

Advanced Research Center for Software Testing and Quality Assurance

Combinatorial Design-based Test Generation

W. Eric Wong

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

NIST



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*)





Here Is the Reason

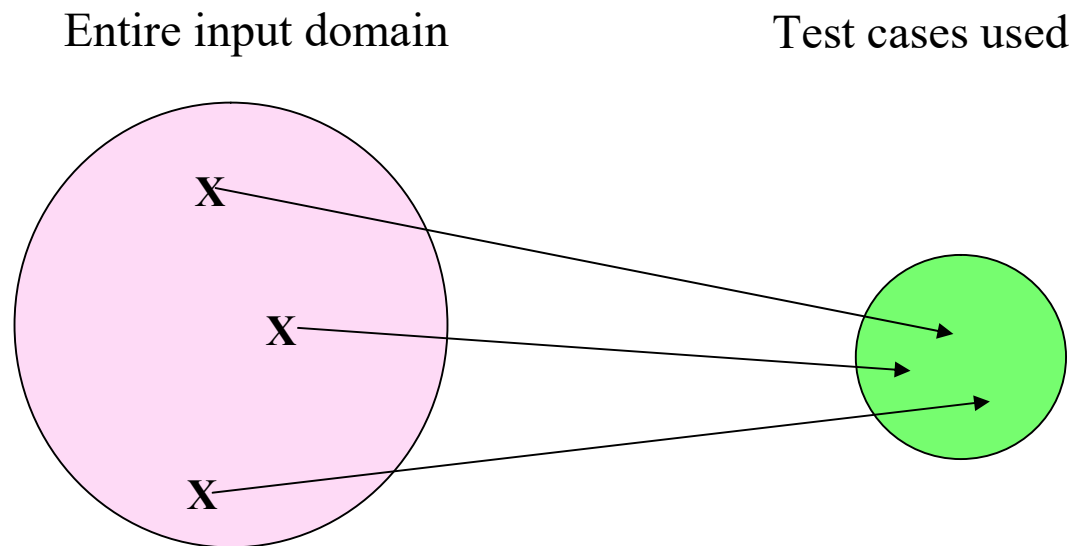
Execution

≠

Fault Detection

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



Do You Know . . .

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

What should we do?

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

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**

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.

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. The left pane displays a System View tree with a hierarchy starting from [Root Node] to [SYSTEM-Package-2-part1], then branching into Bit7, Bit8, Bit9, Bit10, Bit11, Bit12, Bit13, Bit14, Bit33, and Bit34, with a Relations node at the bottom. The right pane shows a Test Result table with 31 rows and 10 columns (BIT7 to BIT34). The table contains numerical data representing test results for each bit across 31 iterations.

	BIT7	BIT8	BIT9	BIT10	BIT11	BIT12	BIT13	BIT14	BIT33	BIT34
1	80	08	40	80	01	00	40	20	04	08
2	40	04	20	80	00	01	20	08	02	04
3	20	02	10	80	02	00	00	04	01	02
4	10	01	08	80	01	01	80	02	00	01
5	08	00	01	80	00	00	40	01	08	00
6	04	20	00	80	02	01	20	00	04	10
7	02	08	80	80	00	00	80	40	01	04
8	00	02	40	80	02	01	00	08	08	01
9	80	01	20	40	02	00	80	04	02	00
10	40	00	10	40	01	01	40	02	00	10
11	20	20	08	40	00	00	20	01	08	08
12	10	04	01	40	01	00	00	00	04	02
13	08	02	00	40	01	01	40	40	02	01
14	04	01	80	40	00	01	00	20	08	04
15	02	00	40	40	01	01	20	08	01	02
16	00	08	20	40	00	00	00	02	00	02
17	80	04	10	20	00	01	80	01	01	01
18	40	02	08	20	02	00	80	00	04	00
19	20	01	01	20	02	01	20	40	02	10
20	10	00	00	20	02	00	00	20	00	08
21	08	20	80	20	01	01	40	04	00	04
22	04	04	40	20	02	00	20	02	00	00
23	02	02	20	20	01	00	00	01	08	10
24	00	08	10	20	00	01	20	00	02	08
25	80	00	08	10	02	00	00	40	04	04
26	40	20	01	10	01	01	80	20	01	01
27	20	08	00	10	00	00	80	08	08	00
28	10	02	80	10	00	00	20	01	02	10
29	08	01	40	10	02	00	40	00	01	08
30	04	00	20	10	00	00	40	04	04	01
31	02	20	10	10	01	01	00	02	02	00

Combinatorial Testing at NIST

- NIST (National Institute of Standards and Technology)

NIST National Institute of Standards and Technology
Information Technology Laboratory

SEARCH CSRC:

Computer Security Division

Computer Security Resource Center

CSRC Home About Projects / Research Publications News & Events

CSRC HOME > GROUPS > SNS > ACTS

AUTOMATED COMBINATORIAL TESTING SOFTWARE (ACTS)

Combinatorial methods have been shown to significantly reduce cost and increase quality for software and system testing. The basis for combinatorial testing is the interaction rule, which is based on analysis of thousands of software failures. The rule states that most failures are induced by single

Interactions (Kul)	Single Failures (%)	Two-way Interactions (%)	Three-way Interactions (%)	Four-way Interactions (%)
1	70	40	30	20
2	100	70	50	30
3	100	100	80	50
4	100	100	100	100

- 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)

<i>A sample system</i>		
<i>a</i>	<i>b</i>	<i>c</i>
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



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

Combinatorial Testing: Example II (3)

<i>A sample system</i>		
<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



<i>An efficient test set</i>			
	<i>a</i>	<i>b</i>	<i>c</i>
t_1	1	4	8
t_2	1	5	9
t_3	1	6	7
t_4	2	4	9
t_5	2	5	7
t_6	2	6	8
t_7	3	4	7
t_8	3	5	8
t_9	3	6	9



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]



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

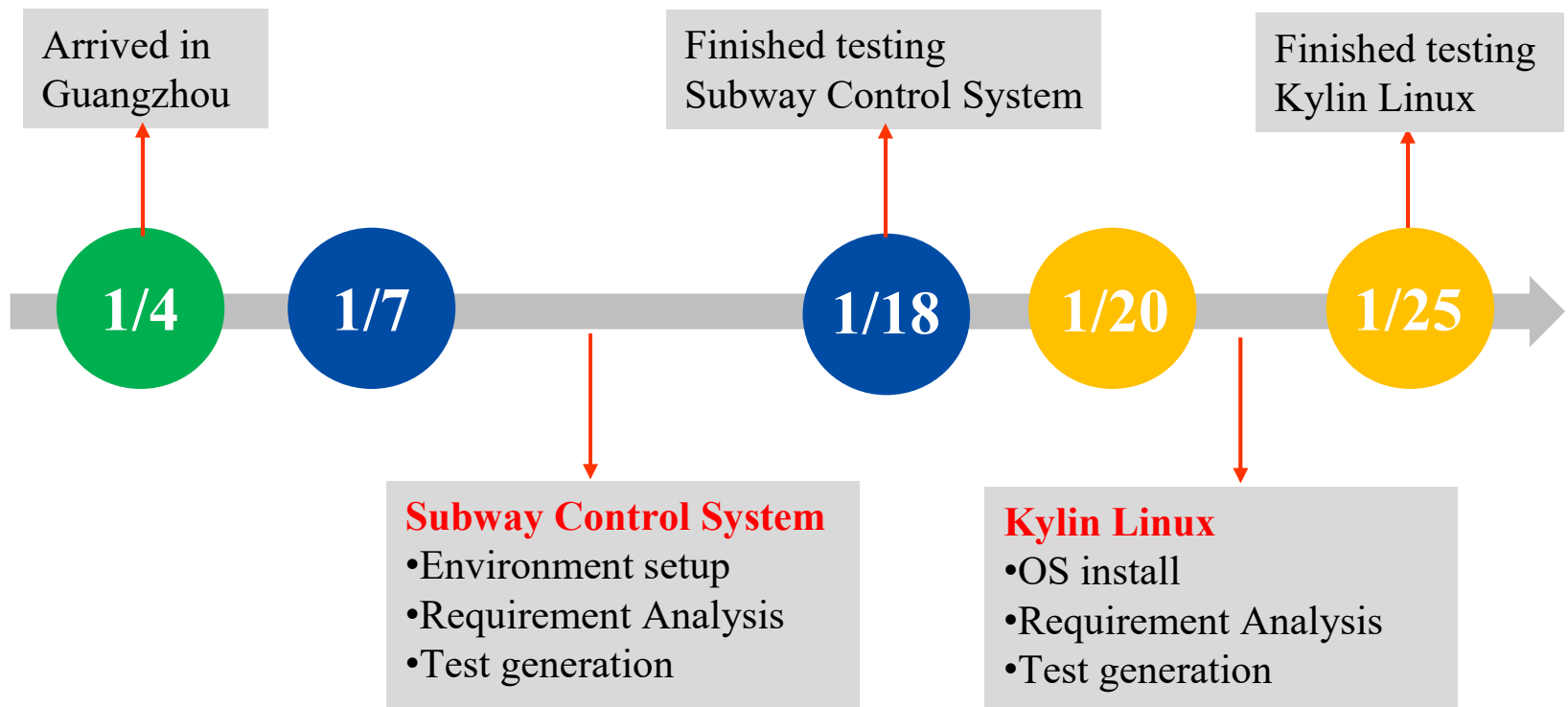
Part I

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

Schedule

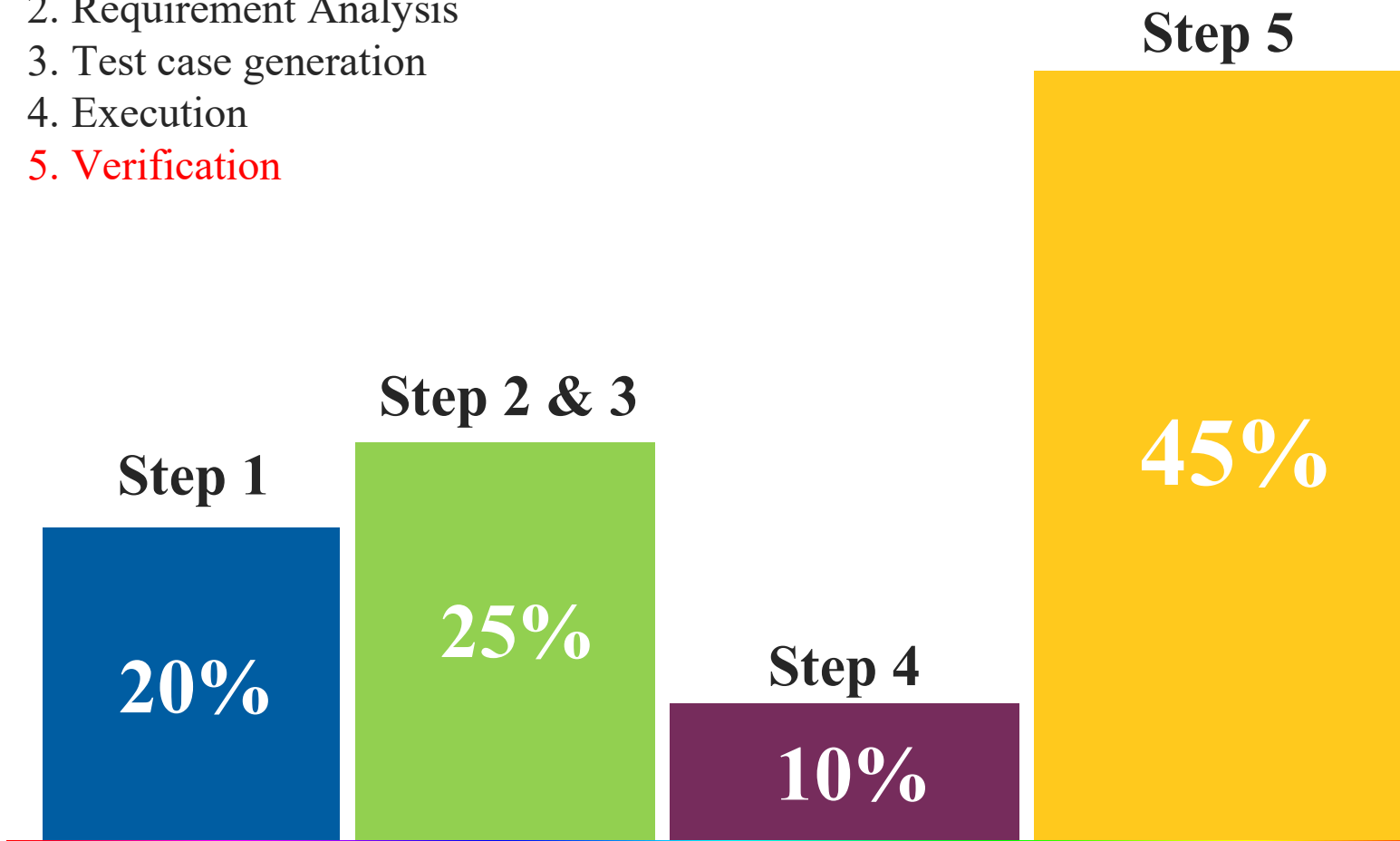
Timeline



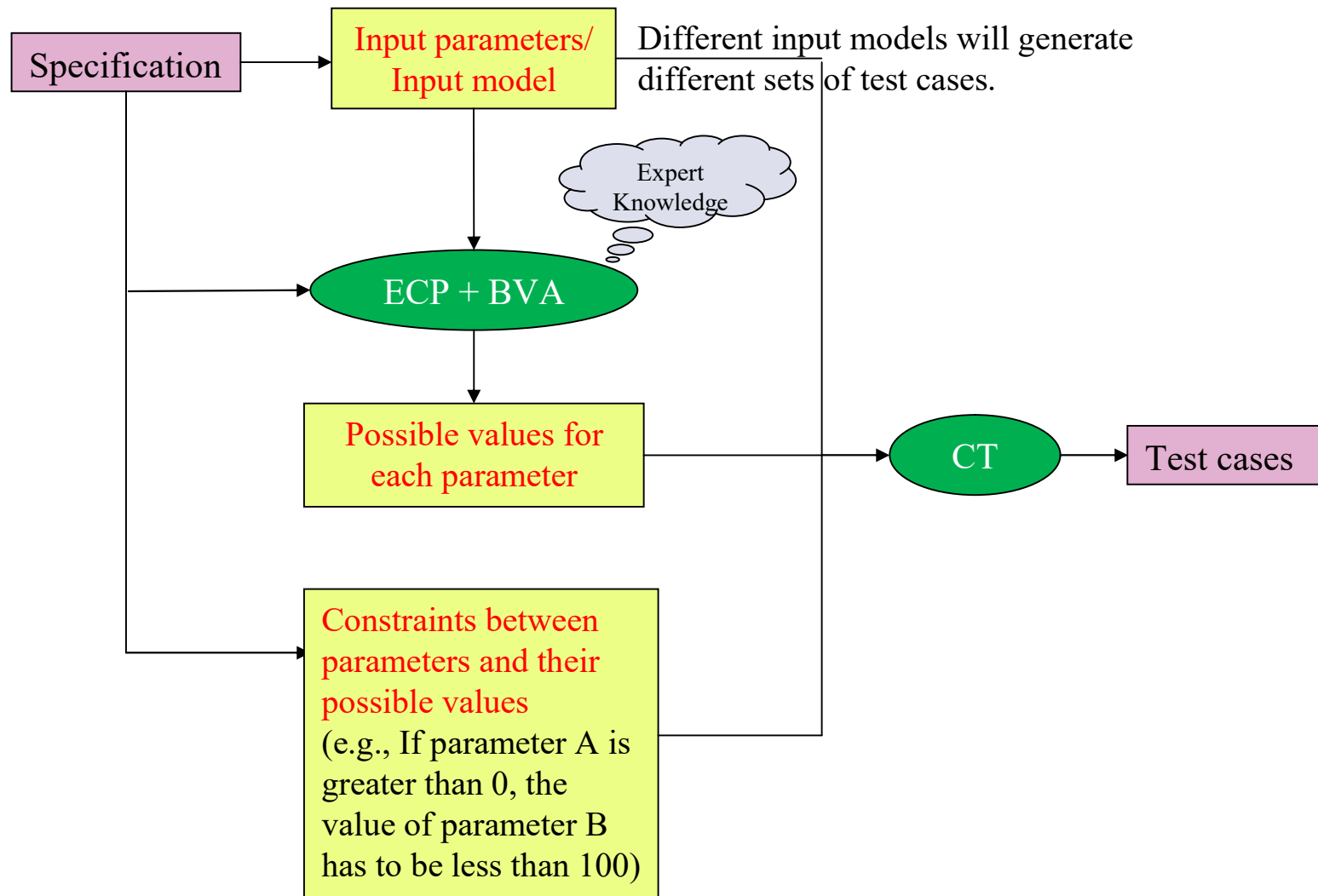
Effort Distribution

Major Steps:

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



Test Generation – CT + ECP + BVA (1)



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:

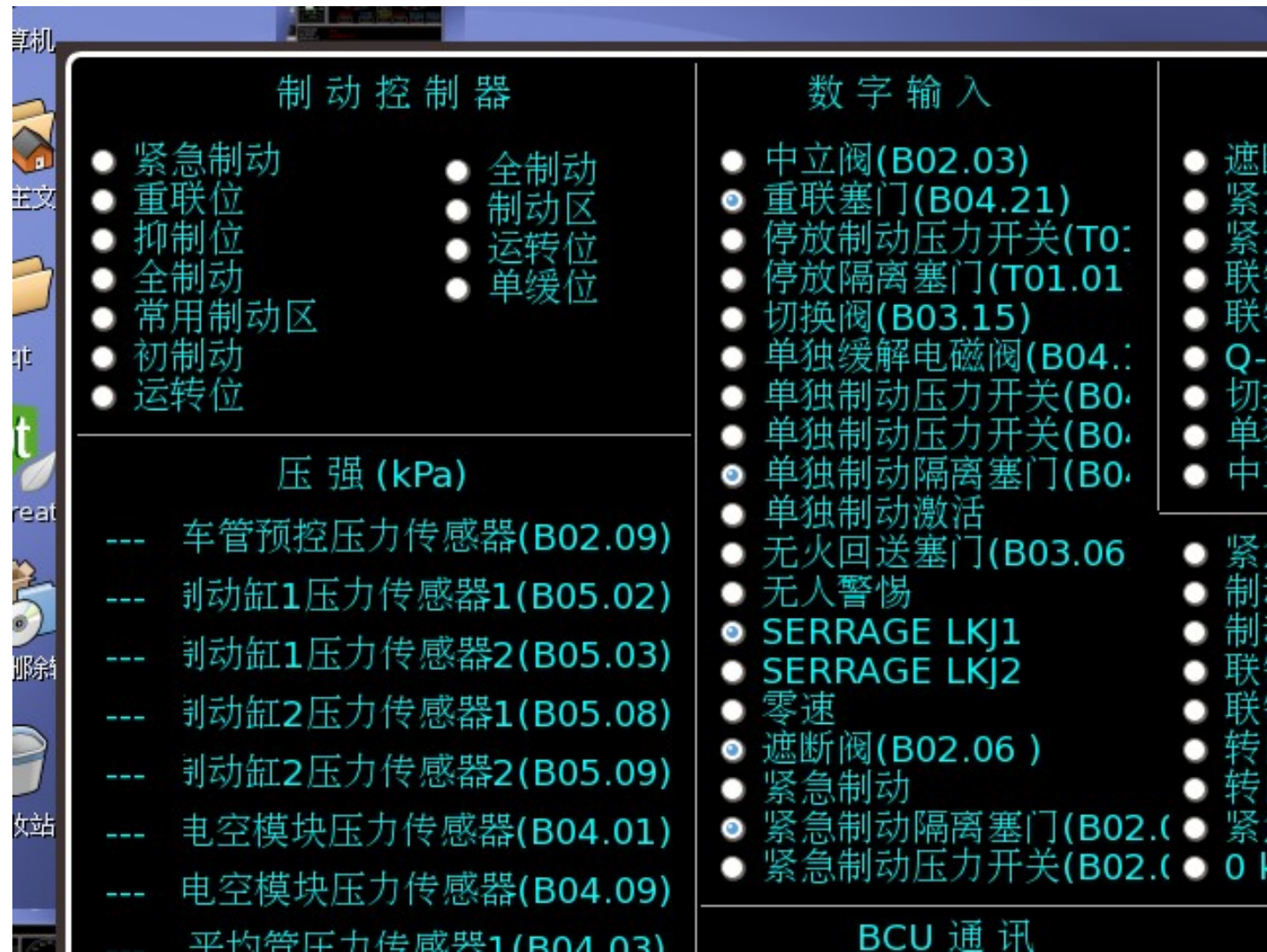
Subway Control System



The GUI of the System (1)



The GUI of the System (2)



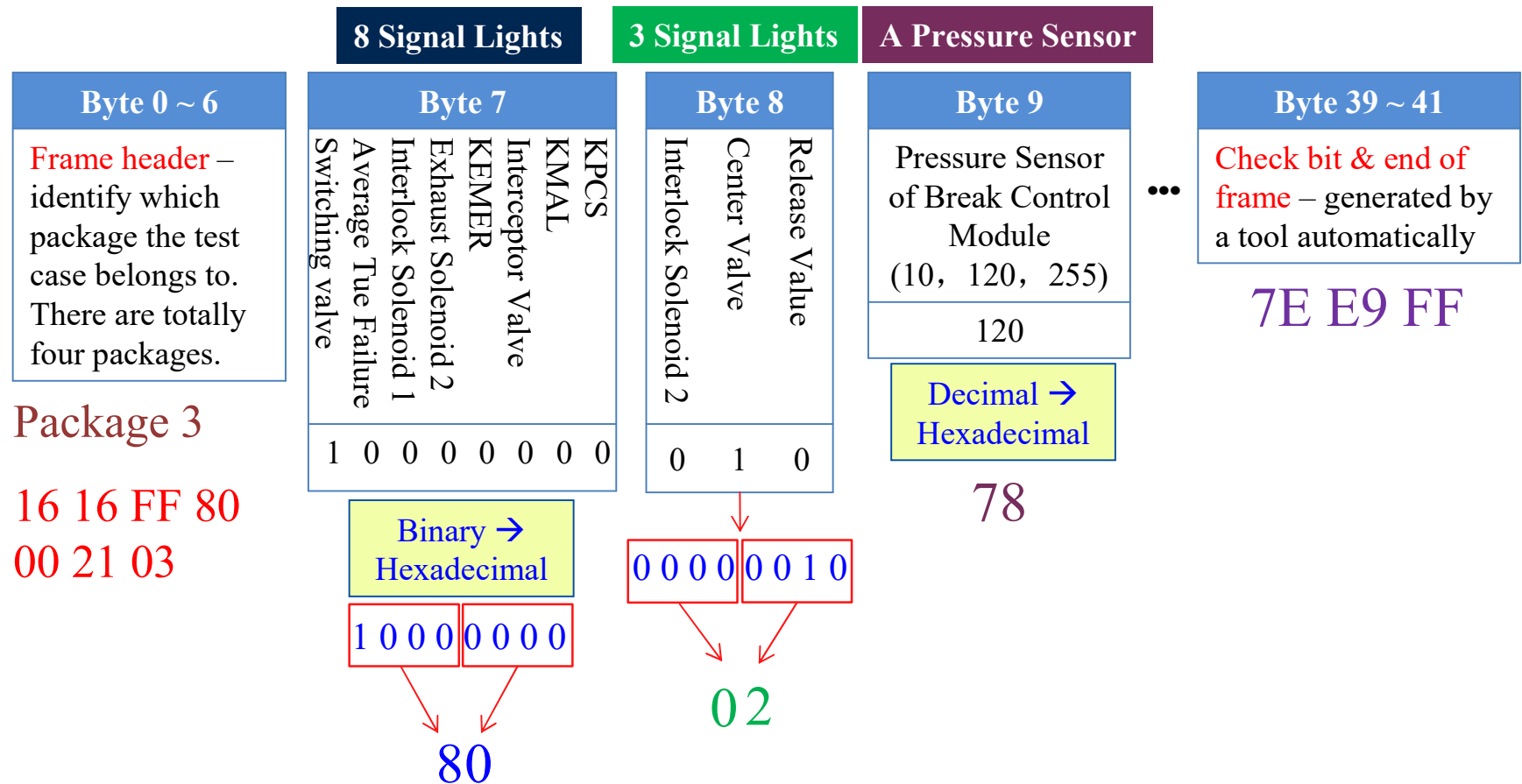


A Sample Test Case

**1616FF80002103800278000078E7
03E703E703402004404001580258
02580258025802646458027EE9FF**

How to interpret such a monster?

Decompose the Test Case



1616FF80002103800278000078E703E703E70340200440400158025802580258025802646458027EE9FF

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

Select Possible Values for Each Byte (2)

Package 3 - Byte 9	
Pressure Sensor of Brake Control Module	
10	
120	
255	

Decimal → Hexadecimal	
10	0A
120	78
255	FF

Possible values for parameter
“Package 3 - Byte 9”:
0A; 78; FF



Take Advantage of Tester's Experiences

**Expert Knowledge can also help
select possible values for each byte**

Select Possible Values for Each Byte (3)

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

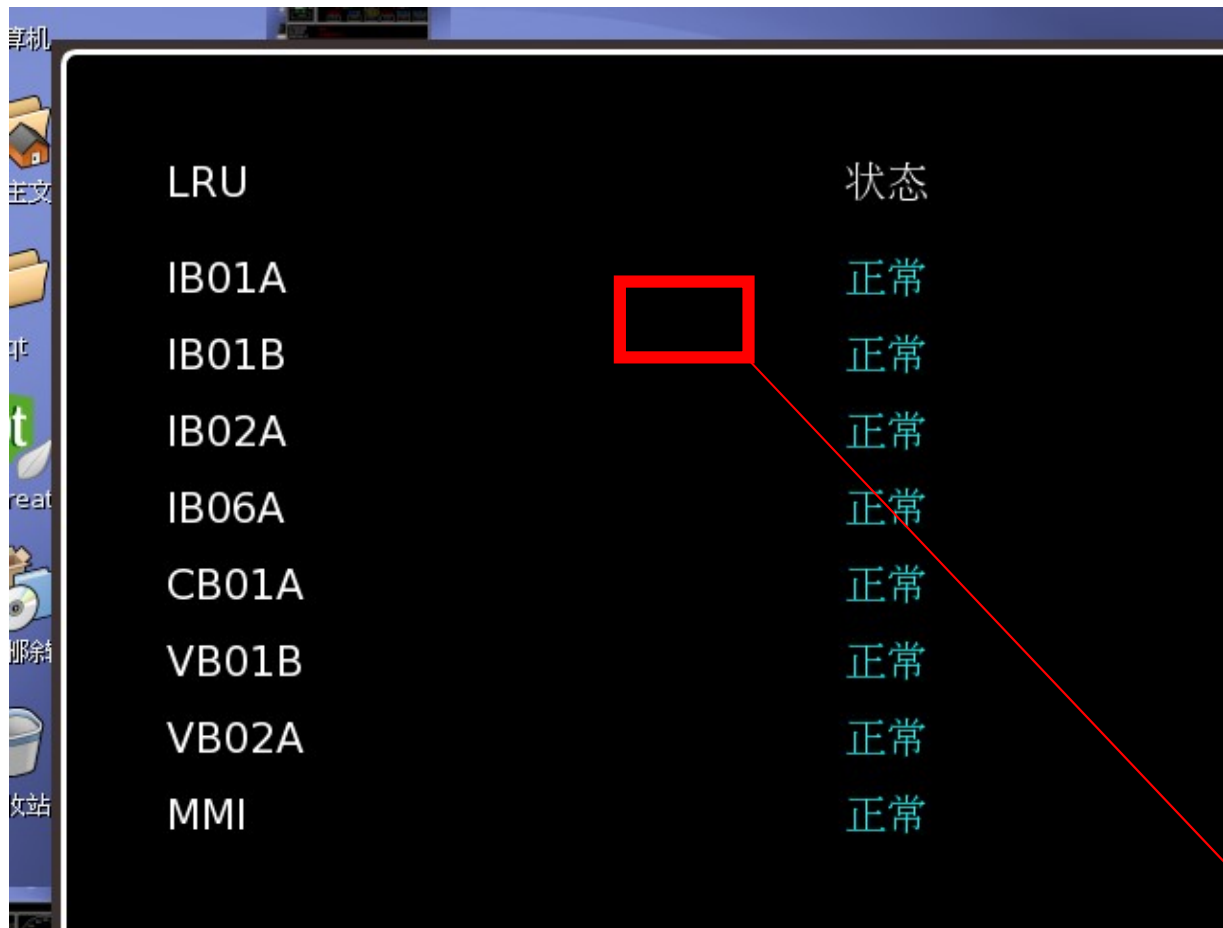
Possible values for “Package 1 - Byte 20”:
20; 10; 08; 04; 02; 01; 00 **30; 0C; 03**

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

Sample Bugs We Have Detected (1)

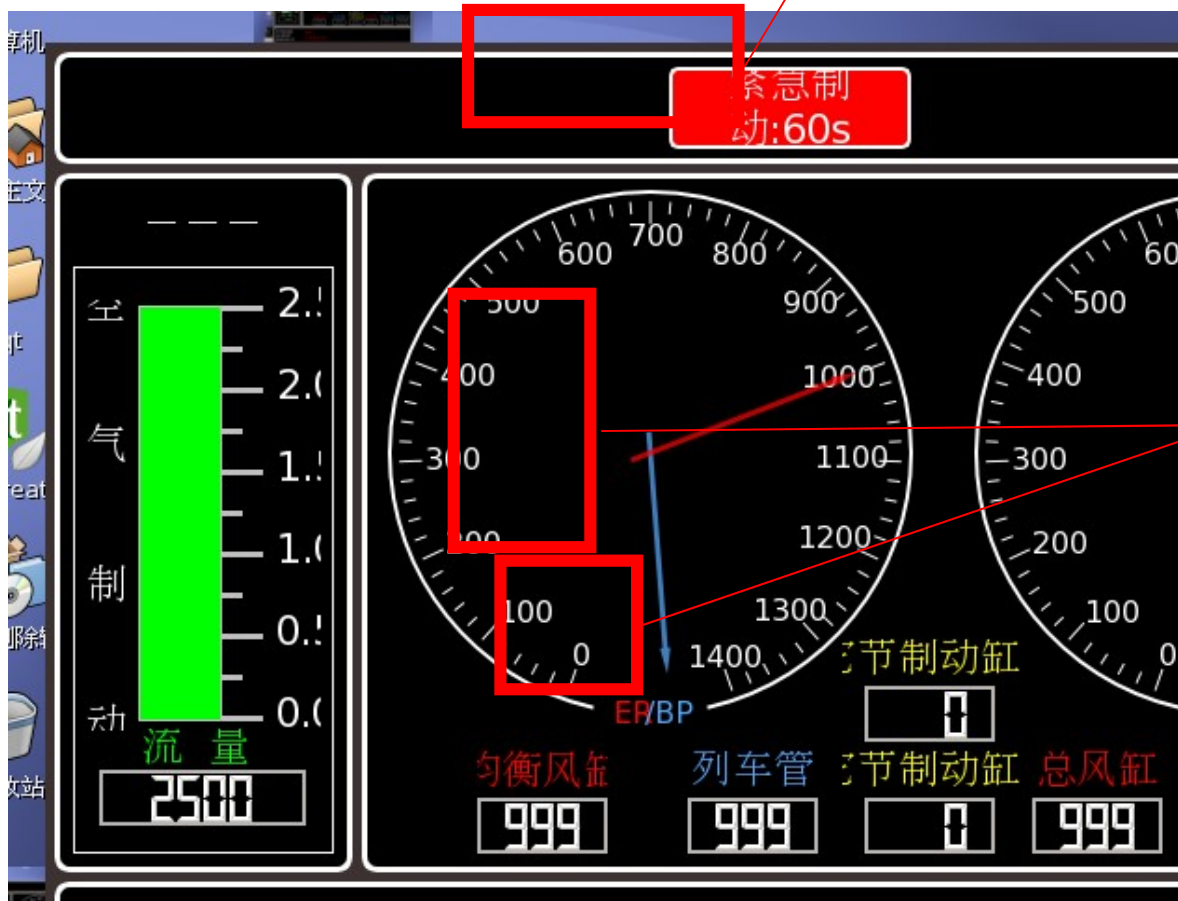


LRU	状态
IBO1A	正常
IBO1B	正常
IBO2A	正常
IBO6A	正常
CB01A	正常
VB01B	正常
VB02A	正常
MMI	正常

After the execution of a test case we generated, the status of module IBO2A should be “**Abnormal**” instead of “**Normal**”

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)

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.

A decorative vertical bar on the left side of the slide, transitioning from dark blue at the top to light blue at the bottom. Two horizontal lines, each with a rainbow gradient, are positioned above and below the main text area.

Case Study II:

Kylin Linux

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.

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



Coverage Criteria

Decision coverage

MC/DC coverage

Statement coverage

Condition coverage

Focusing on single “Segment of Code”

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

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*

Combinatorial Testing & Symbolic Execution

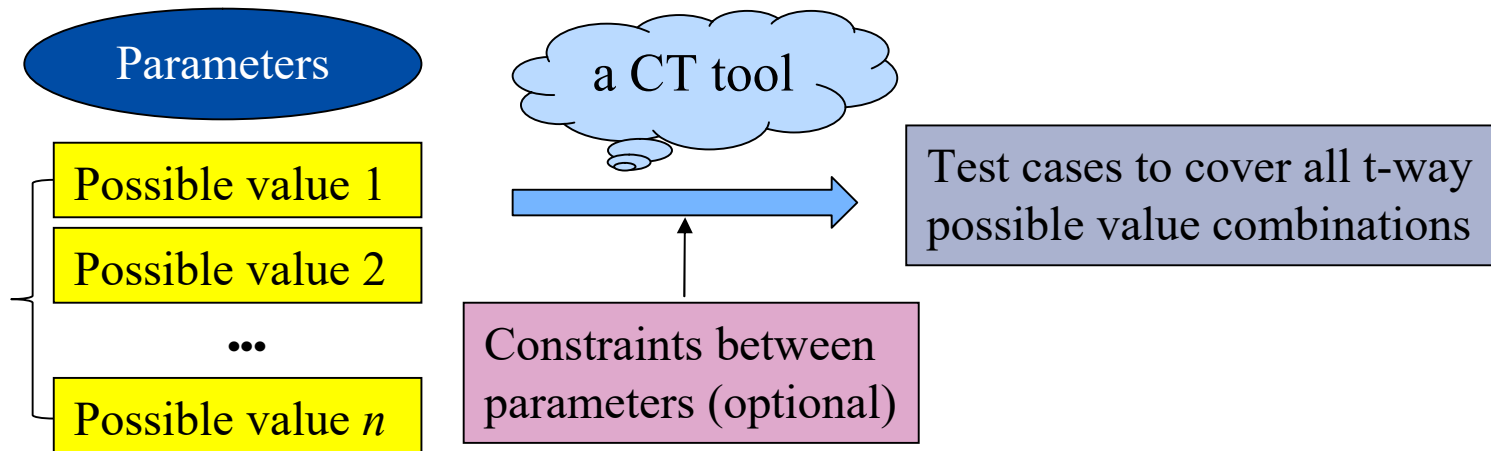
**Black-box
Combinatorial Testing**

Combinatorial Decision Testing

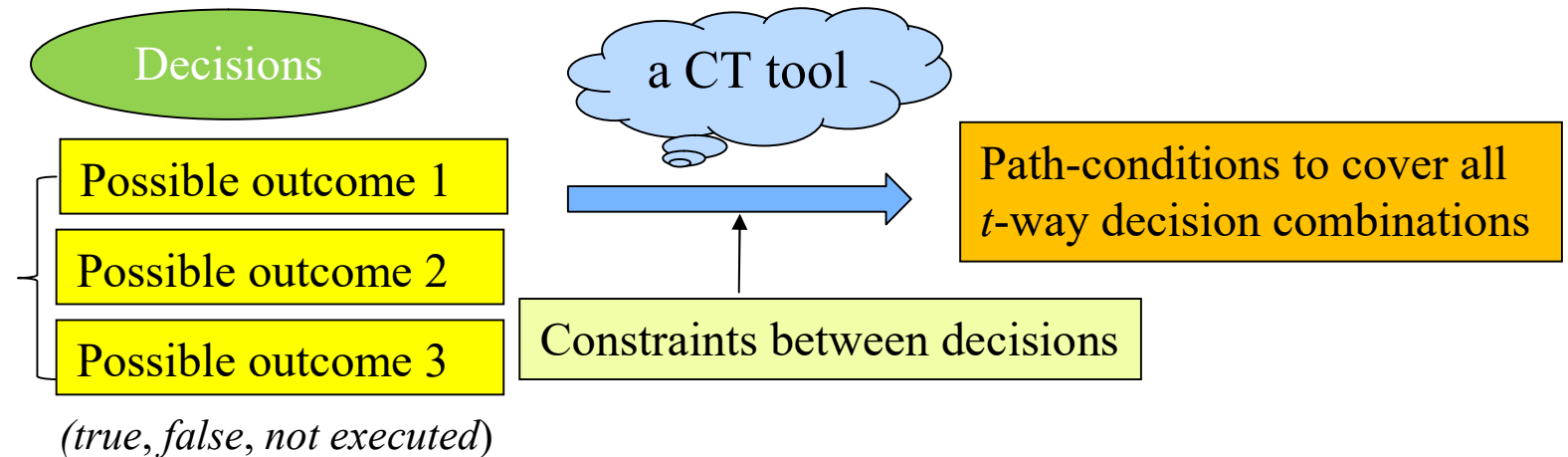
**White-box
Symbolic Execution**

Black-Box CT versus White-Box CT

- CT in black-box settings



- CT in white-box settings





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

Step 1: Identify Program Decisions

```
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**

Step 4: Generation of a t -way Path-condition Set

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

$\{d_1 = T, d_2 = T\}$	$\{d_1 = T, d_2 = F\}$	$\{d_1 = F, d_2 = T\}$	$\{d_1 = F, d_2 = F\}$
$\{d_1 = T, d_3 = T\}$	$\{d_1 = T, d_3 = F\}$	$\{d_1 = T, d_3 = N\}$	$\{d_1 = F, d_3 = T\}$
$\{d_1 = F, d_3 = F\}$	$\{d_1 = F, d_3 = N\}$	$\{d_1 = T, d_4 = T\}$	$\{d_1 = T, d_4 = F\}$
$\{d_1 = T, d_4 = N\}$	$\{d_1 = F, d_4 = T\}$	$\{d_1 = F, d_4 = F\}$	$\{d_1 = F, d_4 = N\}$
$\{d_1 = T, d_5 = T\}$	$\{d_1 = T, d_5 = F\}$	$\{d_1 = F, d_5 = T\}$	$\{d_1 = F, d_5 = F\}$
$\{d_1 = T, d_6 = T\}$	$\{d_1 = T, d_6 = F\}$	$\{d_1 = T, d_6 = N\}$	$\{d_1 = F, d_6 = T\}$
$\{d_1 = F, d_6 = F\}$	$\{d_1 = F, d_6 = N\}$	$\{d_2 = T, d_3 = T\}$	$\{d_2 = T, d_3 = F\}$
$\{d_2 = F, d_3 = N\}$	$\{d_2 = T, d_4 = T\}$	$\{d_2 = T, d_4 = F\}$	$\{d_2 = T, d_4 = N\}$
$\{d_2 = F, d_4 = N\}$	$\{d_2 = T, d_5 = T\}$	$\{d_2 = T, d_5 = F\}$	$\{d_2 = F, d_5 = T\}$
$\{d_2 = F, d_5 = F\}$	$\{d_2 = T, d_5 = T\}$	$\{d_2 = T, d_5 = F\}$	$\{d_2 = F, d_5 = T\}$
$\{d_2 = F, d_6 = T\}$	$\{d_2 = F, d_6 = F\}$	$\{d_2 = F, d_6 = N\}$	$\{d_3 = T, d_4 = N\}$
$\{d_3 = F, d_4 = T\}$	$\{d_3 = F, d_4 = F\}$	$\{d_3 = N, d_4 = N\}$	$\{d_3 = T, d_5 = T\}$
$\{d_3 = T, d_5 = F\}$	$\{d_3 = F, d_5 = T\}$	$\{d_3 = F, d_5 = F\}$	$\{d_3 = N, d_5 = T\}$
$\{d_3 = N, d_5 = F\}$	$\{d_3 = T, d_6 = T\}$	$\{d_3 = T, d_6 = F\}$	$\{d_3 = T, d_6 = N\}$
$\{d_3 = F, d_6 = T\}$	$\{d_3 = F, d_6 = F\}$	$\{d_3 = F, d_6 = N\}$	$\{d_3 = N, d_6 = T\}$
$\{d_3 = N, d_6 = F\}$	$\{d_3 = N, d_6 = N\}$	$\{d_4 = T, d_5 = T\}$	$\{d_4 = T, d_5 = F\}$
$\{d_4 = F, d_5 = T\}$	$\{d_4 = F, d_5 = F\}$	$\{d_4 = T, d_6 = T\}$	$\{d_4 = T, d_6 = F\}$
$\{d_4 = T, d_6 = N\}$	$\{d_4 = F, d_6 = T\}$	$\{d_4 = F, d_6 = F\}$	$\{d_4 = F, d_6 = N\}$
$\{d_4 = N, d_5 = T\}$	$\{d_4 = N, d_5 = F\}$	$\{d_4 = N, d_6 = T\}$	$\{d_4 = N, d_6 = F\}$
$\{d_4 = N, d_6 = N\}$	$\{d_5 = T, d_6 = T\}$	$\{d_5 = T, d_6 = F\}$	$\{d_5 = F, d_6 = N\}$

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

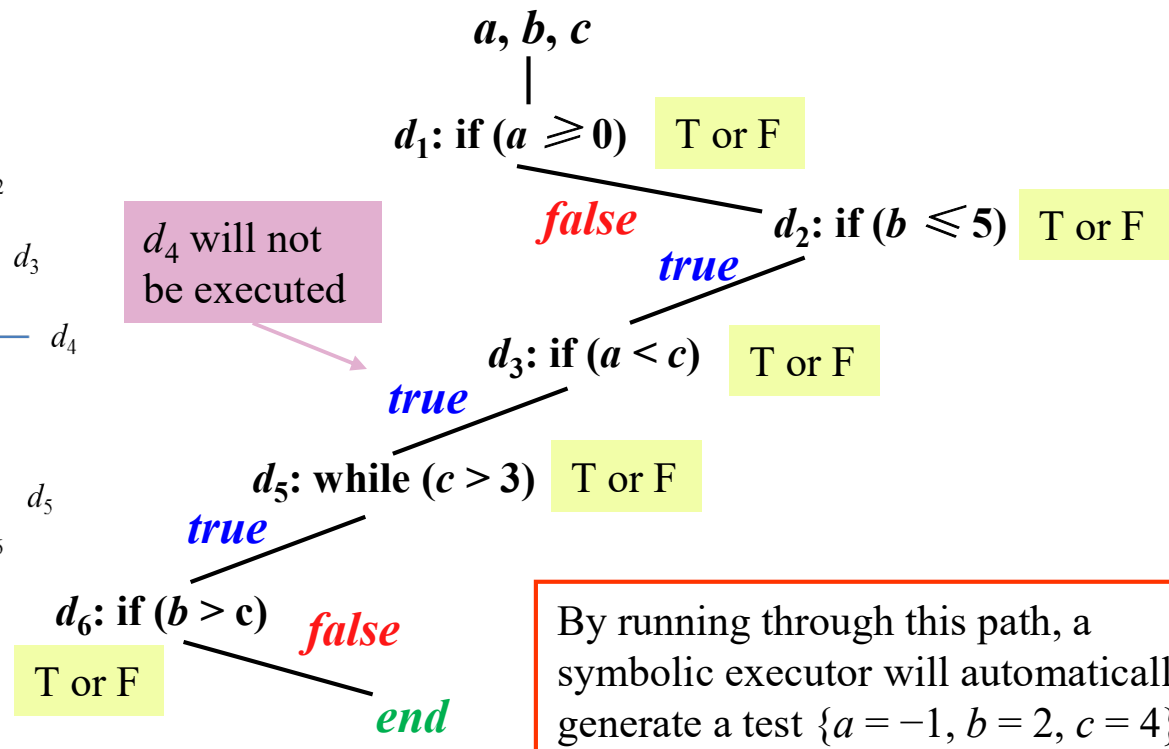
Step 5: Generation of Test Cases (1)

- 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 $d_1 = F, d_2 = T, d_3 = T, d_4 = N, d_5 = T, d_6 = F$

```
1: input a, b, c;
2: int k = 0;
3: if (a ≥ 0) ← d1
4:   k = k + 1;
5: if (b ≤ 5) { ← d2
6:   k = k + 2;
7:   if ((a < c)) { ← d3
8:     k = k × 2;
9:   } else if (a > 5) { ← d4
10:    k = k + 5;
11:   }
12: }
13: while (c > 3) { ← d5
14:   if (b > c) { ← d6
15:     k = k - 2;
16:   }
17:   c = c - 1;
18: }
19: print(k);
```



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_4	1	6	0
t_4	p_6	0	5	4
t_5	p_7	6	4	4
t_6	p_8	5	2	4
t_7	p_9	-1	2	-2
t_8	p_{10}	-1	6	4
t_9	p_{11}	1	6	8
t_{10}	p_{12}	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



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

Conclusion

**Formal Engineering-based
Automated**

Easy-to-use & Effective in Bug Detection

Instead of ad-hoc random approach



Question

