CS 575
Software Testing and Analysis

Ozyegin University
Graduate School of Engineering

Potential Project Topics and Examples

Hasan Sözer
hasan.sozer@ozyegin.edu.tr

Review of Potential Project Topics

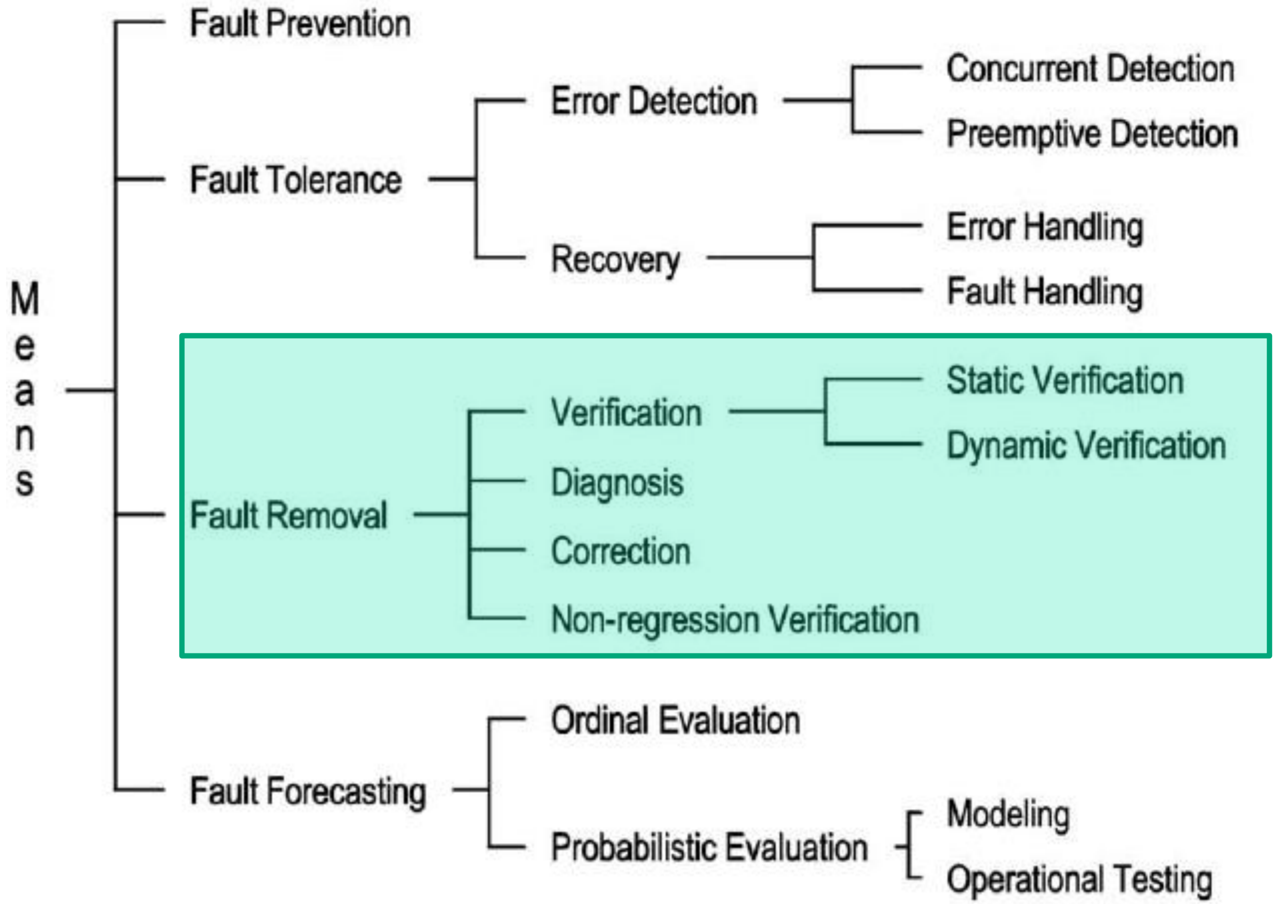
- Methods, tools, techniques related to Software Testing and Analysis
- Also a short **overview** of the subject material

- Towards determining a project topic and scope
  - … and preparing a **project proposal**

Proposal Contents

- Group members
  - if teamwork is considered
- Application domain
  - embedded, Web, mobile, stand-alone application, etc.
- The problem being addressed
  - measuring reliability, detecting errors, diagnosing faults, etc.
- The proposed solution approach
  - type of methods, techniques, tools considered
- Deliverables
  - a framework, integrated tool-set, evaluation report, etc.

Scope



[Avizienis 2004]

Outline

- Static Code Analysis
- Model-Based Testing
- Combinatorial Testing
- Concolic Testing
- Spectrum-based Fault Localization
- Mutation Testing
- Test Automation
  - Mobile Applications
  - Web Applications

New Ideas Welcome!

- A new application domain
- Specific types of faults, errors, failures
- Different concerns
  - Maintenance of test cases, focus on user-perceived failures, etc.

- Our discussion is just to inspire you
- Possible topics are not limited to the discussed examples

Static Code Analysis

- Analyzing source code without executing it
- Finding potential faults

- Bug Patterns
- Programming Rules
- Scalable but subject to false positives

- Extending tools with custom rules?
- Automatically filtering out false positives?

Static Code Analysis Tools

- Findbugs
- PMD
- Klocwork (commercial)
- CppCheck
- Frama-C
- Clang

Analysis for Program Slicing

- Focus on a variable that causes the failure
- Slice the program to filter out the irrelevant parts
- Makes it easier to debug the program

```
Pass = 0 ;
Fail = 0 ;
Count = 0 ;
while (!eof()) {
TotalMarks=0;
scanf("%d",Marks);
if (Marks >= 40)
Pass = Pass + 1;
if (Marks < 40)
Fail = Fail + 1;
Count = Count + 1;
TotalMarks = TotalMarks+Marks ;
}
printf("Out of %d, %d passed and %d failed\n",Count,Pass,Fail) ;
average = TotalMarks/Count;
/* This is the point of interest */
printf("The average was %d\n",average) ;
PassRate = Pass/Count*100 ;
printf("This is a pass rate of %d\n",PassRate) ;
```

```
while (!eof()) {
TotalMarks=0;
scanf("%d",Marks);
Count = Count + 1;
TotalMarks = TotalMarks+Marks;
}
average = TotalMarks/Count;
printf("The average was %d\n",average) ;
```

Example by Mark Harman

Backward vs. Forward Slicing

- Forward Slicing: include lines that are affected by the variable in the rest of the program
- Makes it easier to maintain the program

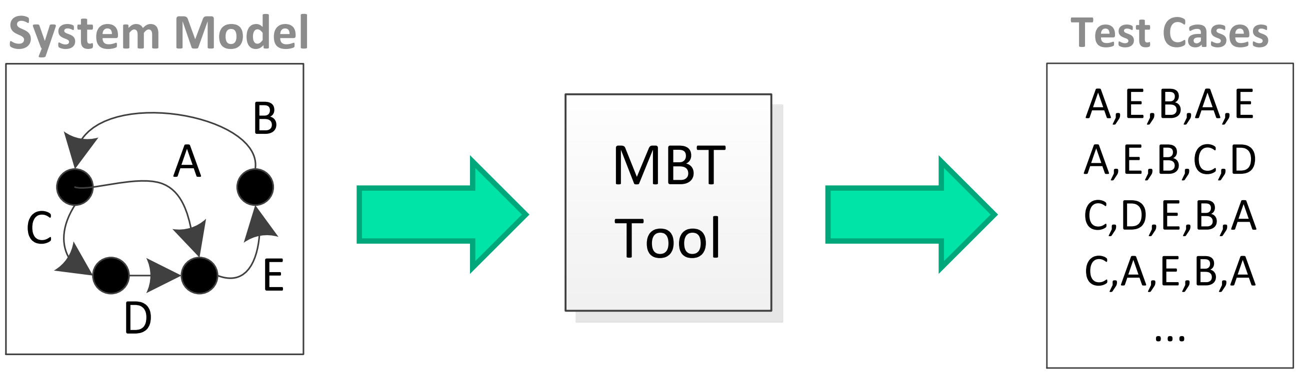
```
x = 1; /* considering changing this line */
y = 3;
p = x + y ;
z = y -2 ;
if(p==0)
r++ ;
```

```
/* Change to first line will affect */
p = x + y ;
if(p==0)
r++ ;
```

Example by Mark Harman

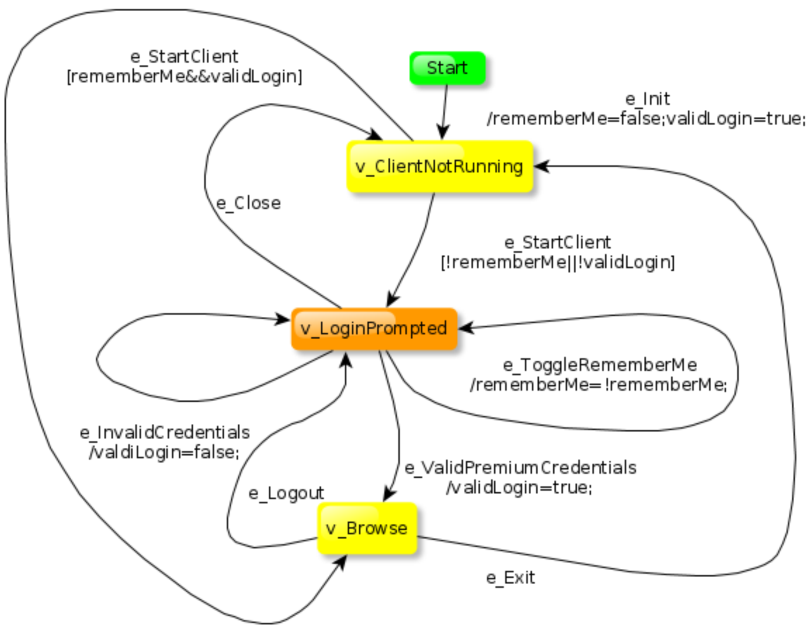
Model-Based Testing

- Automatically generating test cases based on a model of the system



Model-Based Testing Tools

- GraphWalker: Generates Junit test cases
- MaTeLo (commercial)



- Learning the usage model
- Reflecting usage profile
- Analyze coverage

- http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html
- http://robertvbinder.com/open-source-tools-for-model-based-testing/

Combinatorial Testing

- Exploring different combinations of parameters and configuration parameters
- Systematically generate combinations to be tested
  - e.g., IE on Vista, IE on XP, Firefox on Vista, ...
- Rationale: Test cases should be varied and include possible “corner cases”

Pairwise testing

- Generate combinations that efficiently **cover all pairs (triples,…) of classes**

- Rationale: most failures are triggered by single values or combinations of a few values. Covering pairs (triples,…) reduces the number of test cases, but reveals most faults

Example for Pairwise testing

based on the slides of M. Pezze and M. Young

- 432 (3x4x3x4x3) test cases if we consider all combinations

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	

Pairwise combinations: 17 test cases

Language	Color	Display Mode	Fonts	Screen Size
English	Monochrome	Full-graphics	Minimal	Hand-held
English	Color-map	Text-only	Standard	Full-size
English	16-bit	Limited-bandwidth	-	Full-size
English	True-color	Text-only	Document-loaded	Laptop
French	Monochrome	Limited-bandwidth	Standard	Laptop
French	Color-map	Full-graphics	Document-loaded	Full-size
French	16-bit	Text-only	Minimal	-
French	True-color	-	-	Hand-held
Spanish	Monochrome	-	Document-loaded	Full-size
Spanish	Color-map	Limited-bandwidth	Minimal	Hand-held
Spanish	16-bit	Full-graphics	Standard	Laptop
Spanish	True-color	Text-only	-	Hand-held
Portuguese	-	-	Monochrome	Text-only
Portuguese	Color-map	-	Minimal	Laptop
Portuguese	16-bit	Limited-bandwidth	Document-loaded	Hand-held
Portuguese	True-color	Full-graphics	Minimal	Full-size
Portuguese	True-color	Limited-bandwidth	Standard	Hand-held

Combinatorial Testing Tools

- http://www.pairwise.org/tools.asp
- Tcase
- Pict
- ...

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 Darko Marinov and Gul 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		$p=p_0$, $x=x_0$	

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
p → NULL, x=236	$p=p_0$, $x=x_0$	

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
		$x_0>0$
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;
}
```

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

Concrete Execution

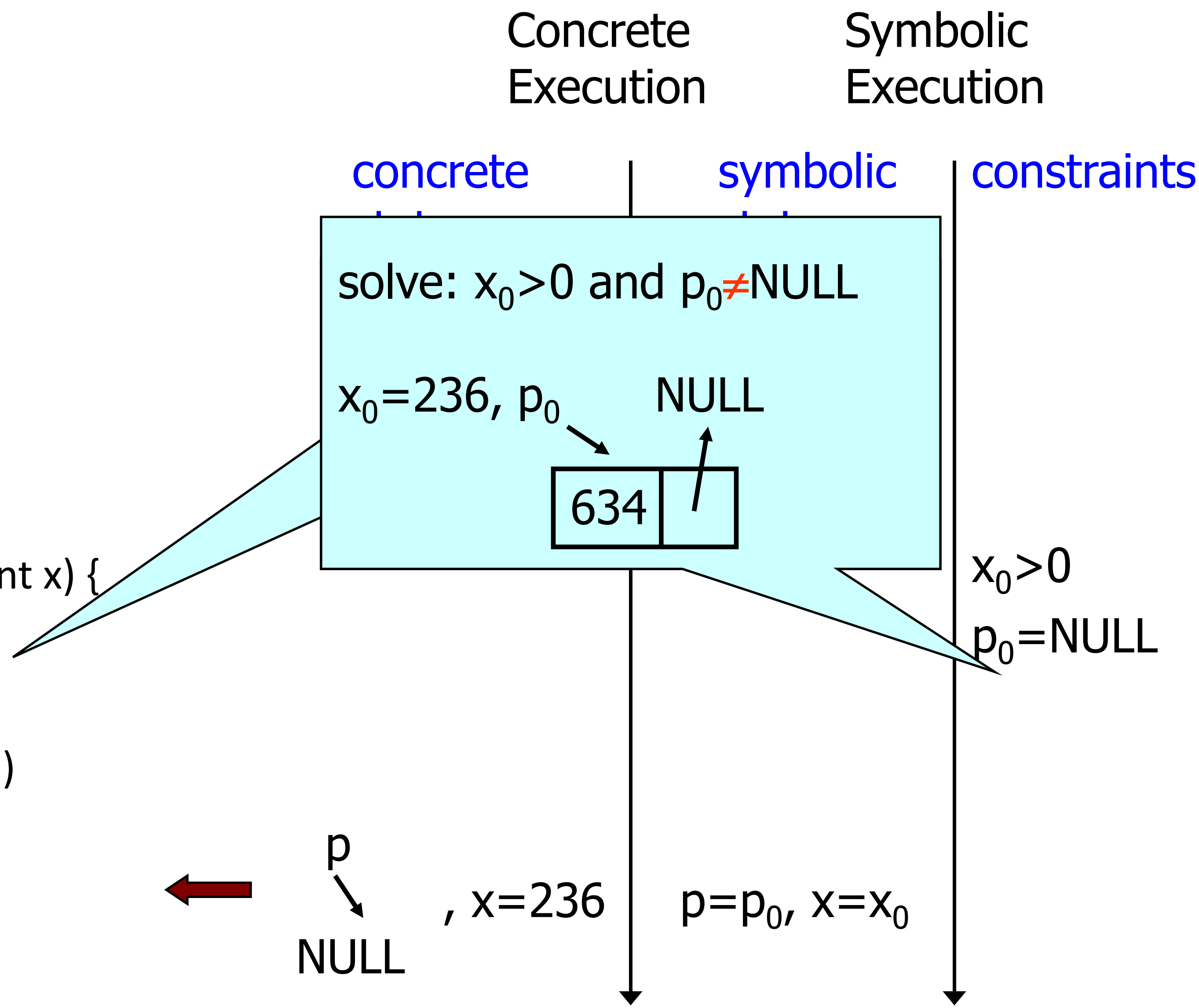
Symbolic Execution

```
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	symbolic	constraints
p → NULL, x=236	$p=p_0$, $x=x_0$	$x_0>0$ p_0=NULL

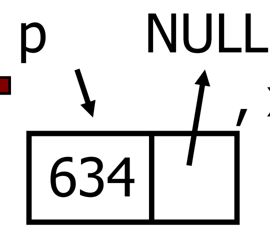
solve: $x_0>0$ and $p_0 \neq$ NULL

```
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

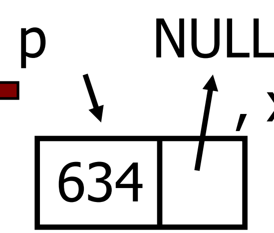
```c
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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 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;
}
```

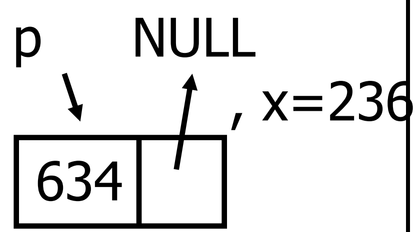
concrete state	symbolic state	constraints
 , x=236	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$

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

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

constraints

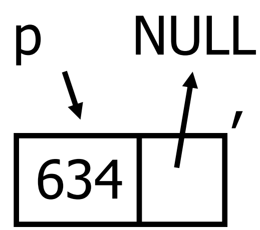
$x_0>0$
$p_0 \neq NULL$

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

$x_0>0$

$p_0 \neq NULL$

$2x_0+1 \neq v_0$

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

Concrete Execution

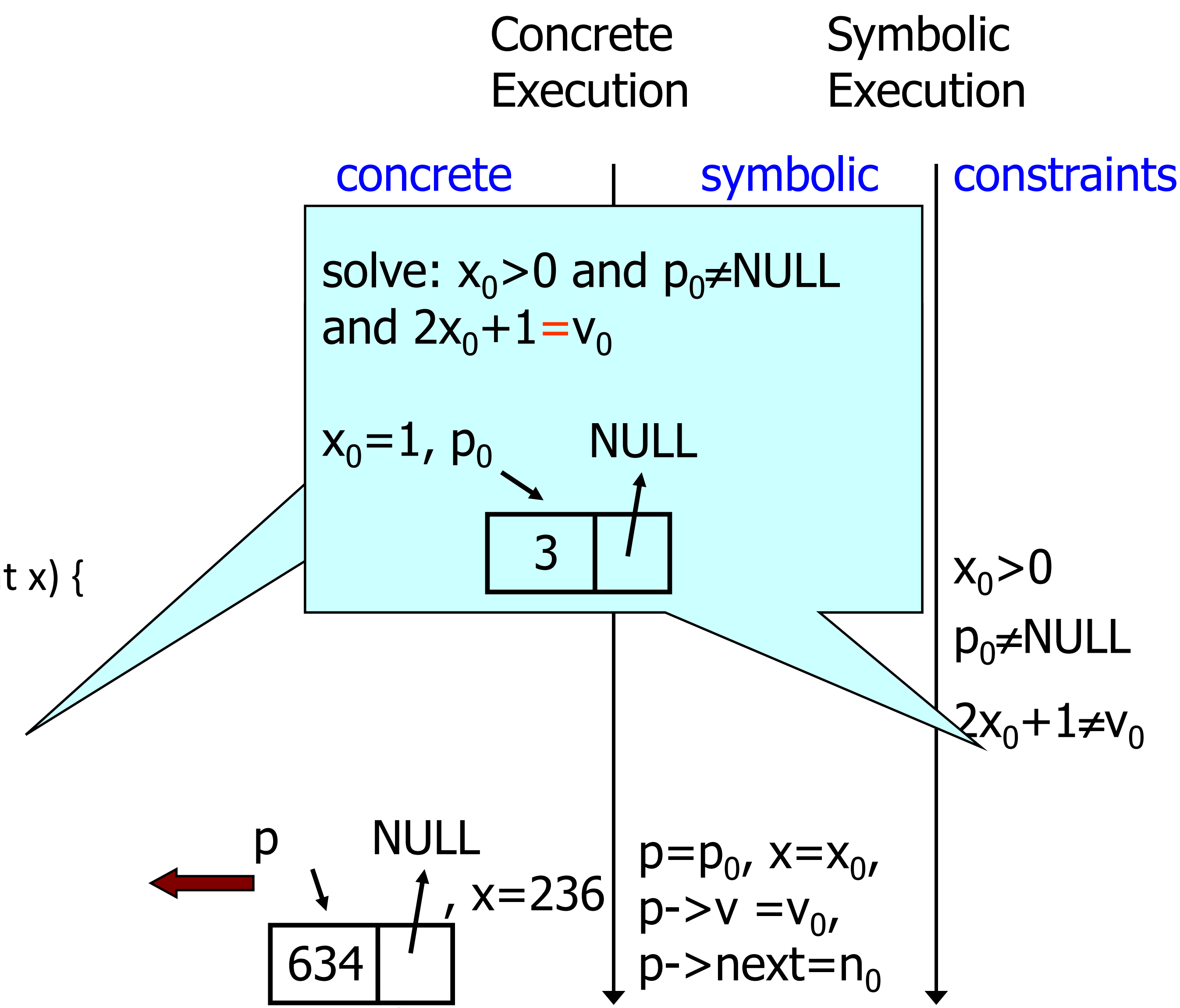
Symbolic Execution

```
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;
}
```

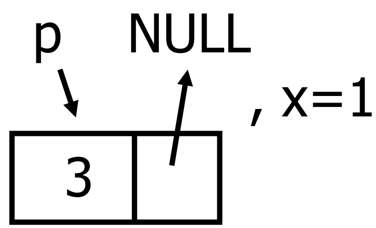
concrete	symbolic	constraints
p → [634 | NULL], x=236	$p=p_0$, $x=x_0$, $p\text{->}v=v_0$, $p\text{->}next=n_0$	$x_0>0$ $p_0 \neq NULL$ $2x_0+1 \neq v_0$

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

```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;
}
```

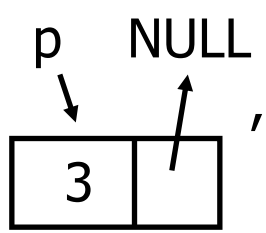


```c
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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

```
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

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

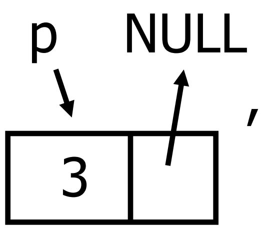
$x_0>0$

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

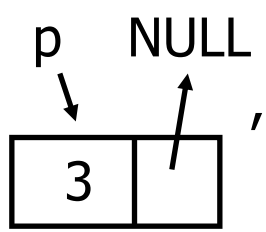
$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

constraints

$x_0 > 0$

$p_0 \neq NULL$

```
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;
}
```

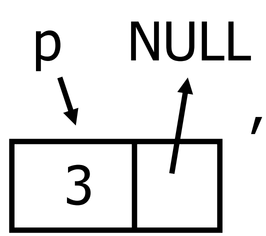
Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
 , x=1	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$ $p_0 \neq NULL$ $2x_0+1=v_0$

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

$x_0>0$

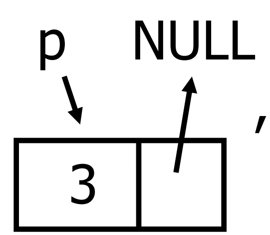
$p_0 \neq NULL$

$2x_0+1=v_0$

$n_0 \neq p_0$

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

```
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;
}
```

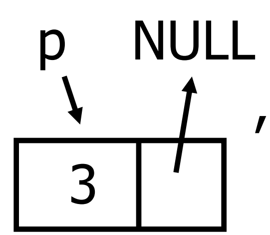
Concrete Execution

Symbolic Execution

concrete state | symbolic state | constraints

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

.



, x=1

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

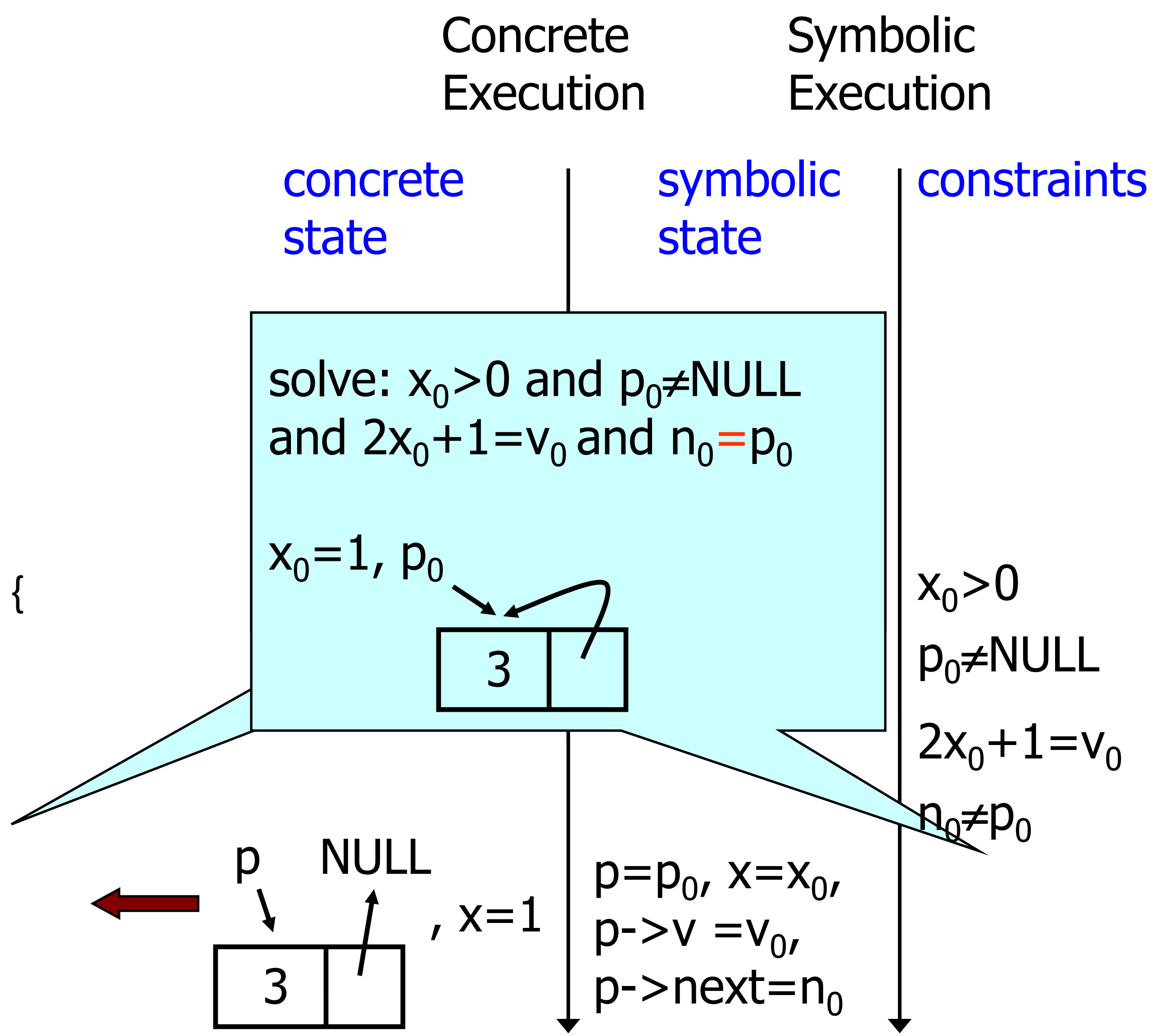
$x_0>0$

$p_0 \neq NULL$

$2x_0+1=v_0$

$n_0 \neq p_0$

```
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

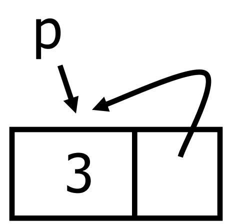
Symbolic Execution

```
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

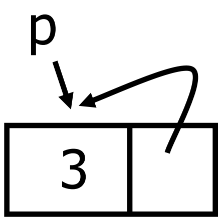
$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

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

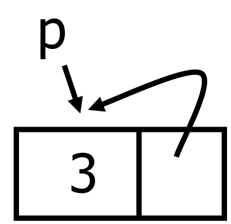
symbolic state

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

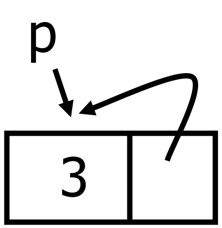
constraints

$x_0>0$

```
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;
}
```

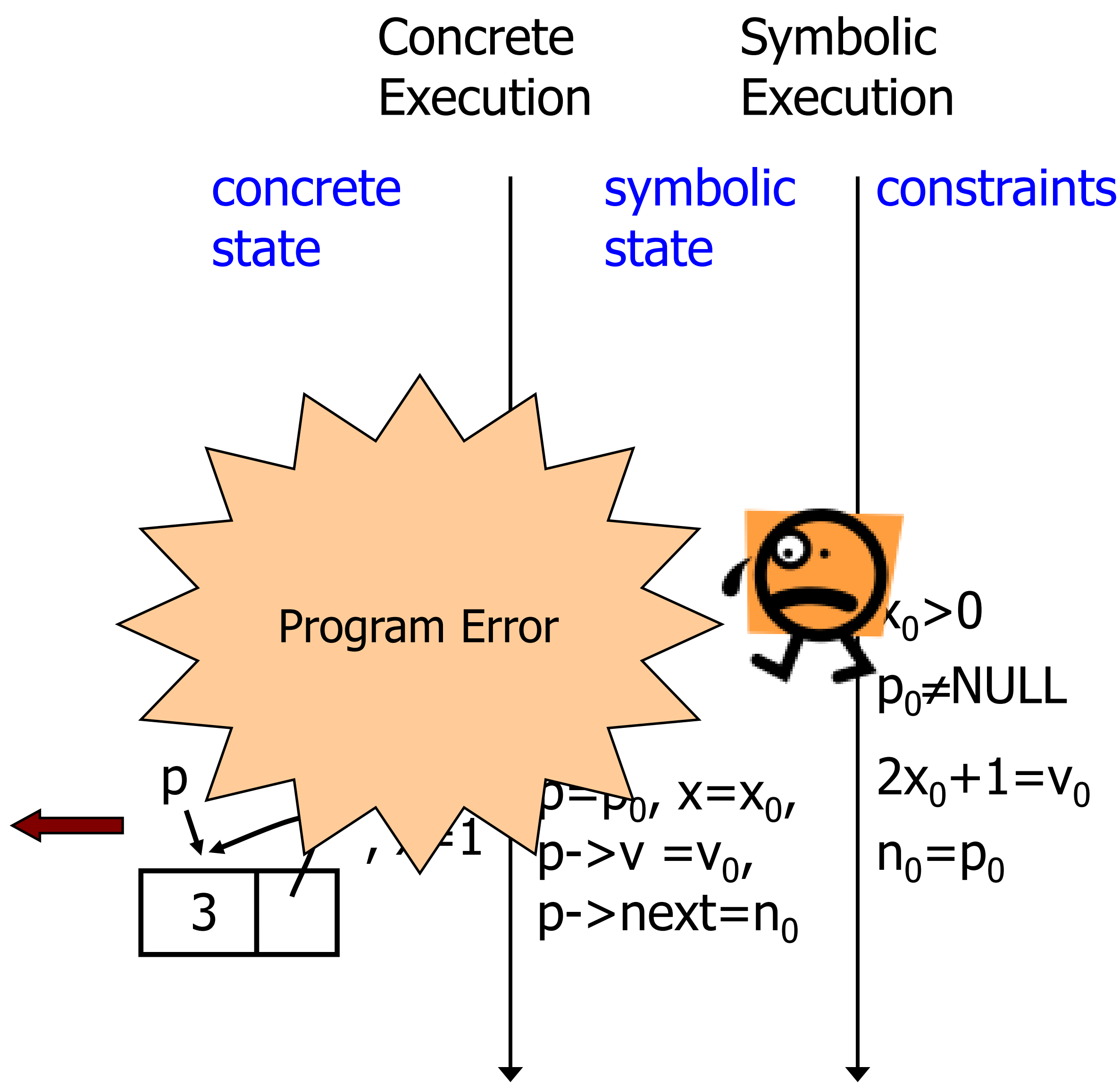
Concrete Execution		Symbolic Execution
concrete state	symbolic state	constraints
 , x=1	$p=p_0$, $x=x_0$, $p\text{->}v=v_0$, $p\text{->}next=n_0$	$x_0>0$ $p_0 \neq NULL$

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;
}
```

```
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;
}
```



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.

Spectrum-based Fault Localization

- Helping developers for debugging
- Correlating execution traces with test results
  - Which program locations are executed?
  - Which components are involved?
  - Which branches are taken?
  - …
- Example slides follow for so called **block-hit spectra**
  - Adopted slides of Rui Abreu, Peter Zoeteweij and Arjan van Gemund

Block / function hit spectra

Function hit spectrum

1: function *i* called
0: function *i* not called

x_1	x_2	…	x_i	…	x_n

Block hit spectrum

1: block *i* executed
0: block *i* not executed

Block:
- C statement (compound stmt)
- cases of a switch statement

Collected Data

n blocks

m cases

x_{11}	x_{12}	…	x_{1n}
x_{21}	x_{22}	…	x_{2n}
…	…	…	…
x_{m1}	x_{m2}	…	x_{mn}

e_1
e_2
…
e_m

Collected Data

x_{11}	x_{12}	…	x_{1n}
x_{21}	x_{22}	…	x_{2n}
…	…	…	…
x_{m1}	x_{m2}	…	x_{mn}

e_1
e_2
…
e_m

Row *i*: the blocks that are executed in case *i*

Collected Data

x_{11}	x_{12}	…	x_{1n}		e_1
x_{21}	x_{22}	…	x_{2n}		e_2
…	…	…	…		…
x_{m1}	x_{m2}	…	x_{mn}		e_m

Column *j* : the test cases in which block *j* was executed

Collected Data

x_{11}	x_{12}	…	x_{1n}
x_{21}	x_{22}	…	x_{2n}
…	…	…	…
x_{m1}	x_{m2}	…	x_{mn}

e_1
e_2
…
e_m

$e_i=1$: error in the i-th test
$e_i=0$: no error in the i-th test

Calculation of Similarity

block *j*

x_{11}	x_{12}	…	x_{1n}
x_{21}	x_{22}	…	x_{2n}
…	…	…	…
x_{m1}	x_{m2}	…	x_{mn}

error vector

e_1
e_2
…
e_m

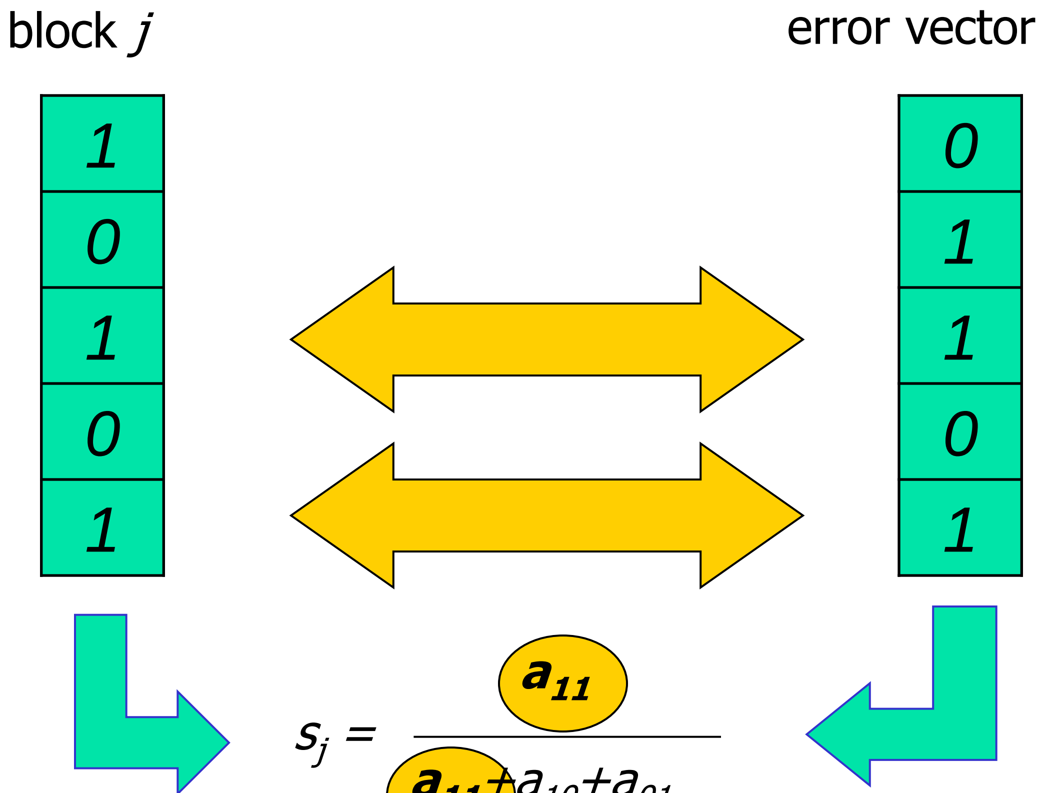
similarity s_j

Jaccard Similarity Coefficient

block j	error vector
1	0
0	1
1	1
0	0
1	1

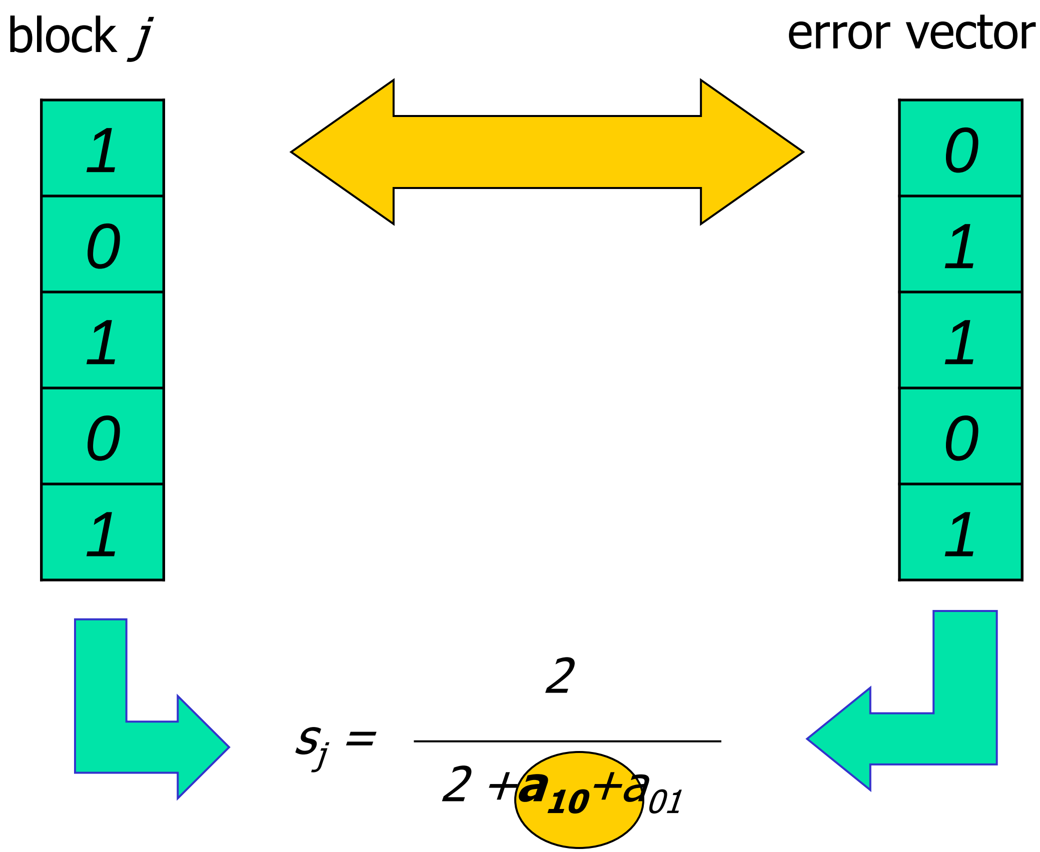
$$s_j = \frac{a_{11}}{a_{11}+a_{10}+a_{01}}$$

Jaccard Similarity Coefficient

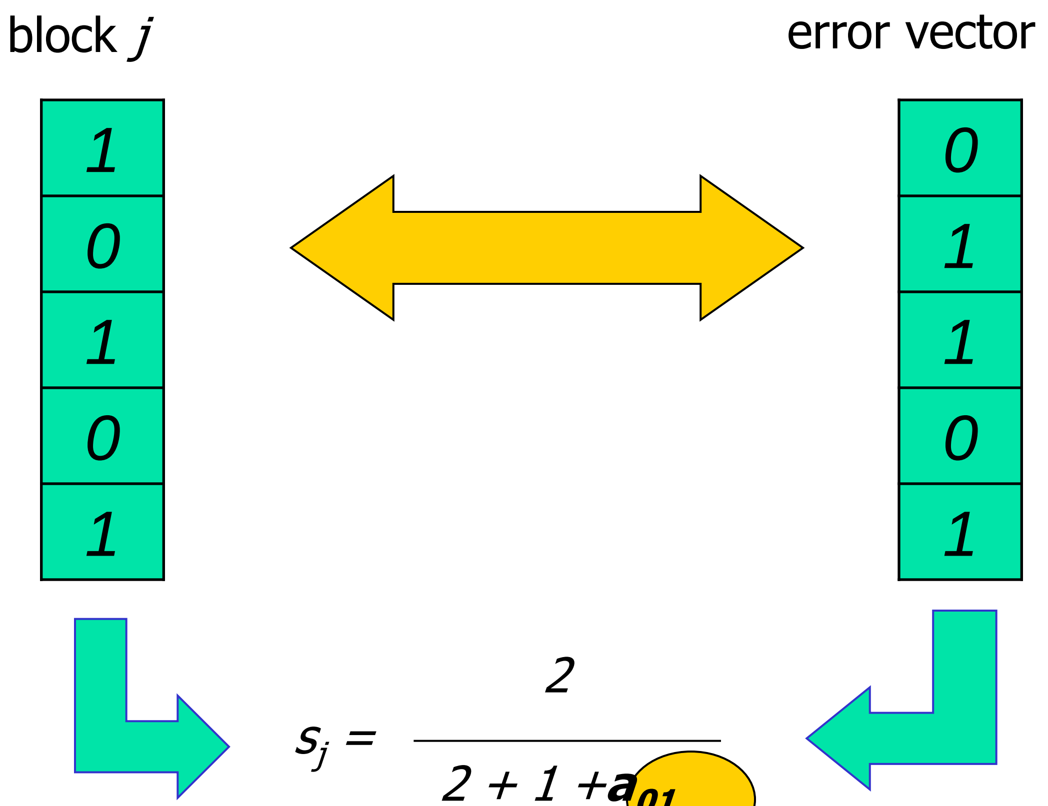


$$s_j = \frac{a_{11}}{a_{11}+a_{10}+a_{01}}$$

Jaccard Similarity Coefficient



Jaccard Similarity Coefficient



$$s_j = \frac{2}{2 + 1 + a_{01}}$$

Jaccard Similarity Coefficient

block *j*	error vector
1	*0*
0	*1*
1	*1*
0	*0*
1	*1*

$$s_j = \frac{2}{2 + 1 + 1}$$

Similarity for Each Block

n blocks

error vector

m cases

$$
\begin{array}{|c|c|c|c|}
\hline
x_{11} & x_{12} & \dots & x_{1n} \\
\hline
x_{21} & x_{22} & \dots & x_{2n} \\
\hline
\dots & \dots & \dots & \dots \\
\hline
x_{m1} & x_{m2} & \dots & x_{mn} \\
\hline
s_1 & s_2 & \dots & s_n
\end{array}
\qquad
\begin{array}{|c|}
\hline
e_1 \\
\hline
e_2 \\
\hline
\dots \\
\hline
e_m \\
\hline
\end{array}
$$

The block with the highest s_i most likely contains the fault.

Example

n blocks

error vector

m cases

0	1	1	1	1	0	0
0	0	0	1	0	1	1
1	1	1	1	0	0	0
0	0	0	0	0	0	0
1	1	0	1	1	0	1

0
1
1
0
1

Example

n blocks

m cases

0	1	1	1	1	0	0
0	0	0	1	0	1	1
1	1	1	1	0	0	0
0	0	0	0	0	0	0
1	1	0	1	1	0	1
⅔	½	¼	¾	¼	⅓	⅔

error vector

0
1
1
0
1

Example

n blocks

error vector

m cases

0	1	1	1	1	0	0
0	0	0	1	0	1	1
1	1	1	1	0	0	0
0	0	0	0	0	0	0
1	1	0	1	1	0	1
$\frac{2}{3}$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{3}{4}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{2}{3}$

0
1
1
0
1

Fault Localization Tools

- Zoltar
- Gzoltar: Eclipse plug-in for Java Applications
  - Paper: W. E. Wong, R. Gao, Y. Li, R. Abreu and F. Wotawa, "A Survey on Software Fault Localization," in *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707-740, Aug. 1 2016.
- Application to new case studies?
- New type of applications?
- A comparison of tools? Similarity metrics?

Mutation Testing

- Testing your tests by injecting faults to your program

- Assume, we have a program P
- Makes 100 different copies of P
- A bug is injected to each copy
- Run all the tests on each of the 100 copies with bugs
- Let’s say, all the tests passed for 80 of them
- What would you think about your tests then?

Mutants

- A **mutant** is a copy of a program with a mutation
- A **mutation** is a syntactic change (a seeded bug)
  - e.g., change *(i < 0)* to *(i <= 0)*

- A mutant is **killed** if it fails on at least one test case

- If many mutants are killed, infer that the test suite is also effective at finding real bugs

Mutation Testing Assumptions

- Competent programmer hypothesis:
  - Programs are nearly correct
    - Real faults are small variations from the correct program
    - => Mutants are reasonable models of real buggy programs

- Coupling effect hypothesis:
  - Tests that find simple faults also find more complex faults
    - Even if mutants are not perfect representatives of real faults, a test suite that kills mutants is good at finding real faults too

Mutation Testing Tools

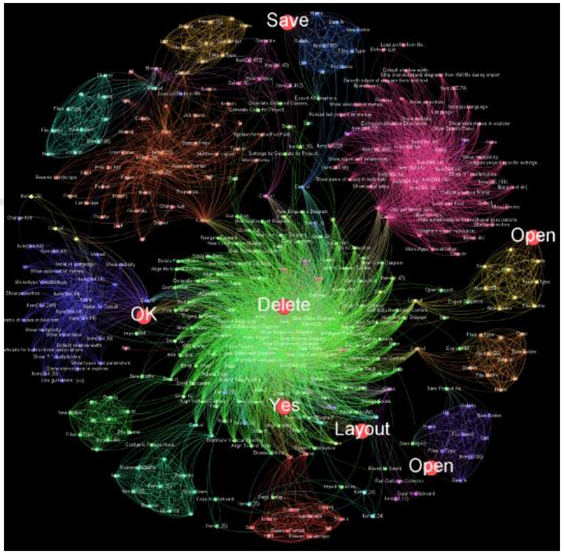
- MuClipse: open-source tool for Java
- Judy
- PIT

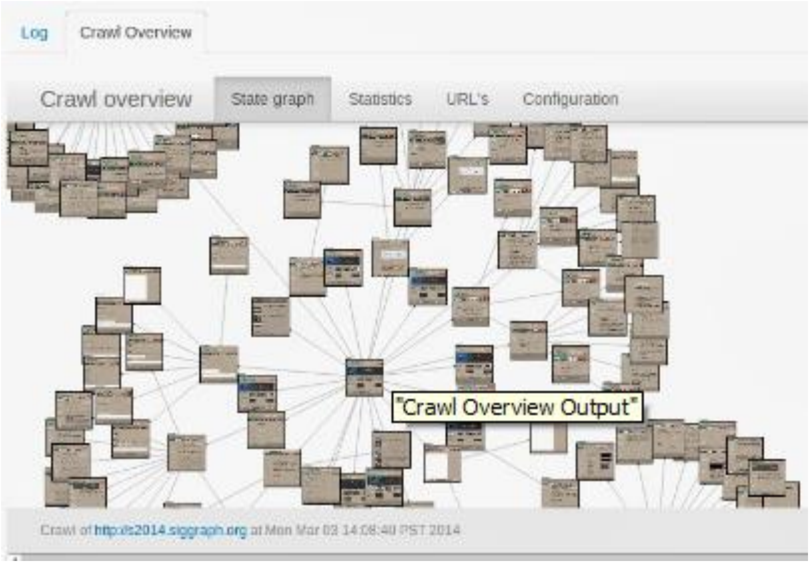
  - Paper: Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. IEEE Trans. Softw. Eng. 37, 5 (September 2011), 649-678

- Application to new case studies?
- A comparison of tools?
- A new mutation testing tool with new type of operators?

Test Automation

- Exploring user interfaces
- Automated Execution of test cases
- GUI
  - Abbot for Java
  - Eggplant
  - Ranorex for .NET
  - Sikuli
- Web
  - Crawljax
- Mobile
  - Android GUITAR
  - Calabash



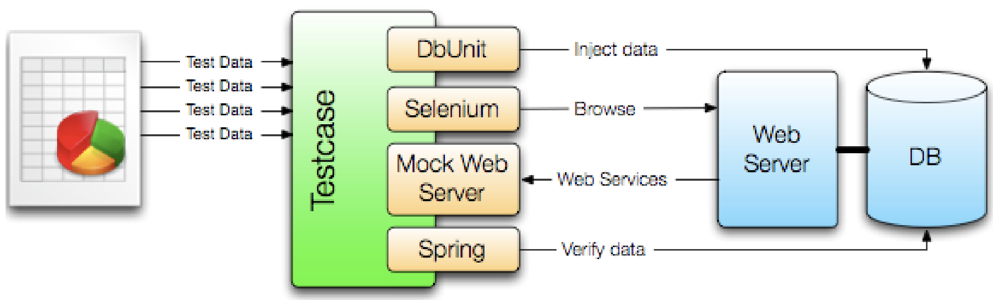


Test Automation Tools for Mobile Applications

- https://github.com/RobotiumTech/robotium
- https://google.github.io/android-testing-support-library/docs/uiautomator/
- https://google.github.io/android-testing-support-library/docs/espresso/
- https://robotium.com/products/robotium-recorder
- https://developer.android.com/studio/test/espresso-test-recorder.html
- http://www.ranorex.com/mobile-automation-testing/android-test-automation.html
- http://developer.android.com/tools/help/monkey.html

Data Driven Acceptance Tests

- DDSteps (http://ddsteps.sourceforge.net/)



- FitNesse

fit.ActionFixture		
start	eg.Page	
enter	location	http://google.com
check	title	Google
enter	link	Jobs
check	title	About Google
enter	link	Press
enter	link	Review
check	title	Google Press Room

Questions so far..