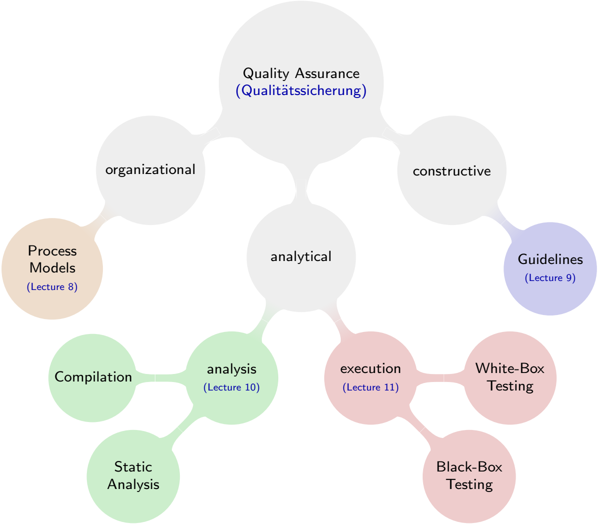
11. Product-Line Testing – Handout

Software Product Lines | Thomas Thüm, Sebastian Krieter, Timo Kehrer, Elias Kuiter | June 21, 2023

universität ulm — u^b UNIVERSITÄT BERN — OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

Recap: Quality Assurance [Ludewig and Lichter 2013]



Lectures on Quality Assurance

how to **avoid** variability bugs (esp. feature interactions) ...

- with processes (e.g., domain scoping) [Lecture 8]
- with guidelines [Lecture 9]

how to **find** variability bugs ...

- **statically** [Lecture 10]
- **dynamically** [Lecture 11]
  - challenges of product-line testing in Part a
  - black-box testing in Part b
  - white-box testing in Part c

11. Product-Line Testing

11a. Challenges of Product-Line Testing

Recap: Software Testing

Test-Case Design in Single-System Engineering

Testing All Configurations

Testing One Configuration

Sample-Based Testing

Summary

11b. Combinatorial Interaction Testing

11c. Solution-Space Sampling

Recap: Software Testing

Software Testing [Sommerville]

"Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use."

Validation Testing [Sommerville]

"Demonstrate to the developer and the customer that the software meets its requirements."

Defect Testing [Sommerville]

"Find inputs or input sequences where the behavior of the software is incorrect, undesirable, or does not conform to its specification."

Stages of Testing [Sommerville]

1. "**Development testing**, where the system is tested during development to discover bugs and defects"
2. "**Release testing**, where a separate testing team tests a complete version of the system before it is released to users"
3. "**User testing**, where users or potential users of a system test the system in their own environment"

Manual vs Automated Testing [Sommerville]

"In **manual testing**, a tester runs the program with some test data and compares the results to their expectations. [...] In **automated testing**, the tests are encoded in a program that is run each time the system under development is to be tested."

Recap: Test-Case Design

Systematic Test [Ludewig and Lichter 2013]

A systematic test is a test, in which

1. the **setup** is defined,
2. the **inputs** are chosen systematically,
3. the **results** are documented and evaluated by criteria being defined prior to the test.

Test Case [Ludewig and Lichter 2013]

In a test, a number of test cases are executed, whereas each test case consists **input values** for a single execution and **expected outputs**. An **exhaustive test** refers a test in which the test cases exercise all the possible inputs.

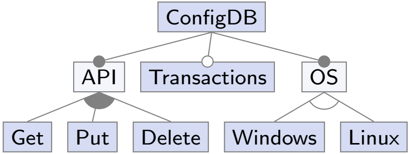
Goal [Ludewig and Lichter 2013]

Detect a large number of failures with a low number of test cases. A test case (execution) is **positive**, if it detects a failure, and **successful** if it detects an unknown failure.

An ideal test case is . . . [Ludewig and Lichter 2013]

- representative: represents a large number of feasible test cases
- failure sensitive: has a high probability to detect a failure
- non-redundant: does not check what other test cases already check

Testing All Configurations



$Transactions \rightarrow Put \vee Delete$

Recap: 26 Valid Configurations [Lecture 4]

$\{C, G, W\}$	$\{C, G, L\}$
$\{C, P, W\}$	$\{C, P, L\}$
$\{C, G, P, W\}$	$\{C, G, P, L\}$
$\{C, D, W\}$	$\{C, D, L\}$
$\{C, G, D, W\}$	$\{C, G, D, L\}$
$\{C, P, D, W\}$	$\{C, P, D, L\}$
$\{C, G, P, D, W\}$	$\{C, G, P, D, L\}$
$\{C, P, T, W\}$	$\{C, P, T, L\}$
$\{C, G, P, T, W\}$	$\{C, G, P, T, L\}$
$\{C, D, T, W\}$	$\{C, D, T, L\}$
$\{C, G, D, T, W\}$	$\{C, G, D, T, L\}$
$\{C, P, D, T, W\}$	$\{C, P, D, T, L\}$
$\{C, G, P, D, T, W\}$	$\{C, G, P, D, T, L\}$

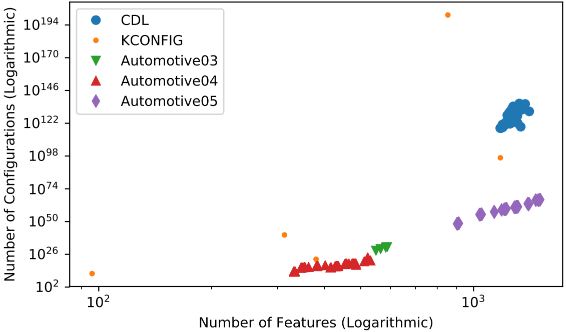
Discussion

- only feasible for **small** product lines (few valid configurations)
- redundant test effort
- large product lines: not feasible to generate and compile all configurations
  - (some) large product lines: even number of valid configurations is unknown

Recap: Feature Model of the Linux Kernel

Testing All Configurations

Recap: Industrial Configuration Spaces [Lecture 1]

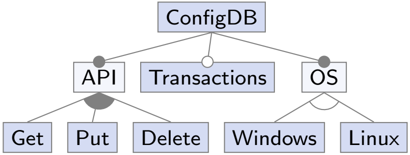


Why being complete on the configurations then?

Edsger W. Dijkstra (1972)

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." [The Humble Programmer]

Testing One Configuration



$Transactions \rightarrow Put \vee Delete$

Recap: 26 Valid Configurations

[Lecture 4]

$\{C, G, W\}$	$\{C, G, L\}$
$\{C, P, W\}$	$\{C, P, L\}$
$\{C, G, P, W\}$	$\{C, G, P, L\}$
$\{C, D, W\}$	$\{C, D, L\}$
$\{C, G, D, W\}$	$\{C, G, D, L\}$
$\{C, P, D, W\}$	$\{C, P, D, L\}$
$\{C, G, P, D, W\}$	$\{C, G, P, D, L\}$
$\{C, P, T, W\}$	$\{C, P, T, L\}$
$\{C, G, P, T, W\}$	$\{C, G, P, T, L\}$
$\{C, D, T, W\}$	$\{C, D, T, L\}$
$\{C, G, D, T, W\}$	$\{C, G, D, T, L\}$
$\{C, P, D, T, W\}$	$\{C, P, D, T, L\}$
$\{C, G, P, D, T, W\}$	$\{C, G, P, D, T, L\}$

Discussion

- applicable to large product lines
- strategy in practice: all-yes-config (configuration with many features selected)
- no redundant test effort (from configurations)
- often unfeasible to test all features wit a single configuration (e.g., `Windows` and `Linux`)

$\Rightarrow$ unnoticed feature interactions [Lecture 9]

What about interactions with missing features?

Sample-Based Testing

Intuition

- to analyze the product line, just analyze **some products**
- sample refers to a subset of all valid configurations
- common technique to test a product line
- sample configurations chosen by experts, randomly, or systematically

Advantages and Challenges

+ lower effort than testing all configurations
+ higher chance to detect defects than testing one configuration
– how many configurations to test? which configurations to test?

Challenges of Product-Line Testing – Summary

Lessons Learned

- recap on software testing and test-case design
- testing all configurations
- testing one configuration
- sample-based testing

Further Reading

- Varshosaz et al. 2018: overview on sampling literature
- Sampling Database: database on sampling algorithms and evaluations

Practice

Recap on feature interactions: What are examples of interactions that cannot be detected statically (cf. Lecture 10) and could be missed when testing a single configuration only?

11. Product-Line Testing

Recap: Black-Box Testing

Motivation

[Ludewig and Lichter 2013]

- source code not always available (e.g., outsourced components, obfuscated code)
- errors are not equally distributed

Black-Box Testing

[Ludewig and Lichter 2013]

- test-case design based on specification
- source code and its inner structure is ignored (assumed as a black-box)

Sample Configuration $\neq$ Test Case

- test case: concrete inputs and expected outputs for a program
- sample configuration: selection of features to derive the program
- both needed when testing product lines
- often confused in the literature
- test case derivation
  - out of scope here
  - global tests (i.e., identical for all configurations)
  - product-line implementation technique used to automatically derive configuration-specific tests

[Lecture 8]

- on next slides: idea of black-box testing applied to derive sample configuration

Pairwise Interaction Testing

Configurations with the Interaction Get $\wedge$ Put

$\{C, G, W\}$	$\{C, G, L\}$
$\{C, P, W\}$	$\{C, P, L\}$
$\{C, G, P, W\}$	$\{C, G, P, L\}$
$\{C, D, W\}$	$\{C, D, L\}$
$\{C, G, D, W\}$	$\{C, G, D, L\}$
$\{C, P, D, W\}$	$\{C, P, D, L\}$
$\{C, G, P, D, W\}$	$\{C, G, P, D, L\}$
$\{C, P, T, W\}$	$\{C, P, T, L\}$
$\{C, G, P, T, W\}$	$\{C, G, P, T, L\}$
$\{C, D, T, W\}$	$\{C, D, T, L\}$
$\{C, G, D, T, W\}$	$\{C, G, D, T, L\}$
$\{C, P, D, T, W\}$	$\{C, P, D, T, L\}$
$\{C, G, P, D, T, W\}$	$\{C, G, P, D, T, L\}$

Pairwise Interaction Testing

- create a sample $S \subseteq C$, in which every pairwise interaction is covered by at least one configuration
- test every configuration in S

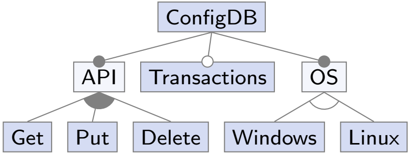
Discussion

- applicable to large product lines
- reduced redundant effort compared to testing all configurations
- full coverage guarantee (opposed to random configurations)
- still requires good test cases (program inputs)
- hard to compute small sample sets

Pairwise Combinations

- four combinations between A and B
  - both selected: $A \wedge B$
  - one selected: $\neg A \wedge B$ and $A \wedge \neg B$
  - none selected: $\neg A \wedge \neg B$

Pairwise Coverage



$Transactions \rightarrow Put \vee Delete$

Interactions to Cover

- exclude abstract features (e.g., *API*, *OS*)
- exclude features contained in every configuration (e.g., C)
- exclude invalid combinations (e.g., $W \wedge L$)

Pairwise Interactions

$G \wedge P$	$\neg G \wedge P$	$G \wedge \neg P$	$\neg G \wedge \neg P$
$G \wedge D$	$\neg G \wedge D$	$G \wedge \neg D$	$\neg G \wedge \neg D$
$G \wedge T$	$\neg G \wedge T$	$G \wedge \neg T$	$\neg G \wedge \neg T$
$G \wedge W$	$\neg G \wedge W$	$G \wedge \neg W$	$\neg G \wedge \neg W$
$G \wedge L$	$\neg G \wedge L$	$G \wedge \neg L$	$\neg G \wedge \neg L$
$P \wedge D$	$\neg P \wedge D$	$P \wedge \neg D$	$\neg P \wedge \neg D$
$P \wedge T$	$\neg P \wedge T$	$P \wedge \neg T$	$\neg P \wedge \neg T$
$P \wedge W$	$\neg P \wedge W$	$P \wedge \neg W$	$\neg P \wedge \neg W$
$P \wedge L$	$\neg P \wedge L$	$P \wedge \neg L$	$\neg P \wedge \neg L$
$D \wedge T$	$\neg D \wedge T$	$D \wedge \neg T$	$\neg D \wedge \neg T$
$D \wedge W$	$\neg D \wedge W$	$D \wedge \neg W$	$\neg D \wedge \neg W$
$D \wedge L$	$\neg D \wedge L$	$D \wedge \neg L$	$\neg D \wedge \neg L$
$T \wedge W$	$\neg T \wedge W$	$T \wedge \neg W$	$\neg T \wedge \neg W$
$T \wedge L$	$\neg T \wedge L$	$T \wedge \neg L$	$\neg T \wedge \neg L$
	$W \wedge \neg L$	$\neg W \wedge L$	

Pairwise Coverage with Six Configurations

$\{C, P, D, T, W\}$
$\{C, G, D, L\}$
$\{C, G, P, T, L\}$
$\{C, G, W\}$
$\{C, P, W\}$
$\{C, D, T, L\}$

T-Wise Interaction Testing

T-Wise Interaction Testing

- generalization of pairwise interaction testing
- t-wise coverage: every t-wise interaction is covered by at least one configuration in the sample
- $t = 1$: every feature is selected and also deselected
- $t = 2$: pairwise interaction coverage
- $t = 3$: every valid combination of three features covered

$t = 3$ Interactions

for the features G, P, and D:

$G \wedge P \wedge D$	$\neg G \wedge P \wedge D$
$G \wedge P \wedge \neg D$	$\neg G \wedge P \wedge \neg D$
$G \wedge \neg P \wedge D$	$\neg G \wedge \neg P \wedge D$
$G \wedge \neg P \wedge \neg D$	$\neg G \wedge \neg P \wedge \neg D$

Algorithms for Combinatorial Interaction Testing

A Greedy Algorithm

idea: select configuration that cover most missing interactions in each step

1. randomly choose first configuration
2. find next optimal configuration
3. repeat step 2 until all interactions are covered

Challenges and Optimizations

- non-deterministic: different sample for each run (cf. Step 1)
  - starting with all-yes-config? $\Rightarrow$ covers more code
- iterating all valid configurations does not scale (cf. Step 2)
- greedy strategy: optimal configuration in each step does not guarantee optimal sample

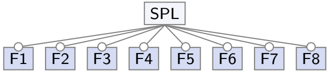
ICPL

[Johansen et al. 2012]

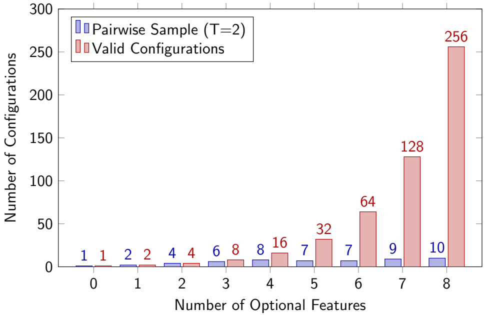
- widespread greedy algorithm
- iterates over all interactions
  - identifies core and dead features early
  - identifies invalid and already covered interactions
  - utilizes parallelization
- incrementally increases t up to desired value
- performance shown on next slides

Efficiency of Combinatorial Interaction Testing [Johansen et al. 2012]

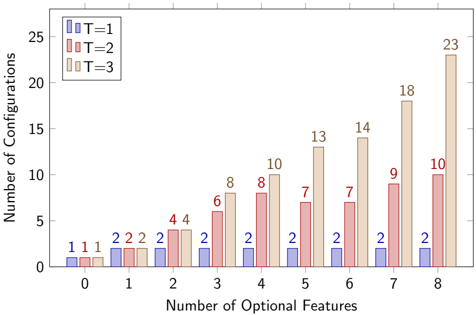
Assumption: All Features are Optional



Number of Configurations in Pairwise Sample

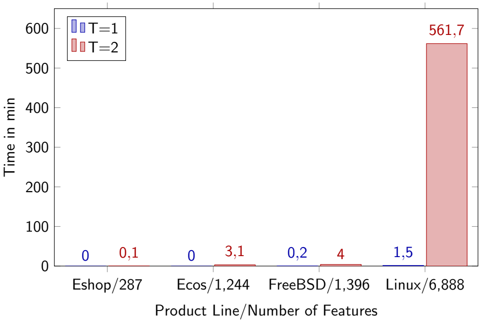


Number of Configurations in T-Wise Sample



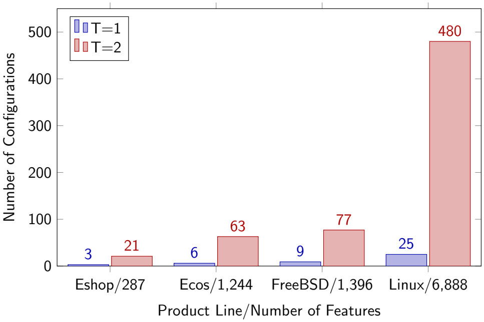
Efficiency of Combinatorial Interaction Testing [Johansen et al. 2012]

Time in Minutes to Compute Sample



- about 9h for Linux
- 480 configuration in pairwise sample

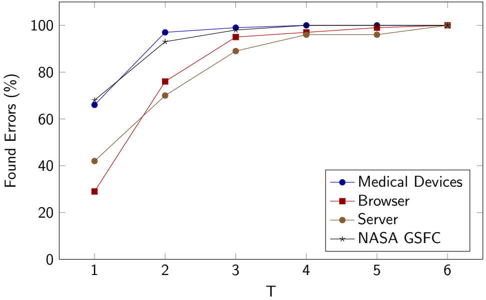
Number of Configurations in Sample



- Linux kernel v2.6.28.6 (February 2009)
- 6,888 features
- 187,193 clauses in conjunctive normal form

Effectiveness of Combinatorial Interaction Testing

Effectiveness of Interaction Testing [Kuhn et al. 2004]



Trade-Off

large t: high coverage (more effective)

small t: low testing effort (more efficient)

Combinatorial Interaction Testing – Summary

Lessons Learned

- recap on black-box testing
- combinatorial interaction testing: pairwise testing, t-wise testing
- efficiency: number of configurations, time to compute sample
- effectiveness: percentage of found defects

Further Reading

- Johansen et al. 2012 – popular t-wise sampling algorithm ICPL
- Krieter et al. 2020 – alternative t-wise sampling algorithm YASA

Practice

Why is it hard to find a good trade-off between efficiency and effectiveness?

11. Product-Line Testing

Recap: Coverage in White-Box Testing [Ludewig and Lichter 2013]

White-Box Testing

- inner structure of test object is used
- idea: coverage of structural elements
  - code translated into control flow graph
  - specific test case (concrete inputs)
    derived from logical test case (conditions)
    derived from path in control flow graph

Coverage Criteria

1. **statement coverage** :
   all statements are executed for at least one test case
2. **branching coverage** : statement coverage and all branches of branching statement are executed
3. **term coverage** :
   branching coverage and all terms used in a branching statement (n) are combined exhaustively (2^n) (simplified)

Can You Spot Problems in the Elevator Product Line?

[Varshosaz et al. 2018]

```
1  class ControlUnit {
2    Elevator elevator;
3    ElevatorState state, nextState;
4    //#if FIFO
5    Req req = new Req();
6    //#elif DirectedCall
7    Req req = new UndReq(this);
8    Req dreq = new DirReq(null);
9    //#else
0    Req req = new Req(this);
1    //#endif
2    void run() {
3      while (true) {
4        calculateNextState();
5        setDirection(nextState);
16         //#if DirectedCall
17         sortQueue();
18         //#endif
19     }
20   }
21   void calculateNextState() {
22     //#if Sabbath
23     if (sabbathNextState()) return;
24     //#endif
25     //#if Service
26     if (serviceNextState()) return;
27     //#endif
28     //#if FIFO
29     callButtonsNextState(dreq);
30     //#endif
31   }
32   //#if Service
33   boolean serviceNextState() {...}
34   //#endif
35   //#if Sabbath
36   boolean sabbathNextState() {...}
37   //#endif
38   //#if DirectedCall
39   void callButtonsNextState(Req d)
         {...}
40   void sortQueue() {...}
41   //#endif
42 }
```

- Line 29: compiler error (i.e., field `dreq` undefined) when $\mathit{FIFO} \land \neg \mathit{DirectedCall}$
- Line 8: null pointer exception when $\mathit{DirectedCall} \land \neg \mathit{FIFO}$
- both problems detectable with pairwise coverage, but presence conditions are more complicated in practice
- also: pairwise coverage often too much effort for large configuration spaces / continuous integration

Coverage of Ifdef Blocks

[Tartler et al. 2012]

- every block selected for at least one configuration in the sample (cf. statement coverage)

Presence-Condition Coverage

Presence-Condition Coverage [Krieter et al. 2022]

- application of t-wise interaction testing to presence conditions
- recap presence condition: formula specifying exactly those configurations under which a block is present
- t-wise presence condition coverage: every t-wise interaction of presence conditions is covered by at least one configuration in the sample
- $t = 1$: every block is selected and also deselected (i.e., more than Tartler's coverage of ifdef blocks)
- $t = 2$: every combination of two blocks covered
- $t = 3$: every combination of three blocks covered

T=3 Presence-Condition Interactions

for the blocks a, b, and c with presence conditions A, B, and C:

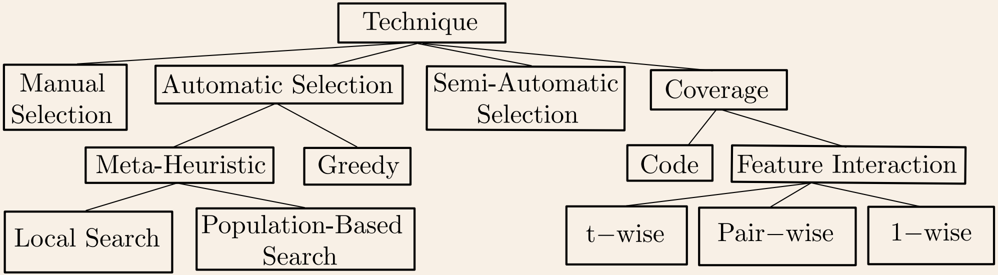
$A \wedge B \wedge C$	$\neg A \wedge B \wedge C$
$A \wedge B \wedge \neg C$	$\neg A \wedge B \wedge \neg C$
$A \wedge \neg B \wedge C$	$\neg A \wedge \neg B \wedge C$
$A \wedge \neg B \wedge \neg C$	$\neg A \wedge \neg B \wedge \neg C$

Presence-Condition Coverage [Krieter et al. 2022]

- coverage of solution space (not problem space)
- aka. solution-space sampling
- for same t: often fewer configurations and similar effectiveness than feature interaction coverage
- also feasible by translating presence conditions into feature model [Hentze et al. 2022]

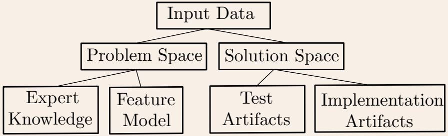
Overview on Coverage Criteria

Techniques & Coverage Criteria [Varshosaz et al. 2018]



Input for Sampling Algorithms

Input Data [Varshosaz et al. 2018]



Further Domain Knowledge [Varshosaz et al. 2018]

- in addition to feature model
- e.g., configurations chosen by experts
- e.g., specialized feature model for sampling

Part 2: Combinatorial Interaction Testing

- (Problem-Space Sampling)
- **feature model** used to consider only valid configurations

Part 3: Solution-Space Sampling

- mapping from features to **implementation artifacts**
- **feature model** used to consider only valid configurations

Combinatorial Reduction of Tests [Kim et al. 2011]

- which configurations matter for each test?
- analyze **unit tests** and **impl. artifacts**
- **feature model** used to consider only valid configurations

Solution-Space Sampling – Summary

Lessons Learned

- recap on white-box testing and coverage criteria
- coverage of ifdef blocks
- t-wise presence condition coverage
- overview on techniques, coverage criteria, input data for sampling

Further Reading

- Tartler et al. 2012: covering every ifdef block (but not their absence)
- Krieter et al. 2022: solution-space sampling as discussed in this part
- Hentze et al. 2022: translation of presence conditions into feature model + reuse of problem-space sampling

Practice

Does the order of configurations matter during testing? [Al-Hajjaji et al. 2019]

FAQ – 11. Product-Line Testing

Lecture 11a

- What is the goal of quality assurance for software product lines?
- How can product lines be tested?
- Why is testing product lines challenging?
- What are (dis-)advantages of testing all configurations?
- What are (dis-)advantages of testing only one configuration?
- What is sample-based testing?
- What is a sample?
- How can a sample be computed?

Lecture 11b

- How is black-box testing used for testing product lines?
- What is the difference between a test configuration and a test case?
- What are (dis-)advantages of combinatorial interaction testing?
- What is pairwise interaction testing?
- What is t-wise interaction testing?
- When does a sample achieve 100% pairwise coverage?
- How can a t-wise sample be computed?

Lecture 11c

- How can white-box testing be used for testing product lines?
- What are potential problems with t-wise interaction testing?
- What is presence condition coverage?
- What are different techniques for t-wise sampling?
- Which additional inputs can be used for sampling algorithms?
- How efficient and how effective are sampling algorithms?