Software Product Lines
Part 9: SPL Analysis and Outlook

Daniel Strüber, Radboud University
with courtesy of: **Sven Apel, Christian Kästner, Gunter Saake**

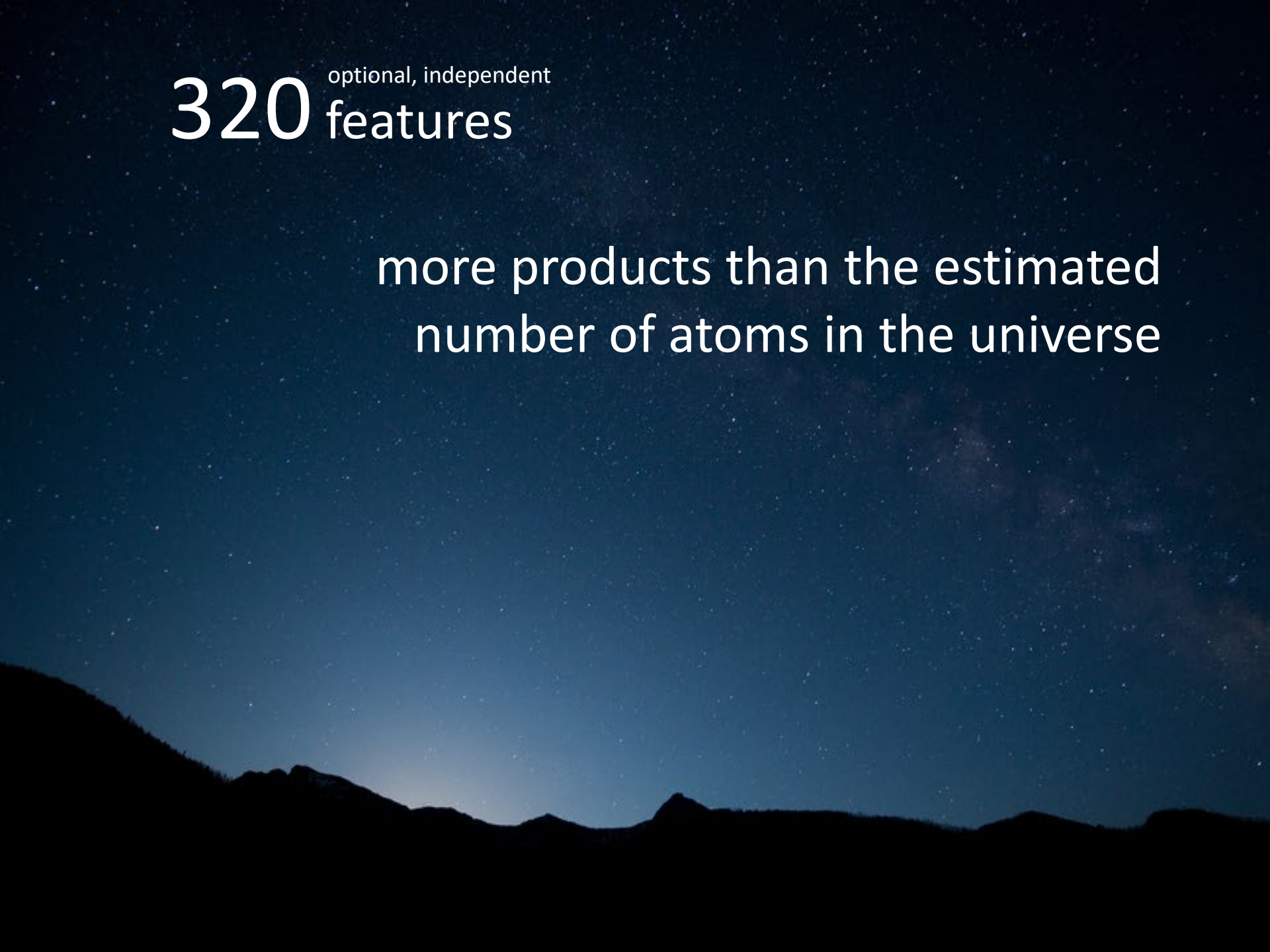
The Problem


Variability = Complexity

33 optional, independent features

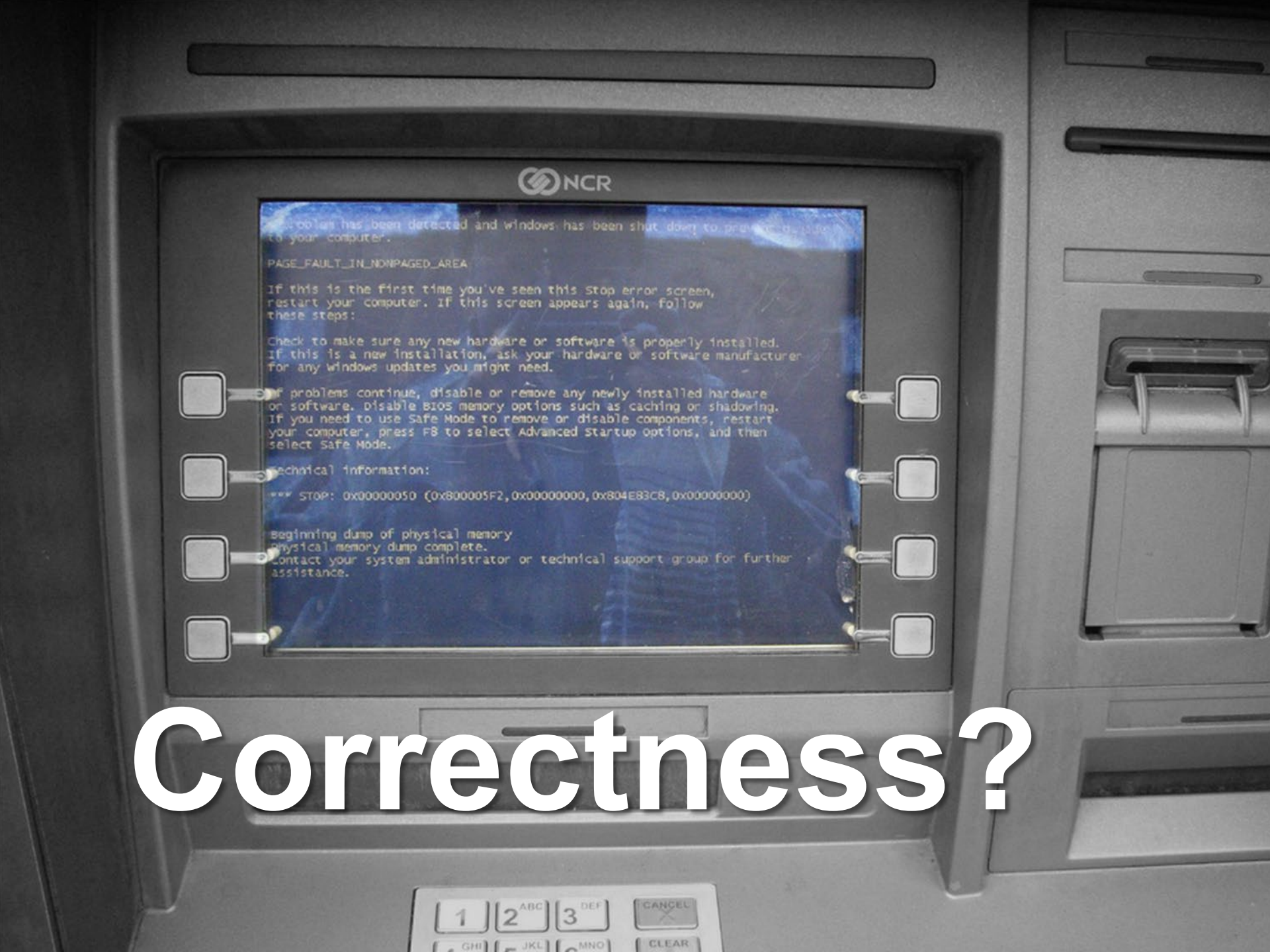
one product for each

person on the planet


320 optional, independent features
more products than the estimated number of atoms in the universe

2000 features

10000 features



Correctness?

Printer firmware

Check products

2000 features
100 printers
30 new printers per year

Linux kernel

Check products

8000 features

? products

Check product line

Implementation with 10000 features

#ifdef, Frameworks, FOP, AOP, ...

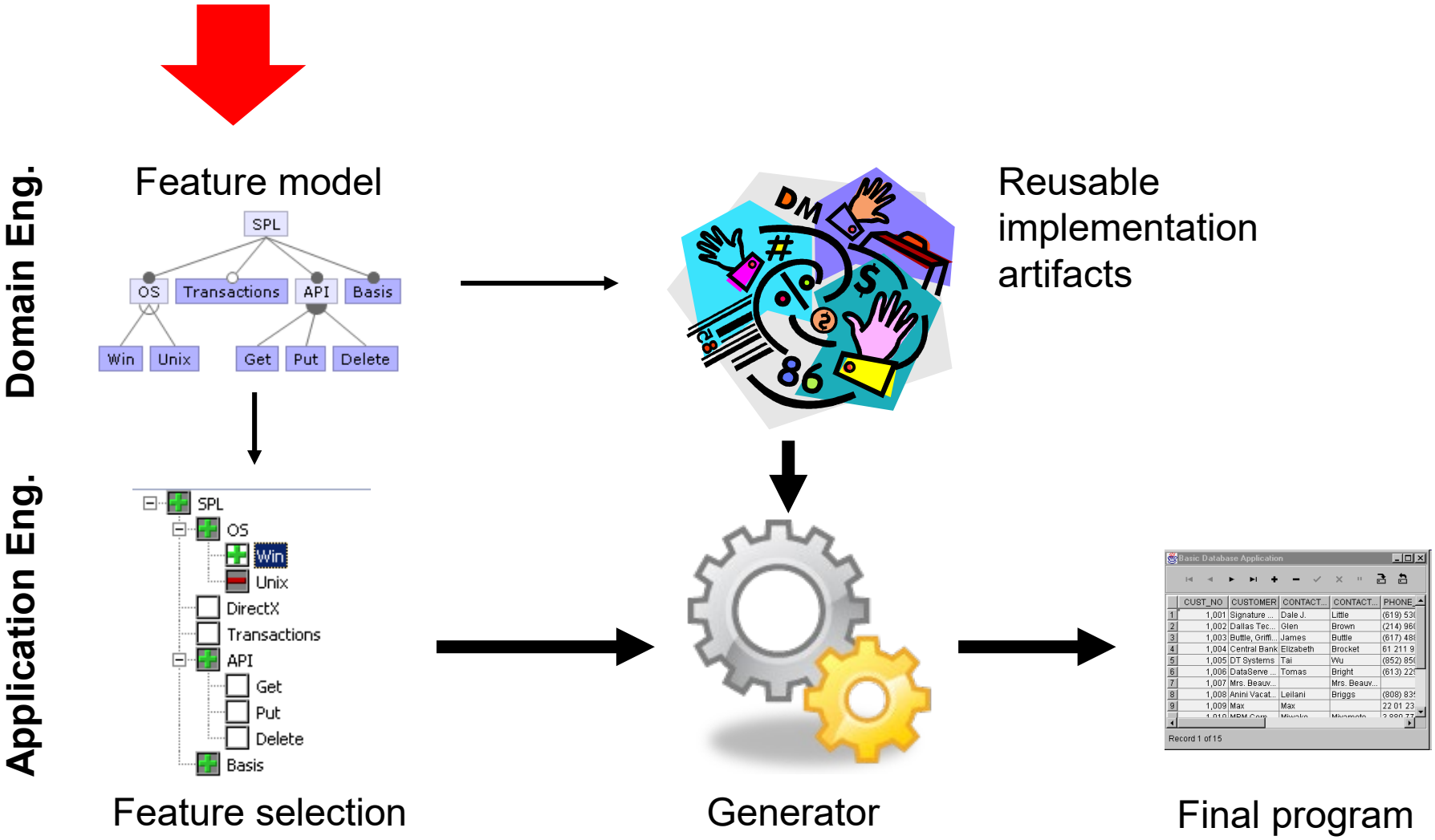
Linux kernel

Agenda

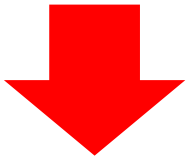
- Analysis of feature models
- Analysis of implementation
  - In isolation
  - With information from the feature model

Analysis of feature models

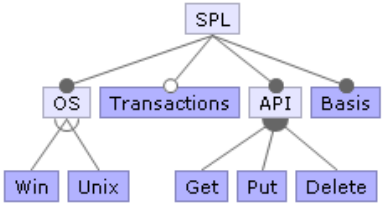
Questions to the feature model



Questions to the feature model



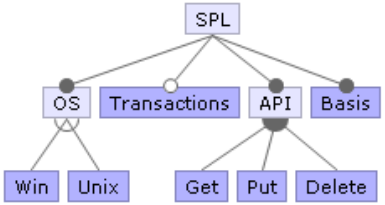
Feature model

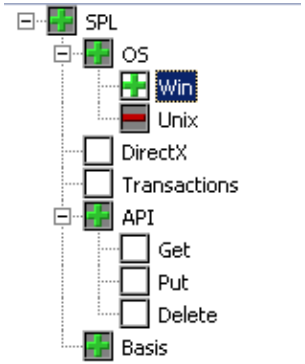


- Is the feature model consistent?
- Which features have to be selected?
- Which features cannot be selected?
- How many valid products exist?
- Are two given feature models equivalent?

Questions to a configuration

Feature model



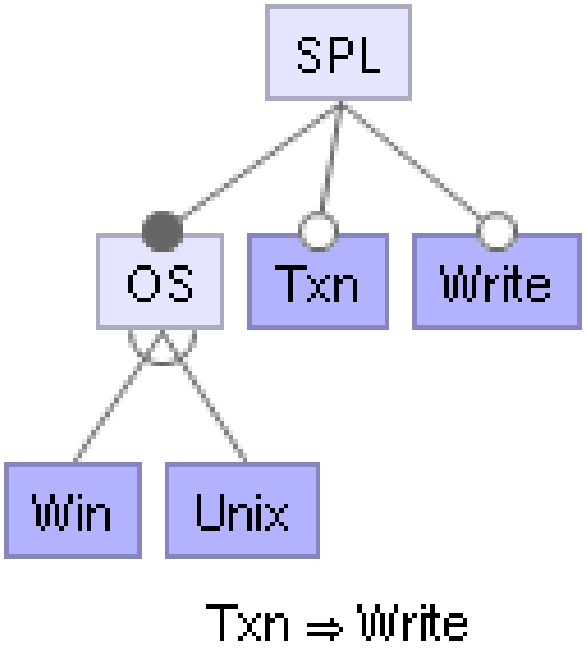


Feature selection

- Is the feature selection valid?
- Which other features have to be selected?
- Which features cannot be selected anymore?

Recap: Feature models & propositional logic

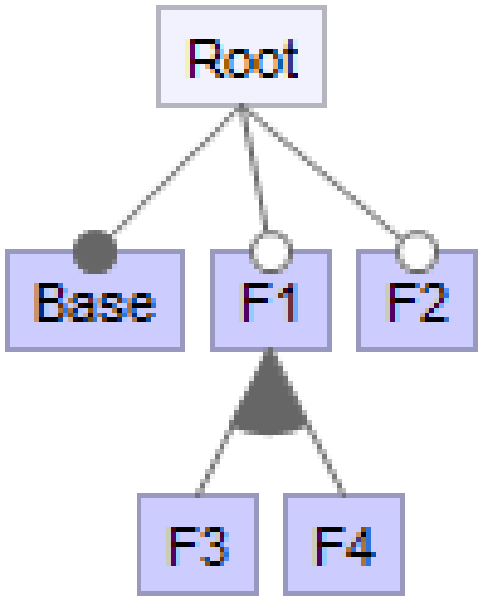
- We can represent feature models as
  - A list of configurations
  - Propositional expression
  - Feature diagram
  - …



$$SPL \wedge$$
$$OS \wedge$$
$$(Unix \vee Win) \wedge \neg(Unix \wedge Win) \wedge$$
$$(Txn \Rightarrow Write)$$

Analysis of feature models

- Are these feature selections valid?
  - {Root, Base, F1, F4}
  - {Root, Base, F2, F3}

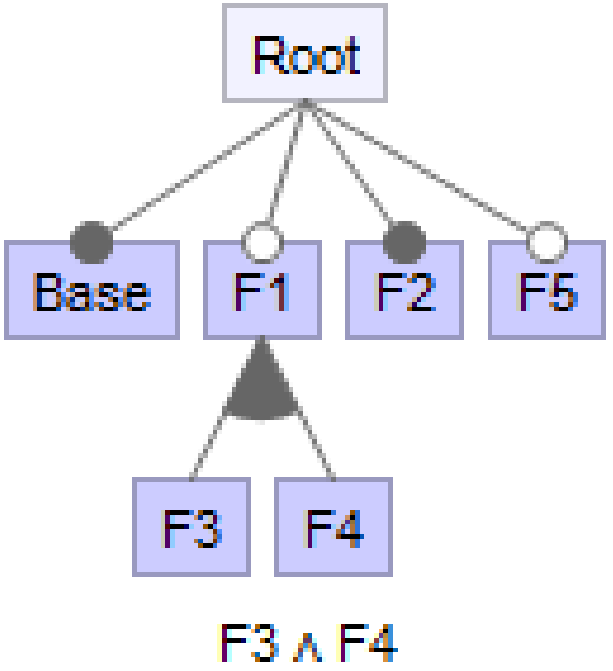


- Replace variables in formula
  - **true** if feature selected, **false** otherwise
- Formula evaluates to **true** for a valid selection

Is the feature model consistent?

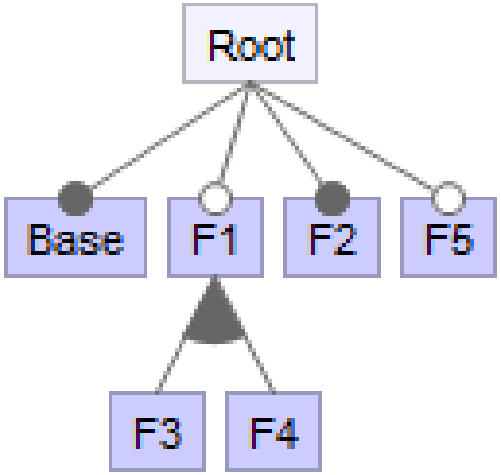
- Does there exist at least one product?

- Formula satisfiable?
- Can use a **SAT solver** to check
  - Tool that checks the satisfiability of a propositional formula
  - Input: a formula F
  - Output: an example assignment satisfying F, or the information that F is insatisfiable
- Query to SAT solver: SAT(FM)



Dead features

- Given a feature model
  - Which features can be selected?
  - Which features have to be selected?
  - Which feature cannot be selected?



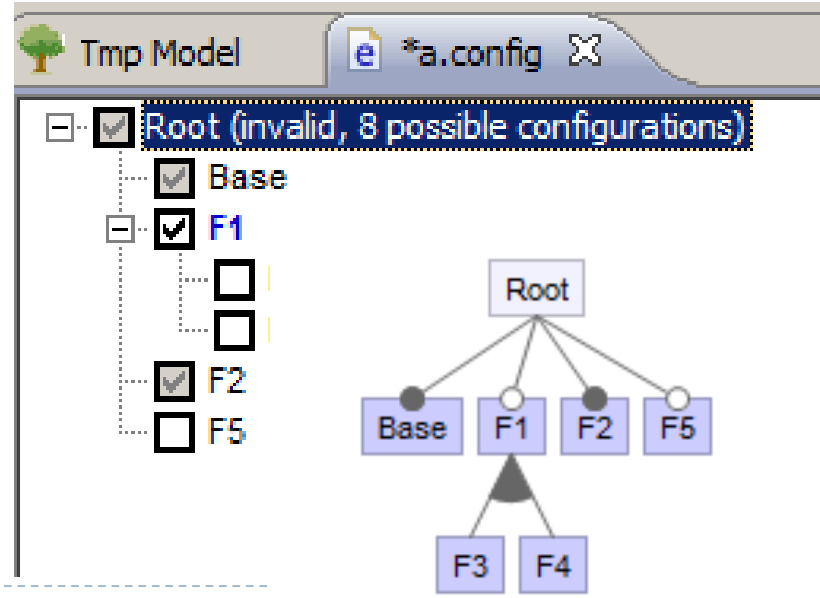
- Feature F selectable if SAT(FM ∧ F)
- Feature F deselectable if SAT(FM ∧ ¬F)

Partial configurations

- Given a partial configuration (feature selection)
  - Which features can still be selected?
  - Which features have to be selected?

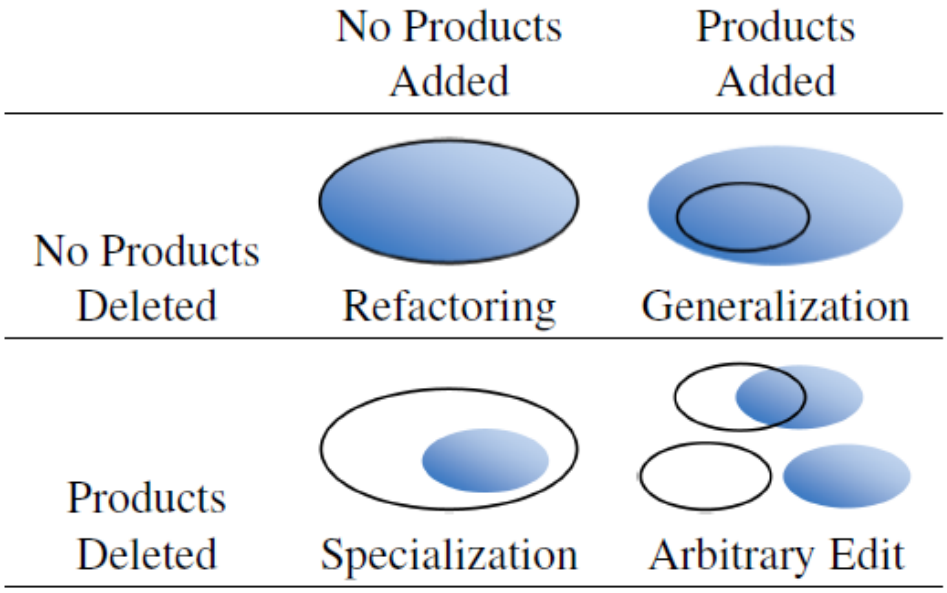
- Add selected features to formula, then same as before

- Feature F still selectable if SAT(FM ∧ Cfg ∧ F)
- Feature F still desectable if SAT(FM ∧ Cfg ∧ ¬ F)



Changes to a feature model

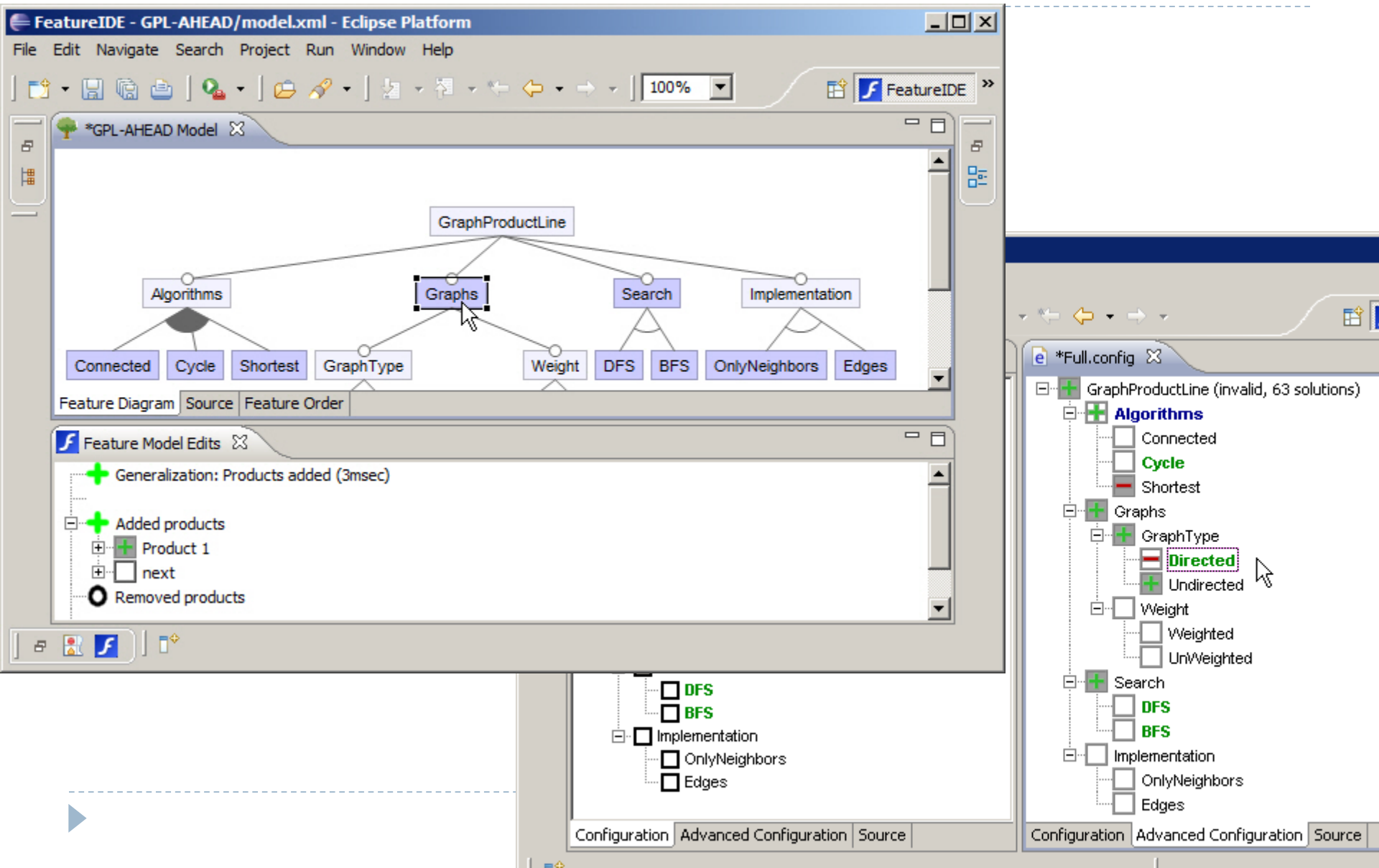
- How does a change affect the configurations?
- Refactorings, specializations, generalizations



- Refactoring if: ¬ SAT(¬(FM1 <=> FM2))

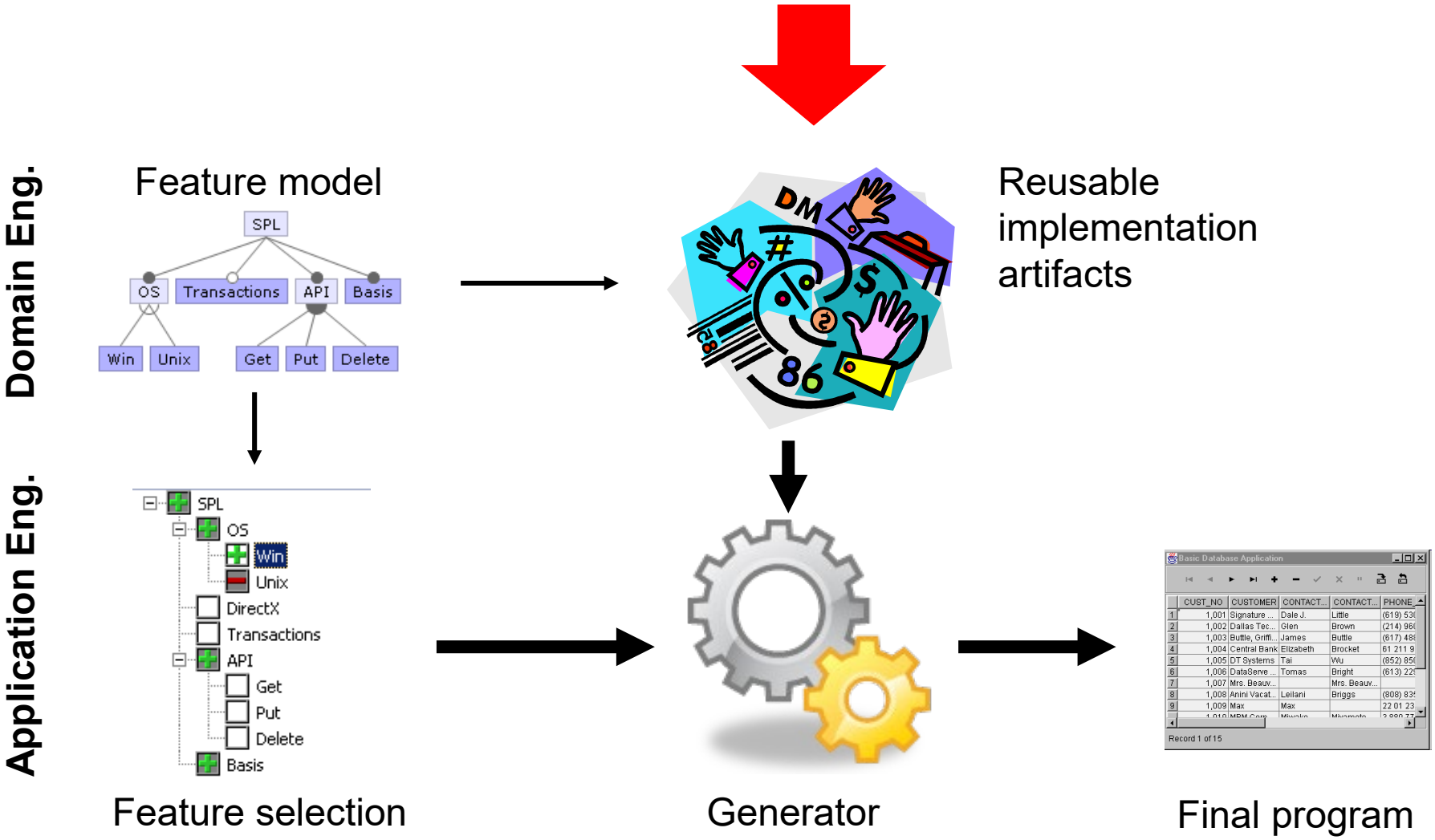
FM1	FM2	¬(FM1 ↔ FM2)
F	F	**F**
F	T	**T**
T	F	**T**
T	T	**F**

Reasoning in FeatureIDE



Analysis of the implementation

Questions to the implementation



Questions to the implementation

- Can a code block be selected?
- Does a feature affect the code base?
- Are there any bugs?
  - syntax errors
  - type errors

Reusable implementation artifacts

Presence conditions

```
#include <stdio.h>

#ifdef WORLD
char * msg = "Hello World\n";
#endif
#ifdef BYE
char * msg = "Bye bye!\n";
#endif

main() {
  printf(msg);
}
```

true

WORLD

BYE

true

Presence conditions

```
line 1
#ifdef A
line 2
    #ifndef B
    line 3
    #endif
line 4
#elif defined(X)
line 5
#else
line 6
#endif
```

true

A

$A \wedge \neg B$

A

$\neg A \wedge X$

$\neg A \wedge \neg X$

Conjunction of ifdefs and ifndefs yields the **presence condition** (PC) of a line

Dead code

```
line 1
#ifdef A
line 2
   #ifndef A
   line 3
   #endif
line 4
#elif defined(X)
line 5
#else
line 6
#endif
```

true

A

$A \wedge \neg A$

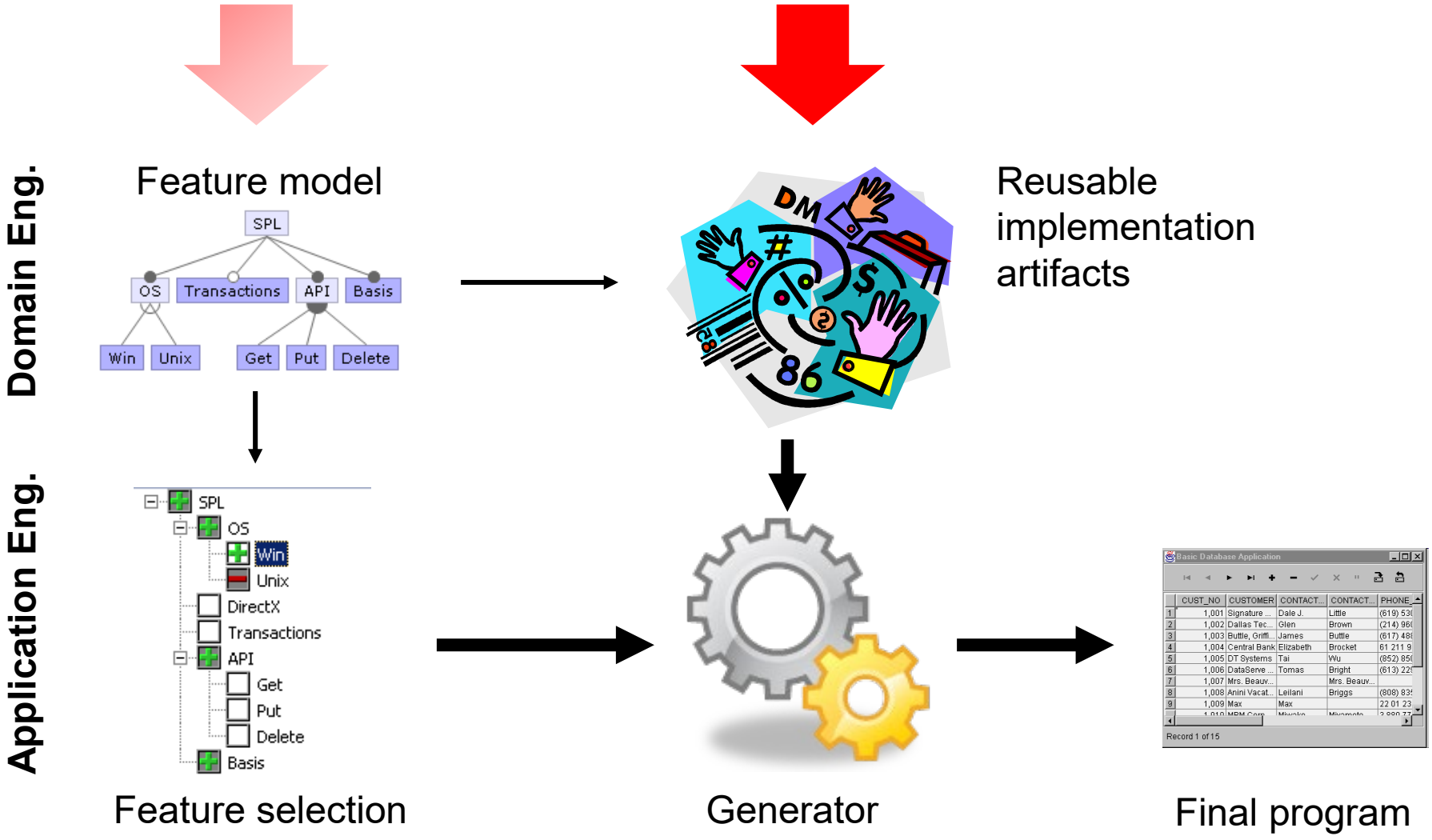
A

$\neg A \wedge X$

$\neg A \wedge \neg X$

Analysis:
SAT(PC(Block i))

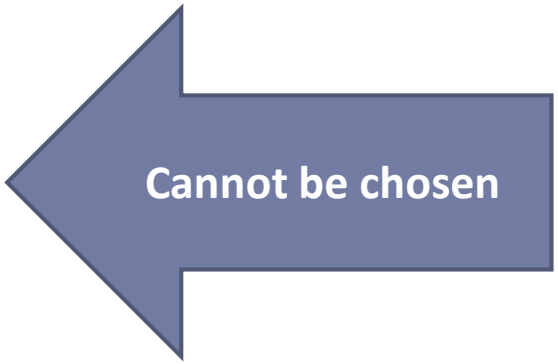
Questions to the implementation with information from the feature model



Dead source code

```
line 1                  true
#ifdef A                A
line 2
   #ifdef B
   line 3               A ∧ B
   #endif
line 4                  A
#elif defined(X)
line 5                  ¬A ∧ X
#el
lin
#e
```

$FM = (A \vee B) \wedge \neg(A \wedge B)$



Analysis:
SAT(FM ∧ PC(Block i))

Using feature model information to analyse the code

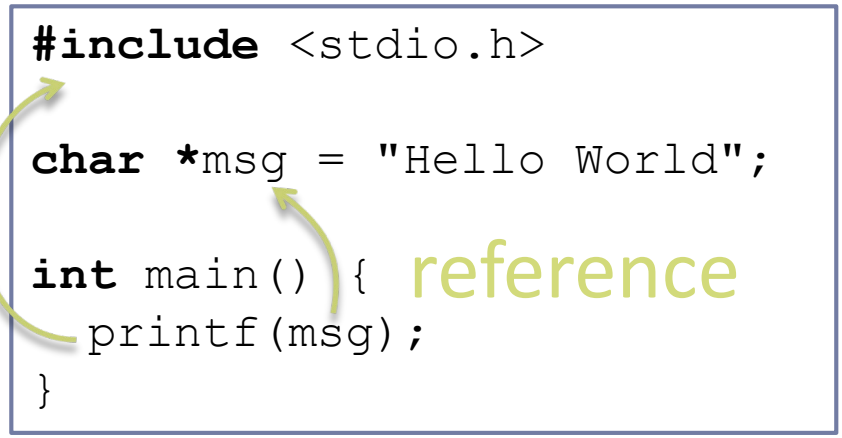
Additional questions

- Which feature modules are never included in a product?
- Which features do not affect the code?

Combined analysis of feature model and implementation

Type systems for product lines

Type checking

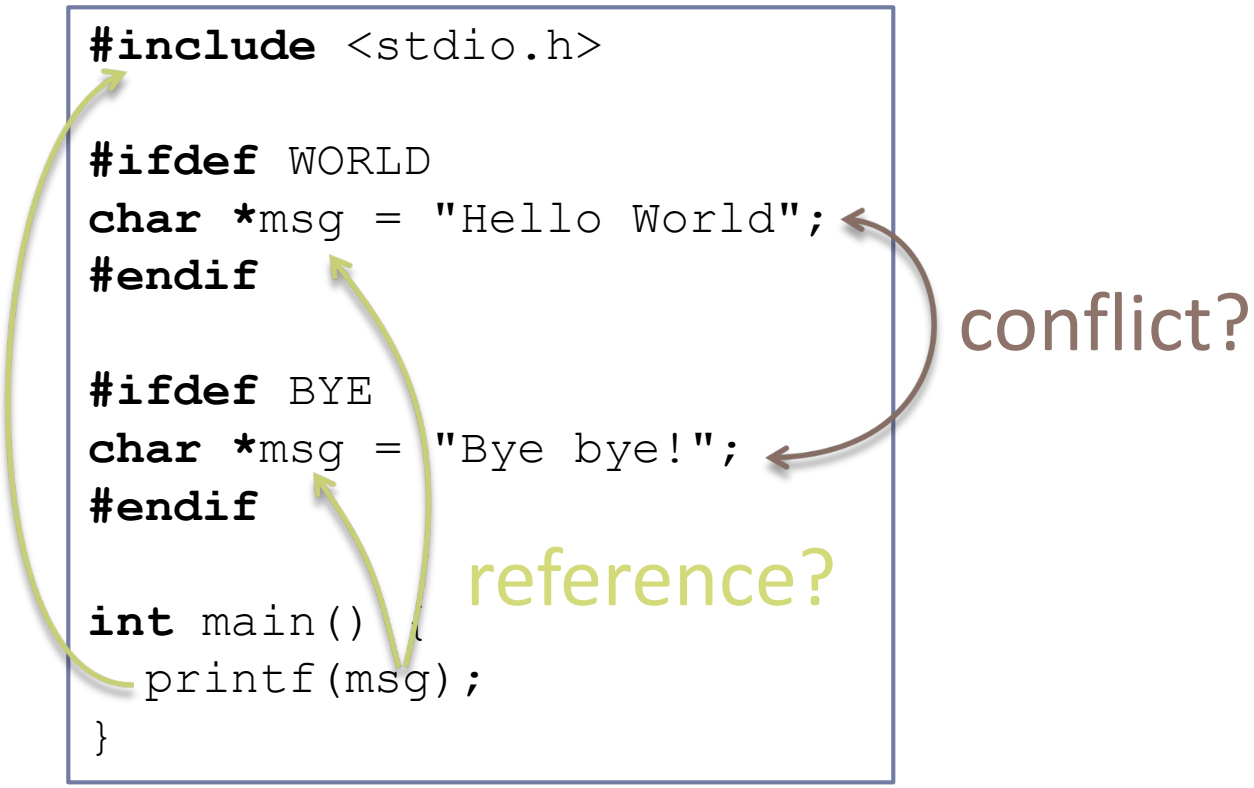


- Type error: referenced variable/method doesn‘t exist

Variability-aware type checking

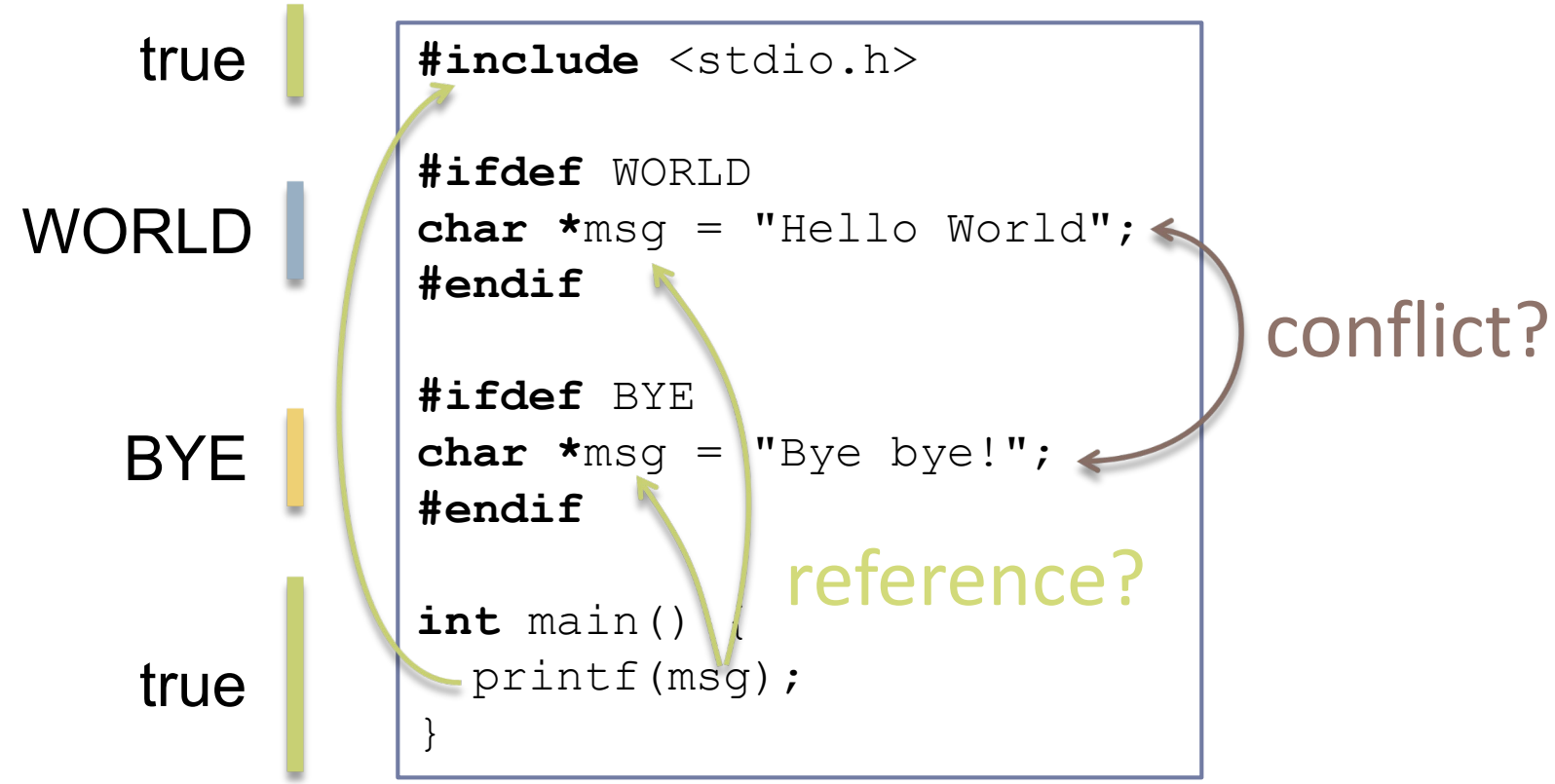
```
#include <stdio.h>

#ifdef WORLD
char *msg = "Hello World";
#endif

#ifdef BYE
char *msg = "Bye bye!";
#endif

int main() {
  printf(msg);
}
```

Variability-aware type checking

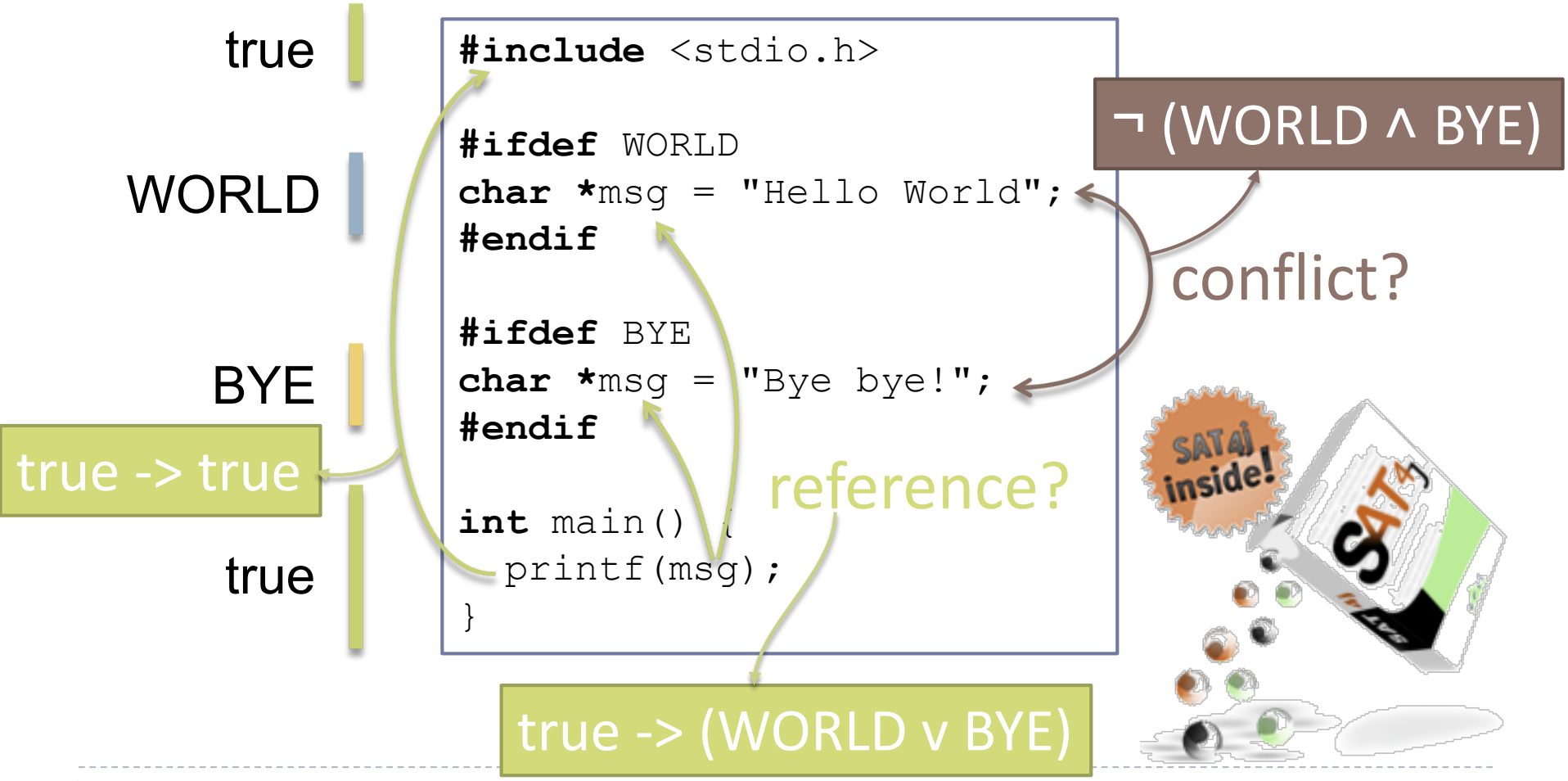


Variability-aware type checking

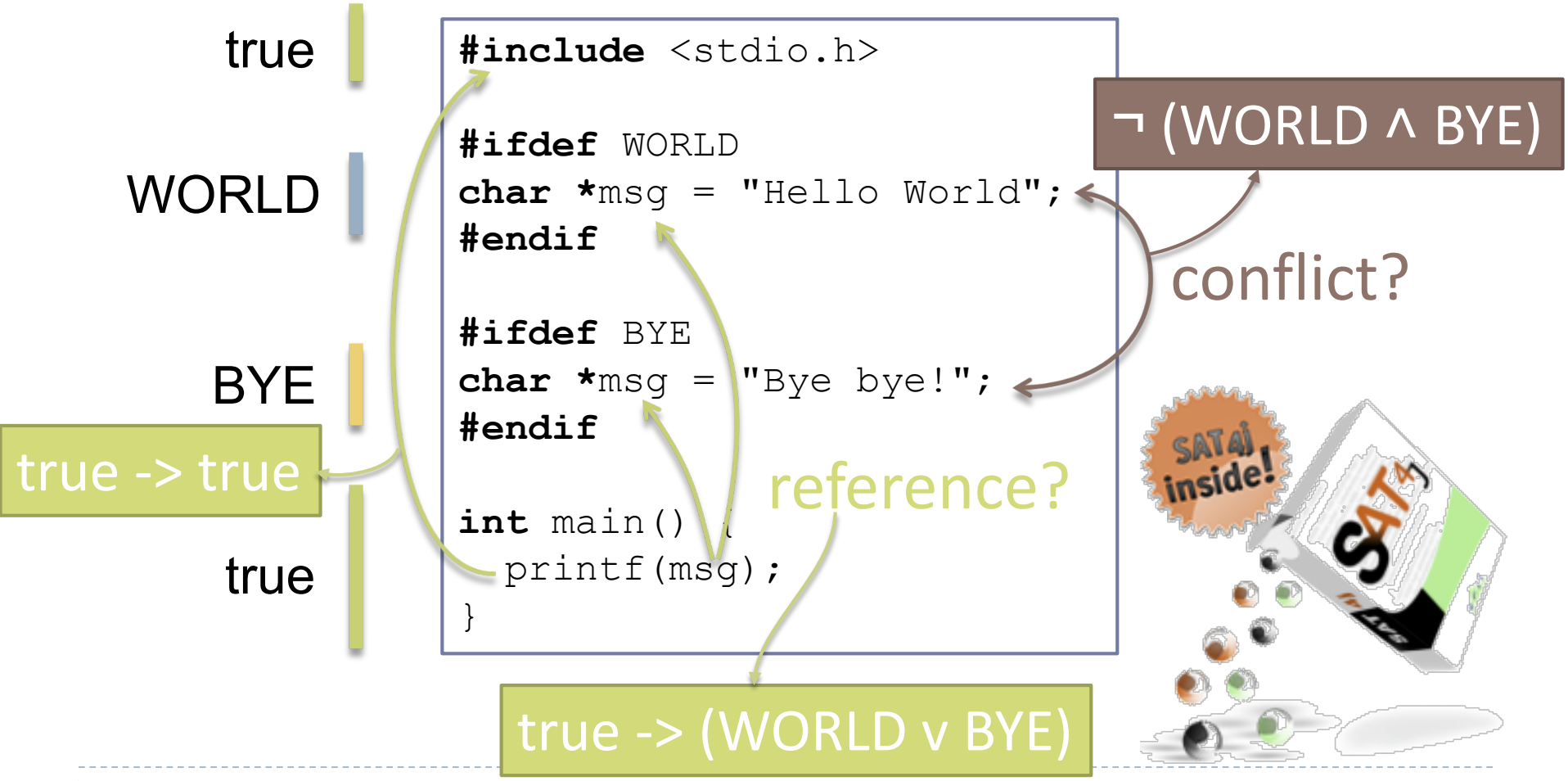
Presence conditions:



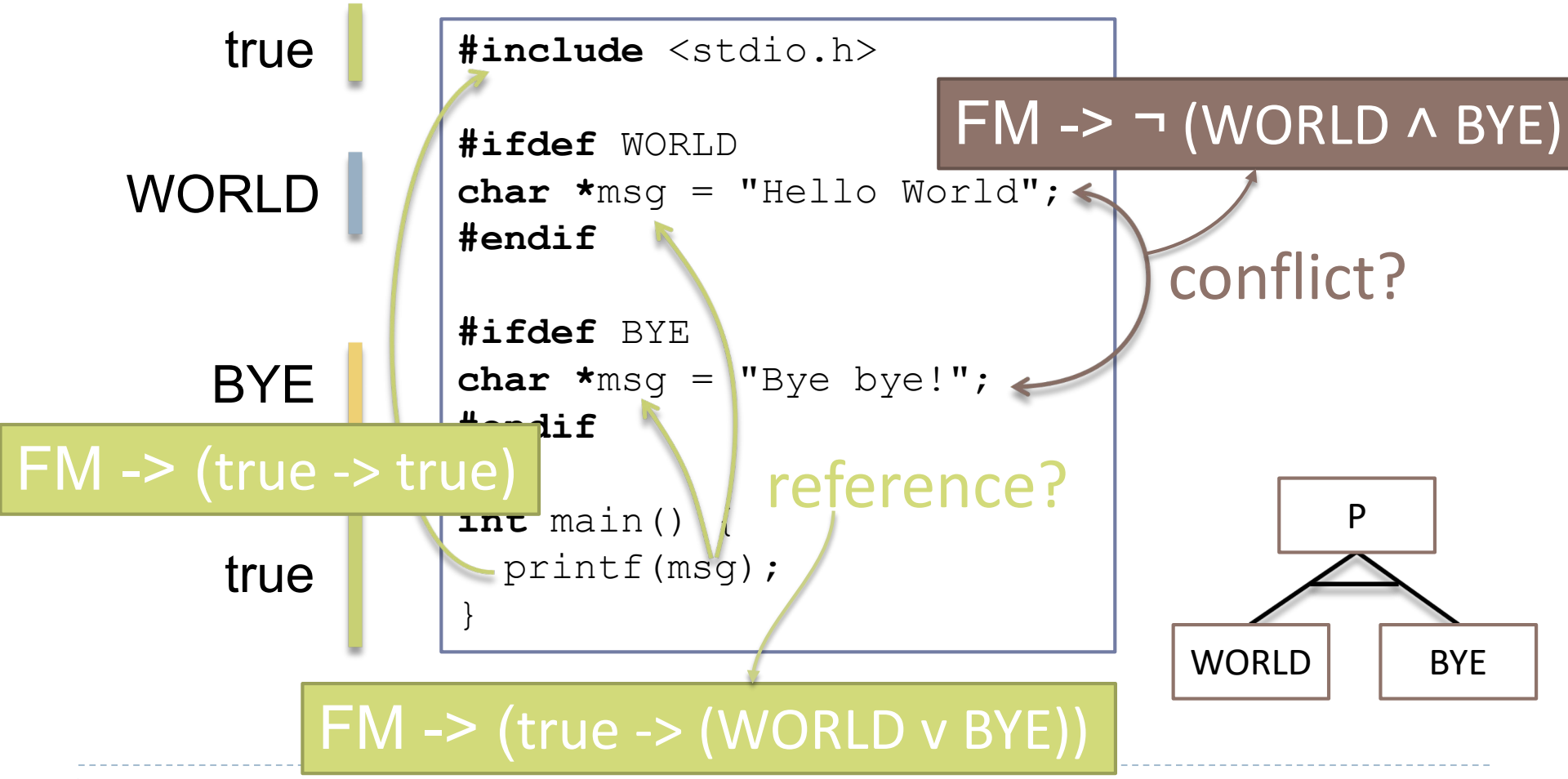
Variability-aware type checking



Reachability: PC(Source) -> PC(Target)
Conflicts: ¬(PC(Def1) ∧ PC(Def2))



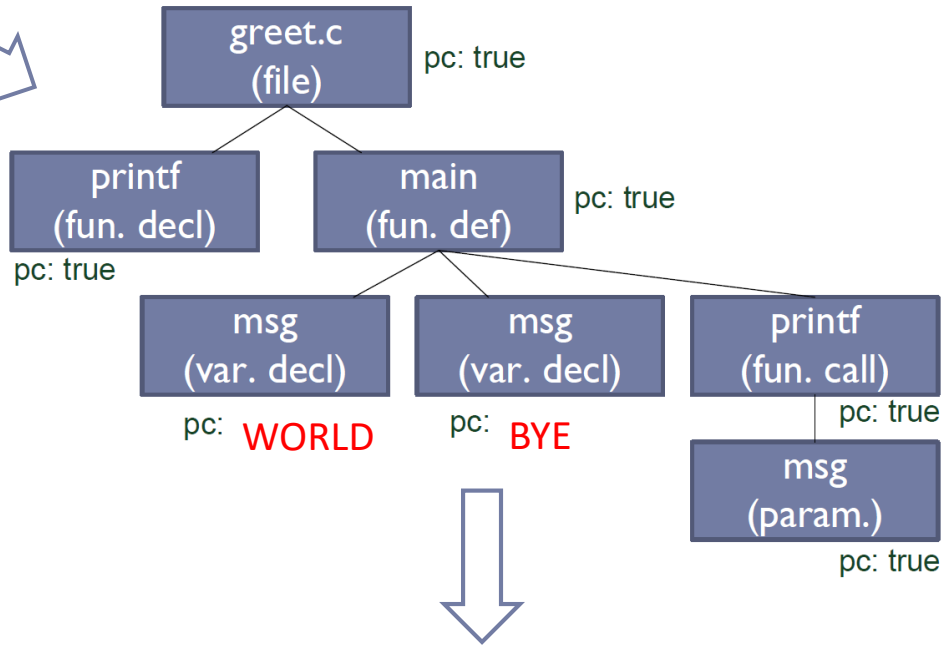
Also consider feature model



Underlying concepts

```
#include <stdio.h>

#ifdef WORLD
char * msg = "Hello World\n";
#endif
#ifdef BYE
char * msg = "Bye bye!\n";
#endif

main() {
  printf(msg);
}
```

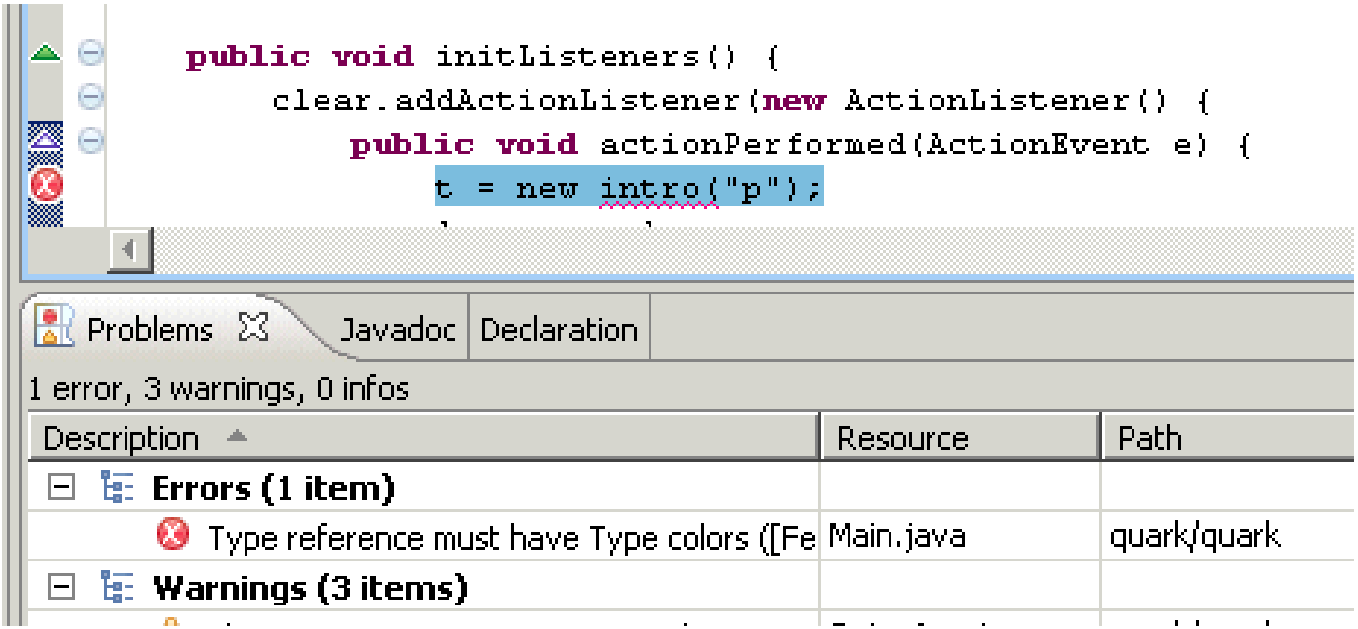
AST enriched with variability information

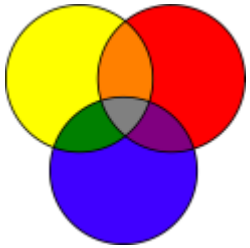


Name	Type	Scope	Presence Condition
printf	char * → int	0	true
msg	char *	0	WORLD
msg	char *	0	¬WORLD

Extended mechanism for reference lookup

Type system implemented in CIDE





http://fosd.net/cide

Type system implemented in CIDE

```
public class Test {
        private static String msg_hi = "Hello world!";
        private static String msg_bye = "Bye bye!";
        public static void main(String[] args) {
                System.out.println(msg_hi);
                System.out.println(msg_bye);
        }
}
```

Similar type checker for AOP/FOP:

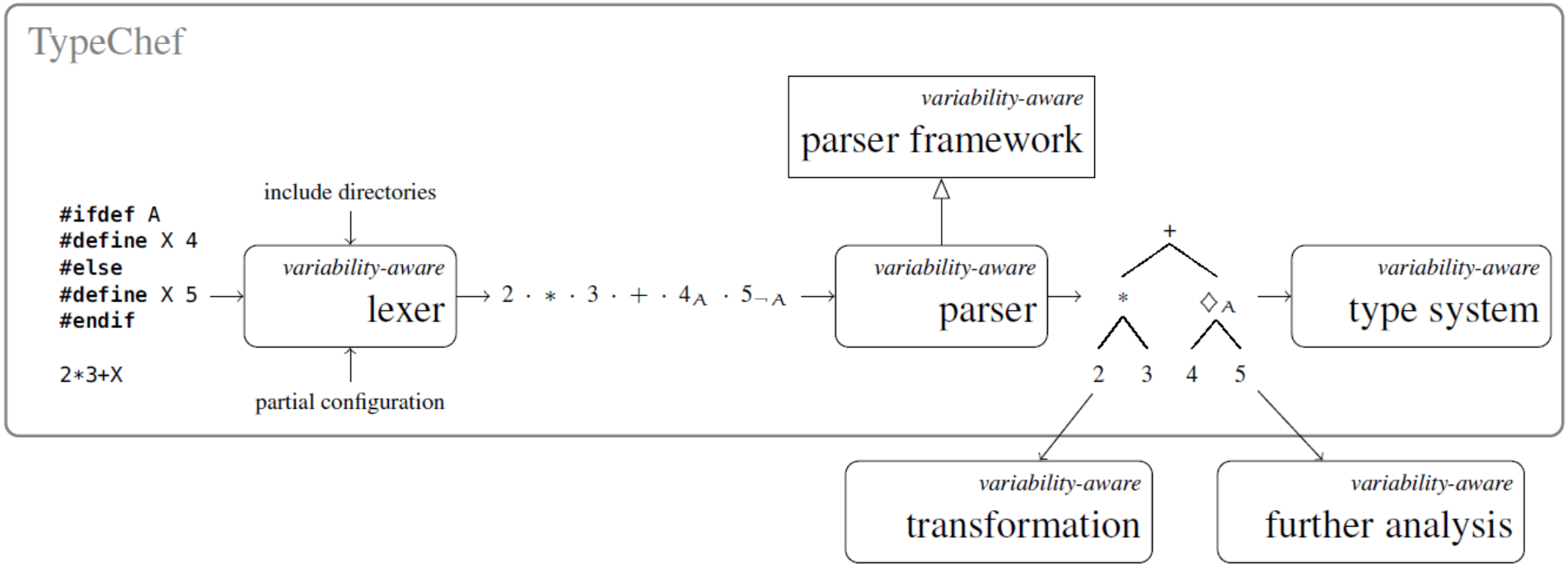
http://www.fosd.de/fuji

Variability-Aware

Parser
Type System
Static Analysis
Bug Finding
Testing
Model Checking
Theorem Proving
…

TypeChef

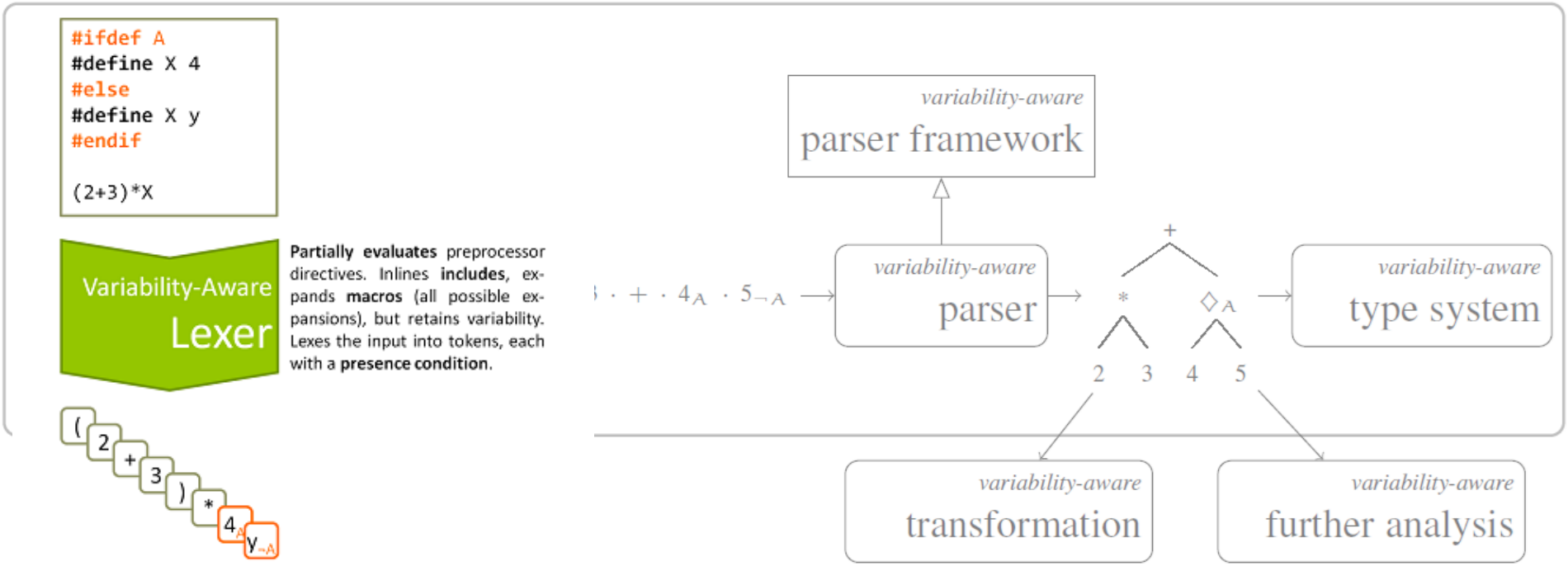
TypeChef



*Research project with the goal of **analyzing ifdef variability in C code** with the goal of **finding variability-induced bugs** in large-scale real-world systems.*

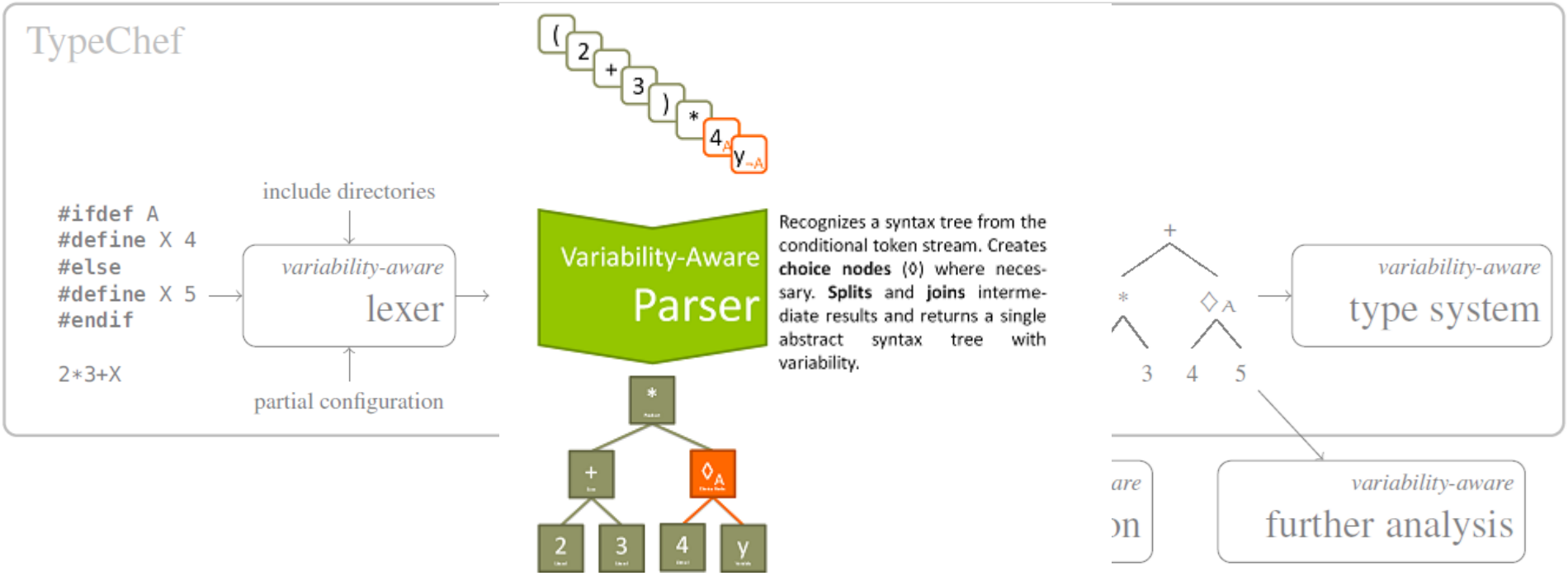
https://ckaestne.github.io/TypeChef/

TypeChef



Partially pre-processes *the code in a variability-aware fashion and produces a* ***conditional token stream***

TypeChef



***Parses** the conditional token stream as an **abstract syntax tree that contains the variability** in form of choice nodes.*

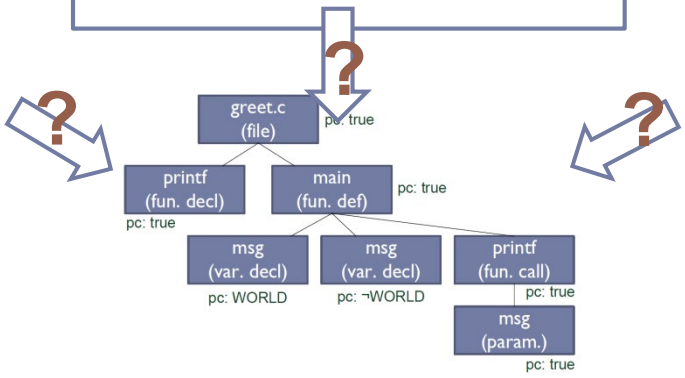
Parsing C code before preprocessing is hard

have to expand macros before parser

```
#define P(msg) \
    printf(msg);

main() {
  P("Hello\n")
  P("World\n")
}
```

undisciplined annotations*

```
if (!initialized)
#ifdef DYNAMIC
  if (enabled)
#endif
    init();
```

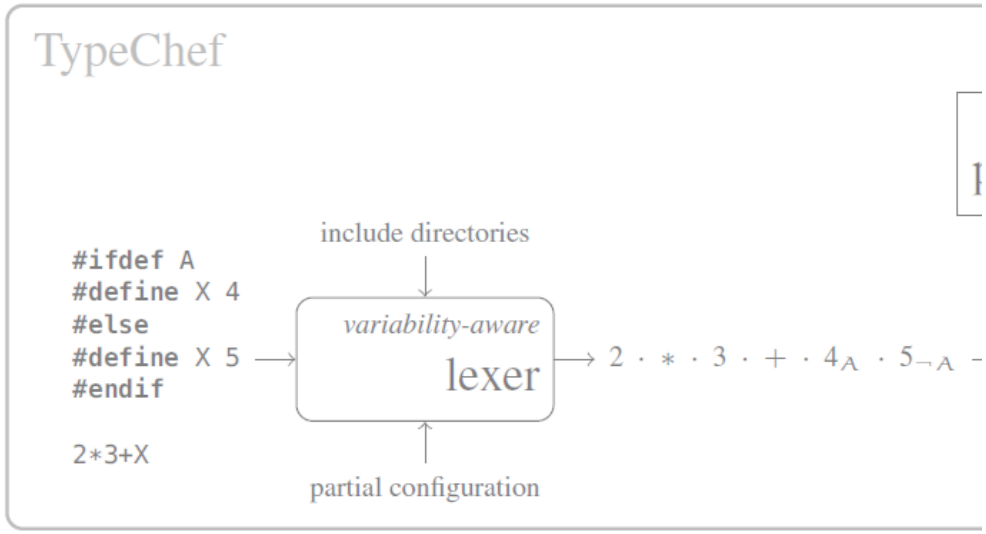


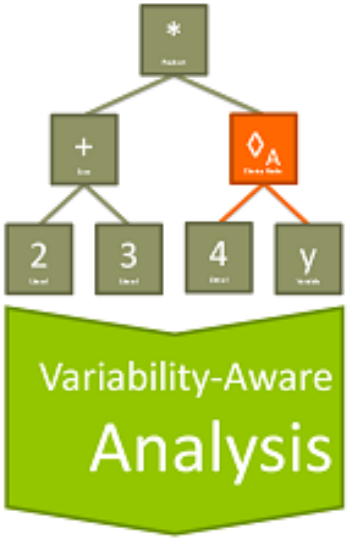
alternative macros

```
#ifdef BIGINT
#define SIZE 64
#else
#define SIZE 32
#endif

allocate(SIZE);
```

* study of 40 open source projects: 16% of annotations is undisciplined

TypeChef





Detects errors in **all configurations** by analyzing the abstract syntax tree with variability. For example, checks if variables can be resolved in all configurations. Possible variability-aware analyses include **type checking**, **static analysis**, and **model checking**.

```
Error Report:
› Syntax correct in all config.
› Variable y not declared in
    configurations without A.
```

*Eventually, TypeChef can rely on a **variability-aware type system** to perform **type checking** on these trees, **variability-aware data-flow analysis** to perform data-flow analysis...*

Summary

Summary

- Variability ⇔ Complexity
- Analyses on feature models **and** implementation are required
- Enumerating and analysing all products in general infeasible
  - ➔ Variability-aware analysis (TypeChef, CIDE, Fuji)

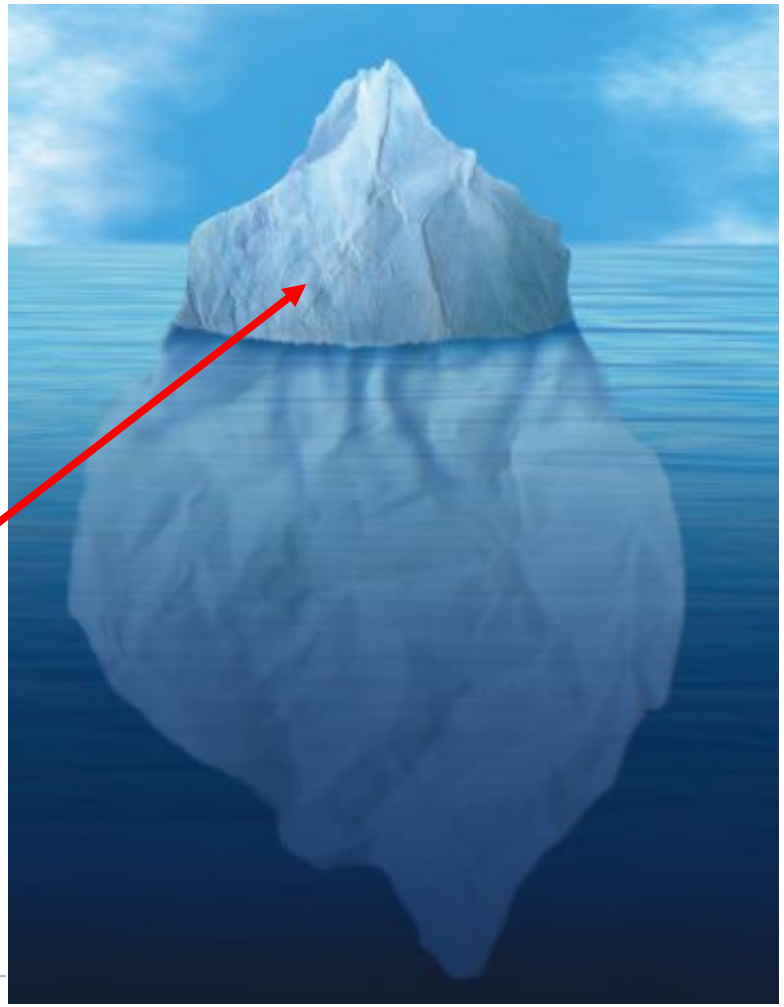
Software Product Lines: Big Picture and Outlook

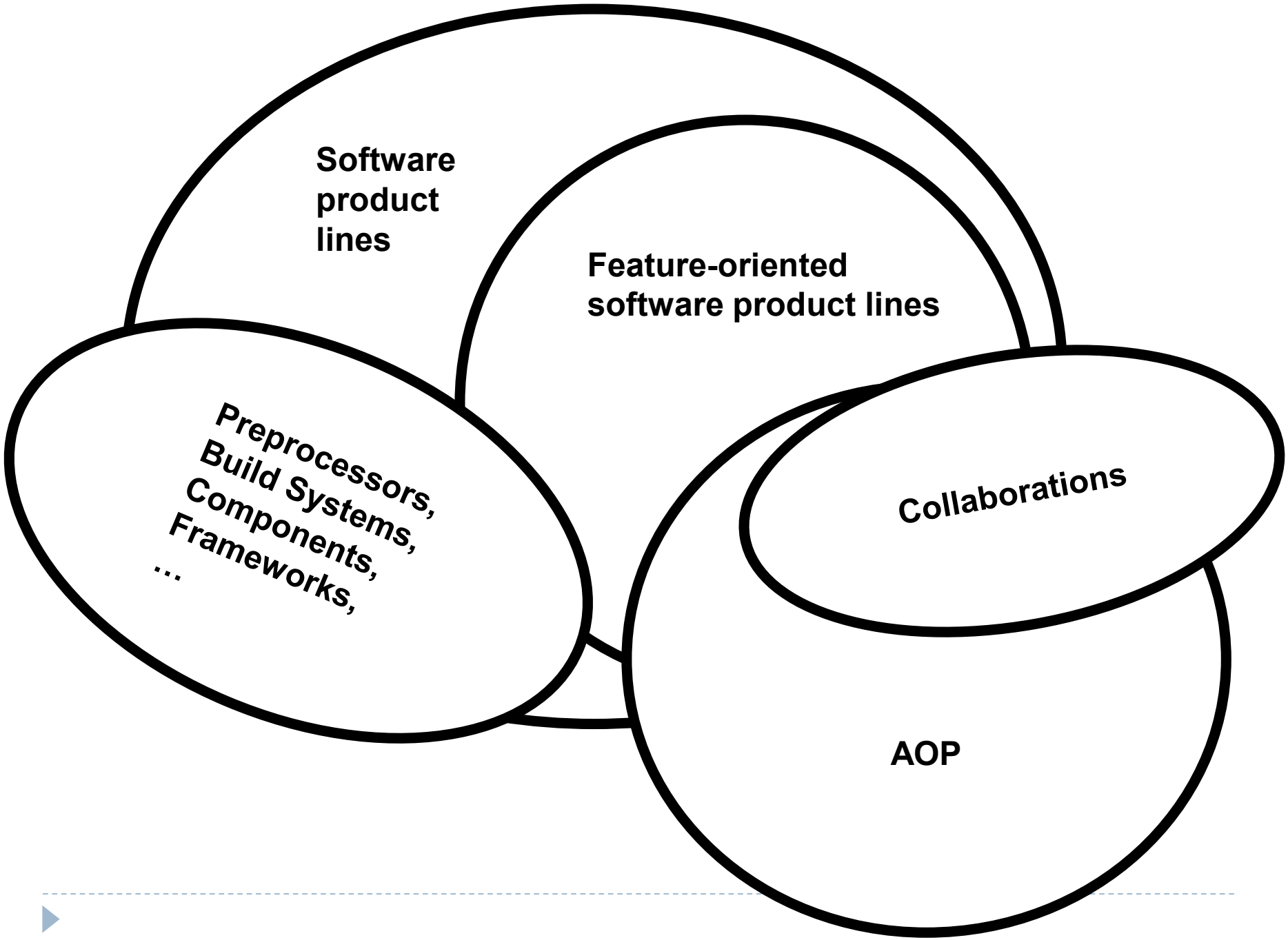
What have we learned?

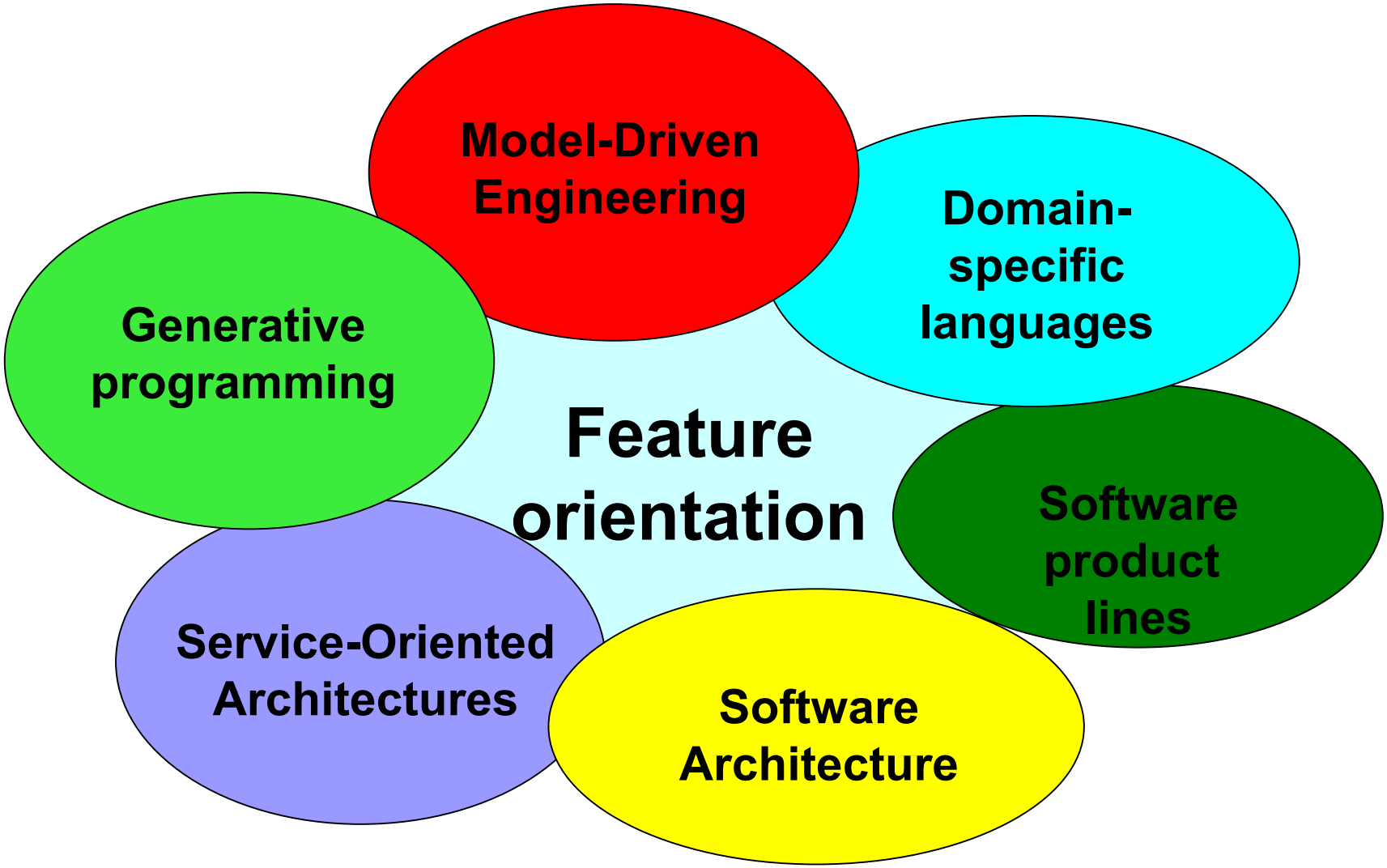
- Basics of software product lines
- Classical implementation techniques (parameters, #ifdefs, frameworks, components…)
  - …and their limitations
- Vision of feature-oriented software product lines
  - Language support for features: collaborations, roles, aspects, etc.
  - Product line analysis

Perspective?

- How do the course contents fit into the big picture of software engineering?




Software product lines
Feature-oriented software product lines
Preprocessors, Build Systems, Components, Frameworks, …
Collaborations
AOP


Model-Driven Engineering
Domain-specific languages
Generative programming
Feature orientation
Software product lines
Service-Oriented Architectures
Software Architecture

Further product line topics

- Definition of variability
- Domain analysis, scoping, requirements engineering
- ROI assessment, product management, market analysis, risk management
- Organisation
  - Project initiation, financing
  - Organisational planning, roles, responsibilities
  - Processes
- Testing, Verification

Product line topics in 2024

SPLC 2024
28th ACM International Systems and Software Product Line Conference
Luxembourg

SPLC 2024 | IMPORTANT DATES | COMMITTEES | PROGRAM | REGISTRATION | CALLS

28TH ACM INTERNATIONAL SYSTEMS AND SOFTWARE PRODUCT LINE CONFERENCE (SPLC 2024)

Search on the website

Search ...

Check out our latest news

Special Issue on Trends in Systems and Software Product Line Engineering
10/07/2024

Early registration deadline extended!
29/07/2024

The call for submissions to the special issue is now available online!
23/07/2024

- https://2024.splc.net/

Product line topics in 2024

- **Analysing and Testing**
  1. Pragmatic Random Sampling of the Linux Kernel: Enhancing the Randomness and Correctness of the conf Tool doi.org/10.1145/3646548.3672586
  2. Feature-oriented Test Case Prioritization Strategies: An Evaluation for Highly Configurable System doi.org/10.1145/3646548.3672592

  Security

  3. Should I Bother? Fast Patch Filtering for Statically-Configured Software Variants doi.org/10.1145/3646548.3672585

  Applications

  4. Leveraging Phylogenetics in Software Product Families: The Case of Latent Content Generation in Video Games https://doi.org/10.1145/3646548.3672596

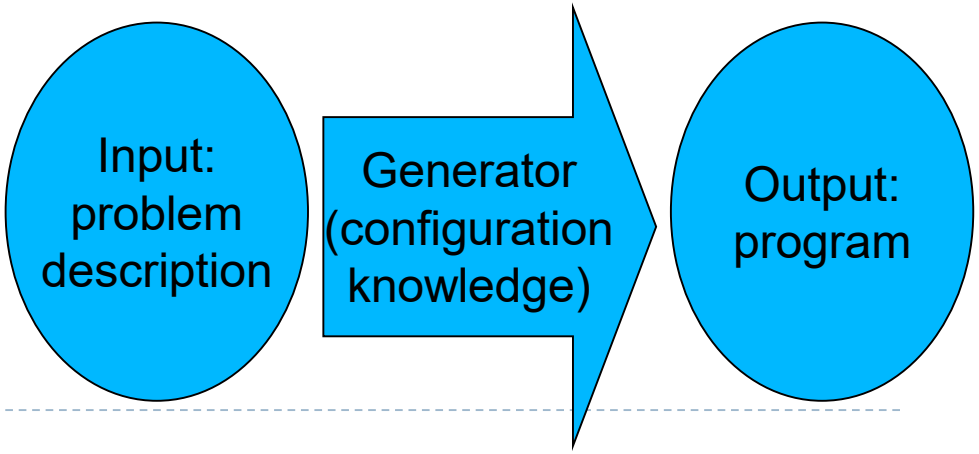
- Maxime Cordy, Daniel Strüber, Mónica Pinto, et al.: SPLC 2024 proceedings. https://dblp.org/db/conf/splc/splc2024a.html

Generative Programming I

- Big chunks of software automatically produce by a code generator, from a specification using code fragments

  „Generative programming (GP) is a style of computer programming that uses automated source code creation through generic classes, prototypes, templates, aspects, and code generators to improve programmer productivity.“

- Application areas
  - Form generators
  - Compiler compilers
  - Parser generators
  - Query optimisation
  - …



Generative Programming II

- FOP as a form of GP
  - Feature selection ⇔ abstract input program
  - Feature modules ⇔ code templates
  - Feature composition ⇔ generation
- GP beyond that
  - Inputs can be more complex
    (e.g. form specification, grammar, domain-specific language…)
  - Each kind of generator
  - Metaprogramming (programs manipulating programs)

Domain-specific languages

- Principle of abstraction
  - Express problems and solutions in a purpose-tailored declarative language
  - Generator/interpreter generates code
- Benefits:
  - less redundancy
  - improved readability
  - fewer irrelevant technical details
  - easier to learn
  - more targeted error messages…

```
Set camera size:
        400 by 300
pixels.
Set camera position:
        100, 100.
Move 200 pixels right.
Move 100 pixels up.
Move 250 pixels left.
Move 50 pixels down.
```

Domain-specific languages II

- Domain-specific modeling languages
- Query languages
- Unix shell scripts
- Spreadsheet programs
- Regular expressions
- Document and data description languages
- Graph description languages
- Form definition languages
- etc.

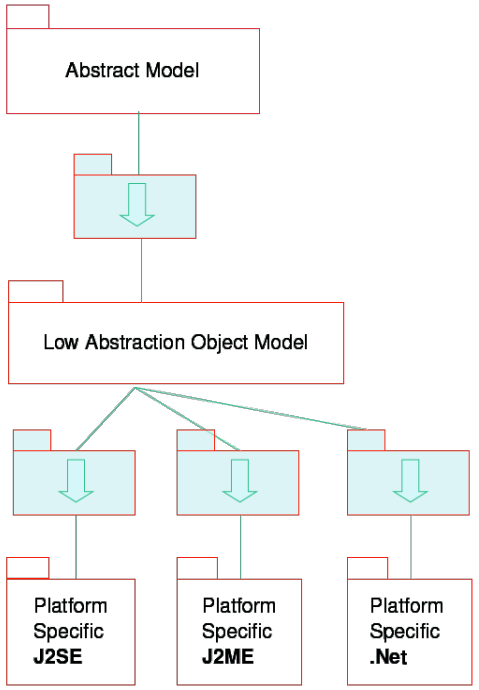
Domain-specific languages III

- DSL can be input for GP
- Connection to product lines/FOP:
  - Feature model ⇔ DSL
  - Feature ⇔ language construct of DSL
  - Feature composition ⇔ translation process

Model-Driven Engineering I

- Software described using design models
- Stepwise transformation towards a running software systems
- Last step in transformation chain is code

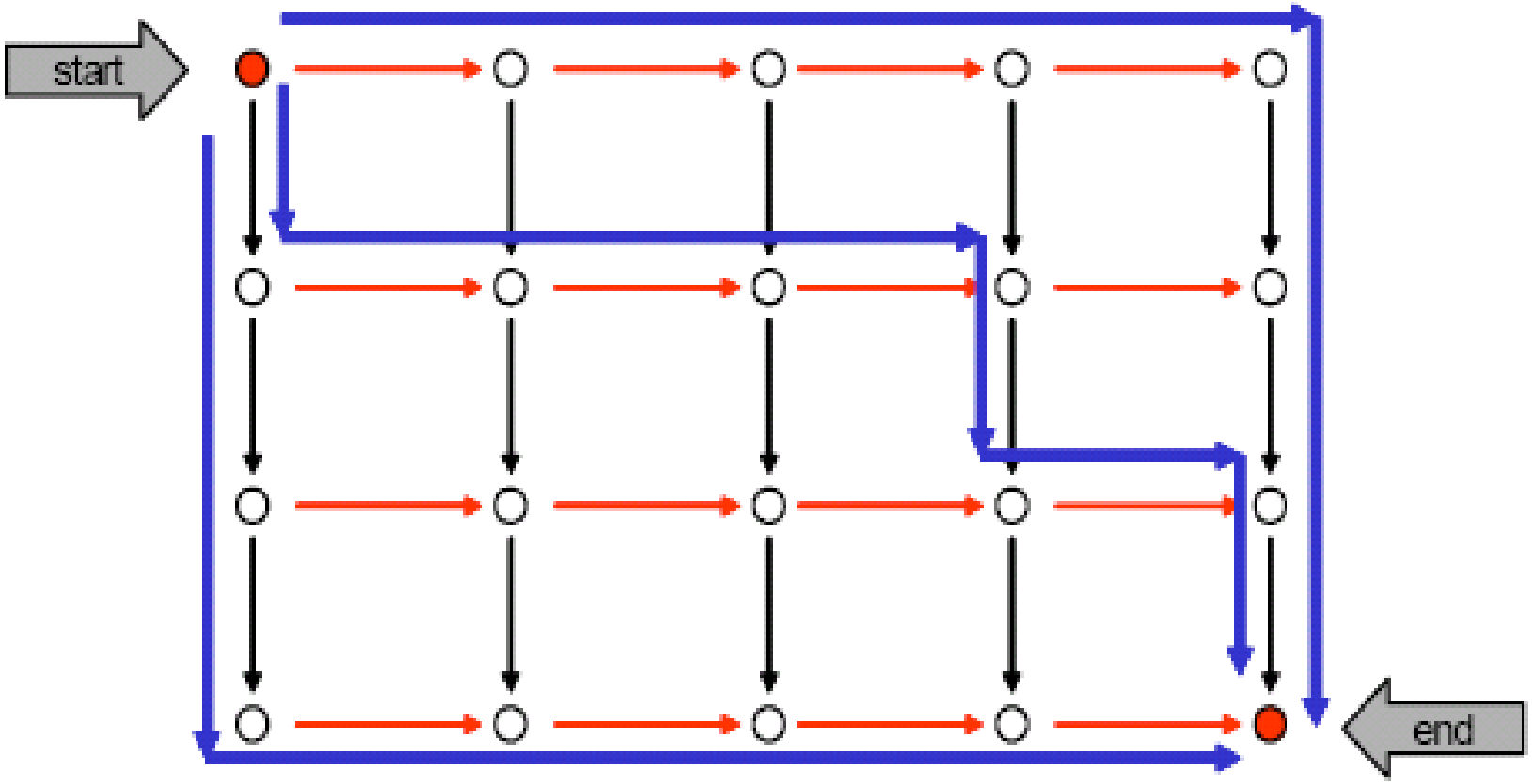
Model-Driven Engineering (MDE) or Model-Driven Development (MDD) refer to the systematic use of models as primary engineering artifacts throughout the engineering lifecycle. MDE/MDD can be applied to software, system, and data engineering. Models are considered as first class entities.



Model-Driven Engineering II

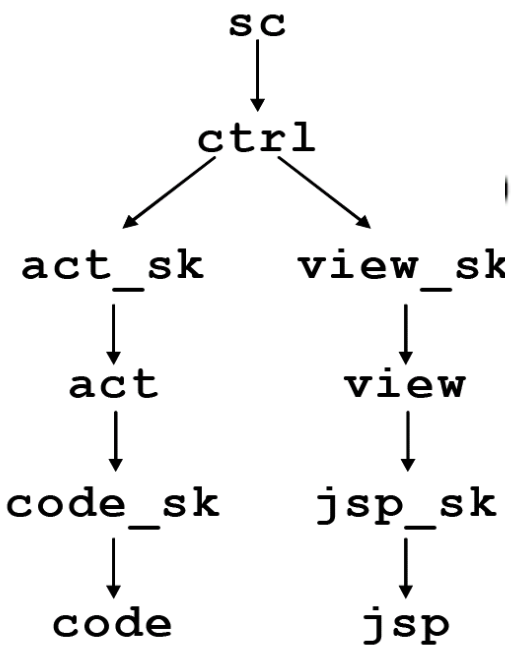
- MDE is a type of generative programming
  - Model as input, new model or code as output
- General-purpose and domain-specific modeling languages
- Different ways of connecting to feature-orientation
  - A) Feature selection as a model that is transformed to code (like in GP/DSL)
  - B) Features can refine models

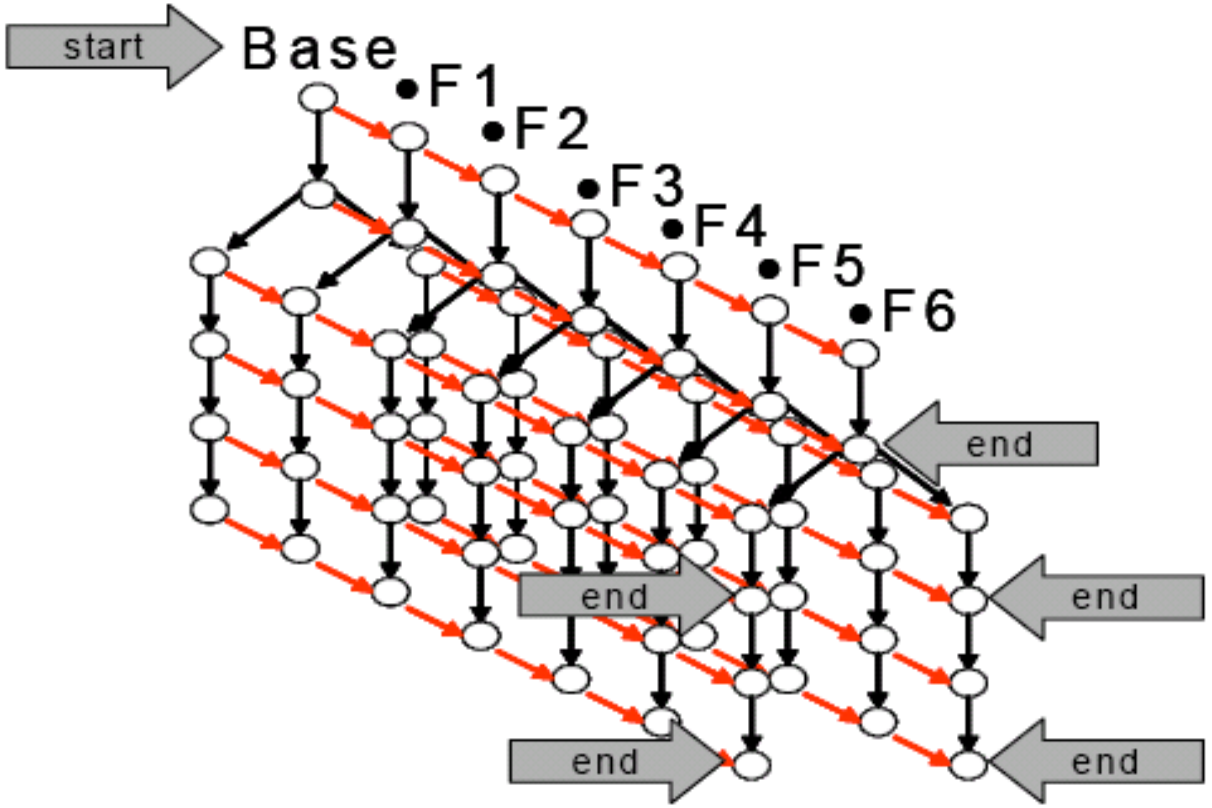
MDE + FOP



- model transformation (MDE) → model/class refinement (FOP)

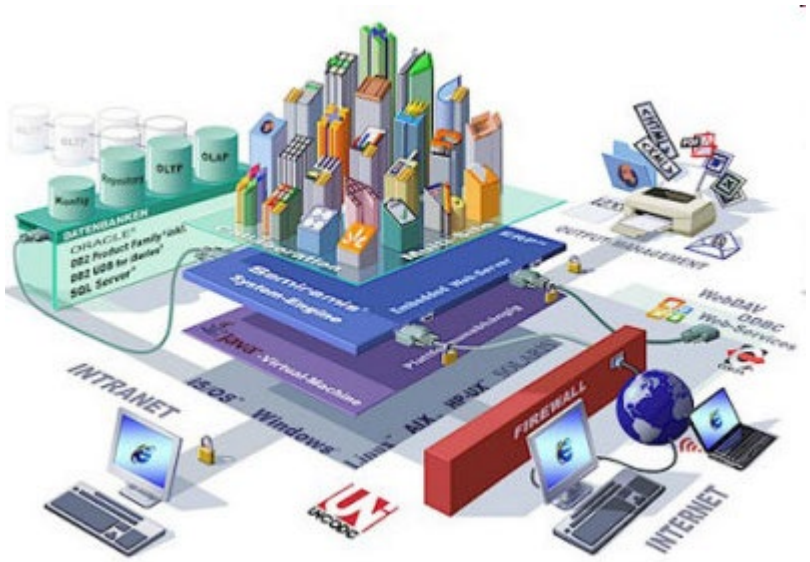
Feature-Oriented Model-Driven Development





Software architecture

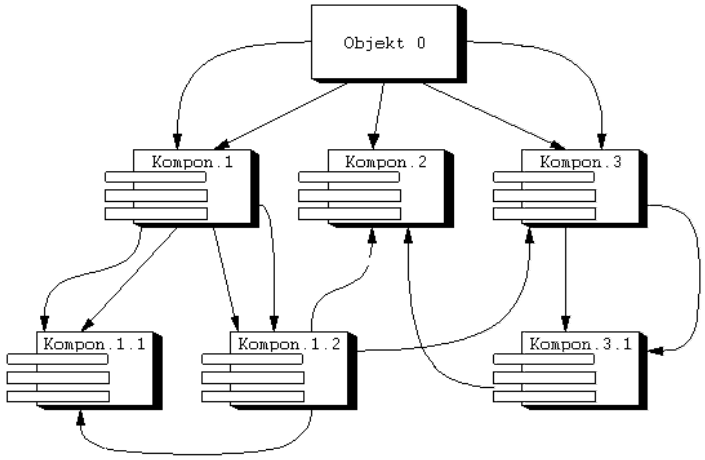
- Coarse-grained design of software system, focus on the main components while abstracting from fine-grained design & code
  - Components, connectors, UML deployment diagrams, etc.



The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them.

Software architecture + FOP

- FOP is a way of structuring software architectures
  - Feature module ⇔ Component
  - Glue module ⇔ Connector (connectors to components)



Summary

- Feature-orientation is a holistic approach to software engineering
- Kinship and overlaps to various trends in research and industry
- Good starting point for delving into fields “Software Engineering” and “Programming languages”, both in academic and industrial contexts

SPL analysis

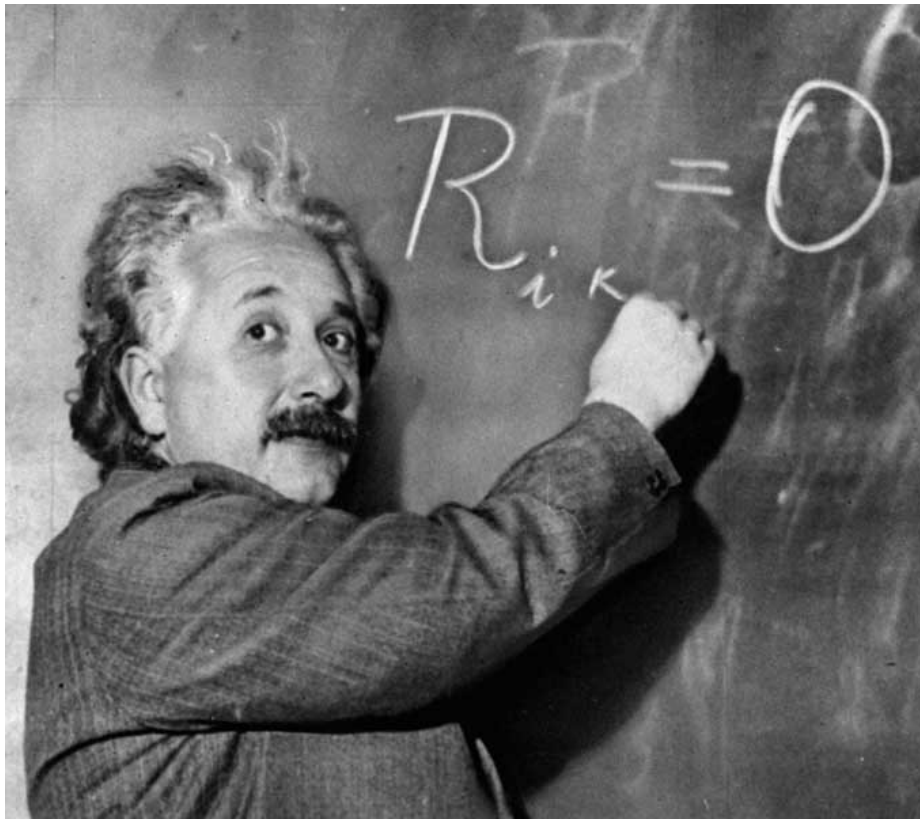
- Metrics, structure analysis
- Finding bugs, static analysis
- Testing
- Model checking, theorem proving
- Detect interactions
- Nonfunctional properties
- Cost-benefit models

Tools and languages

- Analysis tools
- Compiler, transformations
- Version control + configuration management
- Based on existing tools
  - FeatureIDE
  - FeatureHouse
  - Fuji
  - CIDE
  - SPLverifier
  - FeatureVisu

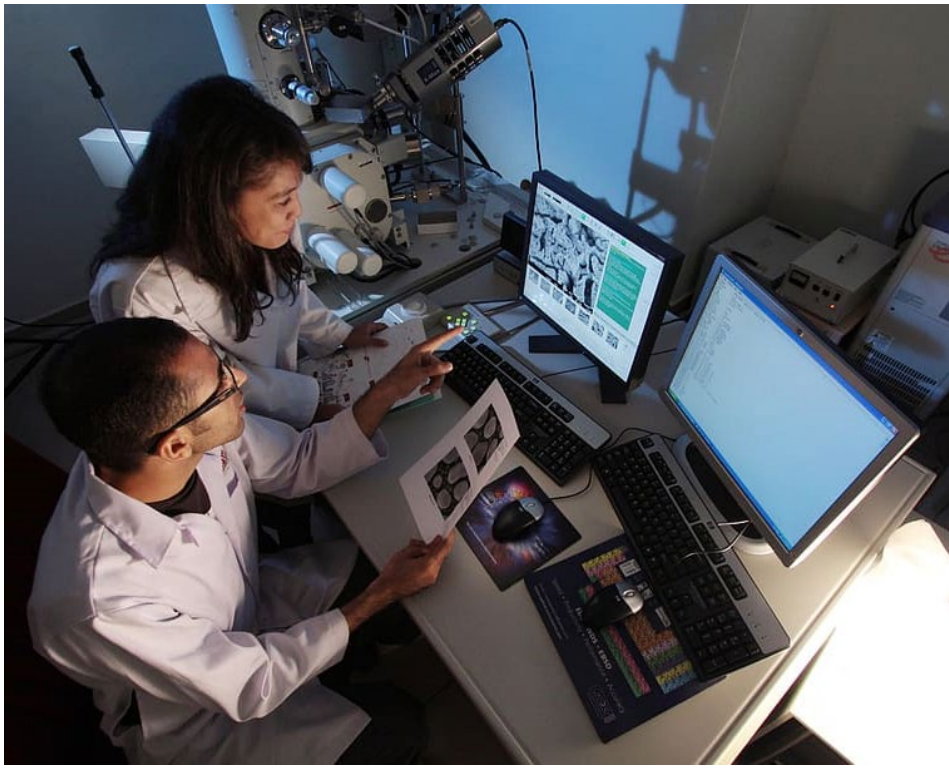
Formal models

- Modeling code aspects of SPLs
  - Structure, interactions
  - Variability
  - Behavior
  - Cost-benefit-ratio
- Building on previous work
  - Choice Calculus
  - Feature Featherweight Java
  - Feature Algebra
  - Program Cubes



Empirical studies

- Benchmarking
  - SPL analysis
  - Tools and languages
- Research artifacts
  - Synthetic data generation
  - Artifact quality



Literature

- David Benavides, Sergio Segura, Antonio Ruiz Cortés: Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35(6): 615-636 (2010)

  [Overview of feature model analyses]
- Christian Kästner and Sven Apel. Feature-Oriented Software Development. In GTTSE IV, volume 7680 of LNCS, pages 346–382. Springer-Verlag, January 2013.

  [Introduction to variability-aware analysis]
- Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. ACM Computing Surveys, 2014.

  [Overview of variability-aware analysis techniques]

Literature I (software product lines in general)

- Paul Clements, Linda Northrop. Software Product Lines – Practice and Pattern. Addison-Wesley. 2002
- Klaus Pohl, Günter Böckle, Frank van der Linden. Software Product Line Engineering. Foundations, Principles, and Techniques. Springer. 2005.

Literature II

- K. Czarnecki and U. Eisenecker. Generative Programming - Methods, Tools, and Applications. Addison-Wesley, 2000. [Introduction to GP]
- M. Mernik, J. Heering, and A. Sloane. When and How to Develop Domain-Specific Languages. ACM Computing Surveys, 37(4), 2005. [Introduction to DSL development]

Literature III

- L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice, Second Edition. Addison-Wesley, 2003
  [Introduction to software architecture]
- D. Schmidt. Model-Driven Engineering. IEEE Computer, 39(2), 2006.
  [Introduction to model-driven engineering]
- T. Erl. Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall, 2005.
  [Introduction to SOA]