

Combinatorial Design-based Test Generation

W. Eric Wong

Department of Computer Science
The University of Texas at Dallas
<http://www.utdallas.edu/~ewong>



1

1

Speaker Biographical Sketch

- Professor & Founding Director
Advanced Research for Software Testing & Quality Assurance
Department of Computer Science
University of Texas at Dallas
- Guest Researcher
Computer Security Division
National Institute of Standards and Technology (NIST)
- Editor-in-Chief, *IEEE Transactions on Reliability*
- Engineer of the Year, 2014, *IEEE Reliability Society*
- Vice President, *IEEE Reliability Society* (2012–2015)
- Secretary, *ACM SIGAPP* (Special Interest Group on Applied Computing)
2009 – 2013
- Steering Committee Chair of the QRS conference (*IEEE International Conference on Software Quality, Reliability and Security*) & IWPD (*IEEE International Workshop on Program Debugging*)



2

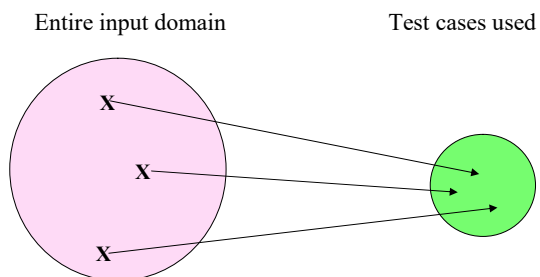
Here Is the Reason

Execution
≠
Fault Detection

3

Test Case Generation

- When to stop testing?
 - Has a significant impact on *quality* and
 - What do we want to achieve?
 - Here is an ideal solution



where x are failure causing inputs

4

Do You Know . . .

- How many failure causing inputs?
- Which one is a failure causing input?

What should we do?

5

Challenges to be Overcome

- Testing individual component is not enough
- We should also focus on interactions on different components

Many system components are tested individually, but often unexpected interactions between components cause failures.

How to do it



Combinatorial Testing

6

Combinatorial Testing – What Is It ?

- CT is an effective test generation technique focuses on testing various combinations among different components of a system
 - Based on a classic mathematics topic – *Combinatorics* (concerning the study of finite or countable discrete structures)
 - N. L. Biggs, “The Root of Combinatorics,” *Historia Mathematica*, 6(2): 109-136, May 1979
 - J. Riordan, “Introduction to Combinatorial Analysis,” *Dover Publications*, September 1980

A Formal Engineering Method
Rather Than an Ad-Hoc Approach

7

Combinatorial Testing: Example I (1)

Input Parameters	Values			
Parameter 1	800	900		
Parameter 2	start	stop		
Parameter 3	on	off	idle	
Parameter 4	up	down	forward	backward

Number of exhaustive combinations among all parameters = $2 \times 2 \times 3 \times 4 = 48$ ← Too many

- Focus on all the combinations among some (e.g., 2) parameters

	Parameter 1	Parameter 2	Parameter 3	Parameter 4
t_1	800	Start	?	?
t_2	800	Stop	?	?
t_3	900	Start	?	?
t_4	900	Stop	?	?
t_5	?	?	on	Up
t_6	?	?	on	Down
t_7	?	?	on	Forward
t_8	?	?	on	Forward
.
.
.

We can reduce the number of test cases by choosing parameter values intelligently.

8

Combinatorial Testing: Example I (2)

- Number of pairwise combinations
(all combinations between 2 parameters)
 - Parameter 1 and Parameter 2: $4 (2 \times 2)$
 - Parameter 1 and Parameter 3: $6 (2 \times 3)$
 - Parameter 1 and Parameter 4: $8 (2 \times 4)$
 - Parameter 2 and Parameter 3: $6 (2 \times 3)$
 - Parameter 2 and Parameter 4: $8 (2 \times 4)$
 - Parameter 3 and Parameter 4: $12 (3 \times 4)$
- Do we really need 44 ($4 + 6 + 8 + 6 + 8 + 12$) test cases ?

Do We Have Any Tool Support ?

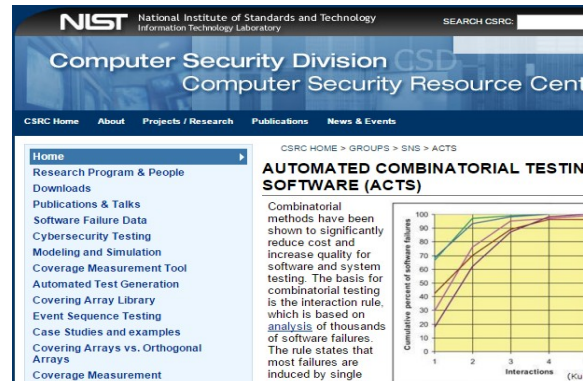
- ACTS is one of the tools which support CT

The screenshot shows the ACTS - ACTS Main Window. On the left is the 'System View' tree, which includes a 'Root Node' and several sub-nodes like 'B07', 'B08', 'B09', 'B10', 'B11', 'B12', 'B13', 'B14', 'B15', 'B16', 'B17', 'B18', 'B19', 'B20', 'B21', 'B22', 'B23', 'B24', 'B25', 'B26', 'B27', 'B28', 'B29', 'B30', 'B31', 'B32', 'B33', 'B34', 'Relations'. On the right is the 'Test Result' table, which displays a grid of test cases. The table has columns for 'Test Result' and 'Statistics'. The 'Test Result' column contains a grid of values (00, 01, 02, 04, 08, 10, 20, 40, 80) for each test case. The 'Statistics' column contains a grid of values (00, 01, 02, 04, 08, 10, 20, 40, 80) for each test case. The table is titled 'Test Result' and 'Statistics'.

Test Result	Statistics
00	00
01	01
02	02
04	04
08	08
10	10
20	20
40	40
80	80

Combinatorial Testing at NIST

- NIST (National Institute of Standards and Technology)



- D. R. Kuhn, D. R. Wallace, A. M. Gallo, "Software Fault Interactions and Implications for Software Testing," *IEEE Transactions on Software Engineering*, 30(6):418-421, June 2004
- J. D. Hagar, T. L. Wissink, and D. R. Kuhn, "Introducing Combinatorial Testing in a Large Organization," *IEEE Computer*, 48(4): 64-72, April 2015

Combinatorial Testing: Example I (3)

- Using ACTS, we only need 12 test cases to cover all pairwise combinations

Test Case	p1	p2	p3	p4
1	900	stop	on	up
2	800	start	off	up
3	900	start	idle	up
4	800	stop	on	down
5	900	start	off	down
6	800	stop	idle	down
7	900	start	on	forward
8	800	stop	off	forward
9	900	stop	idle	forward
10	900	start	on	backward
11	800	stop	off	backward
12	800	stop	idle	backward

Combinatorial Testing: Example II (1)

- Suppose a system has the following three parameters a , b , and c . Each parameter has three possible values:

a	b	c
1	4	7
2	5	8
3	6	9

- There are 27 pairwise combinations

Combinatorial Testing: Example II (2)

A sample system		
a	b	c
1	4	7
2	5	8
3	6	9

An **inefficient** test set (as shown on the right) can cover only 15 pairs:

- $(a$ and $b)$: 9 pairs
- $(a$ and $c)$: 3 pairs
- $(b$ and $c)$: 3 pairs



An inefficient test set			
	a	b	c
t_1	1	4	7
t_2	1	5	7
t_3	1	6	7
t_4	2	4	7
t_5	2	5	7
t_6	2	6	7
t_7	3	4	7
t_8	3	5	7
t_9	3	6	7

Combinatorial Testing: Example II (3)

A sample system		
<i>a</i>	<i>b</i>	<i>c</i>
1	4	7
2	5	8
3	6	9

An **efficient** test set (as shown on the right) can cover 27 pairs:

- (*a* and *b*): 9 pairs
- (*a* and *c*): 9 pairs
- (*b* and *c*): 9 pairs



An efficient test set			
	<i>a</i>	<i>b</i>	<i>c</i>
<i>t</i> ₁	1	4	8
<i>t</i> ₂	1	5	9
<i>t</i> ₃	1	6	7
<i>t</i> ₄	2	4	9
<i>t</i> ₅	2	5	7
<i>t</i> ₆	2	6	8
<i>t</i> ₇	3	4	7
<i>t</i> ₈	3	5	8
<i>t</i> ₉	3	6	9

15

Fun Facts

- Study of Mozilla web browser found 70% of defects with 2-way coverage; ~90% with 3-way; and 95% with 4-way.
[Kuhn *et. al.*, 2002]
- Combinatorial testing of 109 software-controlled medical devices recalled by US FDA uncovered 97% of flaws with 2-way coverage; and only 3 required higher than 2.
[Kuhn *et. al.*, 2004]

16

Two Parts of This Tutorial

- Application of *black-box requirement-based* combinatorial testing to two real-life industry applications
 - Results and lessons learned
- Extension of combinatorial testing to a *white-box structure-based setting*
 - Combinatorial Decision Testing

Part I

Applying Black-box Combinatorial Testing in Industrial Settings

X. Li, R. Gao, W. E. Wong, C. Yang, and D. Li, "Applying Combinatorial Testing in Industrial Settings," in *Proceedings of IEEE International Conference on Software Quality, Reliability, and Security*, pp. 53-60, Vienna, Austria, August 2016

Two Industrial Projects



Subway Control System

Test whether the dashboard correctly presents the data based on different input values



Kylin Linux

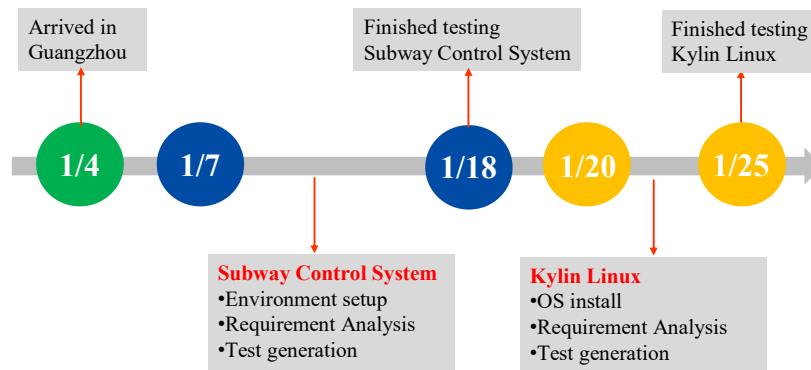
An OS developed by National University of Defense Technology since 2001



19

Schedule

Timeline

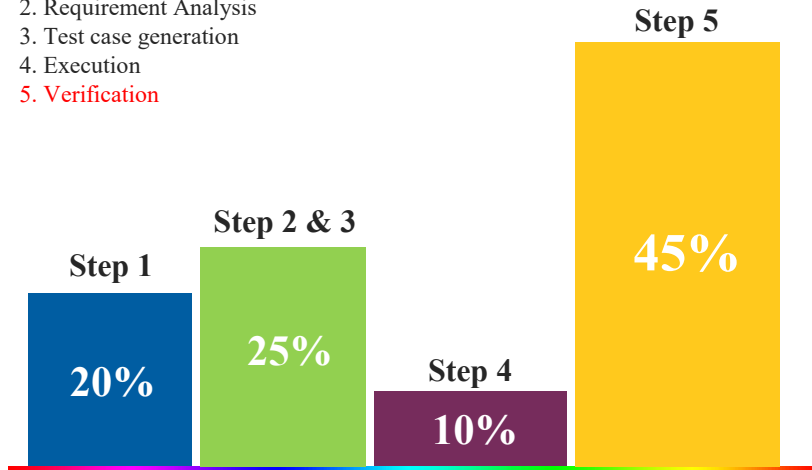


20

Effort Distribution

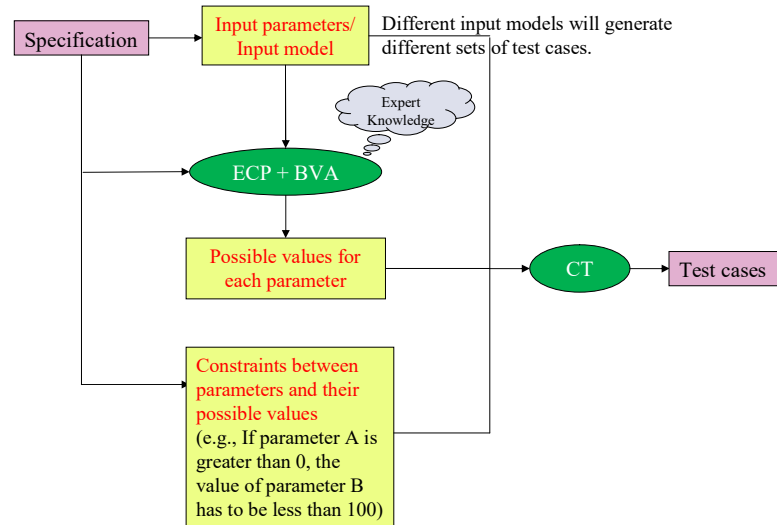
Major Steps:

1. Environment set up
2. Requirement Analysis
3. Test case generation
4. Execution
5. Verification



21

Test Generation – CT + ECP + BVA (1)



22

Test Generation – CT + ECP + BVA (2)

- Equivalence Class Partitioning (ECP)
- Boundary Value Analysis (BVA)

Help us select possible values for each parameter

An Example:

Parameter: Pressure sensor

Input domain: 0 ~ 2500kpa

Equivalence classes:

{ 0 ~ 10kpa }, { 10 ~ 200kpa }, { 200 ~ 2490kpa },
{ 2490 ~ 2500kpa }, { 2500kpa+ }

Possible values: 0, 10, 200, 1250, 2500, 3000

The more possible values selected for each parameter,
the more test cases will be generated

Restriction from CEPREI: Generate no more than 100 test cases!!!

Case Study I:

Subway Control System

The GUI of the System (1)



25

The GUI of the System (2)



26

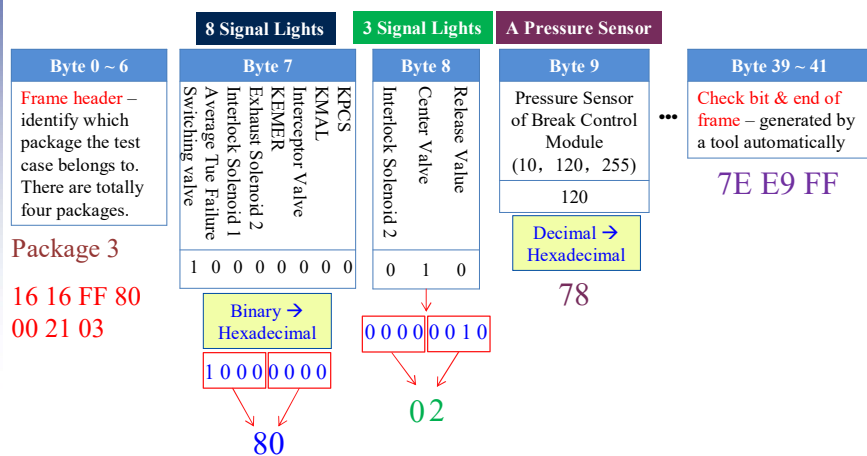
A Sample Test Case

1616FF80002103800278000078E7
03E703E703402004404001580258
02580258025802646458027EE9FF

How to interpret such a monster?

27

Decompose the Test Case



1616FF80002103800278000078E703E703E7034020044040015802580258025802646458027EE9FF

28

Four Data Packages

Package 1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Package 2

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Package 3

0.000

Package 4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Too many parameters!

Instead of selecting possible values for each parameter, we combine parameters into bytes and select possible values for each byte.

29

Select Possible Values for Each Byte (1)

Package 3 - Byte 7							
KPCS	KMAL	Interceptor Valve	KEMER	Exhaust Solenoid 2	Interlock Solenoid 1	Average Tue Failure	Switching Valve
1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0



Binary → Hexadecimal

Switching Valve	1000 0000	80
Average Tue Failure	0100 0000	40
Interlock Solenoid 1	0010 0000	20
Exhaust Solenoid 2	0001 0000	10
KEMER	0000 1000	08
Interceptor Valve	0000 0100	04
KMAL	0000 0010	02
LPCS	0000 0001	01
N/A	0000 0000	00

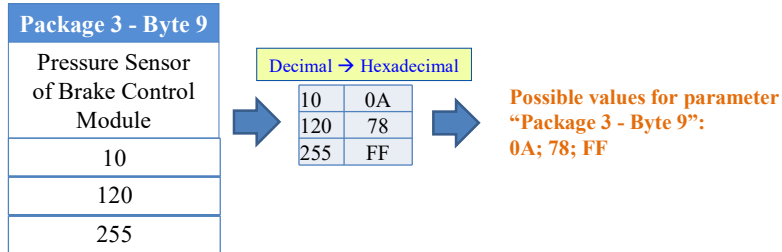


Possible values for parameter "Package 3 - Byte 7":

80; 40; 20; 10; 08; 04; 02; 01; 00

30

Select Possible Values for Each Byte (2)



31

Take Advantage of Tester's Experiences

Expert Knowledge can also help
select possible values for each byte

32

Select Possible Values for Each Byte (3)

Package 1 - Byte 20						
Tube 1	Tube 2	Cylinder 1	Cylinder 2	Valve 1	Valve 2	
1	0	0	0	0	0	
0	1	0	0	0	0	
0	0	1	0	0	0	
0	0	0	1	0	0	
0	0	0	0	1	0	
0	0	0	0	0	1	
0	0	0	0	0	0	
1	1	0	0	0	0	
0	0	1	1	0	0	
0	0	0	0	1	1	

Based on the *expert knowledge*, we also want to turn on the signal lights for Tubes 1 & 2, Cylinders 1 & 2, and Valves 1 & 2 at the same time

Binary → Hexadecimal

Tube 1	0010 0000	20
Tube 2	0001 0000	10
Cylinder 1	0000 1000	08
Cylinder 2	0000 0100	04
Valve 1	0000 0010	02
Valve 2	0000 0001	01
NA	0000 0000	00
Tube 1 & 2	0011 0000	30
Cylinder 1 & 2	0000 1100	0C
Valve 1 & 2	0000 0011	03

Possible values for "Package 1 - Byte 20":
20; 10; 08; 04; 02; 01; 00 **30; 0C; 03**

33

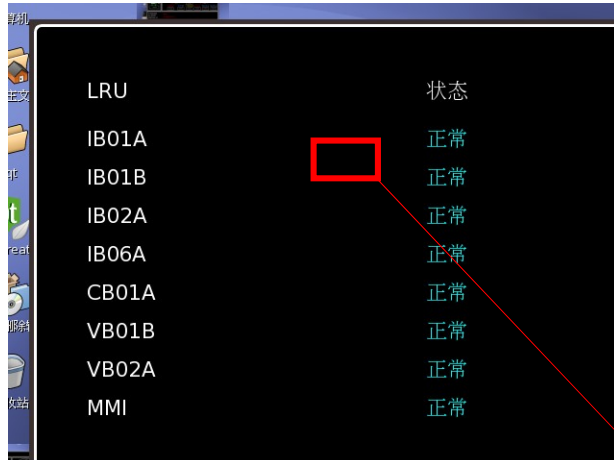
Generate 2-way Test Cases

- Package 1
 - No. of parameters: 30
 - No. of test cases: 49
- Package 2
 - No. of parameters: 58
 - No. of test cases: 77
- Package 3
 - No. of parameters: 61
 - No. of test cases : 80
- Package 4
 - No. of parameters: 50
 - No. of test cases : 90

The number of test cases for each package is less than 100

34

Sample Bugs We Have Detected (1)



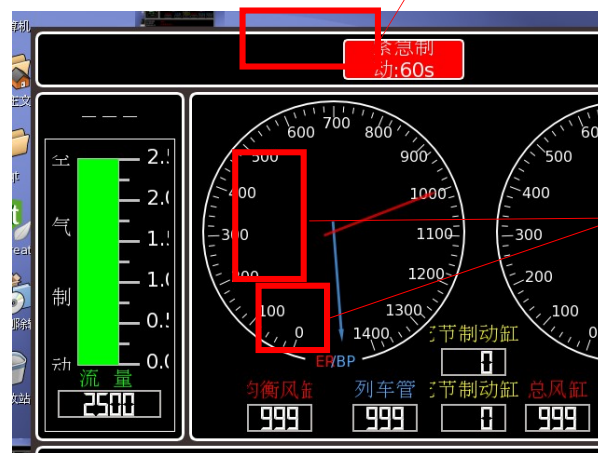
LRU	状态
IB01A	正常
IB01B	正常
IB02A	正常
IB06A	正常
CB01A	正常
VB01B	正常
VB02A	正常
MMI	正常

After the execution of a test case we generated, the status of module IB02A should be “Abnormal” instead of “Normal”

35

Sample Bugs We Have Detected (2)

After the execution of a test case, the signal light for “Emergency Brake” was turned on and could not be turned off (although it should for the subsequent executions that were not used to test the brake.)



The meter does not display the corresponding data correctly (should be 999 instead of 1,400)

36

Bug Detection Results

Less test cases generated, but more bugs detected

		Number of test cases	Number of bugs found	Did CT find all bugs detected by the practitioners?
Package 1	Field Testing	98	2	-
	CT	49	6	Yes
Package 2	Field Testing	102	1	-
	CT	77	5	Yes
Package 3	Field Testing	116	2	-
	CT	80	7	Miss 1
Package 4	Field Testing	122	2	-
	CT	90	4	Yes

Due to time limit, combinations between different packages were not included.

37

Case Study II:

Kylin Linux

38

Results Conclusion

- We use similar approaches to test three functionalities of Kylin
 - File exploration (similar to Windows Explorer)
 - File search
 - Shortcut creation

		Number of test cases	Number of bugs found	Did CT find all bugs detected by the practitioners?
File Exploration	Field Testing	4	1	-
	CT	19	6	Yes
File Search	Field Testing	6	3	-
	CT	17	4	Miss 2
Shortcut creation	Field Testing	-	-	-
	CT	27	1	-

Based on the expert knowledge, testers used some special input values to detect these 2 bugs. However, these values were not selected in our study because the special knowledge was not provided to us.

39

Part II

Combinatorial Decision Testing

R. Gao, L. Hu, W. E. Wong, H. Lu, and S. Huang, "Effective Test Generation for Combinatorial Decision Coverage," in *Proceedings of IEEE International Conference on Software Quality, Reliability, and Security*, pp. 47-54, Vienna, Austria, August 2016

40

Coverage Criteria

Decision coverage

MC/DC coverage

Statement coverage

Condition coverage

Focusing on single “Segment of Code”

41

An Example

```
input  $a, b$ ;  
int  $k = 1$ ;  
if ( $a > 0$ )  
     $k = k + 1$ ;  
if ( $b < 5$ ) {  
     $k = k + 2$ ;  
} else {  
     $k = k - 2$ ;  
}  
print  $5/k$ ;
```

A test suite with 100% statement, decision, and condition coverage

$t_1 = \{a = 2, b = 3\}$

$t_2 = \{a = -1, b = 6\}$

Cannot detect the bug
(because these two test cases cannot make $k = 0$)

To detect the bug, we need to have $a > 0$ and $b \geq 5$

Consider the *combinations* of
different decisions

42

What Should We Do?

Cover all paths?



Obviously too expensive

What Should We Do ?

How to decide which **paths** we should cover?



combinations of decisions

Combinatorial Design

What inputs should we use to cover a particular path?

Symbolic Execution

Extends CT to White-box Settings

- Black-box requirements-based combinatorial testing
 - Testing various combinations among *input parameters (components) of a system*
- White-box-based combinatorial decision testing
 - Testing various combinations among *decisions of a program*

45

Combinatorial Testing & Symbolic Execution

**Black-box
Combinatorial Testing**

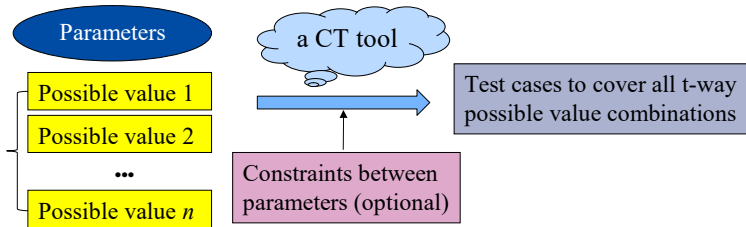
Combinatorial Decision Testing

**White-box
Symbolic Execution**

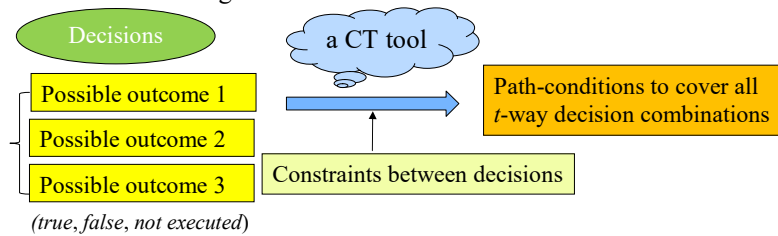
46

Black-Box CT versus White-Box CT

- CT in black-box settings



- CT in white-box settings



47

General Procedure

- Step 1: Identify program decisions
- Step 2: Assignment of decision outcomes
- Step 3: Identification of constraints between decisions
- Step 4: Generation of a t -way path-condition set
- Step 5: Generation of test cases

48


```

1:  input  $a, b, c$ ;
2:  int  $k = 0$ ;
3:  if ( $a \geq 0$ )
4:     $k = k + 1$ ;
5:  if ( $b \leq 5$ ) {
6:     $k = k + 2$ ;
7:    if ( $(a < c)$ ) {
8:       $k = k \times 2$ ;
9:    } else if ( $a > 5$ ) {
10:      $k = k + 5$ ;
11:    }
12:  }
13:  while ( $c > 3$ ) {
14:    if ( $b > c$ ) {
15:       $k = k - 2$ ;
16:    }
17:     $c = c - 1$ ;
18:  }
19:  print( $k$ );

```

Using tools such as IDA (Interactive Dis-Assembler) to identify all program decisions

- Use combinatorial testing tool such as ACTS to generate a t -way path-condition set to cover all possible decision combinations

Path-condition	d_1	d_2	d_3	d_4	d_5	d_6
p_1	F	T	T	N	T	F
p_2	T	T	F	T	F	N
p_3	F	T	F	F	T	T
p_4	T	F	N	N	F	N
p_5	F	T	F	T	T	T
p_6	T	T	T	N	T	T
p_7	T	T	F	T	T	F
p_8	T	T	F	F	T	F
p_9	F	T	F	F	F	N
p_{10}	F	F	N	N	T	T
p_{11}	T	F	N	N	T	F
p_{12}	T	T	T	N	F	N

80 2-way decision combinations need to be covered in this program

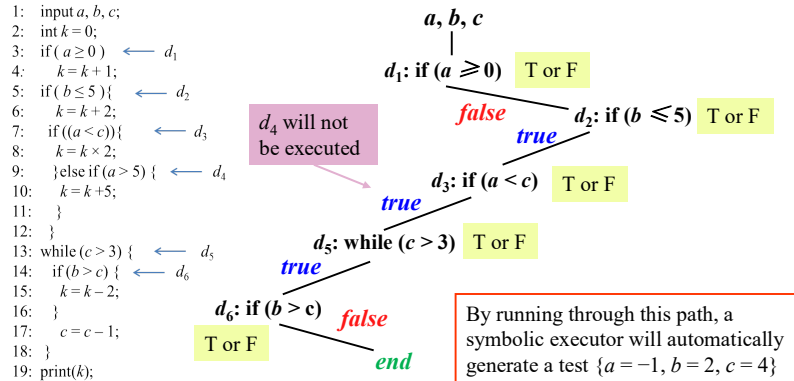
A set of 12 path-conditions can cover all 80 2-way decision combinations

Step 5: Generation of Test Cases (1)

- To satisfy each path-condition, we guide a symbolic executor to run through an execution path and generate the corresponding test case

- Example

– For a path condition $d_1 = F, d_2 = T, d_3 = T, d_4 = N, d_5 = T, d_6 = F$



51

Step 5: Generation of Test Cases (2)

- The generation is not always successful. There still exist some infeasible path-conditions which are hard to identify

Path-condition	d_1	d_2	d_3	d_4	d_5	d_6
p_1	F	T	T	N	T	F
p_2	T	T	F	T	F	N
p_3	F	T	F	F	T	T
p_4	T	F	N	N	F	N
p_5	F	T	F	T	T	T
p_6	T	T	T	N	T	T
p_7	T	T	F	T	T	F
p_8	T	T	F	F	T	F
p_9	F	T	F	F	F	N
p_{10}	F	F	N	N	T	T
p_{11}	T	F	N	N	T	F
p_{12}	T	T	T	N	F	N

Test Case	Corresponding Path-condition	a	b	c
t_1	p_1	-1	2	4
t_2	p_2	6	4	0
t_3	p_3	1	6	0
t_4	p_4	0	5	4
t_5	p_5	6	4	4
t_6	p_6	5	2	4
t_7	p_7	-1	2	-2
t_8	p_8	-1	6	4
t_9	p_9	1	6	8
t_{10}	p_{10}	1	4	3

Two path-conditions cannot be satisfied by any test cases

$(a < 0) \wedge (b \leq 5) \wedge (a \geq c) \wedge (a \leq 5) \wedge (c > 3) \wedge (b > c)$

$(a < 0) \wedge (b \leq 5) \wedge (a \geq c) \wedge (a > 5) \wedge (c > 3) \wedge (b > c)$

Future direction:

Identification of more complex constraints between program decisions

52

Two Approaches

- Traditional Combinatorial Testing
 - Requirements-based
 - Block box
 - Human reads/understands requirements
 - Manually enter inputs
 - Test case generation
- Combinatorial Branch Testing
 - Source code/syntax/semantics based
 - White box
 - Automatically parse/analyze source code
 - No/little manual intervention
 - Test case generation

53

Conclusion

**Formal Engineering-based
Automated**

Easy-to-use & Effective in Bug Detection

Instead of ad-hoc random approach



54

Question

