Combinatorial Testing Explained

Quality Engineered Software and Testing Conference
QUEST 2009
Chicago, IL, USA

Peter Zimmerer
Principal Engineer
Siemens AG, CT SE 1
Corporate Technology
Corporate Research and Technologies
Software & Engineering, Development Techniques
81739 Munich, Germany
peter.zimmerer@siemens.com
http://www.siemens.com/research-and-development/
http://www.ct.siemens.com/
Contents

Introduction – Examples and problem statement

Methods

Experiences

Tools

Summary
Example: Developer unit testing

Testing of a single method of class `java.util.logging.Logger` from J2SE 5.0

- 5 parameters with different number of values:
  ```java
def logrb(Level level,  
            String sourceClass, String sourceMethod,  
            String bundleName, String msg)
```

  *Log a message, specifying message level identifier, source class, source method, name of resource bundle to localize, and string message*

- E.g. level can be SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL, OFF, i.e. 9 different values,
  - 3 class names with 3 method names each,
  - 2 resource bundle names, and 3 string messages

- **486 (\(= 9 \times 3 \times 3 \times 2 \times 3\))** possible parameter combinations exist

**Which parameter combinations shall be selected?**
Example: System black-box testing

Electronic bookstore\(^1\)

- 5 parameters with different number of values

<table>
<thead>
<tr>
<th>TYPE OF CREDIT CARD</th>
<th>CREDIT CARD NUMBER</th>
<th>CREDIT CARD EXPIRATION DATE (YEARS FROM TODAY)</th>
<th>PRODUCT TYPE PURCHASED</th>
<th>QUANTITY PURCHASED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amex</td>
<td>Correct</td>
<td>50</td>
<td>Book</td>
<td>1</td>
</tr>
<tr>
<td>Discover</td>
<td>Incorrect Length</td>
<td>Invalid Year</td>
<td>Video</td>
<td>0</td>
</tr>
<tr>
<td>Visa</td>
<td>Invalid Digits</td>
<td>Today</td>
<td>Software</td>
<td>-1</td>
</tr>
<tr>
<td>MasterCard</td>
<td></td>
<td>Yesterday</td>
<td>Book, Software, Videos</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Invalid Character</td>
<td>Book, Software</td>
<td></td>
</tr>
</tbody>
</table>

- 1200 (= 4 * 3 * 5 * 5 * 4) possible parameter combinations exist

Which parameter combinations shall be selected?

\(^1\)Ref.: Elfriede Dustin, Orthogonally Speaking, STQE Magazine, Sep/Oct 2001
Example: Configuration testing

System under test with 4 components each of which has 3 possible elements

- Calling phone
  - Regular phone
  - Mobile phone
  - Coin phone
- Call type
  - Local call
  - Long distance call
  - Toll free
- Access
  - ISDN
  - PBX
  - Loop
- Called phone
  - Mobile phone
  - Regular phone
  - Pager

- Suite of system test cases exists and has to be executed
- Overall number of possible configurations: \( 81 = 3 \times 3 \times 3 \times 3 \)

Which configurations shall be selected?
Problem statement: Combinatorial testing

Given:

\( n \) independent parameters \( P_1, P_2, \ldots, P_n \)

with \( m_i \) different values each with \( i \in \{1, \ldots, n\} \):

Number of possible combinations:

\[ m_1 \times m_2 \times m_3 \times \ldots \times m_n \]

Testing all possible parameter combinations results in an astronomical number of test cases which is infeasible and inefficient.

Wanted:

An adequate test case design method to reduce the number of test cases while enhancing coverage and quality of tests.

Test case design methods like equivalence class partitioning, boundary value analysis, etc. do not fit here!

Support using risk-based methods is limited.

\(^2\)Parameters can be input, output, valid, invalid.
Methods (1)

Random choice
- Create a test suite by randomly sampling test cases from the complete set of test cases based on some input distribution (often uniform distribution)
- Detection removal efficiency of random choice:
  - There are studies / papers showing advantages of systematic test design
  - There are also studies / papers showing advantages of random testing
    → *But do you really want to design your test cases only randomly?*

All combinations
- Every combination of values of each parameter be used in a test case
- Satisfies n-wise coverage
- Number of test cases = \( m_1 \times m_2 \times m_3 \times \ldots \times m_n \)
  → *But testing all possible parameter combinations results in an astronomical number of test cases which is infeasible and inefficient*
Methods (2)

Cause-effect graphing/decision tables
- Requires dependencies between parameters and can get very complicated and difficult to implement

Tools for cause-effect graphing:
- ATD – Automated Test Designer (AtYourSide Consulting): http://www.atyoursideconsulting.com/
- CaseMaker (Diaz & Hilterscheid): http://www.casemaker.de/
- BenderRBT (Richard Bender): http://www.benderrbt.com/
Each choice

- Each value of each parameter to be included in at least one test case
- Defines 1-wise coverage
- Number of test cases = \( \max_{i=1}^{n} (m_i) \)

Base choice

- Start by identifying one base test case based on any criterion: simplest, smallest, most risky, most likely value, based on operational profile, etc.
- New test cases are created by varying the interesting values of one parameter at a time, keeping the values of the other parameters fixed on the base test case
- Satisfies 1-wise coverage
- Number of test cases = \( 1 + \sum_{i=1}^{n} (m_i - 1) \)

Example:

| Parameter \( P_1 \): 1, 2, 3 |
| Parameter \( P_2 \): 1, 2 |
| Parameter \( P_3 \): 1, 2 |

<table>
<thead>
<tr>
<th>Test case</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1 (base)</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Pairwise combinations – 2-wise coverage

- Empirically known that significant number of failures are caused by parameter interactions that occur in typical, yet realistic situations
- Assume that tests maximizing the interactions between parameters will find more faults
- Test at least all pairwise interactions of parameters, i.e. calculate a set of test cases which include all pairwise parameter combinations
- Assume that the risk of an interaction failure among three or more parameters (threewise, etc.) is balanced against the ability to complete testing within a reasonable budget
- Satisfies 2-wise coverage

Can be extended to n-wise testing

Example: Parameter $P_1$: $1, 2, 3$
Parameter $P_2$: $1, 2$
Parameter $P_3$: $1, 2$

<table>
<thead>
<tr>
<th>Test case</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

$^3$Parameters can be input, output, valid, invalid.
Example – Pairwise combinations – 2-wise coverage

Given a system under test with 4 parameters A, B, C, and D

- Each parameter has 3 possible values
  - Parameter A: a1, a2, a3
  - Parameter B: b1, b2, b3
  - Parameter C: c1, c2, c3
  - Parameter D: d1, d2, d3
- A valid test input data set is e.g. \{a2, b1, c2, d3\}

- Exhaustive testing would require $3^4 = 81$ test cases

Only 9 test cases are already sufficient to cover all pairwise interactions of parameters
Solution: System black-box testing

Electronic bookstore\(^1\)
- 5 parameters with different number of values

<table>
<thead>
<tr>
<th>TYPE OF CREDIT CARD</th>
<th>CREDIT CARD NUMBER</th>
<th>CREDIT CARD EXPIRATION DATE (YEARS FROM TODAY)</th>
<th>PRODUCT TYPE PURCHASED</th>
<th>QUANTITY PURCHASED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amex</td>
<td>Correct</td>
<td>50</td>
<td>Book</td>
<td>1</td>
</tr>
<tr>
<td>Discover</td>
<td>Incorrect Length</td>
<td>Invalid Year</td>
<td>Video</td>
<td>0</td>
</tr>
<tr>
<td>Visa</td>
<td>Invalid Digits</td>
<td>Today</td>
<td>Software</td>
<td>-1</td>
</tr>
<tr>
<td>MasterCard</td>
<td></td>
<td>Yesterday</td>
<td>Book, Software, Videos</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Invalid Character</td>
<td>Book, Software</td>
<td></td>
</tr>
</tbody>
</table>

- 1200 \((= 4 \times 3 \times 5 \times 5 \times 4)\) possible parameter combinations exist

Calculated set of test cases which include all pairwise parameter combinations: 25 (i.e. only \(2\%\) out of 1200!)

\(^1\)Ref.: Elfriede Dustin, Orthogonally Speaking, STQE Magazine, Sep/Oct 2001
All-pairs testing or pairwise testing is a combinatorial testing method that, for each pair of input parameters to a system (typically, a software algorithm) tests all possible discrete combinations of those parameters. Using carefully chosen test vectors, this can be done much faster than an exhaustive search of all combinations of all parameters, by "parallelizing" the tests of parameter pairs. The number of tests is typically $O(nm)$, where $n$ and $m$ are the number of possibilities for each of the two parameters with the most choices.
The reasoning behind all-pairs testing is this: the simplest bugs in a program are generally triggered by a single input parameter. The next simplest category of bugs consists of those dependent on interactions between pairs of parameters, which can be caught with all-pairs testing. Bugs involving interactions between three or more parameters are progressively less common, whilst at the same time being progressively more expensive to find by exhaustive testing, which has as its limit the exhaustive testing of all possible inputs.

Many testing methods regard all-pairs testing of a system or subsystem as a reasonable cost-benefit compromise between often computationally infeasible higher-order combinatorial testing methods, and less exhaustive methods which fail to exercise all possible pairs of parameters. Because no testing technique can find all bugs, all-pairs testing is typically used together with other quality assurance techniques such as unit testing, symbolic execution, fuzz testing, and code review.
Benefits (1)

Dramatically reduced overall number of test cases compared to exhaustive testing

Certainty that test coverage of all pairwise parameter combinations is achieved
- Detects all faults due to single parameters (1-wise coverage)
- Detects all faults due to interaction of two parameters (2-wise coverage)
- Detects many faults due to interaction of multiple parameters

Can be extended to n-wise testing depending on
- risk
- required test intensity
- available resources
- needed time and effort for test setup and test execution
Benefits (2)

Case studies⁴ give evidence that the approach compared to conventional approaches is

- more than **twice as efficient** (measured in terms of detected faults per testing effort) as traditional testing
- about **20% more effective** (measured in terms of detected faults per number of test cases) as traditional testing

Other case studies, e.g.:

- Several papers by Mats Grindal et al. on evaluation of combination strategies (see references)

Benefits (3)

Approach is applicable to generate
- low-level test cases for unit and integration testing
- high-level test cases for system and acceptance testing
- test cases for black-box testing
- test cases for white-box testing (control flow, data flow)
- test cases for model-based testing
  - use cases (e.g. activity diagrams): one basic/main flow – many alternate flows as well as exceptional (error) flows
  - final state machines: events/actions, transitions
- test cases for real-time and concurrency testing (feature interaction testing)
- test cases for robustness testing, redundancy testing
- to generate hardware / software combinations for configuration testing
  - installation testing
  - assembling components and build / deploy / test a system via configuration files
- to generate asset configurations for product line testing
seamlessly with conventional test methods
Example: Control flow graph (1)
Example: Control flow graph (2)

3 IFs in sequence with 2 branches each (1 empty)
- 1 test case for statement coverage
- 2 test cases for branch coverage
- 4 test cases for basis path coverage
- 4 (different!) test cases for pairwise coverage
- 8 test cases for (full) path coverage

Cyclomatic complexity (McCabe):
V(G) = 9 − 7 + 2
V(G) = 3 + 1
V(G) = 4

Pairwise combinations:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th></th>
<th>1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Example: Control flow graph (3)

3 IFs in sequence with 3 branches each (1 empty)
- 2 test cases for statement coverage
- 3 test cases for branch coverage
- 7 test cases for basis path coverage
- 9 (different!) test cases for pairwise coverage
- 27 test cases for (full) path coverage

Cyclomatic complexity (McCabe):
\[ V(G) = 15 - 10 + 2 \]
\[ V(G) = 2 + 2 + 2 + 1 \]
\[ V(G) = 6 + 1 = 7 \]

Pairwise combinations:
Example: Use case

Use case name

Brief description

Basic/Main flow
1. First step
2. Second step
3. Third step

Alternate flow 1
A1 Alternate flow 1
A2 Alternate flow 2
A3 Alternate flow 3
A4 Alternate flow 4

Exceptional (error) flows

End Use Case

Start Use Case
Example: Final state machine

Example: Configuration testing

5 Components/Features/Functionalities \{a, b, c, d, e\} to be combined in a system. Each of them can be included or excluded.

→ How to select subsets for testing?

Power set \( \wp(\{a, b, c, d, e\}) \) = set of all subsets of \{a, b, c, d, e\}

\[
\begin{align*}
\wp(\{a, b, c, d, e\}) &= \{\emptyset, \{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \\
&\quad \{a, b\}, \{a, c\}, \{a, d\}, \{a, e\}, \{b, c\}, \{b, d\}, \{b, e\}, \{c, d\}, \{c, e\}, \{d, e\}, \\
&\quad \{a, b, c\}, \{a, b, d\}, \{a, b, e\}, \{a, c, d\}, \{a, c, e\}, \{a, d, e\}, \{b, c, d\}, \{b, c, e\}, \{b, d, e\}, \{c, d, e\}, \\
&\quad \{a, b, c, d\}, \{a, b, c, e\}, \{a, b, d, e\}, \{a, c, d, e\}, \{b, c, d, e\}, \{a, b, c, d, e\}\}
\end{align*}
\]

\( | \wp(\{a, b, c, d, e\}) | = 2^5 = 1 + 5 + 10 + 10 + 5 + 1 \) (Pascal's triangle)

= 32 possible parameter combinations

Pairwise combinations:

- 1 … included
- 2 … excluded
- - … don’t care value
Variability model
- Common features
- Variable features
  - Variation points: optional features
  - Variation points: alternatives, n out of m
  - Generic interfaces to application

The variability model is an additional dimension for testing
- Additional complexity due to variation points and large number of products to test
- Combinatorial complexity of variability parameters requires intelligent testing methods to select combinations for effective and efficient testing

Example: Product line testing

Product Requirements

Commonalities (C)
Variabilities (V)

in space, i.e. product features

in time, i.e. platform evolution

Example: Microsoft Word 2002
Influences and dependencies

Number of test cases including all pairwise combinations are influenced and depending on

- number of parameters
- maximum parameter cardinality
- variability of parameter cardinality

Strategy:

- Adequate modeling of the parameters is the key
- Try to use a uniform cardinality of the parameters
- That is especially important if using orthogonal and covering arrays
Points to watch and limitations (1)

This kind of combinatorial testing by n-wise testing requires
- that the parameters are directly interdependent and semi-coupled
- that the parameter input is unordered – for example like most API signatures
- that the order to select the parameters is irrelevant / does not matter

This kind of combinatorial testing by n-wise testing is *not* an effective technique
- for parameters with no direct or indirect interaction at all
- for mathematical algorithms and calculations
- for ordered parameter input
- for parameters which require sequential operations
- if order of selection in the permutations matters (sequencing permutations)

See example by Scott Sehlhorst at
http://www.developerdotstar.com/mag/articles/test_smarter_not_harder.html
Points to watch and limitations (2)

Analysis of requirements to identify the relevant parameters and their domains must be done carefully using other test design methods as well. That is very important and it’s engineering work!

- Requires expert domain knowledge
- Parameter modeling can be interface-based (based on implementation) or functionality- / behavior-based (based on requirements)
- The output of any tool depends on the input

As with any other test design method you can never be sure that you missed an important test case

- Remember: testing is always some sort of sampling …
  - This method here is empirically based on the assumption that tests maximizing the interactions between parameters will find more faults
- But use it to reduce your risks!
Points to watch and limitations (3)

Tools do not produce the absolute minimum set of test cases which include all pairwise parameter combinations
- But they produce a very good approximation with feasible, adequate results

Especially for configuration testing it is possible that the number of test cases which include all pairwise parameter combinations is even too high

In general you have no control over the calculated test cases
- I.e. you cannot force which pairs you want to combine in one single test case
- There is some "magic" in the algorithms and in some tools …

Most tools do not provide additional diagnostic output, so
- you have to use elaborate manual checking or
- you have to build your own post-processing checking tool or
- just hope that the calculated test cases really include all pairwise parameter combinations …
Additional requirements

Constraints and dependencies between parameters: unwanted combinations

- This will occur very often in real life
- Example: Installation testing

```
<table>
<thead>
<tr>
<th>Installed version</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Never installed</td>
<td>Default name</td>
</tr>
<tr>
<td>Old version installed</td>
<td>Non default name</td>
</tr>
<tr>
<td>Previous version installed</td>
<td>Same name as previous installation</td>
</tr>
<tr>
<td></td>
<td>Other name as previous installation, old directory not deleted</td>
</tr>
</tbody>
</table>
```

Specific combinations, e.g. high-risky test cases, to be included in the test set: must-have combinations

- If not possible than add these test cases to the calculated set

Specific combinations, for example previously performed test cases, to be excluded from the test set
Example: Installation testing (1)

Installation testing of a real world program

22 parameters, 2-7 values each, including some constraints: OS, access rights, language, IE version, ODBC version, installation source, installation directory, etc.

Number of all possible combinations:
\[ 6 \times 2 \times 3 \times 7 \times 2 \times 4 \times 5 \times 2 \times 3 \times 5 \times 2 \times 5 \times 3 \times 4 \times 2 \times 5 \times 3 \times 7 \times 5 \times 4 \times 7 \times 4 \]
\[ = 4,267,468,800,000 \]

Number of all possible parameter pairs:
\[ 6 \times (2 + 3 + 7 + 2 + 4 + 5 + 2 + 3 + 5 + 2 + 5 + 3 + 4 + 2 + 5 + 3 + 7 + 5 + 4 + 7 + 4) \]
\[ + 2 \times (3 + 7 + 2 + 4 + 5 + 2 + 3 + 5 + 2 + 5 + 3 + 4 + 2 + 5 + 3 + 7 + 5 + 4 + 7 + 4) \]
\[ + 3 \times (7 + 2 + 4 + 5 + 2 + 3 + 5 + 2 + 5 + 3 + 4 + 2 + 5 + 3 + 7 + 5 + 4 + 7 + 4) \]
\[ + \text{other terms} + 7 \times 4 \]
\[ = 3,836 \]
Example: Installation testing (2)

Every test case which combines \( n \) parameters contains \( n \times (n - 1) / 2 \) parameter pairs

- For 22 parameters: 231 parameter pairs per test case

**Boundaries for the number of test cases including all pairwise combinations:**

\[
7 \times 7 = 49 \leq \text{number of test cases} \leq 3,836 \leq 4,267,468,800,000
\]

**Number of test cases including all pairwise combinations\(^5\):** 62

- Number of test cases including all threewise combinations is 491*

**Number of test cases reduced to** \( 1,45 \times 10^{-9}\% \) related to the number of all possible combinations (4,267,468,800,000)!!!

**Number of test cases reduced to** \( 1,616\% \) related to the number of all possible parameter pairs (3,836)

\(^5\)With the PICT tooling
Manual calculation is unfortunately laborious and error prone and cannot be carried out for real-world systems.

Approaches for automated calculation – deterministic (instant, iterative) or non-deterministic (iterative, random)

- Libraries on *orthogonal arrays* and *covering arrays*
  - Strategy: Map your parameters to a given pre-calculated array
  - Example: Taguchi orthogonal array selector
    (http://www.freequality.org/Default.aspx?page=60)

- Algorithms based on *orthogonal arrays* and *covering arrays*

- Different (heuristic) algorithms to calculate pairwise, threewise, etc. combinations
Orthogonal arrays – Covering arrays (1)

An orthogonal array is a two-dimensional matrix of values
- Fixed-level, unmixed, symmetrical: same-sized alphabet over all of the columns
- Mixed-level, asymmetrical: mixed alphabet sizes for different columns

Orthogonal arrays have interesting properties:
- Select any 2 columns and all pairs occur \textit{the same number of times}.
  I.e. an orthogonal array covers all 2-way interactions.
- Orthogonal arrays are balanced
- Orthogonal arrays are symmetric

Covering arrays are a generalization of orthogonal arrays

Covering arrays have interesting properties:
- Select any 2 columns and all pairs occur \textit{at least once}.
  I.e. a covering array covers all 2-way interactions as well.
Orthogonal arrays – Covering arrays (2)

Orthogonal arrays are likely to produce more test cases than covering arrays

Orthogonal arrays are likely to produce more test cases than the different all pairs techniques (but also dependent on the calculation algorithm!)

Fixed-level orthogonal arrays (n parameters with m values each): in most cases they do not exactly match to the current test problem because of the different cardinality of parameters. This results in more test cases leaving "don‘t care values".

→ Tool by STSC (Greg Daich)

Mixed-level orthogonal arrays: there are so many different options, thus finding an orthogonal array of matching size can be difficult
Example: Taguchi orthogonal array selector

### L₉ Orthogonal Array

<table>
<thead>
<tr>
<th>Experiment No.</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

*These combinations are not available – try running a smaller experiment*

Some numbers

10 Parameters with 3 values each
- Number of possible combinations: $3^{10} = 59,049$
- Number of test cases including all pairwise combinations: 15

13 Parameters with 3 values each
- Number of possible combinations: $3^{13} = 1,594,323$
- Number of test cases including all pairwise combinations: 15

10 Parameters with 5 values each
- Number of possible combinations: $5^{10} = 9,765,625$
- Number of test cases including all pairwise combinations: 42

10 Parameters with 10 values each
- Number of possible combinations: $10^{10} = 10,000,000,000$
- Number of test cases including all pairwise combinations: 120

40 Parameters with 3 values each
- Number of possible combinations: $3^{40} = 12,157,665,459,056,928,801$
- Number of test cases including all pairwise combinations: 21
Tools

AETG Web Service (Telcordia):  http://aetgweb.argreenhouse.com/
AllPairs (James Bach):  http://www.satisfice.com/
CTE XL (DaimlerChrysler):  http://www.systematic-testing.com/functional_testing/cte.php
Combinatorial Test Services CTS (IBM alphaWorks):  
IBM Intelligent Test Case Handler (IBM alphaWorks):  
Jenny (Bob Jenkins):  http://burtleburtle.net/bob/math/jenny.html
Pairwise Independent Combinatorial Testing PICT (Microsoft):  
available at: http://download.microsoft.com/download/f/5/5/f55484df-8494-48fa-8dbd-8c6f76cc014b/pict33.msi
Pro-Test (SigmaZone):  http://www.sigmazone.com/protest.htm
TConfig (Alan Williams):  http://www.site.uottawa.ca/~awilliam/
AETG Web Service (Telcordia)

AETG Web Home

The AETG™ Web Service generates test cases from a model of your requirements. It uses combinatorial design techniques to find a minimal test-case set that covers all pairwise interactions among input values.

By using the AETG Web Service, you will be able to create robust and efficient test cases faster and more comprehensively than ever before. You will achieve:

- High-quality test cases
- Rapidly generated test plans
- Reduced incidence of mission-critical software bugs

To illustrate the small size of a test set generated by the AETG system, consider a test problem with 13 input parameters and 3 values for each. AETG will generate just 15 test cases to cover all pairs of values, as compared to over 1.5 million required for exhaustive testing. The generated test cases guarantee that each parameter value will occur at least once with every other parameter value.

Our web interface makes it easy to use the AETG system. Because this interface can be used with almost any web browser, there is no need to maintain it on your local system.

Please sign up for a free trial account so you can evaluate the AETG Web Service on your testing challenges.

The AETG system is covered by U.S. Patent number 5,542,042.
### AllPairs (James Bach)

**Operating System** | **Printer** | **Duplex**
--- | --- | ---
Win98 | HP 4050 | Y
Win2K | HP 4100 | N
WinXP | |

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>case</td>
<td>Operating</td>
<td>Printer</td>
<td>Duplex</td>
<td>pairings</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Win98</td>
<td>HP 4050</td>
<td>Y</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Win98</td>
<td>HP 4100</td>
<td>N</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>Win2K</td>
<td>HP 4050</td>
<td>N</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>Win2K</td>
<td>HP 4100</td>
<td>Y</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>WinXP</td>
<td>HP 4050</td>
<td>Y</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>WinXP</td>
<td>HP 4100</td>
<td>N</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Status</th>
<th>var1</th>
<th>var2</th>
<th>value1</th>
<th>value2</th>
<th>appearances</th>
<th>cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating</td>
<td>Printer</td>
<td>Win98</td>
<td>HP 4050</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Operating</td>
<td>Printer</td>
<td>Win98</td>
<td>HP 4100</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>Operating</td>
<td>Printer</td>
<td>Win2K</td>
<td>HP 4050</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>Operating</td>
<td>Printer</td>
<td>Win2K</td>
<td>HP 4100</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>Operating</td>
<td>Printer</td>
<td>WinXP</td>
<td>HP 4050</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>17</td>
<td>Operating</td>
<td>Printer</td>
<td>WinXP</td>
<td>HP 4100</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>18</td>
<td>Operating</td>
<td>Duplex</td>
<td>Win98</td>
<td>Y</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>Operating</td>
<td>Duplex</td>
<td>Win98</td>
<td>N</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>Operating</td>
<td>Duplex</td>
<td>Win2K</td>
<td>Y</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>21</td>
<td>Operating</td>
<td>Duplex</td>
<td>Win2K</td>
<td>N</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>22</td>
<td>Operating</td>
<td>Duplex</td>
<td>WinXP</td>
<td>Y</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>23</td>
<td>Operating</td>
<td>Duplex</td>
<td>WinXP</td>
<td>N</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>24</td>
<td>Printer</td>
<td>Duplex</td>
<td>HP 4050</td>
<td>Y</td>
<td>2, 1, 5</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Printer</td>
<td>Duplex</td>
<td>HP 4050</td>
<td>N</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>26</td>
<td>Printer</td>
<td>Duplex</td>
<td>HP 4100</td>
<td>Y</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>27</td>
<td>Printer</td>
<td>Duplex</td>
<td>HP 4100</td>
<td>N</td>
<td>2, 2, 6</td>
<td></td>
</tr>
</tbody>
</table>
CTE XL (DaimlerChrysler)
Combinatorial Test Services CTS (IBM alphaWorks)

//Step 1. Set up all the types and attributes
CTSBooleanType boolType;
CTSEnumType strType2(2);
CTSEnumType strType3(3);
CTSIntegerType intType(3);
strType3.setValue(0,&CTSString("WINDOWS"));
strType3.setValue(1,&CTSString("LINUX"));
strType3.setValue(2,&CTSString("AIX"));
strType2.setValue(0,&CTSString("NETSCAPE"));
strType2.setValue(1,&CTSString("EXPLORE"));
CTSAttribute a1("param1", boolType);
CTSAttribute a2("param2", strType3);
CTSAttribute a3("param3", strType2);
CTSAttribute a4("param4", intType);

//Step 2. Create the test profile
TestCaseProfile profile;
profile.addAttribute(a1);
profile.addAttribute(a2);
profile.addAttribute(a3);
profile.addAttribute(a4);

//Step 3. Create an empty test suite
TestSuite suite(profile);

//Step 4. Use the build method to create the pairwise covering suite
float covMeasure = suite.build(2,1000, true);

//Step 5. Print the test suite to stdout
suite.instantiateDONTCAREs();
cout << suite << endl;

Output:
false WINDOWS NETSCAPE 0
false LINUX EXPLORE 1
true WINDOWS EXPLORE 2
true AIX NETSCAPE 1
true LINUX NETSCAPE 0
false AIX EXPLORE 2
true LINUX NETSCAPE 2
false AIX EXPLORE 0
false WINDOWS NETSCAPE 1
IBM Intelligent Test Case Handler (IBM alphaWorks)

A successor of the Combinatorial Test Services CTS

Eclipse plug-in
Jenny (Bob Jenkins)

Command Prompt

D:\temp\Example>jenny.exe -help

jenny:
Given a set of feature dimensions and withouts, produce tests
covering all n-tuples of features where all features come from
different dimensions. For example (=, <, >, <=, >=, !) is a
dimension with 6 features. The type of the left-hand argument is
another dimension. Dimensions are numbered 1..65535, in the order
they are listed. Features are implicitly named a..z, A..Z.

2 Dimensions are given by the number of features in that dimension.
-h prints out these instructions.
-n specifies the n in n-tuple. The default is 2 (meaning pairs).
-w gives withouts. -w1b4ab says that combining the second feature
of the first dimension with the first or second feature of the
fourth dimension is disallowed.
-ofoo.txt reads old jenny testcases from file foo.txt and extends them.

The output is a testcase per line, one feature per dimension per
testcase, followed by the list of all allowed tuples that jenny could
not reach.

Example: jenny -n3 3 2 2 -w2b3b 5 3 -w1c3b4ace5ac 8 2 2 3 2
This gives ten dimensions, asks that for any three dimensions all
combinations of features (one feature per dimension) be covered,
plus it asks that certain combinations of features
(like (1c, 3b, 4c, 5c)) not be covered.

D:\temp\Example>jenny.exe -n2 6 2 3 7 2 4 5 2 3 5 2 5 3 4 2 5 3 7 5 4 7 4 -w13a2
2c -w13a22d -w17a18d -w17a18e -w17a18f -w17c18d -w17c18e -s12920000 | wc
65 1430 5330

D:\temp\Example>jenny.exe -n2 6 2 3 7 2 4 5 2 3 5 2 5 3 4 2 5 3 7 5 4 7 4 -w13a2
2c -w13a22d -w17a18d -w17a18e -w17a18f -w17c18d -w17c18e -s12920000
1a 2b 3b 4b 5a 6d 7a 8a 9b 10c 11b 12c 13c 14a 15a 16a 17b 18c 19b 20d 21b 22b
1b 2a 3a 4a 5b 6c 7e 8b 9a 10b 11a 12d 13b 14b 15b 16d 17c 18g 19e 20c 21d 22a
1c 2a 3c 4f 5b 6b 7d 8a 9c 10e 11b 12e 13c 14c 15b 16b 17a 18b 19a 20b 21e 22c
1d 2b 3c 4e 5a 6a 7b 8b 9c 10a 11a 12b 13b 14d 15a 16c 17b 18d 19d 20a 21a 22d
Pairwise Independent Combinatorial Testing PICT (Microsoft)

Command Prompt

D:\temp\Example>pict
Pairwise Independent Combinatorial Testing

Usage: pict model [options]

Options:
/o:N    - Order of combinations (default: 2)
/d:C    - Separator for values (default: )
/a:C    - Separator for aliases (default: )
/n:C    - Negative value prefix (default: )
/e:file - File with seeding rows
/r[:N]  - Randomize generation, N - seed
/c      - Case-sensitive model evaluation
/s      - Show model statistics

D:\temp\Example>pict Input.txt /o:2 /r:53 | wc

Used seed: 53
63 - 1  1386    8254

D:\temp\Example>
Pairwise Independent Combinatorial Testing PICT (Microsoft): Example for a model file

```
# Machine 1
#
OS_1: Win2000, WinXP
SKU_1: Professional, Server|AdvServer, Datacenter, WinPowered
LANG_1: EN, DE

# Machine 2
#
OS_2: Win2000, WinXP
SKU_2: Professional, Server|AdvServer, Datacenter
LANG_2: EN, DE

# WinXP is always Professional in our case
#
if [OS_1] = "WinXP" then [SKU_1] = "Professional";
if [OS_2] = "WinXP" then [SKU_2] = "Professional";

# No German WinPowered
#
if [SKU_1] = "WinPowered" then [LANG_1] = "EN";

# Let's not test the same OS on both sides
#
[OS_1] <> [OS_2];
```
Additional special features of PICT (Microsoft)

Mixed-strength generation: define different orders of combinations for different subsets of parameters
- E.g. use threewise for subsets of parameters, other factors are "combined in" pairwise

Hierarchy of test parameters (sub-models)
- Model test domains with a clear hierarchy of test parameters
- Limit the combinatorial explosion of certain parameter interactions, e.g. for "environmental" parameters which are more difficult to be controlled and set

Negative testing
- Each negative test case should have only one invalid value (input masking)

Expected results: map expected results onto the domain model
- E.g. partitioning in three domains: Pass, OutOfSize, InvalidValue

Weights
- E.g. a certain parameter value is more important than others
PICT – Example (Jacek Czerwonka)

Type: Simple, Spanned, Striped, Mirror, RAID-5
VolumeSize: 10, 100, 1000, 10000, 40000
FileSystem: FAT, FAT32, NTFS
ClusterSize: 512, 1024, 2048, 4096, 8192, 16384
Compression: On, Off

Exhaustive (all possible parameter combinations): 900 test cases
Pairwise: 31 test cases  Threewise: 151 test cases

Special requirements:
- Cannot really apply FAT to a 40GB volume
- \{VolumeSize, FileSystem and ClusterSize\} have to be tested more thoroughly
- Someone could type -10 in VolumeSize and ClusterSize textboxes
- Some very common combinations must be tested
- Simple type and NTFS are most common user’s choice
TestCaseGenerator (Mark Bulmahn)

GUI wrapper for PICT
http://www.codeplex.com/TestCaseGenerator
A total of 3836 combinations out of 3836 were found in the constraints, previous test, and the test case matrix.
TConfig (Alan Williams)

Enter parameters

List of parameters:

Parameters:

Add...
Edit...
Remove...

Number of parameters: 22

OK Cancel

TConfig - Test Configuration Generator

Parameter mode

- Same # values
- Differing # values
- Named

Generation algorithm

- Recursive block method
- IPO method

Degree of interaction coverage: 2
Number of parameters: 22
Maximum number of values per parameter: 7
Number of configurations: 91

<table>
<thead>
<tr>
<th>1</th>
<th>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2 2 2 2 2 1 2 2 2 2 2 2 1 2 2 2 2 2 2 2</td>
</tr>
<tr>
<td>3</td>
<td>3 - 3 3 - 1 3 - 3 3 - 3 3 1 - 3 3 3 3 3 3</td>
</tr>
<tr>
<td>4</td>
<td>4 - - 4 - 1 4 - - 4 - - 4 - - 1 - - 4 - - 4 4 4 4 4 4</td>
</tr>
<tr>
<td>5</td>
<td>5 - - 5 - 1 5 - - 5 - - 5 - - 1 - - 5 - - 5 5 5 5 5 5</td>
</tr>
<tr>
<td>6</td>
<td>6 - - 6 - 1 - - - - - - - - - - - 6 - - 6</td>
</tr>
<tr>
<td>7</td>
<td>- - - 7 - 1 - - - - - - - - - - - 7 - - 7</td>
</tr>
<tr>
<td>8</td>
<td>3 - - 6 - 2 1 2 3 4 - - - - - - - - - - - 5 - - 5 - - 7</td>
</tr>
<tr>
<td>9</td>
<td>4 - - 7 1 2 2 - - 5 - - - - - - - - - - - 2 1 2 3 4 5 - - 7</td>
</tr>
<tr>
<td>10</td>
<td>5 - - 1 2 2 3 - - - - - - - - - - - - - - - - - 1 2 2 - - 4 - - 6 - - 1 2</td>
</tr>
<tr>
<td>11</td>
<td>6 - - 1 2 - - 2 4 - - - - - - - - - - - - 1 2 3 2 - - 5 - - 7 1 2 3</td>
</tr>
<tr>
<td>12</td>
<td>- - - 1 2 3 - - 2 5 - - - - - - - - - - - 1 2 3 - - 2 - - - - - 1 2 3 4</td>
</tr>
<tr>
<td>13</td>
<td>1 2 3 4 - - 2 - - 1 2 - - 4 - - 2 - - - - - 1 2 3 4 5</td>
</tr>
<tr>
<td>14</td>
<td>2 - - - 5 - - 2 - - 1 2 3 - - - - - - - - - - - - - - - - 1 2 3 4 - - 6</td>
</tr>
<tr>
<td>15</td>
<td>5 - 2 4 - 3 1 - - - 2 4 - - 3 1 3 - - - - - - - - - - - - - - - - 7 2 4 6</td>
</tr>
</tbody>
</table>

Number of configurations: 91

About TConfig

TConfig Version 2.1

Authors:

- Alan Williams
- Johnson H. Ho
- Alexandre Lareau

OK
## Tool characteristics

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>AETG</th>
<th>AllPairs</th>
<th>CTE-XL</th>
<th>CTS</th>
<th>Jenny</th>
<th>PICT</th>
<th>Pro-Test</th>
<th>TConfig</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cost and licensing</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free, open source, public domain, for non-commercial usage (no guarantee)</td>
<td>×</td>
<td>×</td>
<td>× (eval)</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Commercial</td>
<td>×</td>
<td></td>
<td>× (on request)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Calculation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pairwise</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Threewise</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Orthogonal array algorithm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>User interface</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Command line interface</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programming interface (C++-API)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graphical user interface</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Additional requirements</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constraints and dependencies</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Specific combinations, e.g. high-risky test cases, to be included in the test set</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specific combinations, e.g. previously performed test cases, to be excluded from the test set</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>×</td>
</tr>
</tbody>
</table>
Tool calculation results

```
<table>
<thead>
<tr>
<th>Example</th>
<th># Params</th>
<th>Cardinality</th>
<th>Without constraints</th>
<th>With constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installation</td>
<td>22</td>
<td>2-7</td>
<td>58</td>
<td>68</td>
</tr>
<tr>
<td>testing 1</td>
<td></td>
<td></td>
<td>88</td>
<td>88</td>
</tr>
<tr>
<td>Installation</td>
<td>15</td>
<td>2-5</td>
<td>25</td>
<td>29-36</td>
</tr>
<tr>
<td>testing 2</td>
<td></td>
<td></td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>14</td>
<td>15</td>
<td>11-15</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>28</td>
<td>24</td>
<td>22-27</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>58</td>
<td>48</td>
<td>53-80</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>18</td>
<td>19</td>
<td>16-20</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>49</td>
<td>45</td>
<td>44-49</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>96</td>
<td>79</td>
<td>62-80</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>149</td>
<td>177</td>
<td>152-161</td>
</tr>
<tr>
<td>10</td>
<td>26</td>
<td>919</td>
<td>1094</td>
<td>952</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>16</td>
<td>22</td>
<td>19-21</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>56</td>
<td>51</td>
<td>51-57</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>230</td>
<td>210</td>
<td>210-218</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>860</td>
<td>526</td>
<td>708</td>
</tr>
<tr>
<td>22</td>
<td>7</td>
<td>99,102</td>
<td>120</td>
<td>105</td>
</tr>
<tr>
<td>40</td>
<td>3</td>
<td>21</td>
<td>29</td>
<td>26</td>
</tr>
</tbody>
</table>
```

* Dependent on a parameter which seeds the random number generator
Cem Kaner, James Bach, Bret Pettichord: Lessons Learned in Software Testing


DevelopSense: http://www.developsense.com/testing/PairwiseTesting.html
References – Tooling

AllPairs (McDowell): http://www.mcdowella.demon.co.uk/allPairs.html
CaseMaker (Diaz & Hilterscheid): http://www.casemaker.de/
CMU StatLib: http://lib.stat.cmu.edu/designs/
Luminary Software (McGregor): http://www.luminary-software.com/freeResources.htm
Quick Design (QD) (Richard Bender): http://www.benderrbt.com/
SAS Orthogonal Arrays: http://support.sas.com/techsup/technote/ts723.html
Seilevel (OATS Technique):
http://upiia.uab.es/teach/a21291/apunts/provaOO/OATS.pdf
Smartware Technologies (SmartTest): http://www.smartwaretechnologies.com/
Software Technology Support Center (STSC) (Greg Daich):
Taguchi Orthogonal Array Selector:
Testcover.com: http://www.testcover.com/
Test Vector Generator (TVG): http://sourceforge.net/projects/tvg/
Summary

Combinatorial testing occurs very often
- All testing levels: model, unit, integration, system, and acceptance testing
- Black-box and white-box testing, model-based testing
- Real-time and concurrency testing (feature interaction testing)
- Robustness testing, redundancy testing
- Configuration testing, installation testing
- Product line testing

Approach
- Analyze requirements to identify relevant parameters (input, output, valid, invalid) of the system under test and their domains constraints and dependencies between parameters
- Calculate a set of test cases which include all pairwise / n-wise parameter combinations. That is a necessary precondition for adequate testing.
- Use it seamlessly with other test design methods

There exist many tools with different properties and characteristics to support the laborious calculation!