10. Product-Line Analyses – Handout

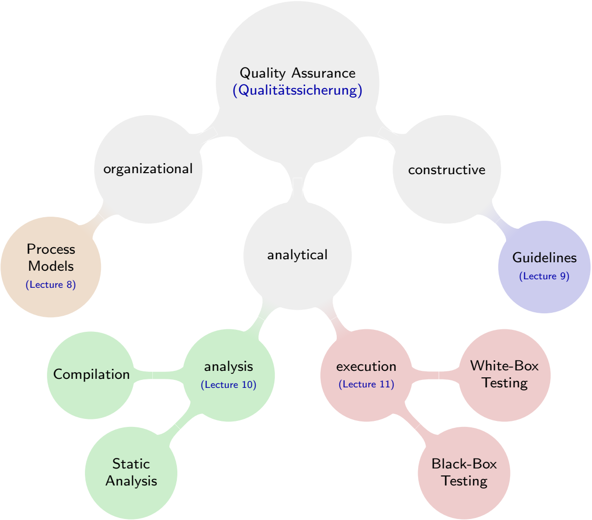
Software Product Lines | Elias Kuiter, Thomas Thüm, Timo Kehrer | June 9, 2023

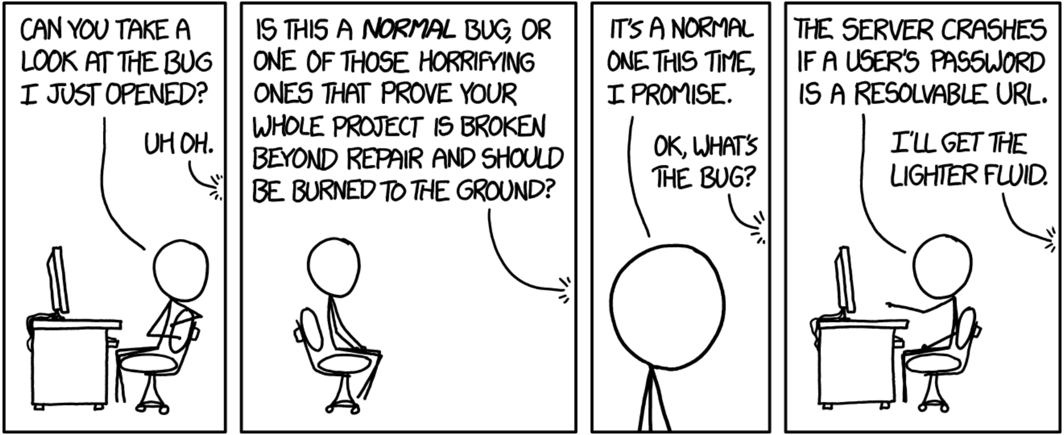
universität ulm · UNIVERSITÄT BERN · OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

10. Product-Line Analyses

Recap: Quality Assurance [Ludewig and Lichter 2013]

- last lecture:
  how to **avoid** variability bugs (esp. feature interactions)
- this + next lecture:
  how to **find** variability bugs




CAN YOU TAKE A LOOK AT THE BUG I JUST OPENED?
UH OH.
IS THIS A NORMAL BUG, OR ONE OF THOSE HORRIFYING ONES THAT PROVE YOUR WHOLE PROJECT IS BROKEN BEYOND REPAIR AND SHOULD BE BURNED TO THE GROUND?
IT'S A NORMAL ONE THIS TIME, I PROMISE.
OK, WHAT'S THE BUG?
THE SERVER CRASHES IF A USER'S PASSWORD IS A RESOLVABLE URL.
I'LL GET THE LIGHTER FLUID.

Automated Analysis of Product Lines

Typical Program Analyses

- code metrics
- type checking
- theorem proving
- data-flow analysis
- performance analysis
- ...

What is a Program Analysis?

- analyzes properties of a **program** (e.g., correctness, performance, and safety)
- can be used to automatically find bugs, bottlenecks, and other **vulnerabilities**

Asking Questions About Product Lines

- Which product has the most lines of code? [ref]
- Which products have type errors? [ref]
- Which products violate specifications? [ref]
- Which products have unsafe data flows? [ref]
- Which is the fastest product? [ref]
  Which product has the smallest binary? [ref]
- ...

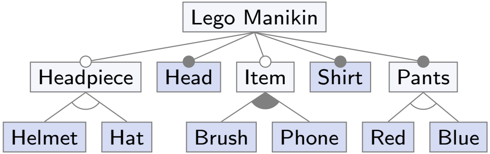
What is a Product-Line Analysis?

- analyzes properties of an entire **product line**
- can be roughly classified by its **strategy**:
  - product-based
  - feature-based
  - family-based

Product-Based Strategies

Intuition

- to analyze the product line, just analyze **each product**
  - individually
  - in isolation
  - possibly in parallel
- e.g., compile and verify each product



$Helmet \rightarrow \neg Phone$

Product-Based Strategies

Algorithm

Require: a product line pl; algorithms γ, α, σ

$C \leftarrow AllSAT(\phi(FM_{pl}))$ ▷ enumerate valid config's

$results \leftarrow []$

for all $S \in C$ **do** ▷ for each valid config

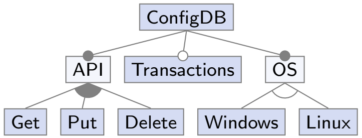
  $p \leftarrow \gamma(S)$ ▷ generate product

  $results \mathrel{+}= \alpha(p)$ ▷ add analysis result

end for

return $\sigma(results)$

- γ **generates** (e.g., compiles) products (e.g., `make`, `gradle`, `FeatureHouse`, `npm`, …)
- α **analyzes** the product (e.g., run verifier)
- σ **summarizes** the results (e.g., each individual call to α must succeed)



$Transactions \rightarrow Put \vee Delete$

$$
\begin{array}{ll}
\sigma([\alpha(\gamma(\{C, G, W\})) & \alpha(\gamma(\{C, G, L\})) \\
\quad \alpha(\gamma(\{C, P, W\})) & \alpha(\gamma(\{C, P, L\})) \\
\quad \alpha(\gamma(\{C, G, P, W\})) & \alpha(\gamma(\{C, G, P, L\})) \\
\quad \alpha(\gamma(\{C, D, W\})) & \alpha(\gamma(\{C, D, L\})) \\
\quad \alpha(\gamma(\{C, G, D, W\})) & \alpha(\gamma(\{C, G, D, L\})) \\
\quad \alpha(\gamma(\{C, P, D, W\})) & \alpha(\gamma(\{C, P, D, L\})) \\
\quad \alpha(\gamma(\{C, G, P, D, W\})) & \alpha(\gamma(\{C, G, P, D, L\})) \\
\quad \alpha(\gamma(\{C, P, T, W\})) & \alpha(\gamma(\{C, P, T, L\})) \\
\quad \alpha(\gamma(\{C, G, P, T, W\})) & \alpha(\gamma(\{C, G, P, T, L\})) \\
\quad \alpha(\gamma(\{C, D, T, W\})) & \alpha(\gamma(\{C, D, T, L\})) \\
\quad \alpha(\gamma(\{C, G, D, T, W\})) & \alpha(\gamma(\{C, G, D, T, L\})) \\
\quad \alpha(\gamma(\{C, P, D, T, W\})) & \alpha(\gamma(\{C, P, D, T, L\})) \\
\quad \alpha(\gamma(\{C, G, P, D, T, W\})) & \alpha(\gamma(\{C, G, P, D, T, L\}))])
\end{array}
$$

Classification of Strategies

Product-Based Strategy

- analyze individual **products**

\+ sound, complete

\+ uses off-the-shelf generator γ and analysis α

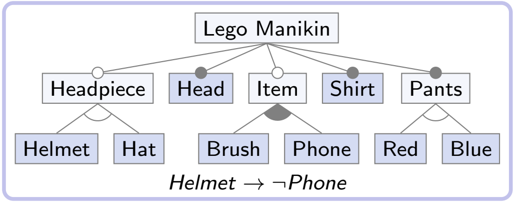
– redundant effort

– does not scale well

Feature-Based Strategies

Intuition

- to analyze the product line, just analyze **each feature** individually
- ignore all relations to other features
- e.g., compile and verify each component
  $\Rightarrow$ requires **interfaces between features** (components, services, plug-ins)

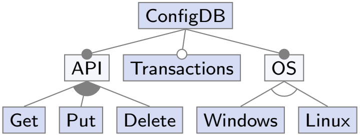


Feature-Based Strategies

Algorithm

Require: a product line *pl*; algorithms α, σ
  results $\leftarrow$ []
  for all $f \in F_{pl}$ **do** ▷ for each feature
    results $\mathrel{+}= \alpha(f)$ ▷ add analysis result
  end for
  return $\sigma(results)$

- α **analyzes** the feature (e.g., compiles and verifies the component)
- σ **summarizes** the results (see product-based)



$Transactions \rightarrow Put \vee Delete$

$\sigma([\alpha(C)$ – e.g., compile and verify base code
$\alpha(G)$ – e.g., compile and verify feature Get
$\alpha(P)$ – ...
$\alpha(D)$
$\alpha(T)$
$\alpha(W)$
$\alpha(L)])$

Classification of Strategies

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 Strategies

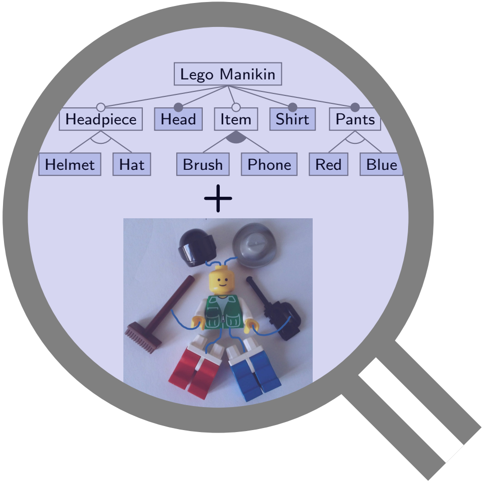
Intuition

- analyze the product line (or **family**) as a whole
- requirement: the analysis should give the same result as a product-based analysis
- makes use of the feature model and artifacts
- analysis is **hand-crafted**, no generic algorithm
  ⇒ typically: reduction to SAT problems

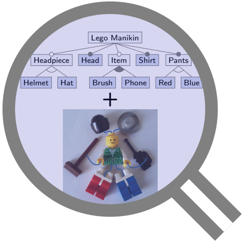
Today's Examples

- analyzing **feature mappings**
- analyzing **variable code**

⇒ here: for **conditional compilation** and **feature-oriented programming**



Classification of Strategies



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

Analysis Strategies – Summary

Lessons Learned

- product-line analyses are needed for quality assurance
- **product-based**: simple, but does not scale
- **feature-based**: fairly simple, but misses interactions
- **family-based**: efficient, but most complex

Further Reading

- Apel et al. 2013, Chapter 10
- Thüm et al. 2014

Practice

Can you imagine other analysis strategies than product-based, feature-based, and family-based? How could such strategies look like?

10. Product-Line Analyses

Automated Analysis of Feature Mappings

Recap: A Typical Product Line

- embedded or systems programming (e.g., Linux)
- implemented with conditional compilation
  - build systems (e.g., KBuild)
  - preprocessors (e.g., CPP)
- feature traceability only implicit
  $\Rightarrow$ there is code scattering and tangling

Recap: Feature Mapping

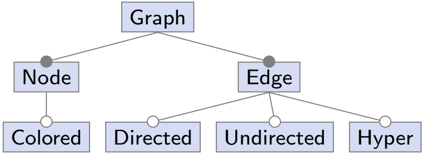
- specifies which features correspond to which artifacts (individual files/lines, components/feature modules/aspects)
- connects the problem space to the solution space

Asking Questions About the Feature Mapping

- Is the code even included in any product?
- Are there contradictory or unnecessary preprocessor annotations in the code?
- How scattered and tangled is the code?
- ...

Automated Analysis of Feature Mappings

Running Example: Graph Product Line

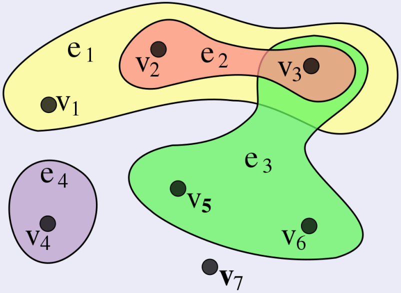


$\neg(\mathit{Directed} \land \mathit{Undirected})$

$\mathit{Hyper} \rightarrow \mathit{Undirected}$

$\mathit{Directed} \lor \mathit{Hyper}$

An Undirected Hypergraph



Presence Conditions

Presence Condition

A **presence condition (PC)** for a code location (i.e., a line or file) is a formula that describes the circumstances under which the code location is included in a product.

- useful for implementation techniques with code scattering and tangling
- e.g., build systems (file PCs) or preprocessors (line PCs)
- here: line PCs for the C preprocessor

Presence Conditions

$\top$
$\top$
Colored
Colored
Colored
$\top$
$\top$
$\top$
Directed
Directed
$\neg Dir \wedge Hyper$
$\neg Dir \wedge Hy \wedge Un$
$\neg Dir \wedge Hy \wedge Un$
$\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$
$\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$
$\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$
$\neg Dir \wedge \neg Hy$
$\neg Dir \wedge \neg Hy \wedge \neg Dir$
$\neg Dir \wedge \neg Hy \wedge \neg Dir$
$\neg Dir \wedge \neg Hy \wedge \neg Dir$
$\neg Dir \wedge \neg Hy$
$\top$

graph.cpp

```
class Node {
  string label;
#ifdef COLORED
  string color;
#endif
};

class Edge {
#ifdef DIRECTED
  Node fromNode, toNode;
#elifdef HYPER
#ifdef UNDIRECTED
  set<Node> nodeSet;
#elifdef DIRECTED
  map<Node, set<Node>> nodeMap;
#endif
#else
#ifndef DIRECTED
  pair<Node, Node> nodePair;
#endif
#endif
};
```

Detecting Dead Code

Dead Code

A line or file of code is **dead** when

- no product includes it.
- or, equivalently:
  its presence condition PC is contradictory (i.e., $PC \Rightarrow \bot$).

calculated by querying a **satisfiability solver** whether PC is not satisfiable (i.e., $\neg SAT(PC)$)

What causes dead code?

- confusion due to nested `#ifdef`
- domain modeling mistakes
- can be intended! [Hentze et al. 2021]

Presence Conditions

$\top$
$\top$
Colored
Colored
Colored
$\top$
$\top$
$\top$
Directed
Directed
$\neg Dir \land Hyper$
$\neg Dir \land Hy \land Un$
$\neg Dir \land Hy \land Un$
$\neg Dir \land Hy \land \neg Un \land Dir$
$\neg Dir \land Hy \land \neg Un \land Dir$
$\neg Dir \land Hy \land \neg Un \land Dir$
$\neg Dir \land \neg Hy$
$\neg Dir \land \neg Hy \land \neg Dir$
$\neg Dir \land \neg Hy \land \neg Dir$
$\neg Dir \land \neg Hy \land \neg Dir$
$\neg Dir \land \neg Hy$
$\top$

graph.cpp

```
class Node {
  string label;
#ifdef COLORED
  string color;
#endif
};

class Edge {
#ifdef DIRECTED
  Node fromNode, toNode;
#elifdef HYPER
#ifdef UNDIRECTED
  set<Node> nodeSet;
#elifdef DIRECTED
  map<Node, set<Node>> nodeMap;
#endif
#else
#ifndef DIRECTED
  pair<Node, Node> nodePair;
#endif
#endif
};
```

Detecting Superfluous Annotations

Superfluous Annotation

An annotation is **superfluous**

- when it can be omitted without consequences.
- or, equivalently:
  its presence condition PC is implied by the enclosing presence condition PC' (i.e., $PC' \Rightarrow PC$).

calculated by querying a **satisfiability solver** whether $PC' \land \neg PC$ is not satisfiable (i.e., $\neg SAT(PC' \land \neg PC)$)

- $PC' = \neg Dir \land \neg Hy$
- $PC = \neg Dir \land \neg Hy \land \neg Dir$

Presence Conditions

$\top$
$\top$
$Colored$
$Colored$
$Colored$
$\top$
$\top$
$\top$
$Directed$
$Directed$
$\neg Dir \land Hyper$
$\neg Dir \land Hy \land Un$
$\neg Dir \land Hy \land Un$
$\neg Dir \land Hy \land \neg Un \land Dir$
$\neg Dir \land Hy \land \neg Un \land Dir$
$\neg Dir \land Hy \land \neg Un \land Dir$
$\neg Dir \land \neg Hy$
$\neg Dir \land \neg Hy \land \neg Dir$
$\neg Dir \land \neg Hy \land \neg Dir$
$\neg Dir \land \neg Hy \land \neg Dir$
$\neg Dir \land \neg Hy$
$\top$

graph.cpp

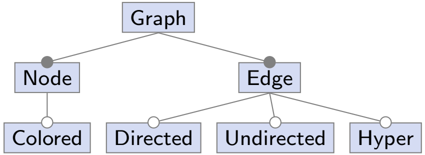
```
class Node {
  string label;
#ifdef COLORED
  string color;
#endif
};

class Edge {
#ifdef DIRECTED
  Node fromNode, toNode;
#elifdef HYPER
#ifdef UNDIRECTED
  set<Node> nodeSet;
#elifdef DIRECTED
  map<Node, set<Node>> nodeMap;
#endif
#else
#ifndef DIRECTED
  pair<Node, Node> nodePair;
#endif
#endif
};
```

Joining the Problem and Solution Space

- right now, we only consider **line PCs** (from the preprocessor)
- but: a line is only included if its file is included, too
  ⇒ we also have to consider **file PCs** (from the build system)
- also: we want to ignore invalid configurations
  ⇒ we also have to consider the **feature model** FM
- idea: **join** feature model, file, and line presence condition:
  $PC_{location} := \Phi(FM) \wedge PC_{file} \wedge PC_{line}$

Suppose we have the feature model ...



$\neg(Directed \wedge Undirected) \wedge (Hyper \rightarrow Undirected) \wedge (Directed \vee Hyper)$

... and two files: `node.cpp` ...

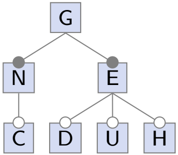
```
class Node {
  string label;
#ifdef COLORED
  string color;
#endif
};
```

... and `edge.cpp`

```
class Edge {
#ifdef DIRECTED
  Node fromNode, toNode;
#elifdef HYPER
#ifdef UNDIRECTED
  set<Node> nodeSet;
#elifdef DIRECTED
  map<Node, set<Node>> nodeMap;
#endif
#else
#ifndef DIRECTED
  pair<Node, Node> nodePair;
#endif
#endif
};
```

Joining the Problem and Solution Space

Problem Space



$\neg(Directed \wedge Undirected)$

$Hyper \rightarrow Undirected$

$Directed \vee Hyper$

$\downarrow \Phi$

$$Graph \wedge Node \wedge Edge$$
$$\wedge \neg(Directed \wedge Undirected)$$
$$\wedge (Hyper \rightarrow Undirected)$$
$$\wedge (Directed \vee Hyper)$$

Solution Space $\rightarrow$

File PC `node.cpp`

Node

Line PCs `node.cpp`

$\top$
$\top$
Colored
Colored
Colored
$\top$

`node.cpp`

```
class Node {
  string label;
#ifdef COLORED
  string color;
#endif
};
```

File PC `edge.cpp`

Edge

Line PCs `edge.cpp`

$\top$
Directed
Directed
$\neg Dir \wedge Hyper$
$\neg Dir \wedge Hy \wedge Un$
$\neg Dir \wedge Hy \wedge Un$
$\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$
$\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$
$\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$
$\neg Dir \wedge \neg Hy$
$\neg Dir \wedge \neg Hy \wedge \neg Dir$
$\neg Dir \wedge \neg Hy \wedge \neg Dir$
$\neg Dir \wedge \neg Hy \wedge \neg Dir$
$\neg Dir \wedge \neg Hy$
$\top$

`edge.cpp`

```
class Edge {
#ifdef DIRECTED
  Node fromNode, toNode;
#elifdef HYPER
#ifdef UNDIRECTED
  set<Node> nodeSet;
#elifdef DIRECTED
  map<Node, set<Node>> nodeMap;
#endif
#else
#ifndef DIRECTED
  pair<Node, Node> nodePair;
#endif
#endif
};
```

Joining the Problem and Solution Space

Feature-Model Formula

$Graph \land Node \land Edge$
$\land \neg(Directed \land Undirected)$
$\land (Hyper \rightarrow Undirected)$
$\land (Directed \lor Hyper)$

File PC `edge.cpp`

Edge

Line PCs `edge.cpp`

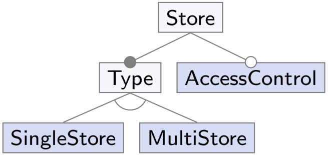
$\top$
$Directed$
$Directed$
$\neg Dir \land Hyper$
$\neg Dir \land Hy \land Un$
$\neg Dir \land Hy \land Un$
$\neg Dir \land Hy \land \neg Un \land Dir$
$\neg Dir \land Hy \land \neg Un \land Dir$
$\neg Dir \land Hy \land \neg Un \land Dir$
$\neg Dir \land \neg Hy$
$\neg Dir \land \neg Hy \land \neg Dir$
$\neg Dir \land \neg Hy \land \neg Dir$
$\neg Dir \land \neg Hy \land \neg Dir$
$\neg Dir \land \neg Hy$
$\top$

`edge.cpp`

```
class Edge {
#ifdef DIRECTED
  Node fromNode, toNode;
#elifdef HYPER
#ifdef UNDIRECTED
  set<Node> nodeSet;
#elifdef DIRECTED
  map<Node, set<Node>> nodeMap;
#endif
#else
#ifndef DIRECTED
  pair<Node, Node> nodePair;
#endif
#endif
};
```

$$
\begin{aligned}
PC_{location} &:= \Phi(FM) \land PC_{\texttt{edge.cpp}} \land PC_{\texttt{pair<Node, Node> nodePair;}} \\
&= G \land N \land E \land \neg(D \land U) \land (H \rightarrow U) \land (D \lor H) \land E \land \neg D \land \neg H \land \neg D \\
&\Leftrightarrow G \land N \land E \land \neg(D \land U) \land (H \rightarrow U) \land (D \lor H) \land E \land \neg D \land \neg H \land \neg D \\
&\Rightarrow (D \lor H) \land \neg D \land \neg H \\
&\Rightarrow \bot \text{ -- so this code is dead after all!}
\end{aligned}
$$

Analyzing Feature Modules



Feature-Model Formula

$\Phi(FM) = Store \wedge Type \wedge (SS \vee MS) \wedge (\neg SS \vee \neg MS)$

Valid Configurations

$\{SS\}$	$\{MS\}$
$\{SS, AC\}$	$\{MS, AC\}$

Feature module *SingleStore*

```
class Store {
  private Object value;
  Object read() { return value; }
  void set(Object nvalue) { value = nvalue; }
}
```

Feature module *MultiStore*

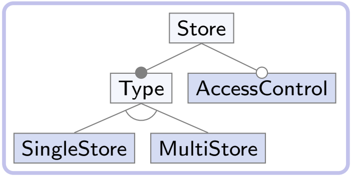
```
class Store {
  private LinkedList values = new LinkedList();
  Object read() { return values.getFirst(); }
  Object[] readAll() { return values.toArray(); }
  void set(Object nvalue) { values.addFirst(nvalue); }
}
```

Feature module *AccessControl*

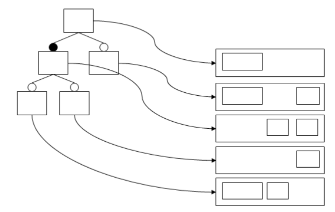
```
refines class Store {
  private boolean sealed = false;
  Object read() {
    if (!sealed) { return Super.read(); }
    else { throw new RuntimeException("Access␣denied!"); }
  }
  void set(Object nvalue) {
    if (!sealed) { Super.set(nvalue); }
    else { throw new RuntimeException("Access␣denied!"); }
  }
}
```

Is there dead code? Are there superfluous annotations?

Analyzing Feature Modules



Recap: 1:1 Feature Mapping!



Feature module *SingleStore*

```
class Store {
  private Object value;
  Object read() { return value; }
  void set(Object nvalue) { value = nvalue; }
}
```

Feature module *MultiStore*

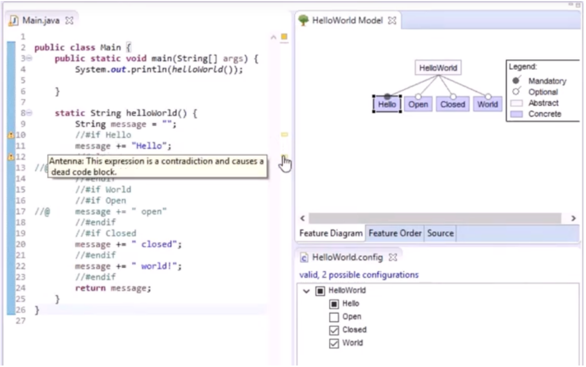
```
class Store {
  private LinkedList values = new LinkedList();
  Object read() { return values.getFirst(); }
  Object[] readAll() { return values.toArray(); }
  void set(Object nvalue) { values.addFirst(nvalue); }
}
```

Feature module *AccessControl*

```
refines class Store {
  private boolean sealed = false;
  Object read() {
    if (!sealed) { return Super.read(); }
    else { throw new RuntimeException("Access␣denied!"); }
  }
  void set(Object nvalue) {
    if (!sealed) { Super.set(nvalue); }
    else { throw new RuntimeException("Access␣denied!"); }
  }
}
```

~~Is~~ Are there ~~dead code~~ dead features? Are there ~~superfluous annotations~~ redundant constraints?

Feature-Mapping Analyses in FeatureIDE



demo video available (minute 3 and 4): dead code block, superfluous annotations, generation of all products, error propagation, unit testing

Discussion

- we can now **identify anomalies**:
  - dead (unused) code
  - mistakes in preprocessor annotations
  - disagreements between problem and solution space
- but: we only analyze the feature mapping and **ignore the actual code**
  - pro: simple, language-independent
  - con: can only find simple anomalies
- difficulty depends on the feature traceability (harder for conditional compilation than for FOP)

Analyzing Feature Mappings – Summary

Lessons Learned

- feature-mapping analyses alleviate the impact of code scattering and tangling
- they are usually not necessary when there is good feature traceability
- they cannot detect bugs in the actual code

Further Reading

- Apel et al. 2013, Chapter 10

Practice

Above, we assumed that we know all presence conditions already. How can we automatically extract presence conditions from code that uses the C preprocessor? What problems might occur?

10. Product-Line Analyses

Automated Analysis of Variable Code

Asking Questions About the Feature Mapping . . .

- Are there contradictory or unnecessary preprocessor annotations in the code?
- Is the code even included in any product?
- If so, in how many products is the code included?
- . . .

only finds code-agnostic anomalies

. . . and the Variable Code

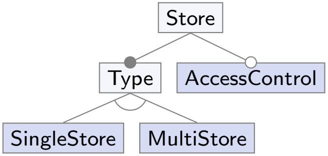
- Can every product be generated (e.g., compiled)?
  $\Rightarrow$ to find all **syntax and type errors**
- Do all tests succeed for every product?
  $\Rightarrow$ to find some **runtime and logic errors**
- Does every product adhere to its specification?
  $\Rightarrow$ to **rule out** runtime and logic errors
- . . .

now: analyze (non-)functional properties of all products

Today's Example

type checking for FOP and conditional compilation

Variability-Aware Type Checking – Analyzing Feature Modules



Feature-Model Formula

$\Phi(FM) = Store \wedge Type \wedge (SS \vee MS) \wedge (\neg SS \vee \neg MS)$

Valid Configurations

$\{SS\}$ $\{MS\}$
$\{SS, AC\}$ $\{MS, AC\}$

Is there a type error in **any** product?
What about $\{SS, AC\}$?

Feature module *SingleStore*

```
class Store {
  private Object value;
  Object read() { return value; }
  void set(Object nvalue) { value = nvalue; }
}
```

Feature module *MultiStore*

```
class Store {
  private LinkedList values = new LinkedList();
  Object read() { return values.getFirst(); }
  Object[] readAll() { return values.toArray(); }
  void set(Object nvalue) { values.addFirst(nvalue); }
}
```

Feature module *AccessControl*

```
refines class Store {
  private boolean sealed = false;
  Object read() {
    if (!sealed) { return Super.read(); }
    else { throw new RuntimeException("Access␣denied!"); }
  }
  Object[] readAll() {
    if (!sealed) { return Super.readAll(); }
    else { throw new RuntimeException("Access␣denied!"); }
  }
  void set(Object nvalue) {
    if (!sealed) { Super.set(nvalue); }
    else { throw new RuntimeException("Access␣denied!"); }
  }
}
```

Variability-Aware Type Checking – Analyzing Feature Modules

Reachability Condition of *id*

guarantees that a given **reference** to *id* is also **defined** somewhere:

$\Phi(FM) \Rightarrow (PC^{id}_{ref} \rightarrow \bigvee_{def} PC^{id}_{def})$

or, with a SAT solver:
$\neg SAT(\Phi(FM) \wedge PC_{ref} \wedge \bigwedge_{def} \neg PC_{def})$

$\Phi(FM) \Rightarrow (AC \rightarrow SS \vee MS)$ holds,
$\Phi(FM) \Rightarrow (AC \rightarrow MS)$ does not

$\Rightarrow \{SS, AC\}$ has no `readAll`!

Type-Safe Product-Line

in a **type-safe** SPL, all references must always be defined (i.e., **all** reachability conditions must hold)

… and many more conditions …

Feature module *SingleStore*

```
class Store {
  private Object value;
  Object read() { return value; }
  void set(Object nvalue) { value = nvalue; }
}
```

$VM \Rightarrow (AccessControl \Rightarrow SingleStore \vee MultiStore)$

Feature module *MultiStore*

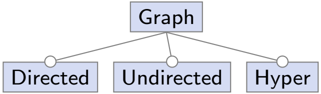
```
class Store {
  private LinkedList values = new LinkedList();
  Object read() { return values.getFirst(); }
  Object[] readAll() { return values.toArray(); }
  void set(Object nvalue) { values.addFirst(nvalue); }
}
```

Feature module *AccessControl*

```
refines class Store {
  private boolean sealed = false;
  Object read() {
    if (!sealed) { return Super.read(); }
    else { throw new RuntimeException("Access denied!"); }
  }
  Object[] readAll() {
    if (!sealed) { return Super.readAll(); }
    else { throw new RuntimeException("Access denied!"); }
  }
  void set(Object nvalue) {
    if (!sealed) { Super.set(nvalue); }
    else { throw new RuntimeException("Access denied!"); }
  }
}
```

$VM \Rightarrow (AccessControl \Rightarrow MultiStore)$

Variability-Aware Type Checking – Analyzing Conditional Compilation



$\neg(\textit{Directed} \wedge \textit{Undirected})$
$\textit{Hyper} \rightarrow \textit{Undirected}$
$\textit{Directed} \vee \textit{Hyper}$

Reachability Condition of *id*

$$\Phi(FM) \Rightarrow (PC_{ref}^{id} \rightarrow \bigvee_{def} PC_{def}^{id})$$

Conflict Condition of *id*, def's d_i

guarantees that no definition of *id* **conflicts** with another:

$$\Phi(FM) \Rightarrow \bigwedge_{d_1 \neq d_2} \neg(PC_{d_1}^{id} \wedge PC_{d_2}^{id})$$

Is *e.nodes* reachable?

$$\Phi(FM) \Rightarrow (\top \rightarrow Dir \vee (Hy \wedge Un) \vee (Hy \wedge Dir))$$

holds, because each graph is directed or an (undirected) hypergraph

Does *e.nodes* conflict?

$$\Phi(FM) \Rightarrow (\neg(Dir \wedge (Hy \wedge Un)) \wedge \neg(Dir \wedge (Hy \wedge Dir)) \wedge \neg((Hy \wedge Un) \wedge (Hy \wedge Dir)))$$

holds, because a graph is never directed and an (undirected) hypergraph at the same time

all reachable, no conflicts

graph.cpp

```
class Node { ... };

class Edge {
#ifdef DIRECTED
  pair<Node, Node> nodes;
#endif
#ifdef HYPER
#ifdef UNDIRECTED
  set<Node> nodes;
#endif
#ifdef DIRECTED
  map<Node, set<Node>> nodes;
#endif
#endif
};

std::ostream& operator<<(
  std::ostream &s, const Edge &e) {
  return s << e.nodes;
}
```

Variability-Aware Type Checking – Discussion

Just the Tip of the Iceberg

- here, we only discussed reachability and conflict conditions
- but: actual type checking requires a table of all identifiers, their types, and their PCs (and a lot more SAT queries)
- the practical difficulty depends:
  - FOP (due to superimposition)
    ⇒ no conflict conditions required
  - good feature traceability (e.g., FOP)
    ⇒ trivial PCs, simpler implementation
  - ignoring the feature model
    ⇒ better performance (false positives!)

The TypeChef Project

[Kästner et al. 2011]

- a variability-aware lexer, parser framework, and type system for C code with `#ifdef`'s
- skips preprocessing, instead builds an abstract syntax tree (AST) annotated with presence conditions
- poster with examples
- does it scale?

Busybox (811 features): "We need 57 minutes to type check all modules." [ref]

Linux (6065 features): "We successfully parsed [it in] roughly 85 hours on a single machine." [ref]

Product-Line Analyses in the Wild – Product-Line Complexity

Six Classes of Product-Line Complexity [Thüm 2021]

In a timeframe of 24h ...

- **NC** Products cannot be generated automatically
- **C1** All products can be generated and **tested**
- **C2** Not C1, but all **products** can be **generated**
- **C3** Not C2, but all **configurations** can be **generated** (AllSAT)
- **C4** Not C3, but the **number of valid configurations** can be computed (#SAT)
- **C5** Not C4, but **whether there is a valid configuration** can be computed (SAT)
- **C6** It cannot be computed whether there is a valid configuration

Examples

- **NC** all product lines with mandatory custom development in application engineering (e.g., components and services with glue code, white-box frameworks)
- **C1** $<$ 2000 products for 1 min per product
- **C2** $<$ 90000 products for 1 s per product
- **C3** $< 10^{13}$ configurations for 1 ns per configuration
- **C4** older versions of Linux/Automotive05
- **C5** newer versions of Linux/Automotive05 (see Sundermann et al. 2020)
- **C6** No example known

Product-Line Analyses in the Wild – Automated Analysis . . .

Lecture 4c

. . . of Feature Models

analyze only the feature model

- void, core/dead features
- decision propagation
- atomic sets, redundant constraints
- . . .

Lecture 10b

. . . of Feature Mappings

analyze the feature mapping (considering the feature model)

- dead code
- superfluous annotations
- degree of code scattering and tangling
- . . .

Lecture 10c

. . . of Variable Code

analyze the variable code (considering the feature model and feature mapping)

- parsing, type checking
- static analysis
- model checking, theorem proving
- . . .

here: **family-based** analysis strategies for **conditional compilation** and **feature-oriented programming**

Analyzing Variable Code – Summary

Lessons Learned

- with family-based analyses of variable code, we can analyze (non-)functional properties of all products at once
- type checking all products at once is possible for product lines up to medium size
- for huge product lines (e.g., Linux), it is infeasible

Further Reading

- Apel et al. 2013, Chapter 10
- Kästner et al. 2011

Practice

Suppose you have a preprocessor-based product line (with `#ifdef`'s). If you could turn it into a single, large runtime-variable product (with `if`'s), you could use an off-the-shelf compiler to find any type error in any product.

Is this possible? What problems might occur?

FAQ – 10. Product-Line Analyses

Lecture 10a

- How to find variability bugs?
- What is a program analysis? What are examples?
- What is a product-line analysis?
- What are principal strategies to analyze product lines? What are (dis-)advantages?
- Given a specific algorithm, classify its analysis strategy!

Lecture 10b

- How to analyze feature mappings?
- What are potential problems in feature mappings?
- What are presence conditions, dead code, superfluous annotations?
- Shall we incorporate the feature model when analyzing feature mappings?
- Shall product-line analyses analyze problem and solution space separately?
- What is special when analyzing the feature mapping of feature modules?
- What are limitations of analyzing feature mappings?
- Given CPP source code, determine its presence conditions, dead code, and superfluous annotations!

Lecture 10c

- What are (examples of) type errors?
- Why are type errors challenging to detect in product lines?
- What is a type-safe product line, reachability condition, conflict condition?
- How does the analysis complexity differ for real-world product lines?
- What are analyses for problem and solution space?
- Give examples for easy and difficult product lines in terms of analysis effort!