12. Evolution and Maintenance – Handout

Software Product Lines | Thomas Thüm, Elias Kuiter, Timo Kehrer | June 28, 2023

Technische Universität Braunschweig

UNIVERSITÄT BERN

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

12. Evolution and Maintenance

Recap and Motivation

Jimmy Koppel, 2019 [corecursive.com]

"Software maintenance is important because the world runs on software, and changing the world means changing the software."

Recap and Motivation

Lehman's Laws of Software Evolution (excerpt) [Lehman et al. 1997]

- Continuing Change: systems must be continually adapted to stay satisfactory
- Increasing Complexity: complexity increases during evolution unless work is done to maintain or reduce it
- Continuing Growth: functionality must be continually increased to maintain user satisfaction
- Declining Quality: quality will decline unless rigorously maintained and adapted to operational environment changes

Essence of the Laws

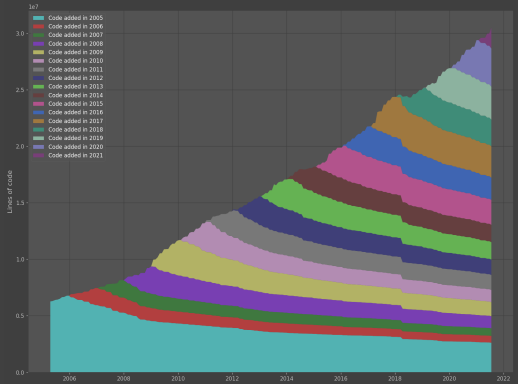
- software that is used will be modified
- when modified, its complexity will increase (unless one does actively work against it)

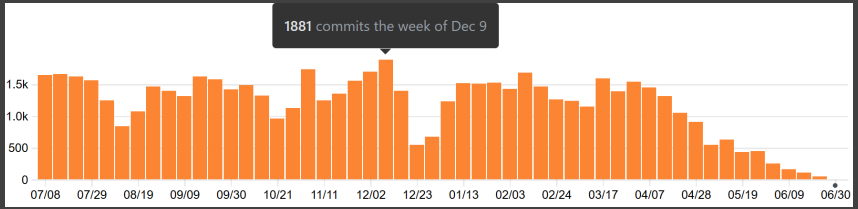
Consequences for Product Lines

- number of features and size of implementation increases over time
- discussed challenges increase over time
  - more software clones
  - harder to trace features
  - automated generation more urgent
  - increasing combinatorial explosion
  - more feature interactions

Evolution of the Linux Kernel

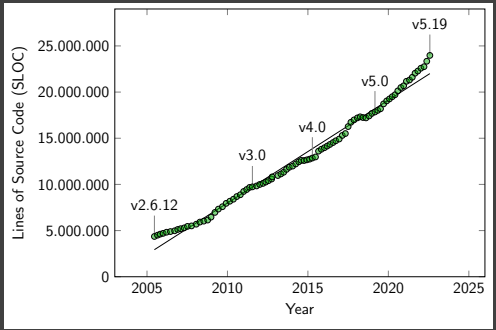
- about 60,000 commits per year
- in peak weeks: new commit every 5 minutes
- in average weeks: every 9 minutes





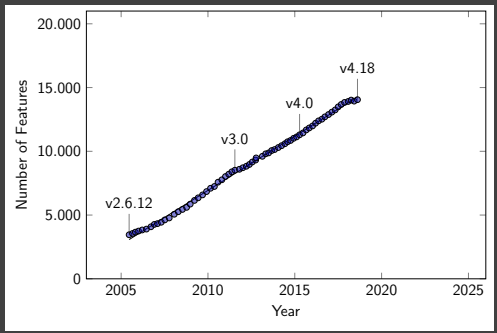
Evolution of the Linux Kernel

Size of the Code Base



- from 4 to 24 millions in 17 years
- about one million LOC added every year
- about 3,000 LOC per day

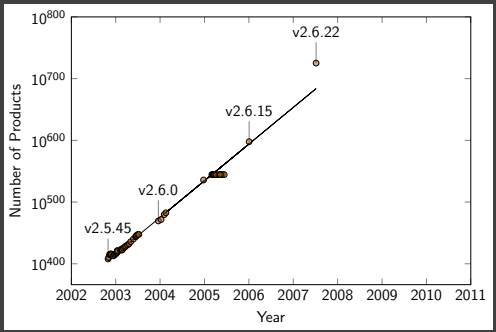
Number of Features



- about 800 new features every year
- about 15 new features every week
- in 2018 four times more features than in 2005

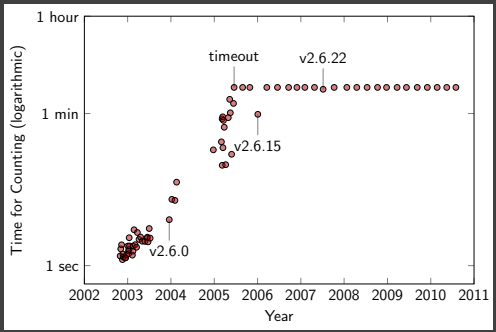
Evolution of the Linux Kernel

Number of Products



- number of products grows by factor 100.000 each month
- the current kernel is likely to have more than 10^{1500} products

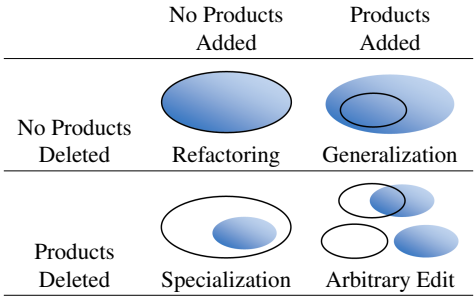
Time to Count Products



- most kernel versions before 2006 can be computed within 1 minute
- most kernel versions after 2006 cannot be computed within 1 hour

Evolution of Feature Models [Thüm et al. 2009]

Classification of Changes



Automated Classification

classification can be reduced to SAT problems

Advantages for Quality Assurance

assumption: only feature model has changed

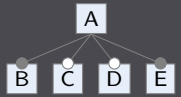
- refactoring: no retest needed
- specialization: cannot produce new faults
- generalization: cannot fix known faults
- arbitrary edit: retest needed

Advantages for Modelers

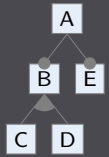
did the configuration space change as intended?

Evolution of Feature Models

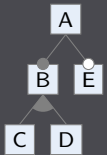
Version 1



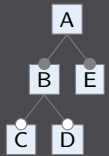
Version 2



Version 3



Version 4



Refactoring

1 to 2, 2 to 1

Generalization

1/2 to 3/4

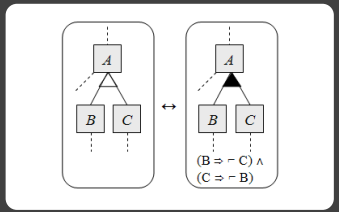
Specialization

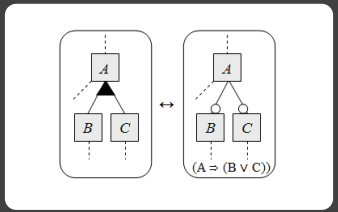
3/4 to 1/2

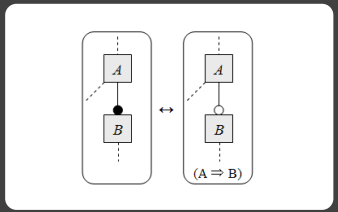
Arbitrary Edit

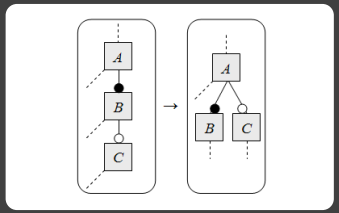
3 to 4, 4 to 3

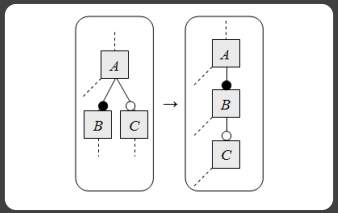
Evolution of Feature Models – Refactorings [Alves et al. 2006]





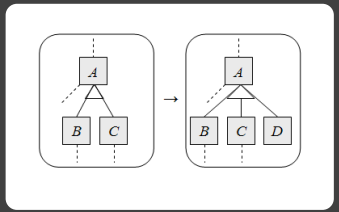


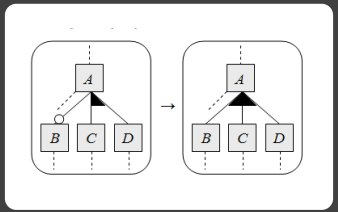


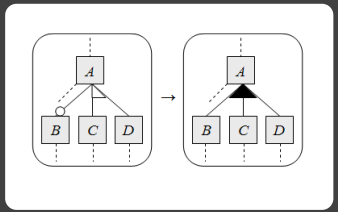


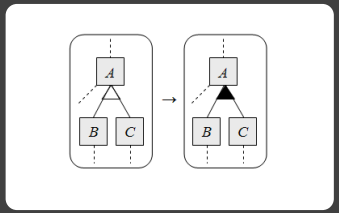
. . .

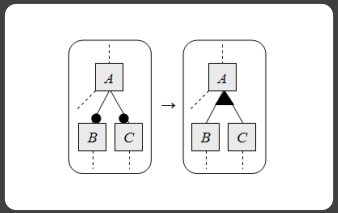
Evolution of Feature Models – Generalizations [Alves et al. 2006]

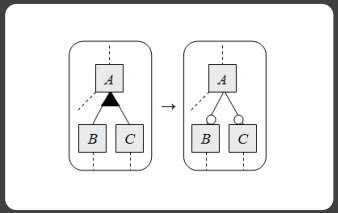




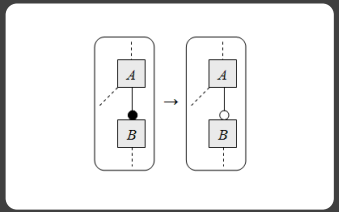


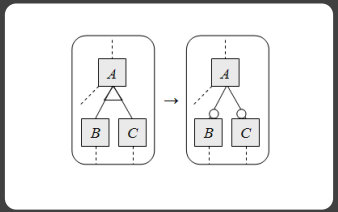


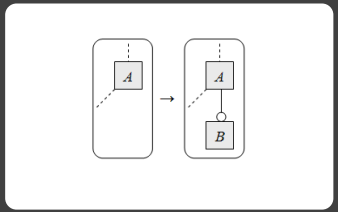


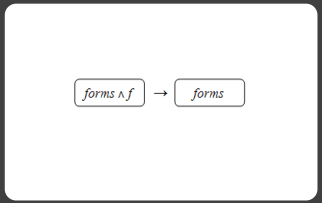


Evolution of Feature Models – Generalizations [Alves et al. 2006]



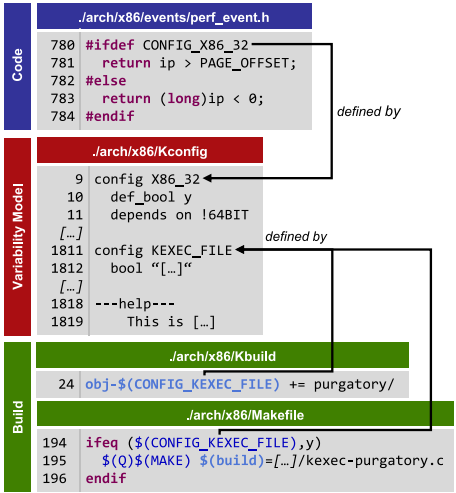






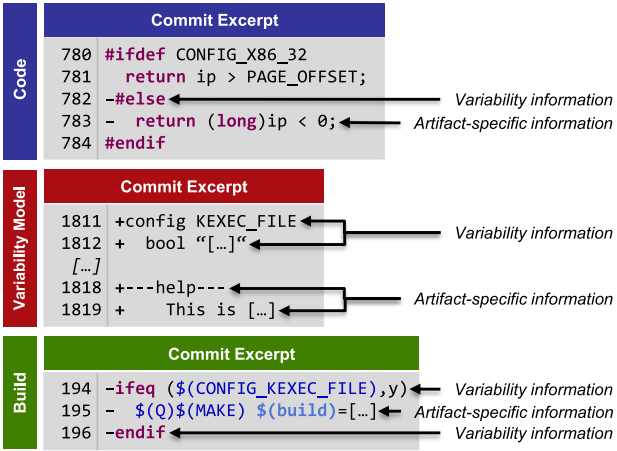
. . .

Co-Evolution of Product Lines [Kröher et al. 2023]



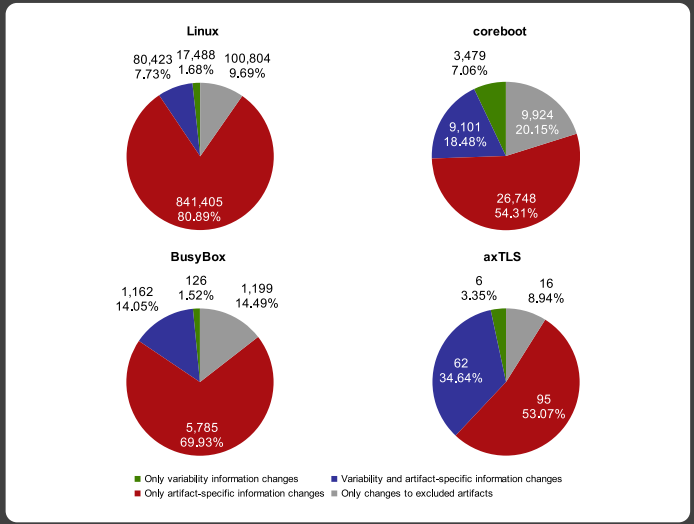
- not only feature models evolve
- but also source code
- and mapping from features to source code
- aka. **co-evolution**
- for conditional compilation:
  1. feature model
  2. build scripts with features
  3. source code with preprocessor statements
- how frequent are those changes?

Co-Evolution of Product Lines [Kröher et al. 2023]



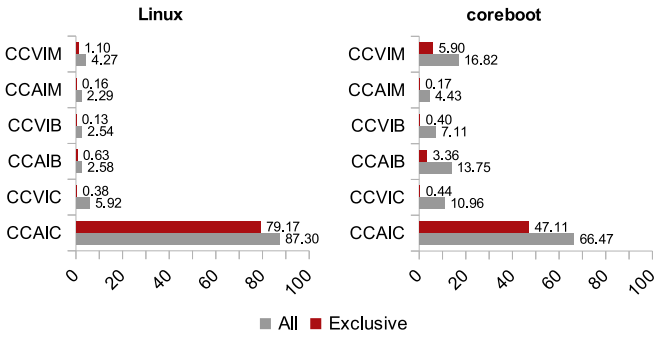
- variability information evolves
- artifact-specific information evolves
- how frequent are those changes?

Co-Evolution of Product Lines [Kröher et al. 2023]



- most changes only affect artifact-specific information
- fewest changes only affect variability information

Co-Evolution of Product Lines [Kröher et al. 2023]



- 1 % of commits in Linux **only** change feature model
- 4 % of commits in Linux change feature model

- <1 % of commits in Linux **only** change build scripts
- 3 % of commits in Linux change build scripts

- 79 % of commits in Linux **only** change source code
- 87 % of commits in Linux change source code

- different percentages for other systems

Product-Line Evolution – Summary

Lessons Learned

- changes are inevitable and occur frequently
- product lines typically grow over time
- different kinds of changes to feature models (e.g., refactorings)
- co-evolution of feature model, feature mapping, artifacts

Further Reading

- Thüm et al. 2009
  — SAT-based classification of feature-model changes
- Alves et al. 2006
  — patterns for feature-model refactorings
- Kröher et al. 2023
  — product-line evolution

Practice

After which changes do we need to analyze and test the product line again?
Do we always need to analyze and test the complete product line again?

12. Evolution and Maintenance

12a. Product-Line Evolution

12b. Product-Line Maintenance

- Recap on Software Maintenance
- Kinds of Maintenance
- Recap on Feature Model Transformations
- Reengineering Tasks
- Summary

12c. Course Summary

Recap on Software Maintenance [Ludewig and Lichter 2013]

Motivation

- for software: no compensation of deterioration, repair, spare parts
- corrections (especially shortly after first delivery)
- modification and reconstruction

Operation and Maintenance Phase [IEEE Std 610.12]

"The period of time in the software life cycle during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements."

Maintenance [IEEE Std 610.12]

"The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment."

Recap on Software Maintenance [Ludewig and Lichter 2013]

Evolution

- new or removed functionality
- larger changes
- often foreseen changes
- results in upgrades, service packs, or cumulative updates

Maintenance

- mostly corrections
- smaller changes
- often unforeseen changes
- results in patches and hot fixes

Minor Release

new minor version: 2.3.1 $\Rightarrow$ 2.4.0

Patch Release

new patch version: 2.3.1 $\Rightarrow$ 2.3.2

Major Release

new major version: 2.3.1 $\Rightarrow$ 3.0.0

not easy to distinguish – there is a continuum between evolution and maintenance

Kinds of Maintenance [Ludewig and Lichter 2013]

Adaptive Maintenance [Ludewig and Lichter 2013]

"Software maintenance performed to make a computer program usable in a changed environment."

desktop application for a new version of an operating system (e.g., from Windows 10 to 11)

Corrective Maintenance [Ludewig and Lichter 2013]

"Maintenance performed to correct faults in software."

feature interaction of Lenovo products fixed with BIOS update

Perfective Maintenance [Ludewig and Lichter 2013]

"Software maintenance performed to improve the performance, maintainability, or other attributes of a computer program."

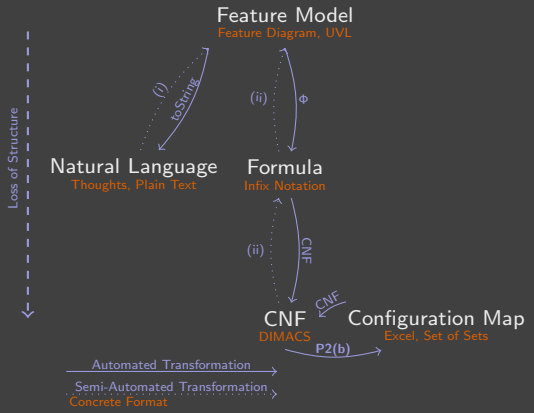
improve start-up time of the Linux kernel

Preventive Maintenance [Ludewig and Lichter 2013]

"Maintenance performed for the purpose of preventing problems before they occur."

code audit of car software before next leap second

Recap on Feature Model Transformations



Problems

P1 How to express feature models **textually**?

P2 How to
- (a) validate configurations and
- (b) get all valid configurations

automatically?

P3 (How to reverse engineer feature models?)

Solutions

P1 Universal Variability Language ⇒ **Syntax**

P2 Propositional Formulas ⇒ **Semantics**
- (a) evaluate feature-model formula
- (b) Lecture 4c

P3 (i) e.g., Bakar et al. 2015
(ii) e.g., Czarnecki and Wasowski 2007

Reengineering Tasks [Ludewig and Lichter 2013]

Reverse Engineering [Chikofsky und Cross]

"Reverse engineering is the process of analyzing a system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction."

create feature model from existing configurations

Forward Engineering [Chikofsky und Cross]

"Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system."

domain engineering [Lecture 8a]

Refactoring [Chikofsky und Cross]

"Refactoring is a transformation from one form of representation to another at the same relative level of abstraction. The new representation is meant to preserve the semantics and external behavior of the original."

feature-model refactorings [Lecture 12a]

Reengineering [Chikofsky und Cross]

"Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."

combination of reverse engineering, refactoring, and forward engineering

Product-Line Maintenance – Summary

Lessons Learned

- what is maintenance?
- what are examples for product-line maintenance?
- what is reengineering?
- what are examples of product-line reengineering?

Further Reading

- Ludewig and Lichter 2013
  — maintenance and reengineering

Practice

How do **product-line adoption strategies** (proactive, reactive, extractive) correlate with **reengineering tasks** (reengineering, refactoring, forward/reverse engineering)?

12. Evolution and Maintenance

12a. Product-Line Evolution

12b. Product-Line Maintenance

12c. Course Summary

Software Product Lines

Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. **Evolution and Maintenance**

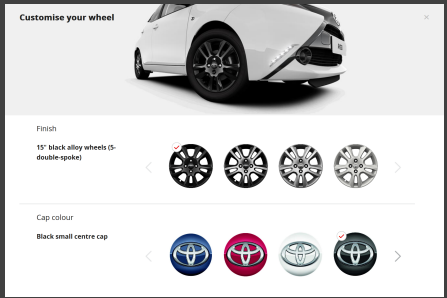
The Vision of Product Lines

Mass Customization

[Apel et al. 2013, p. 4]

- = mass production + customization
- customized, individual goods at costs similar to mass production

Car Configuration



Car Production

Other Domains

bikes, computers, electronics, tools, medicine, clothing, food, financial services, . . . , software?

The Vision of Product Lines

Software Product Line [Northrop et al. 2012, p. 5]

"A **software product line** is

- a set of software-intensive systems

  aka. products or variants

- that share a common, managed set of features

  common set, but not all products have all features in common

- satisfying the specific needs of a particular market segment or mission

  aka. domain

- and that are developed from a common set of core assets in a prescribed way."

  aka. planned, structured reuse

[Software Engineering Institute, Carnegie Mellon University]

The Vision of Product Lines

Product-Line Engineering [Pohl et al. 2005, p. 14]

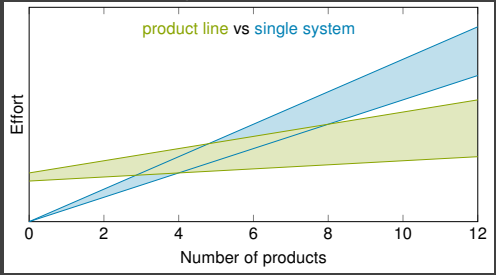
"**Software product-line engineering** is a paradigm to develop software applications (software-intensive systems and software products) using software platforms and mass customization."

Promises of Product Lines [Apel et al. 2013, pp. 9–10]

- tailor-made
- reduced costs
- improved quality
- reduced time-to-market

Idea of Product-Line Engineering

Reduce effort per product by means of an up-front investment for the product line:



Single-System Engineering

classical software development that is not considered as product-line engineering

Modeling Features

Natural Language

"A **configurable database** has an API that allows for at least one of the request types **Get**, **Put**, or **Delete**. Optionally, the database can support **transactions**, provided that the API allows for Put or Delete requests. Also, the database targets a supported operating system, which is either **Windows** or **Linux**."

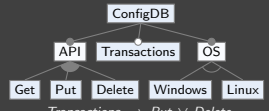
Configuration Map

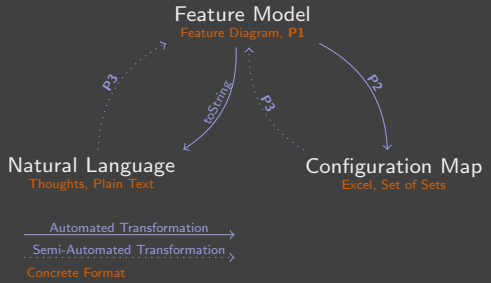
$\{C, G, W\}$ $\quad$ $\{C, G, L\}$

$\vdots$ $\quad$ $\vdots$

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

Feature Diagram (Graphical Feature Model)





Problems

P1 How to express feature models **textually**?

P2 How to (a) validate configurations and (b) get all valid configurations **automatically**?

P3 (How to reverse engineer feature models?)

Modeling Features

Problem Space [Apel et al. 2013, p. 21]

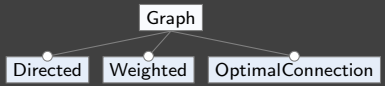
"The **problem space** takes the perspective of stakeholders and their problems, requirements, and views of the entire domain and individual products. Features are, in fact, domain abstractions that characterize the problem space."

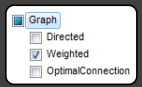
[Lecture 4]

Solution Space [Apel et al. 2013, p. 21]

"The **solution space** represents the developer's and vendor's perspectives. It is characterized by the terminology of the developer, which includes names of functions, classes, and program parameters. The solution space covers the design, implementation, and validation and verification of features and their combinations in suitable ways to facilitate systematic reuse."

[Lecture 2, Lecture 3, Lecture 5, Lecture 6, Lecture 7]






```
class Edge {
  Node first, second;
//#ifdef Weighted
  int weight;
//#endif
  Edge(Node first, Node second) {...}
  boolean equals(Edge e) {
    return (first == e.first
        && second == e.first
//#ifndef Directed
        || first == e.second
        && second == e.first
//#endif
    ) && weight == e.weight;
  }
  void testEquality() {
    Node a = new Node();
    Node b = new Node();
    Edge e = new Edge(a, b);
    Assert.assertTrue(e.equals(e));
} }
```

```
class Edge {
  Node first, second;

  int weight;

  Edge(Node first, Node second) {...}
  boolean equals(Edge e) {
    return (first == e.first
        && second == e.first

        || first == e.second
        && second == e.first
    ) && weight == e.weight;

  }
  void testEquality() {
    Node a = new Node();
    Node b = new Node();
    Edge e = new Edge(a, b);
    Assert.assertTrue(e.equals(e));
} }
```

Product-Line Requirements

Domain Engineering

Domain Analysis

Domain Design

Domain Implementation

Domain Testing

Product Requirements

Application Engineering

Application Analysis

Application Design

Application Implementation

Application Testing

Product

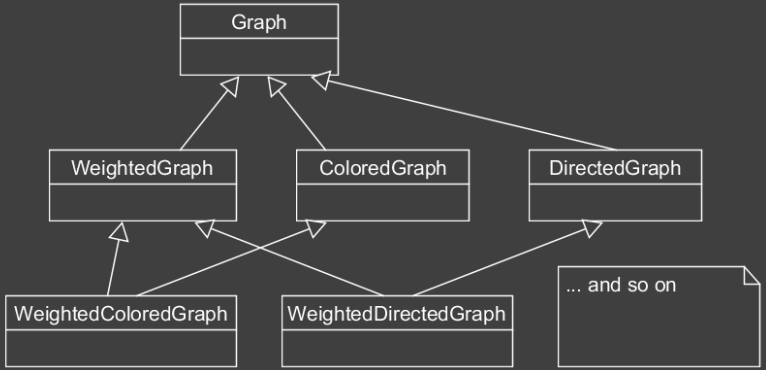
Implementing Features

	Compile-Time Variability	Features	Product Generation	Feature Traceability
Runtime Variability	no (very limited for immutable global variables)	only fine grained	yes (except for preference dialogs)	no
Clone-and-Own	yes (only for implemented products)	no	no (limited generation for clone-and-own with build systems)	no
Build Systems (for Conditional Compilation)	yes	only coarse grained	yes	with tool support
Preprocessors (for Conditional Compilation)	yes	only fine grained	yes	with tool support
Components/Services	yes	only coarse grained	no (except pure exchange)	only coarse grained
Frameworks with Plug-Ins	yes	only coarse grained	yes	only coarse grained
Feature Modules/Aspects	yes	yes	yes	yes

Further Criteria

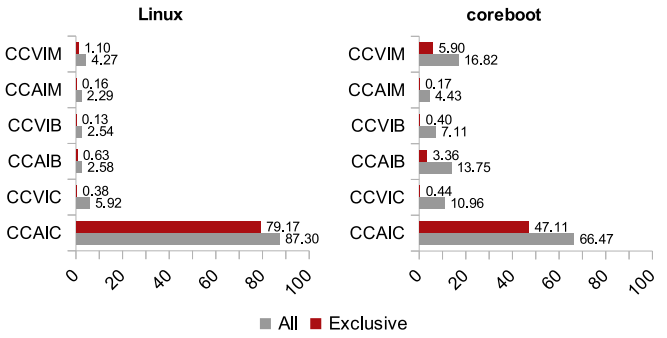
interfaces between features? code duplication necessary? modularization of cross-cutting concerns? ...

Static Modeling of Feature Combinations



Even if multiple inheritance is supported, statically combining features through inheritance is tedious (or infeasible).

Implementing Features [Kröher et al. 2023]



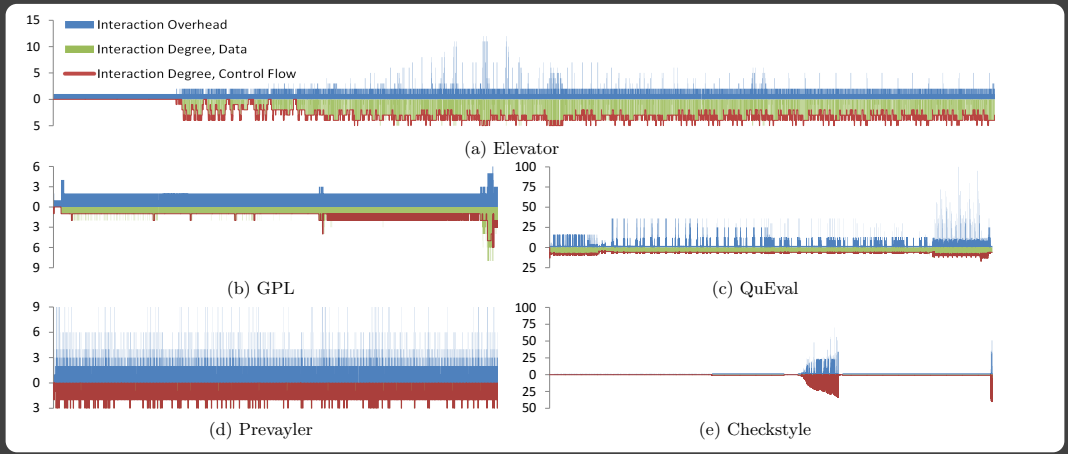
- 1 % of commits in Linux **only** change feature model
- 4 % of commits in Linux change feature model

- <1 % of commits in Linux **only** change build scripts
- 3 % of commits in Linux change build scripts

- 79 % of commits in Linux **only** change source code
- 87 % of commits in Linux change source code

- different percentages for other systems

Execution Traces in Configurable Systems [Meinicke et al. 2016]



insights: not all features interact. some statements may lead to higher interactions than others.

Analyzing Features

Product-Based Strategy

- analyze individual **products**
- \+ sound, complete
- \+ uses off-the-shelf generator γ and analysis α
- – redundant effort
- – does not scale well

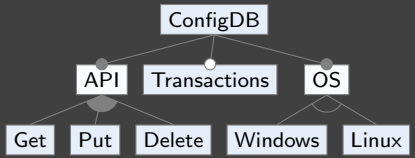
Feature-Based Strategy

- analyze individual **features**
- \+ sound, efficient
- – analysis α requires features with interfaces
- – incomplete: misses all feature interactions

Family-Based Strategy

- analyze the **product line**
- \+ sound, complete, efficient
- – requires careful, hand-crafted analysis α

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

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

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}

Handling Feature Interactions

Solution	Variability	Effort	Size & performance	Quality
S1: Adapt Feature Model		✓	✓	✓
S2: Orthogonal implementation	?	✓	?	✓
S3: Duplicate implementations	✓		✓	
S4: Move source code	✓	✓		
S5: Conditional compilation	✓	✓	✓	
S6: Derivative modules	✓		✓	✓

Course Summary – Summary

Lessons Learned

- how to implement features and variability?
- how to model valid combinations and reason about those?
- how to do quality assurance?
- (how to evolve and maintain a product line?)

Further Reading

see earlier parts

Practice

Questions? Feedback?

FAQ – 12. Evolution and Maintenance

Lecture 12a

- Why do we need to change product lines?
- How does the Linux kernel evolve over time?
- How can changes to feature models be classified?
- What are advantages of classifying changes to feature models?
- Give an example for a refactoring/-generalization/specialization!
- What is referred to as co-evolution of product lines?
- How do feature model, build scripts, and source code co-evolve?

Lecture 12b

- What is the difference between evolution and maintenance?
- Which kinds of maintenance and reengineering are there?
- What are examples for product-line maintenance?
- What are examples for adaptive, corrective, perfective, preventive maintenance?
- What are examples for reverse engineering, forward engineering, reengineering?

Lecture 12c

- What is a software product line?
- When to use which implementation technique for variability?
- How to perform quality assurance for product lines?