Black Box Testing & Combinatorial Interaction Testing

CS453 Automated Software Testing

Shin Yoo | COINSE@KAIST

Black Box Testing

- View program as a black box and ignore the internal structure of the program

Test the behaviour of the program according to its specifications.

Black Box Testing

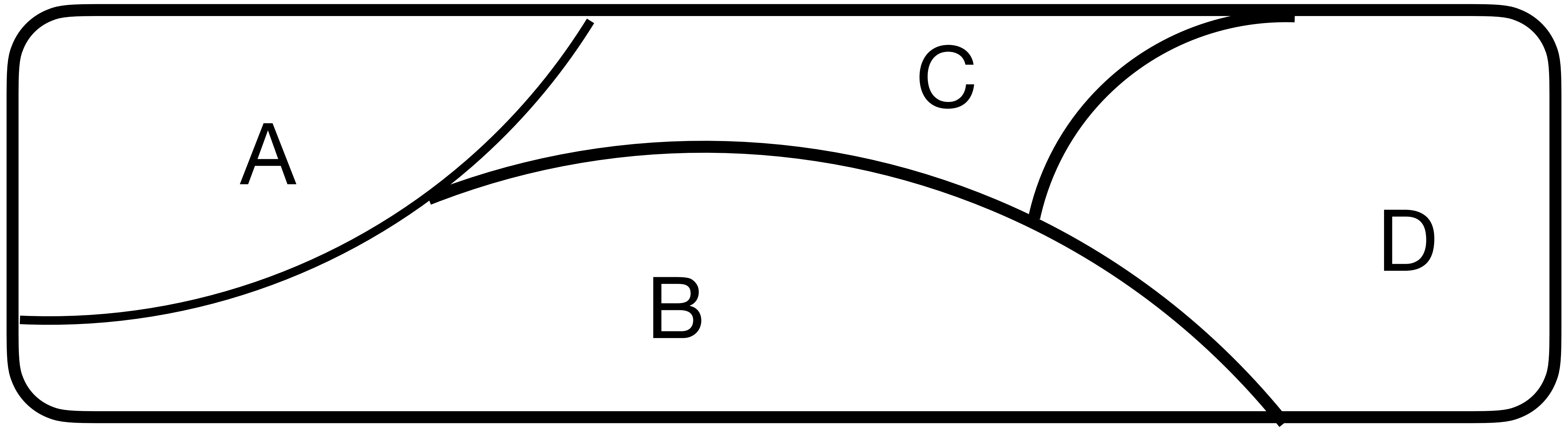
- Also known as Functional Testing or Behavioural Testing
- Test data are derived solely from the specifications
- Exhaustive input testing is impossible!
    - Recall: you cannot test a program to guarantee that it is error free

Black Box Testing

- Use a small subset of all possible inputs.
- A good set of test inputs with higher probability of finding most errors.
- Approaches
    - Random Testing (which can also be white box)
    - Equivalence Class Partitioning
    - Boundary Value Analysis

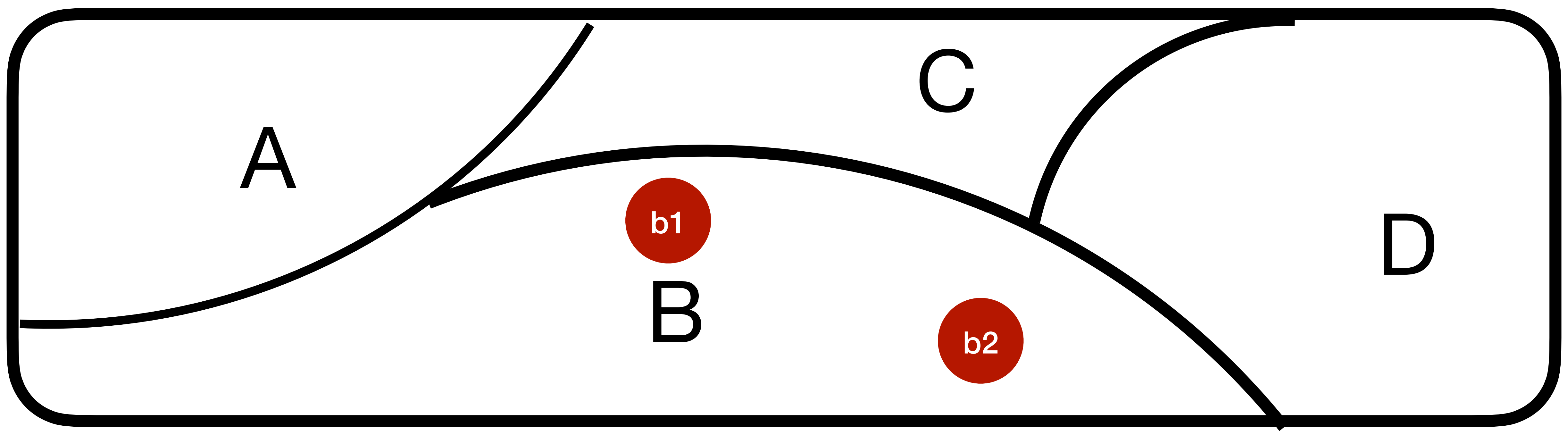
Equivalence Partitioning

- Partition the input domain of a program into a finite number of equivalence classes.
- A Program shows the same behaviour on all elements within an equivalence class.



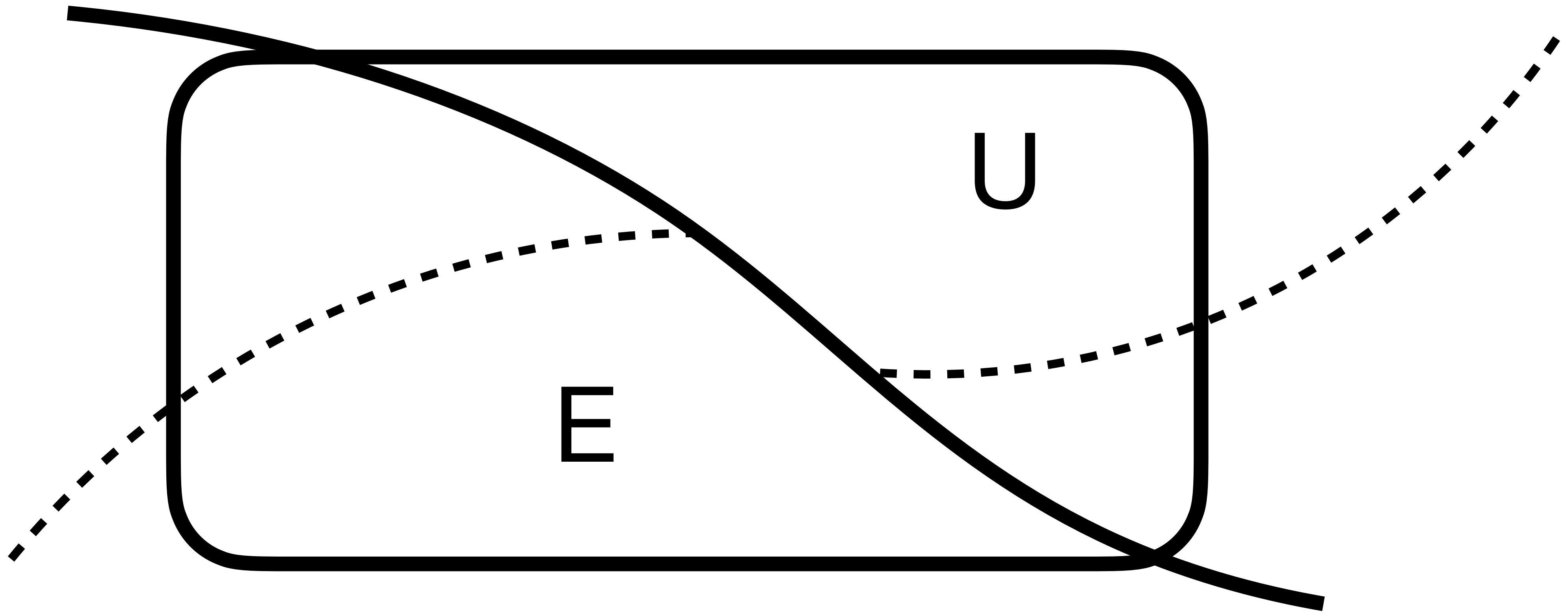
Equivalence Partitioning

- If one test case in an equivalence class detects an error, all other test cases in the equivalence class would be expected to find the same error.
- We can select one test from each equivalence class.



Equivalence Partitioning

- The entire input domain can always be divided into two subsets:
  - Expected or legal inputs (E)
  - Unexpected or illegal inputs (U)
- E and U can be further subdivided into subsets according to the specification of the program.



Equivalence Partitioning

- Consider an application that takes an integer i, which denotes the age of the user:
  - E = {0 <= i <=120}
  - U = {i < 0 , i > 120}
- Suppose the SUT deals with an insurance policy that divides people into different age group at 20 and 70:
  - E1 = {0 <= i <= 20} U1 = {i < 0}
  - E2 = {20 < i <=70} U2 = {i > 120}
  - E3 = {70 < i <= 120}
- The final test input we get from this partitioning:
  - Test inputs I = {-10, 10, 30, 80, 200}

Equivalence Partitioning

- There are many ways to partition an input domain.
- Even from the same equivalence partitioning, two testers might select different tests from the same class
- Effectiveness may depend on the tester's experience.
- *Partition testing can be better, worse, or the same as random testing, depending on how the partitioning is done.* — E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. IEEE Transactions on software Engineering, (7):703–711, 1991.

Off by one error

- Logic errors that involve boundary conditions
- Usually happens due to confusion between “less than” and “less than or equal to”
- Simple, but actually very common

Off by one: Loops

```
for (i = 0; i < 10; i++)
{
   /* Body of the loop */
}
```

Correct

```
for (i = 1; i < 10; i++)
{
   /* Body of the loop */
}
```

Incorrect

```
for (i = 0; i <= 10; i++)
{
   /* Body of the loop */
}
```

Incorrect

```
for (i = 0; i < 11; i++)
{
   /* Body of the loop */
}
```

Incorrect

Off by one: Fenceposts

- Common when counting **boundaries** between things

Things	1	2	3	4	5	6	7	8
Boundaries	1	2	3	4	5	6	7	

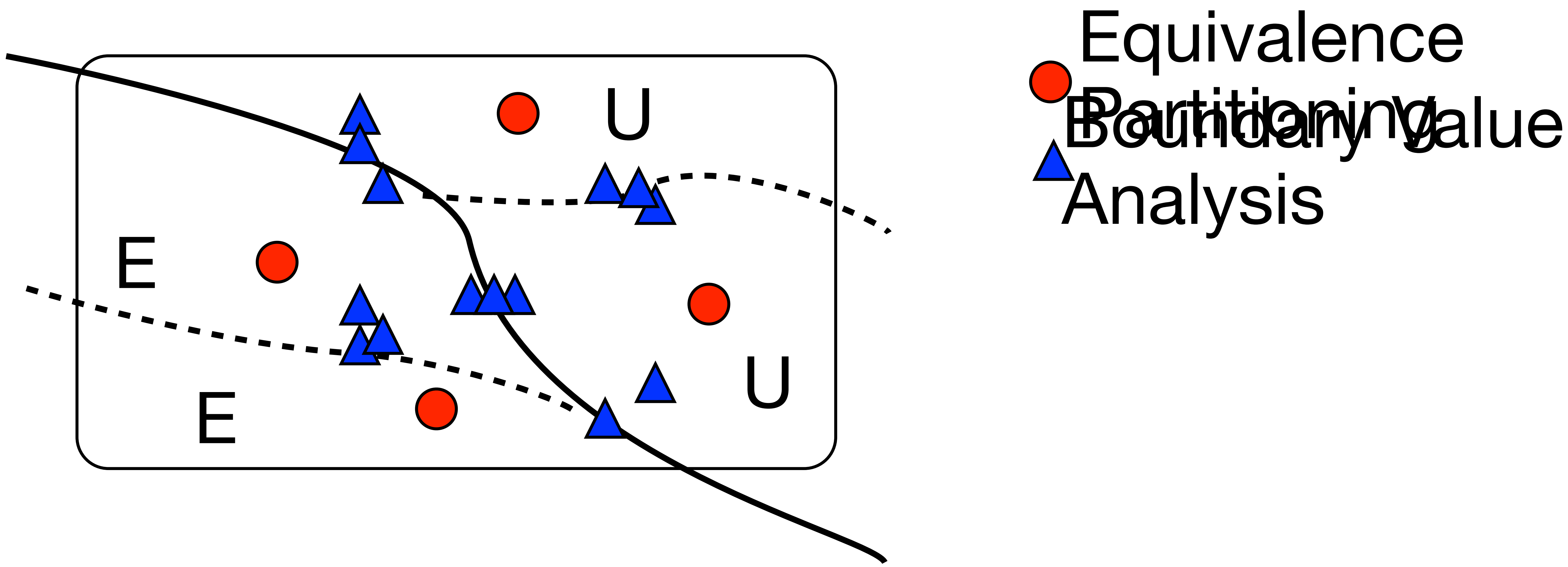
Off by one: `strncat`

```
void foo (char *s)
{
    char buf[15];
    memset(buf, 0, sizeof(buf));
    strncat(buf, s, sizeof(buf)); // should be: sizeof(buf)-1
}
```

The `strncat` function in C implicitly includes the end-of-string null in the number of characters it copies. If you are not careful, you can write outside the array boundary. This has serious security implications.

Boundary Value analysis

- Assumption: Programmers make mistakes in processing value at and near the boundaries of equivalent classes.
- Recommendation: sample on and from near the boundaries



Boundary Value analysis

- Used in conjunction with Equivalence Partitioning
- Targets on faults in the program at the boundaries of equivalence classes.
- For example, from our previous example about insurance policy:
- E1 = {0 <= i <= 20} U1 = {i < 0}
- E2 = {20 < i <=70} U2 = {i > 120}
- E3 = {70 < i <= 120}
- Test inputs = {-1,0,1,19,20,21,69,70,71,119,120,121,}

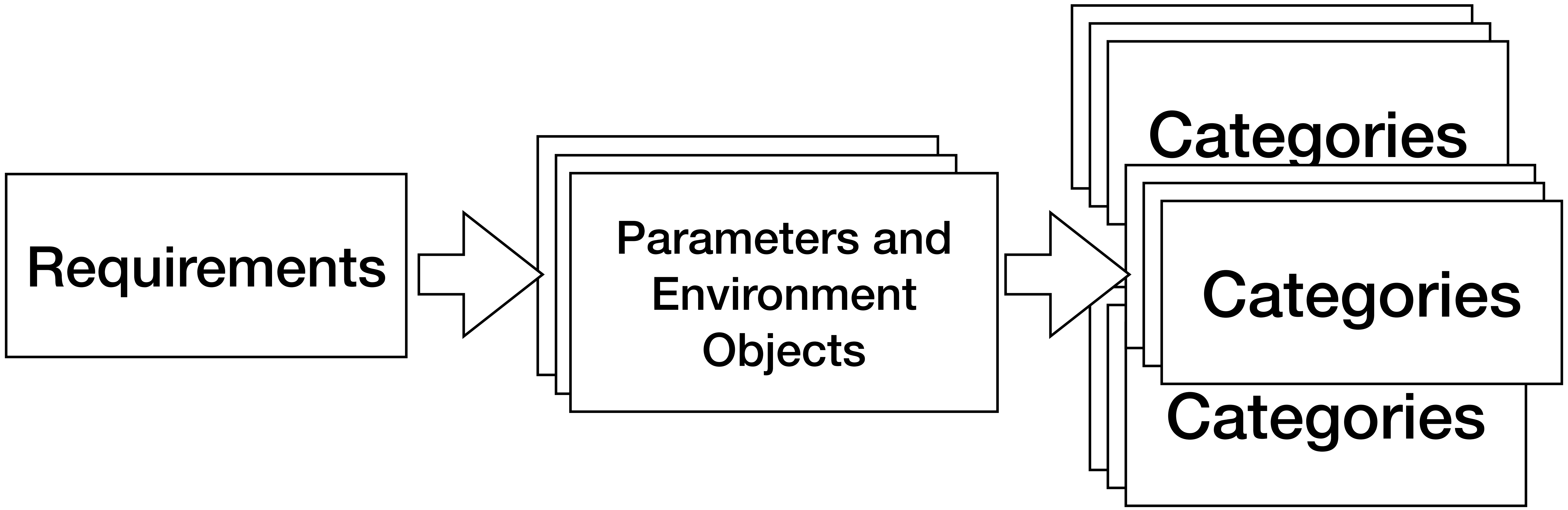
```
if(age < 20)
 { ... } // fault
if(age <= 20)
 { ... } // correct
```

Category Partition

- Most programs have multiple testable attributes, each associated with its own domain of values
- So we need to identify the distinct attributes that can be varied
  - Input
  - Environment and/or configuration
- And then systematically generate combinations of values to be tested

Category Partition

- A systematic approach to generate test data from requirements.



Thomas J. Ostrand and Marc J. Balcer, ‘The Category- Partition Method for Specifying and Generating Functional Tests’, Communications of the ACM, 31(6): 676-686, June 1988.

Category Partition Method

1. Analyse specification
2. Identify parameters and environment objects
3. Identify category (each parameter and environment objects)
    1. Categories are meaningful characteristics of each parameter and environmental object.
4. Partition categories
5. Identify constraints
6. Generate test cases

Example

- Command: `find`
- Syntax: `find <pattern> <file>`
- Function:
  - The find command is used to locate **one or more instances** of a given pattern in a text file. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is written only once, **regardless of the number of times the pattern occurs in it**.
  - The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be enclosed in quotes ("). To include a quotation mark in the pattern, you should escape with a backslash (\”).

Example

- Command: find
- Syntax: find <pattern> <file>
- Examples:
  - find john myfile : displays lines in the file myfile which contain john
  - find " john smith " myfile : displays lines in the file my file which contain the string “ john smith “
  - find " john\” smith" myfile : displays lines in the file my file which contain the string “ john" smith”

Example

- What are the features of this program ?
- What are the parameters and environmental objects?
- For each parameter and environmental object, what are the categories?

Example

- What are the features of this program ?
  - To find the occurrence of pattern strings
  - To print the corresponding lines
- What are the parameters and environmental objects?
  - pattern
  - file name
  - the actual file

Example

- For each parameter and environmental object, what are the categories?
  - Categories are meaningful characteristics of each parameter and environmental object.

Identify Categories

- For the parameter “pattern”:
  - Size
  - Quoting
  - Embedded blanks
  - Embedded quoting
- For the parameter “filename”:
  - Validity
- For the environment “file”
  - Number of occurrences of the pattern
  - Number of occurrences in a single target line

Partition Each Categories

- For the parameter “pattern”:
    - For category “size”:
        - empty
        - single character
        - many characters
        - longer than any line in the file

Partition Categories

- For the parameter “pattern”:
  - For the category “quoting”:
    - pattern is quoted
    - pattern is not quoted
    - pattern is improperly quoted
  - For the category “embedded blanks”:
    - none
    - one embedded blank
    - several embedded blanks

- For the category “embedded quotes”:
  - none
  - one embedded quote
  - several embedded quotes

Partition Categories

- For the parameter “file name”:
    - For the category “validity”:
        - file exists
        - file doesn’t exist
        - omitted
- For the environment “file”:
    - Number of occurrences of patten
        - none
        - exactly one
        - more than one

- Pattern occurrences on target line
    - once
    - more than one

Deriving Test Inputs

- How many tests do we have?
    - 4 * 3 * 3 * 3 * 3 * 3 * 2 = 1944
- Can we reduce them?

Combinatorial Interaction Testing

- Suppose we're booking a flight:
  - 1 choice (airline)
  - 2 choices (city)
  - 2 choices (outgoing date)
  - 2 choices (return date)
- Total: 8 combinations (1 * 2 * 2 * 2)

Airline	City	Out	Return
Jeju	Osaka	20 May	27 May
	Tokyo	21 May	26 May

Combinatorial Interaction Testing

- How about testing the entire Incheon Airport system?
- Possible combinations of city, out, and return dates: 1,799,736,525
- Combinatorial explosion!

Airline	City	Out	Return
79	171 cities	365 days	365 days

Combinatorial Interaction Testing

- Problem: testing all combinations is too expensive.
- Solution: testing all *interactions* between any set of t parameters. Such a solution is known as a covering array.
- Definition: A covering array *CA(t,k,v)* of size *N* is a table with *N* rows and *k* columns. Each field of *CA* contains a value in the range *0, … , v – 1*. *CA* has the following property: every combination of t values between any *t* parameters occurs in at least one row. We call *t* the strength of the covering array.

Combinatorial Interaction Testing

- CIT Problem: find a minimal test suite that covers all t-way interaction. There is a tough combinatorial problem at the foundation: minimum size is not easy to know.
- http://www.public.asu.edu/~ccolbou/src/tabby/catable.html
- Pairwise testing, i.e. CIT with $t = 2$, is the most widely studied testing technique.
  - You are likely to detect any problem that results from interaction between two input parameters.

Pairwise Testing

Combinations between all possible pairs.

Airline	City	Out	Return
Jeju	**Osaka**	20 May	26 May
Jeju	Osaka	20 May	27 May
Jeju	Osaka	21 May	26 May
Jeju	Osaka	21 May	27 May
Jeju	**Tokyo**	20 May	26 May
Jeju	Tokyo	20 May	27 May
Jeju	Tokyo	21 May	26 May
Jeju	Tokyo	21 May	27 May

Pairwise Testing

Combinations between all possible pairs.

Airline	City	Out	Return
Jeju	Osaka	**20 May**	26 May
Jeju	Osaka	20 May	27 May
Jeju	Osaka	**21 May**	26 May
Jeju	Osaka	21 May	27 May
Jeju	Tokyo	20 May	26 May
Jeju	Tokyo	20 May	27 May
Jeju	Tokyo	21 May	26 May
Jeju	Tokyo	21 May	27 May

Pairwise Testing

Combinations between all possible pairs.

Airline	**City**	**Out**	**Return**
Jeju	Osaka	20 May	**26 May**
Jeju	Osaka	20 May	**27 May**
Jeju	Osaka	21 May	26 May
Jeju	Osaka	21 May	27 May
Jeju	Tokyo	20 May	26 May
Jeju	Tokyo	20 May	27 May
Jeju	Tokyo	21 May	26 May
Jeju	Tokyo	21 May	27 May

Pairwise Testing

Combinations between all possible pairs.

Airline	City	Out	Return
Jeju	**Osaka**	**20 May**	26 May
Jeju	Osaka	20 May	27 May
Jeju	Osaka	**21 May**	26 May
Jeju	Osaka	21 May	27 May
Jeju	Tokyo	20 May	26 May
Jeju	Tokyo	20 May	27 May
Jeju	Tokyo	21 May	26 May
Jeju	Tokyo	21 May	27 May

Pairwise Testing

Combinations between all possible pairs.

Airline	City	Out	Return
Jeju	**Osaka**	20 May	**26 May**
Jeju	Osaka	20 May	**27 May**
Jeju	Osaka	21 May	26 May
Jeju	Osaka	21 May	27 May
Jeju	Tokyo	20 May	26 May
Jeju	Tokyo	20 May	27 May
Jeju	Tokyo	21 May	26 May
Jeju	Tokyo	21 May	27 May

Pairwise Testing

Combinations between all possible pairs.

Airline	City	Out	Return
Jeju	Osaka	**20 May**	26 May
Jeju	Osaka	20 May	27 May
Jeju	Osaka	**21 May**	26 May
Jeju	Osaka	21 May	27 May
Jeju	**Tokyo**	20 May	26 May
Jeju	Tokyo	20 May	27 May
Jeju	Tokyo	21 May	26 May
Jeju	Tokyo	21 May	27 May

Pairwise Testing

Combinations between all possible pairs.

Airline	City	Out	Return
Jeju	Osaka	20 May	**26 May**
Jeju	Osaka	20 May	**27 May**
Jeju	Osaka	21 May	26 May
Jeju	Osaka	21 May	27 May
Jeju	**Tokyo**	20 May	26 May
Jeju	Tokyo	20 May	27 May
Jeju	Tokyo	21 May	26 May
Jeju	Tokyo	21 May	27 May

Pairwise Testing

Combinations between all possible pairs.

Airline	City	Out	Return
Jeju	Osaka	**20 May**	**26 May**
Jeju	Osaka	20 May	**27 May**
Jeju	Osaka	21 May	26 May
Jeju	Osaka	21 May	27 May
Jeju	Tokyo	20 May	26 May
Jeju	Tokyo	20 May	27 May
Jeju	Tokyo	21 May	26 May
Jeju	Tokyo	21 May	27 May

Pairwise Testing

Combinations between all possible pairs.

Airline	City	Out	Return
Jeju	Osaka	20 May	**26 May**
Jeju	Osaka	20 May	**27 May**
Jeju	Osaka	**21 May**	26 May
Jeju	Osaka	21 May	27 May
Jeju	Tokyo	20 May	26 May
Jeju	Tokyo	20 May	27 May
Jeju	Tokyo	21 May	26 May
Jeju	Tokyo	21 May	27 May

How many rows do you need?

How many rows do you need?

Single row can contain multiple pairs!

Airline	City	Out	Return
Jeju	Osaka	20 May	27 May
Jeju	Osaka	21 May	26 May
Jeju	Osaka	20 May	26 May
Jeju	Osaka	21 May	27 May
Jeju	Tokyo	21 May	26 May
Jeju	Tokyo	20 May	27 May

How about t=3?

3-way testing

Airline	City	Out	Return
Jeju	Osaka	20 May	27 May
Jeju	Osaka	21 May	26 May
Jeju	Osaka	20 May	26 May
Jeju	Osaka	21 May	27 May
Jeju	Tokyo	21 May	26 May
Jeju	Tokyo	20 May	27 May
Jeju	**Tokyo**	**20 May**	**26 May**
Jeju	**Tokyo**	**21 May**	**27 May**

Application Area

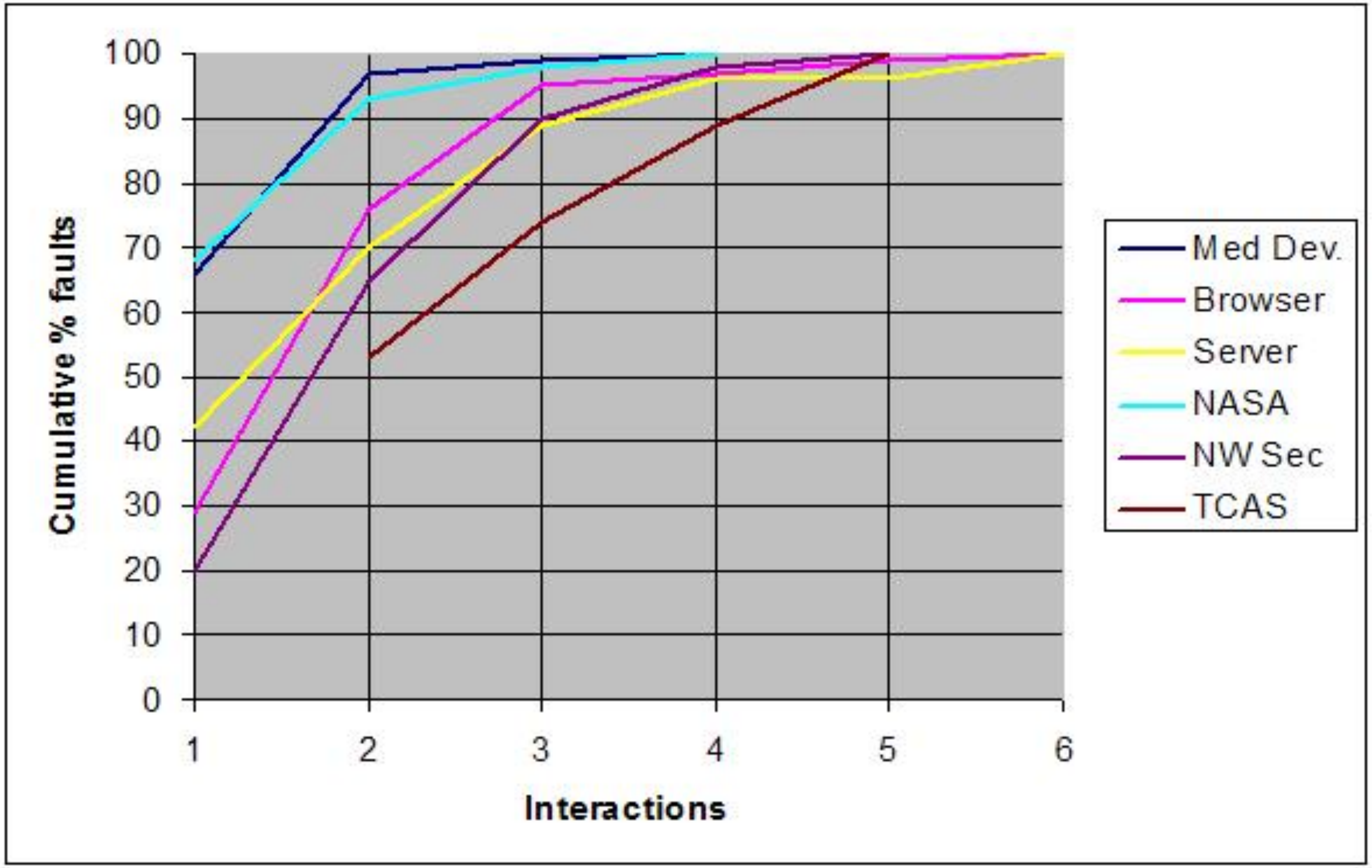
- Input Partitions
- Configurations
- Software Product Line
- …and any configurable systems

Load content	Notify pop-up blocked	Cookies	Warn before add-ons install	Remember downloads
Allow Restrict Block	Yes No	Allow Restrict Block	Yes No	Yes No

Automated Driving Controller	Collision Avoidance Braking	Parallel Parking	Lateral Range Finder	Forward Range Finder
Included None	StandardAvoidance EnhancedAvoidance None	Included None	Included None	Included None

Fault Detection

- Is higher strength always better at detecting faults?
- The answer is “it depends on the target program”, but we can analyse the general trend against a set of known faults. Empirical results state:
  - Pairwise testing discovers at least 53% of the known faults.
  - 6-way testing discovers 100% of the known faults.
- These numbers are estimates and can only provide relative guidance; the exact effectiveness will of course vary case by case.



*

results avilable at http://csrc.nist.gov/groups/SNS/acts/ftfi.html

Prioritisation

- Suppose there is a fault that can be detected by the interaction between “Jeju” and “Tokyo”.
- What is the strategy to detect this as early as possible using this 3-way test suite?

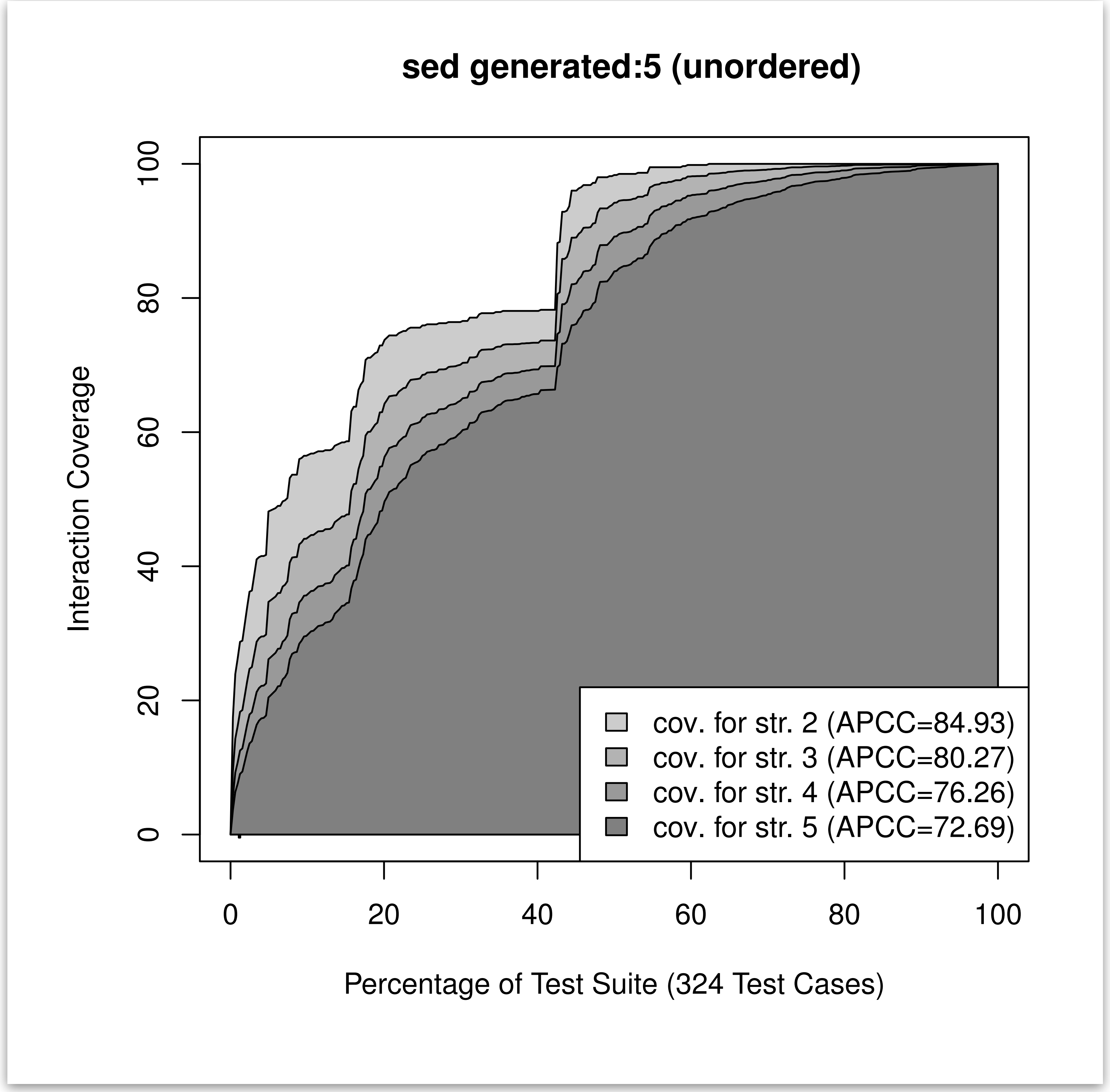
Airline	City	Out	Return
Jeju	Osaka	20 May	27 May
Jeju	Osaka	21 May	26 May
Jeju	Osaka	20 May	26 May
Jeju	Osaka	21 May	27 May
Jeju	Tokyo	21 May	26 May
Jeju	Tokyo	20 May	27 May
Jeju	Tokyo	20 May	26 May
Jeju	Tokyo	21 May	27 May

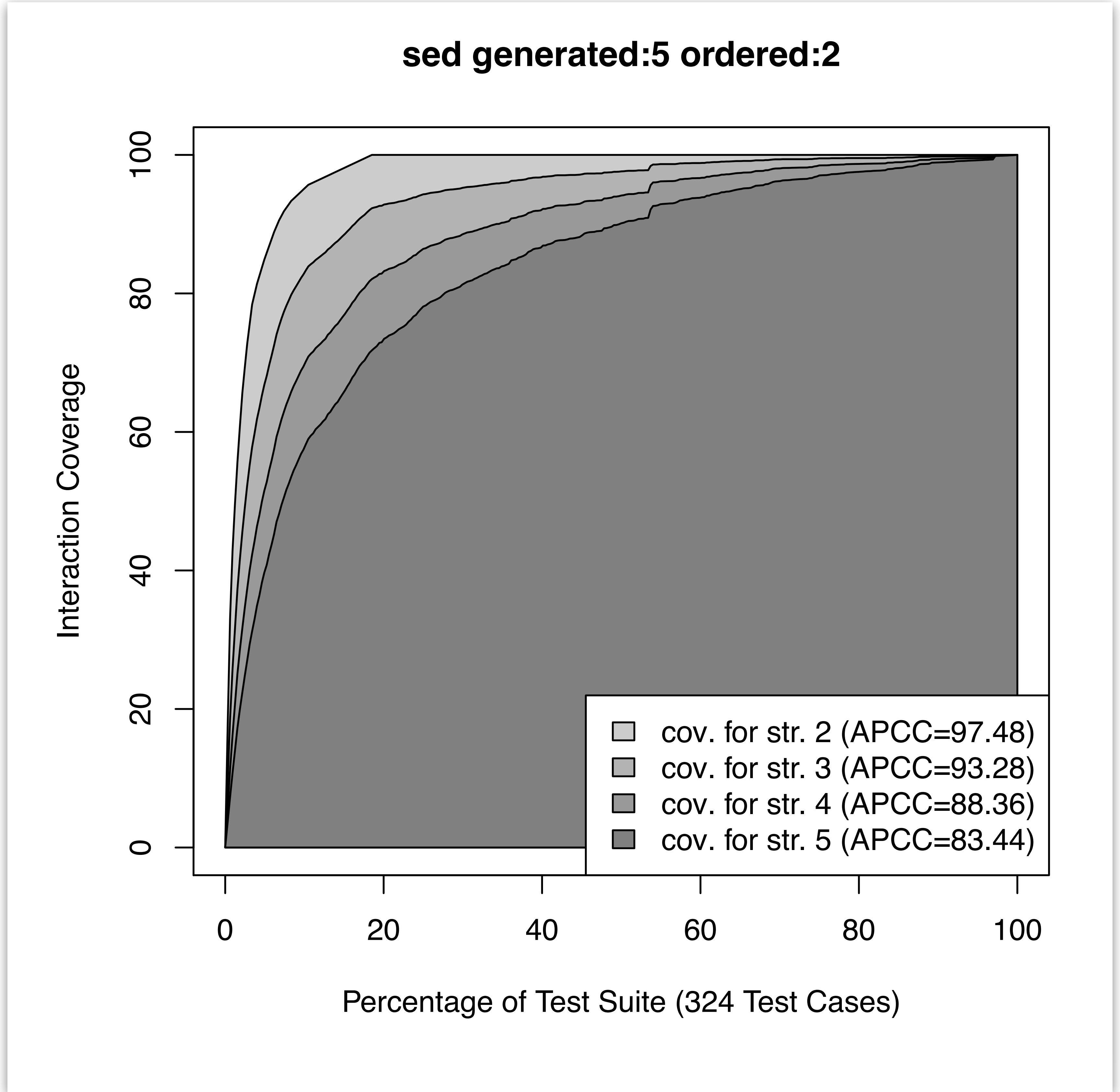
Prioritisation

- Let's count the number of new pairs that we additionally cover by executing each row.
- We should have prioritised based on the number of new pairs!

Airline	City	Out	Return	New Pairs
Jeju	Osaka	20 May	27 May	
Jeju	Osaka	21 May	26 May	
Jeju	Osaka	20 May	26 May	
Jeju	Osaka	21 May	27 May	
Jeju	Tokyo	21 May	26 May	
Jeju	Tokyo	20 May	27 May	
Jeju	Tokyo	20 May	26 May	
Jeju	Tokyo	21 May	27 May	

Prioritisation





Algorithm: Greedy

- Lei, Y., Kacker, R., Kuhn, D. R., Okun, V., & Lawrence, J. (2007, March). IPOG: A general strategy for t-way software testing. In Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the (pp. 549-556). IEEE.
- Example: Three parameters: P1, P2, P3
  - Values for parameter P1: 0,1
  - Values for parameter P2: 0,1
  - Values for parameter P3: 0,1,2
  - Objective: find a pairwise interaction test suite

Greedy algorithm IPOG-Test (int *t*, ParameterSet *ps*)

1. initialize test set *ts* to be an empty set
2. denote the parameters in *ps*, in an arbitrary order, as P_1, $P_2, \cdots$, and P_n
3. add into *ts* a test for each combination of values of the first *t* parameters
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.

Greedy approach - example

Adding all combinations of values between the first 2 parameters:

P_1	P_2
0	0
0	1
1	0
1	1

Greedy algorithm IPOG-Test (int t, ParameterSet ps)

1. initialize test set ts to be an empty set
2. denote the parameters in ps, in an arbitrary order, as P_1, $P_2, \cdots$, and P_n
3. add into ts a test for each combination of values of the first t parameters
4. for (int $i = t + 1$; $i \leq n$; $i++$){
5. let π be the set of t-way combinations of values involving parameter P_i and t-1 parameters among the first i-1 parameters
6.
7.
8.
9.
10.
11.
12.
13.
14.

Greedy approach - example

Set π = pairs to cover involving P_3:

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy algorithm IPOG-Test (int t, ParameterSet ps)

1. initialize test set ts to be an empty set
2. denote the parameters in ps, in an arbitrary order, as P_1, P_2,$\cdots$, and P_n
3. add into ts a test for each combination of values of the first t parameters
4. for (int $i = t + 1$; $i \leq n$; $i++$){
5. let π be the set of t-way combinations of values involving parameter P_i and t-1 parameters among the first i-1 parameters
6. for (each test $\gamma = (v_1, v_2, \cdots, v_{i-1})$ in test set ts) {
7. choose a value v_i of P_i and replace γ with $\gamma' = (v_1, v_2, \cdots, v_{i-1}, v_i)$ so that γ' covers the most number of combinations of values in π
8. remove from π the combinations of values covered by γ' }
9.
10.
11.
12.
13.
14.

Greedy approach - example

Adding values for P_3 in *ts*:

P_1	P_2	P_3
0	0	0
0	1	
1	0	
1	1	

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy approach - example

Adding values for P_3 in *ts*:

P_1	P_2	P_3
0	0	0
0	1	1
1	0	
1	1	

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy approach - example

Adding values for P_3 in *ts*:

P_1	P_2	P_3
0	0	0
0	1	1
1	0	1
1	1	

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy approach - example

Adding values for P_3 in *ts*:

P_1	P_2	P_3
0	0	0
0	1	1
1	0	1
1	1	0

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy algorithm IPOG-Test (int t, ParameterSet ps)

1. initialize test set ts to be an empty set
2. denote the parameters in ps, in an arbitrary order, as P_1, P_2,$\cdots$, and P_n
3. add into ts a test for each combination of values of the first t parameters
4. for (int $i = t + 1$; $i \leq n$; $i++$){
5. let π be the set of t-way combinations of values involving parameter P_i and t-1 parameters among the first i-1 parameters
6. for (each test $\gamma = (v_1, v_2, \cdots, v_{i-1})$ in test set ts) {
7. choose a value v_i of P_i and replace γ with $\gamma' = (v_1, v_2, \cdots, v_{i-1}, v_i)$ so that γ' covers the most number of combinations of values in π
8. remove from π the combinations of values covered by γ' }
9. for (each combination α in set π){
10. if (there exists a test that already covers α) {
11. remove α from π
12. } else {
13. change an existing test, if possible, or otherwise add a new test to cover α and remove it from π
    } } }
14. return ts;

Greedy approach - example

Extending *ts*:

P_1	P_2	P_3
0	0	0
0	1	1
1	0	1
1	1	0

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy approach - example

Extending *ts*:

P_1	P_2	P_3
0	0	0
0	1	1
1	0	1
1	1	0
0	0	2

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy approach - example

Extending *ts*:

P_1	P_2	P_3
0	0	0
0	1	1
1	0	1
1	1	0
0	0	2
1	1	2

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Algorithm: metaheuristic

- Simulated Annealing, a type of local search algorithm, has been proven to be effective against CIT
- B. J. Garvin, M. B. Cohen, and M. B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In 2009 1st International Symposium on Search Based Software Engineering, pages 13–22, May 2009.
- http://cse.unl.edu/~citportal/

Greedy vs. Meta-heuristics

Size comparison (average over 50 runs).

Subject	Greedy	Meta-heuristics
SPIN-S	27	19
SPIN-V	42	36
GCC	24	21
Apache	42	32
Bugzilla	21	16

*results from: Evaluating improvements to a meta-heuristic search for constrained interaction testing. Brady J. Garvin et. al, 2011

Greedy vs. Meta-heuristics

Time (sec.) comparison (average over 50 runs).

Subject	Greedy	Meta-heuristics
SPIN-S	0.2	8.6
SPIN-V	11.3	102.1
GCC	204	1902.0
Apache	76.4	109.1
Bugzilla	1.9	9.1

*results from: Evaluating improvements to a meta-heuristic search for constrained interaction testing. Brady J. Garvin et. al, 2011

Constraints

- Booking a flight

Airline	City	Out	Return
Jeju	Osaka	20 May	27 May
AirFrance	Paris		

Constraints

- Pairwise Test suite: what is wrong?
- Certain combinations should not be allowed.

Airline	City	Out	Return
Jeju	Osaka	20 May	27 May
AirFrance	Paris	20 May	27 May
Jeju	Paris	20 May	27 May
AirFrance	Osaka	20 May	27 May

Constraints

- Hard Constraints: the specific combination of parameter values are prohibited.
- Soft Constraint: not prohibited, but also not necessarily required.
    - For example, recall our `find` example. Any combination that includes an empty file as an input will throw an exception - no need to execute all other combinations that is derived from this input.

Constraints

- With real world applications, constraints tend to reduce the size of CIT test suites even further.
- Sometimes this allows us to use test suites stronger than pairwise.
  - J. Petke, M. Cohen, M. Harman, and S. Yoo. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In Proceedings of the 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013, pages 26–36, 2013.
  - J. Petke, M. B. Cohen, M. Harman, and S. Yoo. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. IEEE Transactions on Software Engineering, 41(9):901– 924, September 2015.

Constraints and Test Suite Generation

- Naive approach: generate an unconstrained test suite, remove violating rows, fill in, repeat.
- More integrated methods: various techniques first generate disallowed tuples from constraints and avoid these.

Summary

- Black box testing
  - Test the functional behaviour of the program according to the specification
- Equivalence Partitioning + Boundary Value Analysis
- Category Partition Method
- Combinatorial Interaction Testing
  - Test interactions between *t* parameters, instead of all possible combinations