Software Testing & Verification Course
University of Bergamo, Italy

Guest Lecture: Combinatorial Methods & Related Modelling Techniques in Testing

Dimitris E. Simos, SBA Research & TU Graz, Austria

May 31, 2017

Agenda of the Lecture

Structure

- Part I: Combinatorial methods in testing (introduction)
- Part II: Configuration testing
- Part III: Input testing

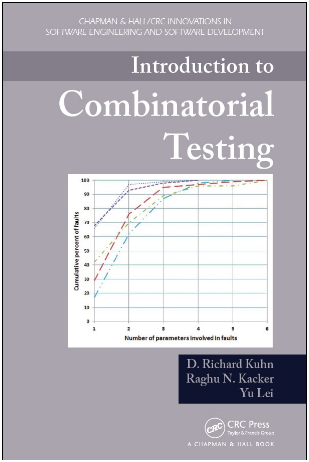
Goal

Learn the basic principles of combinatorial testing

Focus

1. Basic understanding of combinatorial testing principles
2. Software failures and their relation to combinatorial methods
3. Creation of configuration and input models
4. Examples and exercises in CITLAB tool

Further Reading from CT Textbook (Optional)



- **Chapters 1, 2, 3 and 4**

Part I

Combinatorial Methods in Testing

Outline of Part I: Combinatorial Methods in Testing

1. Motivation

Outline of Part I: Combinatorial Methods in Testing

1. Motivation

2. Software Failures and the Interaction Rule

Outline of Part I: Combinatorial Methods in Testing

1. Motivation

2. Software Failures and the Interaction Rule

3. Two Forms of Combinatorial Testing

Outline of Part I: Combinatorial Methods in Testing

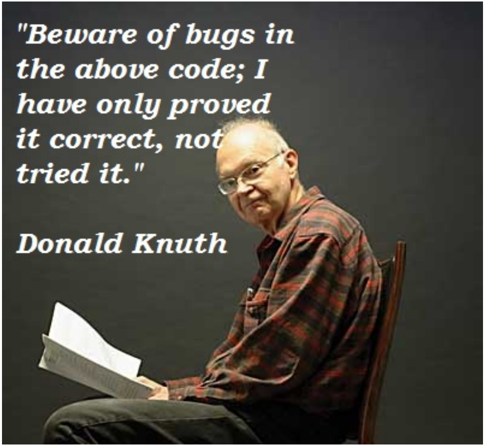
1. Motivation

2. Software Failures and the Interaction Rule

3. Two Forms of Combinatorial Testing

4. Covering Arrays

Should we Care for Software Testing?



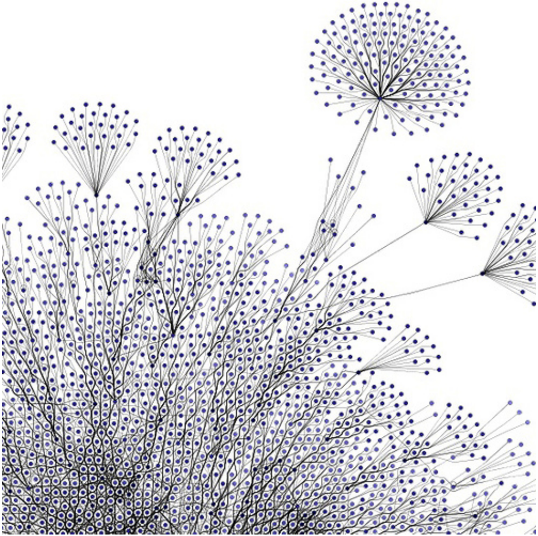
Should we Really Care for Software Testing?

Finding 90% of flaws is pretty good, right?

"Relax, our engineers found 90 percent of the flaws."

I don't think I want to get on that plane.

You cannot Test Everything



A Large Example for Testing

- Suppose we have a system with on-off switches
- 34 switches $= 2^{34} = 1.7 \times 10^{10}$ possible settings
- **What if we knew no failure involves more than 2 switch settings interacting?**

- How do we test this? (**topic of this lecture**)

Motivation for Combinatorial Methods

Key Observations

- Software testing may consume up to half of the overall software development cost
  - **Combinatorial explosion:** Exhaustive search of input space increases time needed exponentially
  - Added level of complexity for real-world testing (modelling behavior of faults)
- How can we estimate the residual risk that remains after testing?
- How can we guarantee aspects of test quality (e.g. test coverage, locating faults)?

In this Lecture

Formulate problems of software testing as combinatorial problems and then use efficient methods to tackle them

Interaction Fault

Interaction Fault

That causes failure **only** when certain values (settings) of two or more factors (parameters or variables) occur together

- One factor fault: Single value of a factor is enough to trigger failure (not interaction fault)

What does an interaction fault look like?

How does an interaction fault manifest itself in code?
Example: altitude_adj == 0 && volume < 2.2 (2-way interaction)

```
if (altitude_adj == 0 ) {
    // do something
    if (volume < 2.2)  { faulty code!  BOOM! }
    else { good code, no problem}
} else {
    // do something else
}
```

A test with altitude_adj == 0 and volume = 1 would find this

t-way Faults from NVD (National Vulnerability Database)

Pairwise (2-way) Interaction Fault

Two particular values of a pair of factors combined together to trigger a software failure

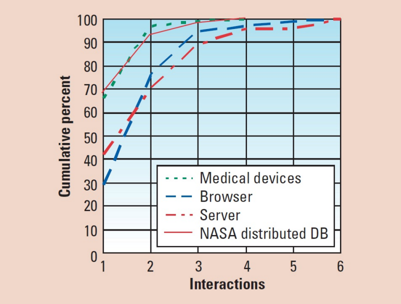
- **Example:** Single character search string in conjunction with a single character replacement string, which causes an “off by one overflow”

3-way Interaction Fault

Three particular values of a triplet of factors combined together to trigger a software failure

- **Example:** Directory traversal vulnerability when `register_globals is enabled` and `magic_quotes is disabled` and .. (dot dot) in the `page parameter`

Empirical Evidence: Fault Coverage vs. Interactions



- Rick Kuhn, Yu Lei, and Raghu Kacker. 2008. Practical Combinatorial Testing: Beyond Pairwise. IT Professional 10, 3 (May 2008), 19-23.

- 1-way interaction: enter value **age** $>$ 100 and device crashes
- 2-way interaction: **age** $>$ 100 and **zip-code** $=$ 5001, DB push fails
- 3-way interaction: **a** $=$ 2 and **b** $=$ *FALSE* and **update** $=$ *Tuesday*, system enters infinite loop

Interaction Rule and its Relation to Software Assurance

Interaction Rule

Most failures are induced by single factor faults or by the **joint combinatorial effect** (interaction) of two factors, with progressively fewer failures induced by interactions between three or more factors

Some Remarks:

- The maximum degree of interaction in actual real-world faults so far observed is **relatively** small (six to eight)
- So tests that cover **all** such few parameter (factor) interactions can be very effective
- In other words, testing all **t-way combinations** can provide strong assurance

Combinatorial Testing (CT)

What is Combinatorial Testing?

Combinatorial Strategy for Higher Interaction Testing

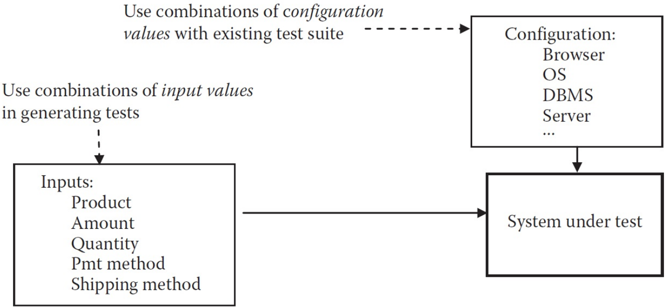
Where it can be Applied?

To system configurations, input data or both

Key Facts:

- Coverage is perceived as a quality measure of **system configurations** (configuration testing) or **test input data** (input testing) of the System under Test (SUT)
- CT guarantees 100% coverage of t-way combinations of k parameters, $t < k$; provided by **mathematical objects**, called covering arrays, that are later transformed to software artifacts
- If all faults are triggered by the interaction of t or fewer parameters, then testing all t-way combinations can provide strong assurance

Two Approaches to Combinatorial Testing



1. Use combinations of configuration parameter values: run the same test set against all t-way combinations of configuration options
2. Use combinations of input parameter values: construct a test suite that covers all t-way combinations of input transaction fields

Configuration Testing: Example

Example

Application must run on any configuration of OS, browser, protocol, CPU and DBMS (very efficient for interoperability testing)

Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv4	Intel	MySQL
2	XP	Firefox	IPv6	AMD	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySQL
5	OS X	IE	IPv4	Intel	Sybase
6	OS X	Firefox	IPv4	Intel	Oracle
7	RHEL	IE	IPv6	AMD	MySQL
8	RHEL	Firefox	IPv4	Intel	Sybase
9	RHEL	Firefox	IPv4	AMD	Oracle
10	OS X	Firefox	IPv6	AMD	Oracle

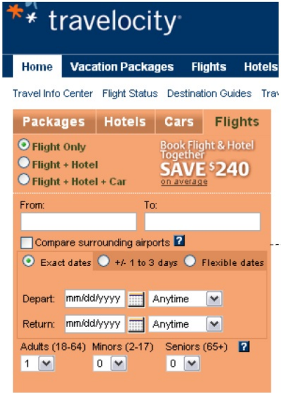
Figure: Pairwise test configurations

- There is no Linux and IE configuration? CA tools can avoid such invalid combinations (**details later**)

Input Testing: Example

Example

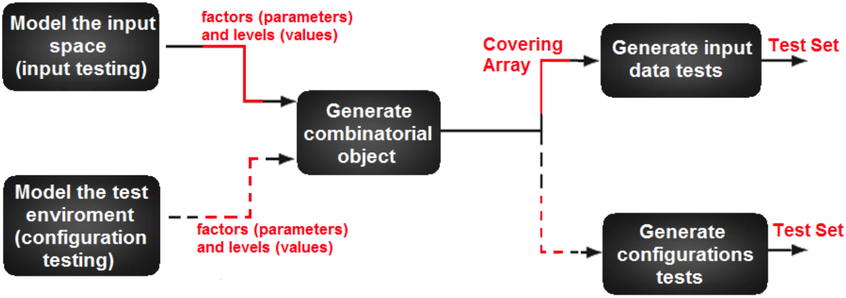
Testing of a booking application (many input fields)



Many values per variable
Need to abstract values
But we can still increase information per test

Plan: flt, flt+hotel, flt+hotel+car
From: CONUS, HI, Europe, Asia …
To: CONUS, HI, Europe, Asia …
Compare: yes, no
Date-type: exact, 1to3, flex
Depart: today, tomorrow, 1yr, Sun, Mon …
Return: today, tomorrow, 1yr, Sun, Mon …
Adults: 1, 2, 3, 4, 5, 6
Minors: 0, 1, 2, 3, 4, 5
Seniors: 0, 1, 2, 3, 4, 5

The Combinatorial Test Design Process (I)



- Modelling of input space or the environment is not exclusive and one might apply either one or both depending on the SUT

The Combinatorial Test Design Process (II)

Combinatorial Test Design Process

1. Model the input space and/or configuration space; The model is expressed in terms of factors (parameters) and respective levels (values)
2. The model is input to (mainly) an **algorithmic procedure** to generate a combinatorial object which is simply an array of symbols
3. Every row of the generated array is used to output a test case for testing the System Under Test (SUT)

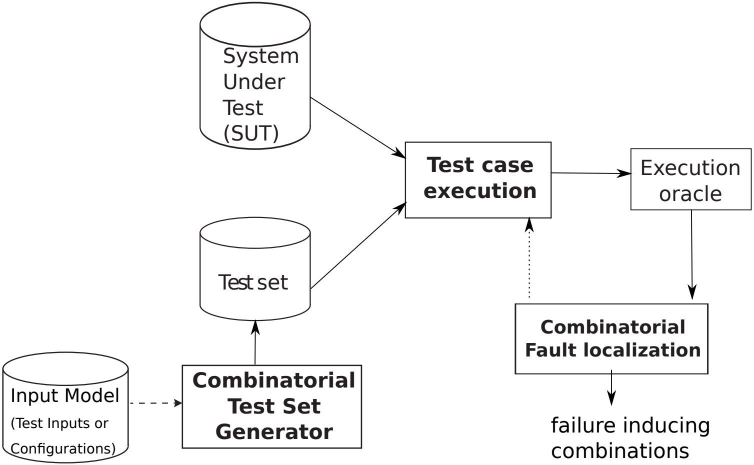
Benefit

Steps 2 and 3 can be automated

Note

We consider combinatorial testing as a black-box testing technique

Overview of a Combinatorial Testing Framework



Challenges for Combinatorial Testing

1. Modelling of the test space (configuration space and/or input space) including **specification** of test factors, test settings and their constraints
2. Efficient generation of t-way test suites, especially involving support of constraints
3. Determination of the **expected** behavior of the SUT for each possible test case and checking whether the **actual** behavior agrees with the expected one
4. Identification of the **failure-inducing** test value combinations from pass/fail results of CT
5. Integration of CT in the **existing infrastructures** for testing

Covering Arrays

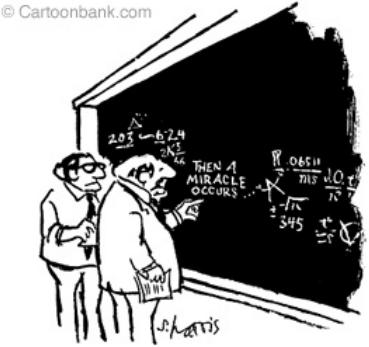
Covering Arrays $CA(N; t, k, v)$ of Strength t

A CA is an $N \times k$ array in which entries are from a finite set S of v symbols each such that each $N \times t$ subarray contains each possible t-tuple at least once

Covering Arrays $CA(N; t, k, v)$ Properties

- Cover all t-way combinations of k input parameters at least once
- The parameter t is called the strength of the array
- Input parameters have v total values each
- Such a mathematical construct has N total rows (tests)

Definition too Mathematical?



"I think you should be more explicit here in step two."

Example of a Covering Array for Software Testing

System with 3 boolean input parameters a, b, c

- Could be function, application, configuration file, etc.
- Exhaustive test set: $2^3 = 8$ tests
- 2-way covering array (test set): 4 tests

a	b	c	(a, b)	(b, c)	(a, c)

Table: 2-way test set (left) covering all pairs of parameters (right)

Example of a Covering Array for Software Testing

System with 3 boolean input parameters a, b, c

- Could be function, application, configuration file, etc.
- Exhaustive test set: $2^3 = 8$ tests
- 2-way covering array (test set): 4 tests

a	b	c	(a, b)	(b, c)	(a, c)
0	0	0	(0, 0)	(0, 0)	(0, 0)

Table: 2-way test set (left) covering all pairs of parameters (right)

Example of a Covering Array for Software Testing

System with 3 boolean input parameters a, b, c

- Could be function, application, configuration file, etc.
- Exhaustive test set: $2^3 = 8$ tests
- 2-way covering array (test set): 4 tests

a	b	c	(a, b)	(b, c)	(a, c)
0	0	0	(0, 0)	(0, 0)	(0, 0)
0	1	1	(0, 1)	(1, 1)	(0, 1)

Table: 2-way test set (left) covering all pairs of parameters (right)

Example of a Covering Array for Software Testing

System with 3 boolean input parameters a, b, c

- Could be function, application, configuration file, etc.
- Exhaustive test set: $2^3 = 8$ tests
- 2-way covering array (test set): 4 tests

a	b	c	(a, b)	(b, c)	(a, c)
0	0	0	(0, 0)	(0, 0)	(0, 0)
0	1	1	(0, 1)	(1, 1)	(0, 1)
1	0	1	(1, 0)	(0, 1)	(1, 1)

Table: 2-way test set (left) covering all pairs of parameters (right)

Example of a Covering Array for Software Testing

System with 3 boolean input parameters a, b, c

- Could be function, application, configuration file, etc.
- Exhaustive test set: $2^3 = 8$ tests
- 2-way covering array (test set): 4 tests

a	b	c	(a, b)	(b, c)	(a, c)
0	0	0	(0, 0)	(0, 0)	(0, 0)
0	1	1	(0, 1)	(1, 1)	(0, 1)
1	0	1	(1, 0)	(0, 1)	(1, 1)
1	1	0	(1, 1)	(1, 0)	(1, 0)

Table: 2-way test set (left) covering all pairs of parameters (right)

Example of a Covering Array for Software Testing

System with 3 boolean input parameters a, b, c

- Could be function, application, configuration file, etc.
- Exhaustive test set: $2^3 = 8$ tests
- 2-way covering array (test set): 4 tests

a	b	c	(a, b)	(b, c)	(a, c)
0	0	0	(0, 0)	(0, 0)	(0, 0)
0	1	1	(0, 1)	(1, 1)	(0, 1)
1	0	1	(1, 0)	(0, 1)	(1, 1)
1	1	0	(1, 1)	(1, 0)	(1, 0)

Table: 2-way test set (left) covering all pairs of parameters (right)

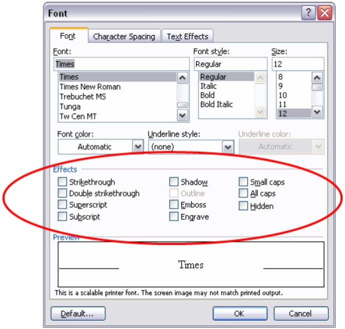
Covering Arrays (CAs) for Software Testing

- Used to generate tests for revealing software faults
- Received attention from standardization bodies such as US NIST

How to Use this Knowledge in Testing?

Example

Testing of a word-processing application having 10 effects to highlight text (each can be on or off)



- The font-processing function receives these settings as input ($2^{10} = 1024$ possible combinations)

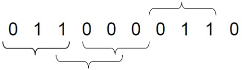
How Many Tests would it Take?

- What if our budget is too limited for these $2^{10} = 1024$ tests?

Testing of 3-way Interactions

- There are $\binom{10}{3} = 120$ 3-way interactions of the application settings (font-processing effects)
- Naively, we need $120 \times 2^3 = 960$ tests
- Since we can pack 3 triples into each test, we need no more than 320 tests

- Each test exercises many triples (3-tuples)



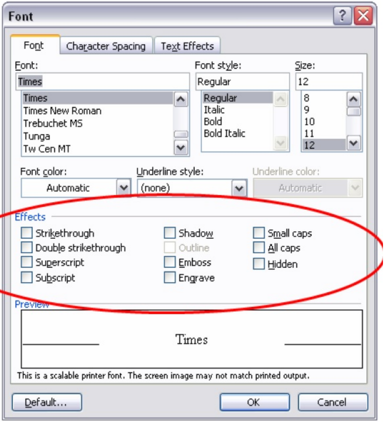
OK, OK, what's the smallest number of tests we need?

Resulting 3-way Test Set for Word Application[a]

[a]Thanks to Rick Kuhn. NIST

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

0 = effect off
1 = effect on



13 tests for **all 3-way combinations**

2^{10} = **1,024** tests for **all combinations**

Covering Arrays give Minimum Tests (I)

Testing Scenario

Testing of an industrial switch network with 29 factors (switches) each one having 2 test settings (ON/OFF)

- **Exhaustive testing:** Number of possible (29-way) tests:
  - $2^{29} = 536,870,912$

Covering Arrays give Minimum Tests (II)

Testing Scenario

Testing of an industrial switch network with 29 factors (switches) each one having 2 test settings (ON/OFF)

- Pairs of test settings, e.g. number of 2-tuples to cover: $\binom{29}{2} \times 2^2 = 406 \times 4 = 1624$
- How many tests needed to cover all such 1624 pairs? (**guess**)

Covering Arrays give Minimum Tests (II)

Testing Scenario

Testing of an industrial switch network with 29 factors (switches) each one having 2 test settings (ON/OFF)

- Pairs of test settings, e.g. number of 2-tuples to cover: $\binom{29}{2} \times 2^2 = 406 \times 4 = 1624$
- How many tests needed to cover all such 1624 pairs? (**guess**)

A CA of only 10 rows can cover all 1624 pairs

- Each row is a test (comprised of test settings)

Part II

Configuration Testing

Outline of Part II: Configuration Testing

5. Preliminaries

Outline of Part II: Configuration Testing

5. Preliminaries

6. Runtime Environment Configurations

Outline of Part II: Configuration Testing

5. Preliminaries
6. Runtime Environment Configurations
7. Invalid Combinations and Constraints

Outline of Part II: Configuration Testing

Configuration Space

Configuration Space

The **configuration space** of a (test) environment *P* consists of all possible (existing) settings of the environment factors (parameters) under which *P* could be used

Example: Test Configuration of a Printer

- Windows 7, DSL connection and a PC with 8 GB of memory, is one possible configuration
- Different versions of OS and printer drivers, can be **combined** to create several test configurations of a printer

Methodology: How to Create a Configuration Model

Model the Configuration Space

1. Identify all possible settings of the environment
2. Map them to configurable parameters
3. Select (combinations of) values of configurable parameters
4. Express the resulting model in terms of factors (parameters) and respective levels (values)

- The model is plugged-into the combinatorial test design process to generate test cases (using a proper algorithmic procedure)
- **Combinatorial Aspect:** Achieve combinatorial coverage of all possible *t*-way configuration parameter values

Configuration Testing: Example

Example

Application must run on any configuration of OS, browser, protocol, CPU and DBMS (very efficient for interoperability testing)

Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv4	Intel	MySQL
2	XP	Firefox	IPv6	AMD	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySQL
5	OS X	IE	IPv4	Intel	Sybase
6	OS X	Firefox	IPv4	Intel	Oracle
7	RHEL	IE	IPv6	AMD	MySQL
8	RHEL	Firefox	IPv4	Intel	Sybase
9	RHEL	Firefox	IPv4	AMD	Oracle
10	OS X	Firefox	IPv6	AMD	Oracle

Figure: Pairwise test configurations

- There is no Linux and IE configuration? CA tools can avoid such invalid combinations (**today's lecture**)

Exercise: Interoperability Testing

Our Task

- App should work on all combinations of platform options
- Develop a set of **test configurations** in CITLAB tool
- That allows testing across all *t*-way combinations of these options

Testing of an Application (Our Configuration Space)

- Five configuration parameters
- A total of 72 configurations may be set
- At $t = 5$ the number of tests is the same as exhaustive testing (**why?**)
- List of configuration options (below)

Parameter	**Values**
Operating system	XP, OS X, RHL
Browser	IE, Firefox
Protocol	IPv4, IPv6
CPU	Intel, AMD
DBMS	MySQL, Sybase, Oracle

CITLAB Input

- CITLAB input includes the names of parameters, types, and possible values

```
Model internet

Parameters:
    Enumerative OS { XP OS_X RHL };
    Enumerative Browser { IE Firefox };
    Enumerative Protocol { IPv4 IPv6 };
    Enumerative CPU { Intel AMD };
    Enumerative DBMS { MySQL Sybase Oracle };
end
```

Output Test Configurations covering 2-way Combinations

Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv6	AMD	MySQL
2	XP	Firefox	IPv4	Intel	Sybase
3	XP	Firefox	IPv6	AMD	Oracle
4	OS_X	IE	IPv4	AMD	MySQL
5	OS_X	IE	IPv6	AMD	Sybase
6	OS_X	Firefox	IPv6	Intel	Oracle
7	RHL	Firefox	IPv4	Intel	MySQL
8	RHL	Firefox	IPv6	AMD	Sybase
9	RHL	IE	IPv4	Intel	Oracle

Number of Combinatorial Tests

- CT results in drastically smaller test sets (even for such a small example)

t	# Tests	% of Exhaustive
2	10	14
3	18	25
4	36	50
5	72	100

What is the Relation of the Test Set to Covering Arrays?

- Fixed-value $CA(N; t, k, v)$: An $N \times k$ matrix such that every t-columns contains all t-tuples **at least once**
- Another notation $CA(N, v^k, t)$
- In past example, not all parameter had the same number of parameter values!

Mixed-value covering array $CA(N, v_1^{k_1}, v_2^{k_2}, \ldots, v_n^{k_n}, t)$ is a variation of fixed value CA

- k_1 columns have v_1 distinct values, k_2 columns have v_2 distinct values,. . .,k_n columns have v_n distinct values,
- $k = k_1 + k_2 + \ldots + k_n$

Hint

To see the abstract mathematical object replace the domain range of the configuration parameters with $\{0, 1, 2\}$ for three-value parameters and $\{0, 1\}$ for two-value parameters

Constraints among Parameter Values

- So far, we have assumed that the set of possible values for parameters never changes
- Thus, a covering array of t-way combinations of possible values would contain combinations that:
  - either would occur in the systems under test
  - or could occur and must therefore be tested

Some Combinations never occur in Practice

The IE browser is never used on Linux systems; so, it would be impossible to create a configuration that specified IE on a Linux system

Practical testing requires the consideration of constraints!

How to deal with Invalid Combinations?

- We cannot simply delete tests with these untestable combinations
- That would result in **losing other combinations** that are essential to test but are not covered by other tests
- **Example:** deleting tests 4 and 5 would mean that we would also lose the test for Linux with the IPv4 and IPv6 protocol

Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv6	AMD	MySQL
2	XP	Firefox	IPv4	Intel	Sybase
3	XP	Firefox	IPv6	AMD	Oracle
4	OS_X	IE	IPv4	AMD	MySQL
5	OS_X	IE	IPv6	AMD	Sybase
6	OS_X	Firefox	IPv6	Intel	Oracle
7	RHL	Firefox	IPv4	Intel	MySQL
8	RHL	Firefox	IPv6	AMD	Sybase
9	RHL	IE	IPv4	Intel	Oracle

Specification of Constraints

- We can define constraints, which tell the tool not to include **specified combinations** in the generated test configurations
- **Example:** `(OS !="XP") => (Browser="Firefox")`
- Constraints (usually) **reduce** the size of the test set
- Result in **revised** test configuration array (below)

```
Constraints:
    # OS != OS.XP => Browser == Browser.Firefox #
end
```

Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv4	AMD	MySQL
2	XP	IE	IPv4	Intel	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS_X	Firefox	IPv4	Intel	MySQL
5	OS_X	Firefox	IPv6	AMD	Sybase
6	OS_X	Firefox	IPv6	AMD	Oracle
7	RHL	Firefox	IPv6	AMD	MySQL
8	RHL	Firefox	IPv4	AMD	Sybase
9	RHL	Firefox	IPv4	Intel	Oracle
10	XP	Firefox	IPv6	Intel	MySQL

Constraints among Parameters

- Other untestable combinations may arise in practice
- Some parameters become inactive when others are set to particular values

Example

- Suppose testers also wanted to consider additional software, i.e. Java and Microsoft .Net
- Desirable to add two additional parameters: `java_version` and `dot_net_version`

However, Java can be present on both Windows and Linux platforms, but we must deal with the problem that .Net will not be present on a Linux system

Specification of Constraints among Parameters

Two different parameter sets:

- If `(OS == "Windows")` then the parameters are `OS, browser, protocol, cpu, dbms, java_version, dot_net_version`
- If `(OS == "Linux")` then the parameters are `OS, browser, protocol, cpu, dbms, java_version`

Note

Practical testing problems may be more complex than this example, and may have **multiple constraints among parameters**

Exercise: Testing Android Configurations

Our Task

- App should work on all combinations of platform options
- Develop a set of **test configurations** in CITLAB tool
- That allows testing across all *t*-way combinations of these options

Testing Android Configurations (Our Configuration Space)

- Resource configuration file for Android apps
- A total of 35 options may be set

HARDKEYBOARDHIDDEN_NO
HARDKEYBOARDHIDDEN_UNDEFINED
HARDKEYBOARDHIDDEN_YES

KEYBOARDHIDDEN_NO
KEYBOARDHIDDEN_UNDEFINED
KEYBOARDHIDDEN_YES

KEYBOARD_12KEY
KEYBOARD_NOKEYS
KEYBOARD_QWERTY
KEYBOARD_UNDEFINED

NAVIGATIONHIDDEN_NO
NAVIGATIONHIDDEN_UNDEFINED
NAVIGATIONHIDDEN_YES

NAVIGATION_DPAD
NAVIGATION_NONAV
NAVIGATION_TRACKBALL
NAVIGATION_UNDEFINED
NAVIGATION_WHEEL

ORIENTATION_LANDSCAPE
ORIENTATION_PORTRAIT
ORIENTATION_SQUARE
ORIENTATION_UNDEFINED

SCREENLAYOUT_LONG_MASK
SCREENLAYOUT_LONG_NO
SCREENLAYOUT_LONG_UNDEFINED
SCREENLAYOUT_LONG_YES

SCREENLAYOUT_SIZE_LARGE
SCREENLAYOUT_SIZE_MASK
SCREENLAYOUT_SIZE_NORMAL
SCREENLAYOUT_SIZE_SMALL
SCREENLAYOUT_SIZE_UNDEFINED

TOUCHSCREEN_FINGER
TOUCHSCREEN_NOTOUCH
TOUCHSCREEN_STYLUS
TOUCHSCREEN_UNDEFINED

Android Configuration Model

- This set of Android options has 172,800 possible configurations
- $3 \times 3 \times 4 \times 3 \times 5 \times 4 \times 4 \times 5 \times 4 = 172,800$ configurations
- A $3^3 4^4 5^2$ system

```
Model Android

Parameters:
    Enumerative HARDKEYBOARDHIDDEN { NO UNDEFINED YES };
    Enumerative KEYBOARDHIDDEN { NO UNDEFINED YES };
    Enumerative KEYBOARD { 12KEY NOKEYS QWERTY UNDEFINED };
    Enumerative NAVIGATIONHIDDEN { NO UNDEFINED YES };
    Enumerative NAVIGATION { DPAD NONAV TRACKBALL UNDEFINED WHEEL };
    Enumerative ORIENTATION { LANDSCAPE PORTRAIT SQUARE UNDEFINED };
    Enumerative SCREENLAYOUT_LONG { MASK NO UNDEFINED YES };
    Enumerative SCREENLAYOUT_SIZE { LARGE MASK NORMAL SMALL UNDEFINED };
    Enumerative TOUCHSCREEN { FINGER NOTOUCH STYLUS UNDEFINED };
end
```

Cost and Practical Considerations

- If each test suite can be run in 15 min
- Roughly 24 staff-years to complete the testing for an app
- With salary and benefit costs for each tester of 150, 000 EUR, the cost of testing an app will be more than 3 million EUR
- Virtually impossible to return a profit for most apps
- Number of combinatorial tests is a **fraction** of an exhaustive test set

t	# Tests	% of Exhaustive
2	29	0.02
3	137	0.08
4	625	0.4
5	2532	1.5
6	9168	5.3

Part III

Input Testing

Outline of Part III: Input Testing

9. Preliminaries

Outline of Part III: Input Testing

Outline of Part III: Input Testing

Outline of Part III: Input Testing

Input Space

Input Space

The **input space** of a (test) program P consists of k-tuples of values that could be input to P during execution

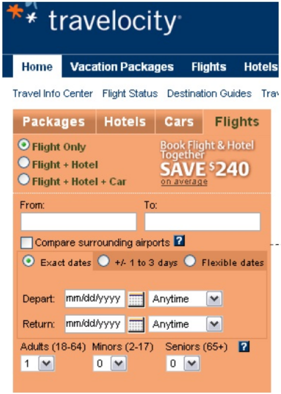
Example: Sample Program

- Consider program P that takes two integers $x > 0$ and $y > 0$ as inputs (i.e. $P(x, y)$)
- The input space of P is the set of **all** pairs of positive integers

Input Testing: Example

Example

Testing of a booking application (many input fields)



Many values per variable

Need to abstract values

But we can still increase information per test

Plan: flt, flt+hotel, flt+hotel+car
From: CONUS, HI, Europe, Asia …
To: CONUS, HI, Europe, Asia …
Compare: yes, no
Date-type: exact, 1to3, flex
Depart: today, tomorrow, 1yr, Sun, Mon …
Return: today, tomorrow, 1yr, Sun, Mon …
Adults: 1, 2, 3, 4, 5, 6
Minors: 0, 1, 2, 3, 4, 5
Seniors: 0, 1, 2, 3, 4, 5

Methodology: How to Create an Input Model

Model the Input Space

1. Identify possible parameters of the (test) program
2. Map them to input parameters
3. Select (combinations of) input data values
4. Express the resulting model in terms of factors (parameters) and respective levels (values)

- The model is plugged-into the combinatorial test design process to generate test cases (using a proper algorithmic procedure)
- **Combinatorial Aspect:** Combinatorial coverage of input data values is required for tests constructed

Testing of F-16 Ventral Fin

- **Problem:** Unknown factors causing failures of F-16 ventral fin
- LANTIRN pod carriage on the F-16

F-16 Ventral Fin Damage on Flight with LANTIRN[a]

[a]Thanks to Rick Kuhn. US NIST

It's not supposed to look like this:

Input Model for Testing of F-16 Ventral Fin

- **Original solution:** Lockheed Martin engineers spent many months with wind tunnel tests and expert analysis to consider interactions that could cause the problem
- **CT solution:** modelling and simulation using CITLAB

Parameter	Values
Aircraft	15, 40
Altitude	5k, 10k, 15k, 20k, 30k, 40k, 50k
Maneuver	hi-speed throttle, slow accel/dwell, L/R 5 deg side slip, L/R 360 roll, R/L 5 deg side slip, Med accel/dwell, R-L-R-L banking, Hi-speed to Low, 360 nose roll
Mach (100^{th})	40, 50, 60, 70, 80, 90, 100, 110, 120

How were the parameter values selected??

Methods to Select Representative Values

Modelling Methods

Category or equivalence partitioning and boundary value analysis

- **Objective:** partition the input space such that any value selected from the **partition** will affect the system under test in the same way as any other value in the same class of the partition
- That is, from a testing standpoint, the values in the same class of a partition are equivalent (hence the name "equivalence class")
- Thus, ideally if a test case contains a parameter x that has value y, replacing y with any other value from the same class of the partition will not affect the test case result
- This ideal may not always be achieved in practice

Testing of an Access Control Module

SUT: Access Control Module

A program that implements a certain policy

Access is allowed if and only if:

- The subject is an employee
  - **AND** the current time is between 9 a.m. and 5 p.m.
  - **AND** it is not a weekend
  - **OR** the subject is an employee with a special authorization code
  - **OR** the subject is an auditor AND the time is between 9 a.m. and 5 p.m. (not constrained to weekdays)

Exercise: Testing an Access Control Module

Our Task

- The values for a particular access attempt would be passed to a module that returns a ”grant” or ”deny” access decision
- Using a **function call** such as `access_decision(emp, time, day, auth, aud)`
- Develop a suitable input parameter model

Testing of an Access Control Module (Our Input Space)

- We are dealing with input parameters rather than configuration options
- Select representative values (supplemented with extreme values) for the hour parameter

```
emp:  boolean;
time:  0..1440;  // time in minutes
day:   {m,tu,w,th,f,sa,su};
auth: boolean;
aud:  boolean;
```

Parameters and Values for Access Control Example

Parameter	Values
emp	0, 1
time	??
day	m,tu,w,th,f,sa,su
auth	0, 1
aud	0, 1

How to Select Representative Values?

- Select values from various points on the range of a parameter (simple approach)
- However, partitions are best determined from the **specification**

Example: Access Control Module

9 AM and 5 PM are significant; so 0540 (9h past midnight in minutes) and 1020 (17h past midnight in minutes) could be used to determine the appropriate partitions

Boundary Value Analysis

- Ideally, the program should behave the same for any of the times within the partitions
- It should not matter whether the time is 4:00 AM or 7:03 AM (the specification treats both these times the same)
- Similarly, it should not matter which time between the hours of 9 AM and 5 PM is chosen
- The access control program should behave the same for 10:20 AM and 2:33 PM

Boundary Value Analysis

Select values at each boundary and at the smallest possible unit on either side of the boundary, for three values per boundary

- One possible **selection of values** for the time parameter would then be: 0000, 0539, 0540, 0541, 1019, 1020, 1021, and 1440

Number of Tests for the Access Control Module

- The total number of combinations is $2 \times 8 \times 7 \times 2 \times 2 = 448$
- Generating covering arrays for $t = 2$ through 4 results in the following number of tests

t	# Tests
2	56
3	112
4	224

Sample of Faulty Code (2-way Interaction Fault)

- If two boolean conditions are **true**, faulty code is executed resulting in a failure

```
if (pressure < 10) {
    //do something
    if (volume > 300) {
                //faulty code! BOOM!
        } else {
                //good code, no problem
        }
} else {
   //do something else
}
```

- The branches `pressure < 10` and `volume > 300` are correct and the fault occurs in the code that is reached when **these conditions are true**
- Any 2-way covering array with values for pressure and volume that **will make the conditions true** can detect the problem

Exercise: Detecting a t-way Fault

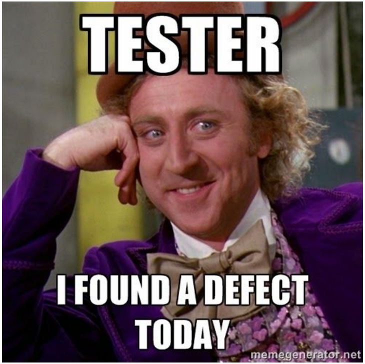
Our Task

Develop a t-way covering array, for suitable *t*, capable of detecting the following fault:

```
if ((A < 10 || B > 0) && C > 90) faulty code
else correct code
```

Solution

A 2-way array is needed, because either `A && C` or `B && C` will cause a branch into the faulty code



Summary

Highlights

1. Applications of **combinatorial methods** to problems of software testing:
   - interaction rule and t-way interaction faults
   - can be used as configuration testing and/or input testing
2. Combinatorial testing guarantees **100% t-way coverage**
   - provided by mathematical objects, called covering arrays
   - many available CA generation tools
3. Many practical exercises for **configuration testing** and **input testing** using the CITLAB tool

Questions - Comments

Thank you for your Attention!

dsimos@ist.tugraz.at