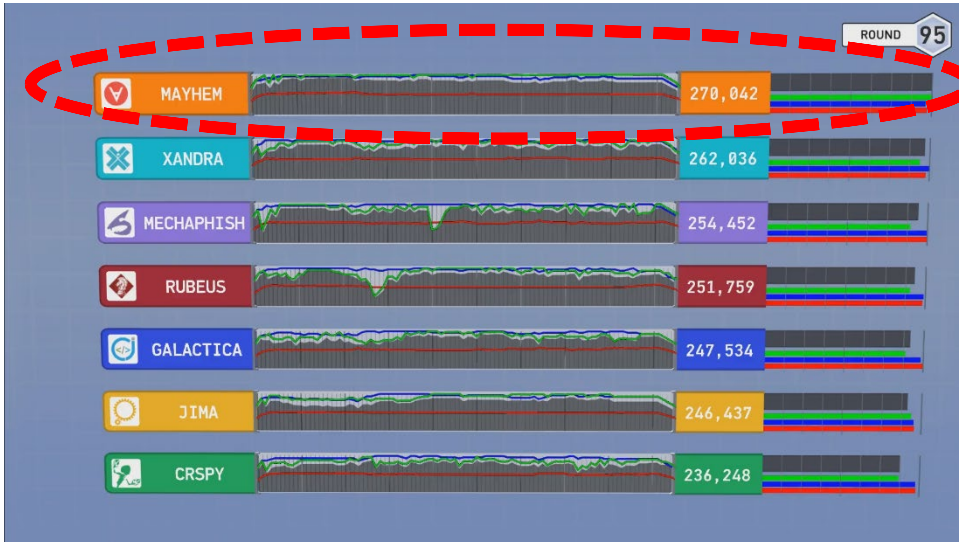
Microsoft Security Risk Detection

- Azure-based cloud service to find security bugs in x86 windows binary
- Based on concolic testing techniques of SAGE

2016 Aug DARPA Cyber Grand Challenge -the world's 1st all-machine hacking tournament

- Each team's Cyber Reasoning System automatically identifies security flaws and applies patches to its own system in a hack-and-defend style contest targeting a new Linux-based OS DECREE
- Mayhem won the best score, which is CMU's concolic testing based tool





Solution for Huge Economic & Social Cost due to SW Bugs

Labor-intensive Manual Testing
Large SW Testing Cost and Time
Low Bug Detection Abilty
Low Product Quality

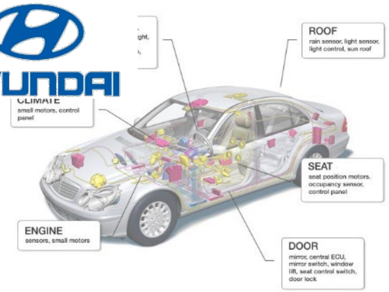
Solution: AI-based Automated Concolic Testing Technique

movie link https://bit.ly/3NS6RrQ

Existing Problems → Developed Solutions

'10-14 Samsung Electronics

Detected dozens of crash bugs in the comm. firmware



'15~20 Hyundai /Mobis

Achieved 90% branch cov. and reduced 80% of labor cost by using auto. testing tech.

'18 LIGnex1

Detected several SW bugs in the 10 programs in the battleships

'20 Natl. Security Research Inst.

Detected SW bugs in the software in the security equipment

현대모비스, AI 기반 소프트웨어 검증시스템 도입..."효율 2배로"

2018-07-22 10:00

댓글 가⁻ 가⁺

'마이스트' 적용...대화형 검색 로봇 '마이봇'도 도입

(서울=연합뉴스) 윤보람 기자 = 현대모비스[012330]가 인공지능(AI)을 활용해 자율주행, 커넥티비티(연결성) 등 미래 자동차 소프트웨어(SW) 개발에 속도를 낸다.

현대모비스는 AI를 기반으로 하는 소프트웨어 검증시스템 '마이스트'(MAIST: Mobis Artificial Intelligence Software Testing)를 최근 도입했다고 22일 밝혔다.

Hyundai Mobis and a research team lead by Prof. Moonzoo Kim at KAIST jointly developed MAIST for automated testing

MAIST automates unit coverage testing performed by human engineers by applying concolic unit testing

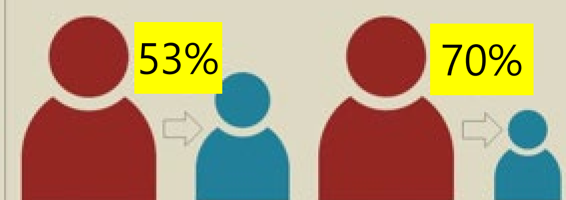
실제 현대모비스가 통합형 차체제어시스템(IBU)과 써라

MAIST can reduce 53% and 70% of manual testing effort for IBU(Integrated Body Unit) and SVM(Surround View Monitoring)

현대모비스는 하반기부터 소프트웨어가 탑재되는 제동, 조향 등 모든 전장부품으로 마이스트를 확대 적용할 계획이다. 글로벌 소프트웨어 연구기지인 인도연구소에도 적용한다.

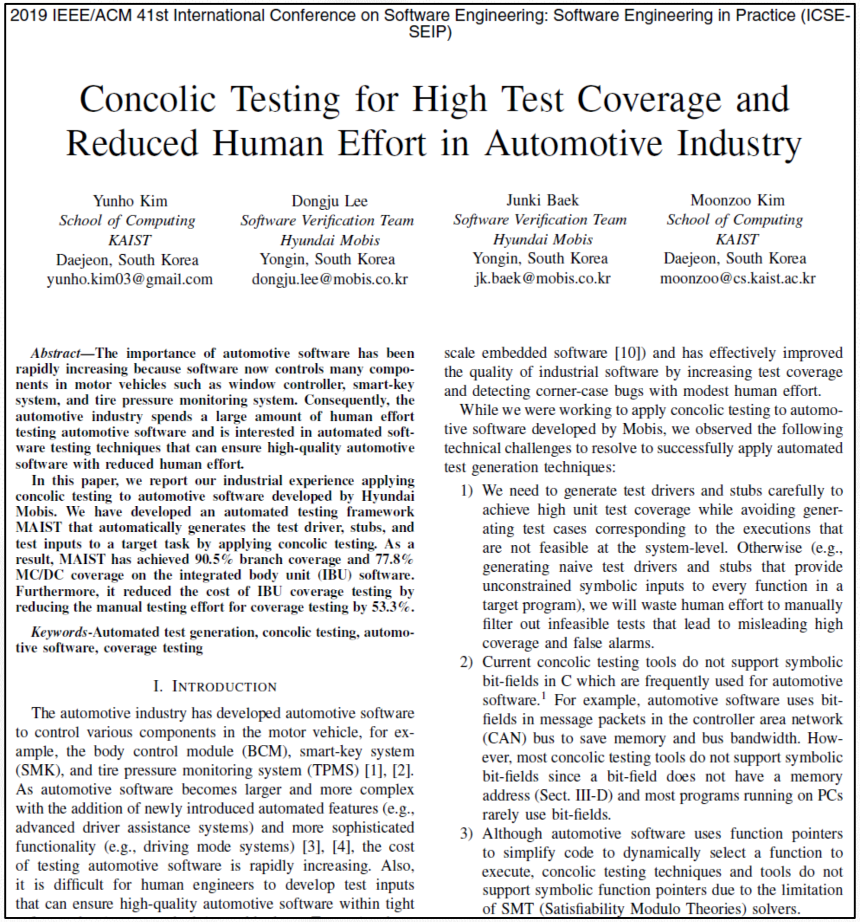
■ **현대모비스 인공지능 도입 사례**

AI 시스템	목적	도입 효과
MAIST	소프트웨어 검증 자동화	IBU 53%, SVM 70% Reduction of manual testing effort
마이봇 (Mobis AI Robot)	소프트웨어 개발문서 검색	딥러닝 기반, 개발문서 20만 건 관리



http://m.yna.co.kr/kr/contents/?cid=AKR20180720158800003&mobile

MAIST Paper Presentation @ ICSE 2019 SEIP Track



2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)

Concolic Testing for High Test Coverage and Reduced Human Effort in Automotive Industry

Yunho Kim, *School of Computing*, *KAIST*, Daejeon, South Korea, yunho.kim03@gmail.com
Dongju Lee, *Software Verification Team*, *Hyundai Mobis*, Yongin, South Korea, dongju.lee@mobis.co.kr
Junki Baek, *Software Verification Team*, *Hyundai Mobis*, Yongin, South Korea, jk.baek@mobis.co.kr
Moonzoo Kim, *School of Computing*, *KAIST*, Daejeon, South Korea, moonzoo@cs.kaist.ac.kr

Abstract—The importance of automotive software has been rapidly increasing because software now controls many components in motor vehicles such as window controller, smart-key system, and tire pressure monitoring system. Consequently, the automotive industry spends a large amount of human effort testing automotive software and is interested in automated software testing techniques that can ensure high-quality automotive software with reduced human effort.

In this paper, we report our industrial experience applying concolic testing to automotive software developed by Hyundai Mobis. We have developed an automated testing framework MAIST that automatically generates the test driver, stubs, and test inputs to a target task by applying concolic testing. As a result, MAIST has achieved 90.5% branch coverage and 77.8% MC/DC coverage on the integrated body unit (IBU) software. Furthermore, it reduced the cost of IBU coverage testing by reducing the manual testing effort for coverage testing by 53.3%.

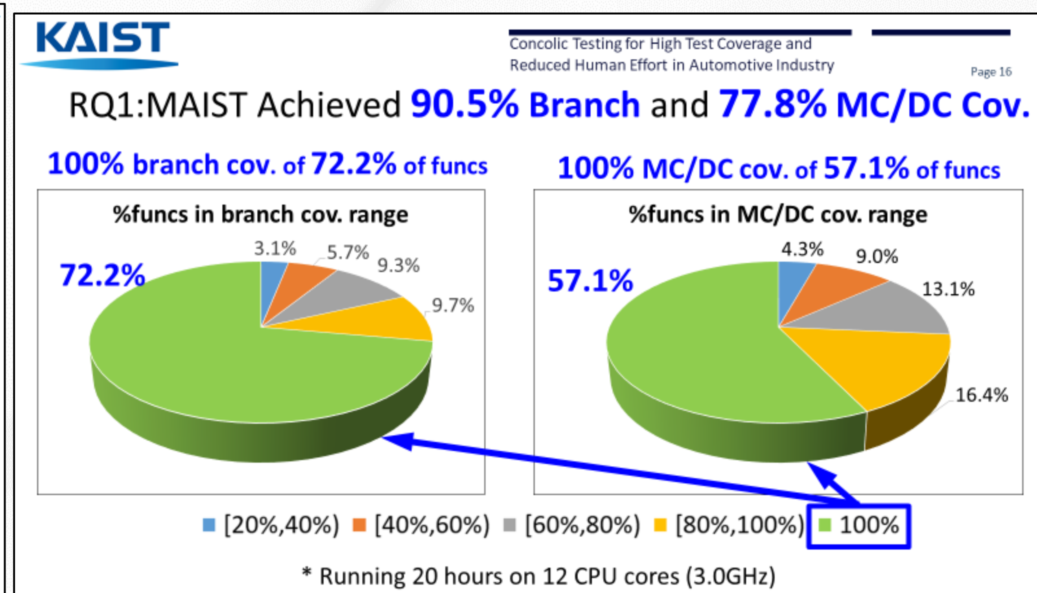
Keywords-Automated test generation, concolic testing, automotive software, coverage testing

I. Introduction

The automotive industry has developed automotive software to control various components in the motor vehicle, for example, the body control module (BCM), smart-key system (SMK), and tire pressure monitoring system (TPMS) [1], [2]. As automotive software becomes larger and more complex with the addition of newly introduced automated features (e.g., advanced driver assistance systems) and more sophisticated functionality (e.g., driving mode systems) [3], [4], the cost of testing automotive software is rapidly increasing. Also, it is difficult for human engineers to develop test inputs that can ensure high-quality automotive software within tight scale embedded software [10]) and has effectively improved the quality of industrial software by increasing test coverage and detecting corner-case bugs with modest human effort.

While we were working to apply concolic testing to automotive software developed by Mobis, we observed the following technical challenges to resolve to successfully apply automated test generation techniques:

1) We need to generate test drivers and stubs carefully to achieve high unit test coverage while avoiding generating test cases corresponding to the executions that are not feasible at the system-level. Otherwise (e.g., generating naive test drivers and stubs that provide unconstrained symbolic inputs to every function in a target program), we will waste human effort to manually filter out infeasible tests that lead to misleading high coverage and false alarms.
2) Current concolic testing tools do not support symbolic bit-fields in C which are frequently used for automotive software.[1] For example, automotive software uses bit-fields in message packets in the controller area network (CAN) bus to save memory and bus bandwidth. However, most concolic testing tools do not support symbolic bit-fields since a bit-field does not have a memory address (Sect. III-D) and most programs running on PCs rarely use bit-fields.
3) Although automotive software uses function pointers to simplify code to dynamically select a function to execute, concolic testing techniques and tools do not support symbolic function pointers due to the limitation of SMT (Satisfiability Modulo Theories) solvers.



ICSE 2019 SEIP Acceptance ratio: 20%

Model Checking vs Concolic Testing

	Model checking	Concolic testing
Analysis approach	Monolithic (i.e., whole analysis should be completed)	Incremental (i.e., analysis results are accumulated step-by-step) - Anytime algorithm
Compositional analysis	No	Yes (analysis of each symbolic execution path is **independent** from each other)
Accuracy	Very high	Very high (per given assert statements) - Known as path model checker
Explicit test inputs	Not generated	Generated
Requires abstraction	Yes	No
Memory consumption	Very high	Low
CPU time consumption	Very high	Very high
External binary library handling	None	Partial
Debugging support	**Limited (except a counter example generated)**	**Fully supported (you can freely use gdb or printf to analyze each concrete execution)**
Scalability	**Very limited**	**Large**

Various Automated SW Analysis Techniques Have Its Own Pros/Cons and Its Best Uses !!!

https://www.escoffier.edu/blog/culinary-arts/different-knives-and-the-best-uses-for-each/

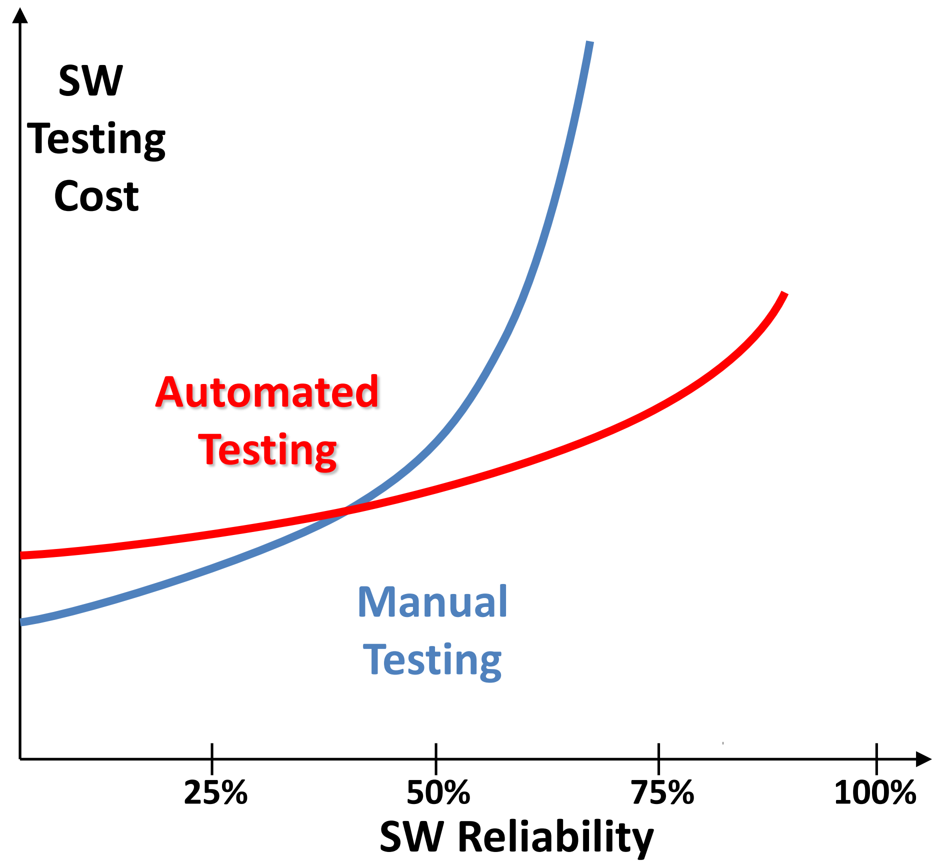
Future Direction

- Tool support will be strengthened for automated SW analysis
  - Ex. CBMC, BLAST, CREST, KLEE, and Microsoft PEX
  - Automated SW analysis will be performed routinely like GCC
  - Labor-intensive SW analysis => automated SW analysis by few experts
- Supports for concurrency analysis
  - Deadlock/livelock detection
  - Data-race detection
- Less user input, more analysis result and less false alarm
  - Fully automatic C++ syntax & type check (1980s)
  - (semi) automatic null-pointer dereference check (2000s)
  - (semi) automatic user-given assertion check (2020s)
  - (semi) automatic debugging (2030s)

Conclusion

- Automated concolic testing is effective and efficient for testing industrial embedded software including vehicle domain as well as consumer electronics domain

- Successful application of automated testing techniques requires expertise of human engineers



Traditional testing

- Manual TC gen
- Testing main scenarios
- System-level testing
- Small # of TCs

Concolic testing

- Automated TC gen
- Testing exceptional scenarios
- Unit-level testing
- Large # of TCs