FeatureHouse: Language-Independent, Automated Software Composition

Sven Apel
Department of Informatics and Mathematics
University of Passau

Christian Kästner
School of Computer Science
University of Magdeburg

Christian Lengauer
Department of Informatics and Mathematics
University of Passau
Software Product Lines (SPLs)

- Set of related software products for one domain generated from common code base
- Products distinguished in terms of features
- What is a feature?
  - Product characteristic
  - Domain abstraction relevant to stakeholders

Product A
Product B
Product C
Problem: Lack of Separation of Concerns

Vision: Implement Features Modularly

Vision: Implement Features Modularly

A feature module is a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option.

— Apel et al.
Software Product Generation based on Features

feature composition
Software Product Generation based on Features
Software Product Generation based on Features

=  

=  

=  

feature composition
Feature (De)Composition

- Aggregation
- Generation
- Model transformation
- Class and plug-in loading
- Superimposition
- Aspect weaving
- ...
Feature (De)Composition

- Aggregation
- Generation
- Model transformation
- Class and plug-in loading
- Superimposition
- Aspect weaving
- ...

Slide - 10
Superimposition

- Informally,
  - the composition of two software artifacts (features),
  - by merging recursively the artifacts' structures,
  - based on nominal and structural similarity.
Superimposition

- Informally,
  - the composition of two software artifacts (features),
  - by merging recursively the artifacts' structures,
  - based on nominal and structural similarity.

- Think of software artifacts in terms of their hierarchical, modular structure, e.g.:

```java
package tools;
class Counter {
    int val = 0;
    void inc() { val++; }
    int get() { return val; }
}
```

software artifact
Superimposition

- Informally,
  - the composition of two software artifacts (features),
  - by merging recursively the artifacts' structures,
  - based on nominal and structural similarity.
- Think of software artifacts in terms of their hierarchical, modular structure, e.g.:

```java
package tools;

class Counter {
    int val = 0;
    void inc() { val++; }
    int get() { return val; }
}
```

software artifact hierarchical structure
**Superimposition**

- Informally,
  - the composition of two software artifacts (features),
  - by merging recursively the artifacts' structures,
  - based on nominal and structural similarity.

- Think of software artifacts in terms of their hierarchical, modular structure, e.g.:

```java
package tools;
class Counter {
    int val = 0;
    void inc() { val++; }
    int get() { return val; }
}
```

software artifact  hierarchical structure
Superimposition

- Informally,
  - the composition of two software artifacts (features),
  - by merging recursively the artifacts' structures,
  - based on nominal and structural similarity.
- Think of software artifacts in terms of their hierarchical, modular structure, e.g.:

```java
package tools;

class Counter {
    int val = 0;
    void inc() { val++; }  
    int get() { return val; }  
}
```

---

**Software artifact**

**Hierarchical structure**
Superimposition

- Informally,
  - the composition of two software artifacts (features),
  - by merging recursively the artifacts' structures,
  - based on nominal and structural similarity.
- Think of software artifacts in terms of their hierarchical, modular structure, e.g.:

```
package tools;
class Counter {
    int val = 0;
    void inc() { val++; }
    int get() { return val; }
}
```

software artifact hierarchical structure
Superimposition

- Informally,
  - the composition of two software artifacts (features),
  - by merging recursively the artifacts' structures,
  - based on nominal and structural similarity.
- Think of software artifacts in terms of their hierarchical, modular structure, e.g.:

```java
package tools;

class Counter {
    int val = 0;
    void inc() { val++; }
    int get() { return val; }
}
```

software artifact  hierarchical structure
Superimposition

Counter

```
package tools;
class Counter {
    int val = 0;
    void inc() { val++;
    }
    int get() { return val; }
}
```

Diagram:

```
     tools
    /   \
  Counter
   /     \
val  get
     / \
    inc
```
Superimposition

Counter

```java
tool package tools;
class Counter {
    int val = 0;
    void inc() { val++; }
    int get() { return val; }
}
```

Backup

```java
tool package tools;
class Counter {
    int back = 0;
    void inc() { back=val; original(); }
    void restore() { val=back; }
}
```
Superimposition

Counter

```java
package tools;
class Counter {
    int val = 0;
    void inc() { val++; }
    int get() { return val; }
}
```

Backup

```java
package tools;
class Counter {
    int back = 0;
    void inc() { back=val; original(); }
    void restore() { val=back; }
}
```

BackupCounter

```java
package tools;
class Counter {
    int val = 0;
    int back = 0;
    void inc() { back=val; val++; }
    int get() { return val; }
    void restore() { val=back; }
}
```
Languages, Tools, and Formal Systems

- **Languages**
  - Jak, Scala, CaesarJ, FeatureC++, Java Layers, Classbox/J, ObjectTeams/J, Lasagne/J

- **Tools**
  - Hyper/J, AHEAD Tool Suite, Jiauzzi, Xak

- **Formal Systems**
  - Jx, J& vc, vObj, Tribe, .FJ, FFJ, FLJ, Deep, gDeep
An Idea

- Capture the essential properties of superimposition in a model and composition tool
  - Language independence
  - General theory of software composition by superimposition
  - Integrate a language of your choice

- **Feature Structure Tree** Model

```
Base = tools : Package
      ⊕ tools.Counter : Class
      ⊕ tools.Counter.val : Field
      ⊕ tools.Counter.inc : Method
      ⊕ tools.Counter.get : Method
```

Feature Algebra (Apel et al., AMAST'08)
Non-Terminal vs. Terminal Nodes

- Non-terminal nodes
  - Identified by name* and type
  - Superimposition proceeds recursively with the children

- Terminal nodes
  - Identified by name* and type
  - Carry further language-specific content
  - Superimposition terminates with composing contents

* Names are mangled.
FSTComposer

Java

Parser

Feature Structure Trees

Pretty Printer

Java
FSTComposer

Java

Feature Structure Trees

Parser

Java

C#

Parser

C#

C

Parser

C

Haskell

Parser

Haskell

Pretty Printer

Pretty Printer

Pretty Printer

Pretty Printer

Feature Structure Trees
Problems

- Manual integration of Java, C#, UML, and Bali
  - Implementation of a parser and a pretty printer per language
    - Tedious (several weeks effort)
    - Error-prone (many bugs)
  - Composition of terminal nodes requires special language-dependent rules
    - $\text{method} \times \text{method} \rightarrow \text{method}$ (overriding)
    - $\text{constructor} \times \text{constructor} \rightarrow \text{constructor}$ (concatenation)
    - $\text{implements} \times \text{implements} \rightarrow \text{implements}$ (union)
    - $\text{extends} \times \text{extends} \rightarrow \text{extends}$ (replacement)
    - ...

- Benefit of language independence is almost lost
An Observation

- Code for supporting different languages is very similar
An Observation

- Code for supporting different languages is very similar

Parser
An Observation

- Code for supporting different languages is very similar

Which elements are represented by terminals and non-terminals?

Parser
An Observation

- Code for supporting different languages is very similar

Which elements are represented by terminals and non-terminals?
An Observation

- Code for supporting different languages is very similar

Which elements are represented by terminals and non-terminals?

How are terminals and non-terminals translated back to source code?
An Insight

- Non-terminals and terminals correspond to production rules in the artifact language's grammar

FST

Counter.java

Counter

val inc get
An Insight

- Non-terminals and terminals correspond to production rules in the artifact language's grammar

FST

Grammar

```
JavaFile : (ClassDecl)*;
ClassDecl : "class" Type "extends" ExtType "{" (VarDecl)* (ClassConstr)* (MethodDecl)* "}";
MethodDeclaration : Type <IDENTIFIER> "(" (Params)? ")" "{" "return" Expression ";" ";" "}";
VarDecl : Type <IDENTIFIER> ";";
```

Counter.java

Counter

val inc get
An Insight

- Non-terminals and terminals correspond to production rules in the artifact language's grammar

**FST**

```
Counter.java
  |___ Counter
    |___ val
    |___ inc
    |___ get
```

**Grammar**

```
JavaFile : (ClassDecl)*;
ClassDecl : "class" Type "extends" ExtType "{" (VarDecl)* (ClassConstr)* (MethodDecl)* "}";
MethodDeclaration : Type <IDENTIFIER> "(" (Params)? ")" "{" "return" Expression ";" "}";
VarDecl : Type <IDENTIFIER> ";";
```
An Insight

- Non-terminals and terminals correspond to production rules in the artifact language's grammar

**Grammar**

```
JavaFile : (ClassDecl)*;
ClassDecl : "class" Type "extends" ExtType "{" (VarDecl)* (ClassConstr)* (MethodDecl)* "}";
MethodDeclaration : Type <IDENTIFIER> "(" (Params)? ")" "{" "return" Expression ";" "}";
VarDecl : Type <IDENTIFIER> ";";
```
An Insight

- Non-terminals and terminals correspond to production rules in the artifact language's grammar

FST

Grammar

JavaFile : (ClassDecl)*;
ClassDecl : "class" Type "extends" ExtType "{" (VarDecl)* (ClassConstr)* (MethodDecl)* "}";
MethodDeclaration :
  Type <IDENTIFIER> "(" (Params)? ")" "{" "return" Expression ";" "}";
VarDecl : Type <IDENTIFIER> ";";

Counter.java

Counter

val inc get
An Insight

- Non-terminals and terminals correspond to production rules in the artifact language's grammar

**FST**
- Counter.java
  - Counter
    - val
    - inc
    - get

**Grammar**
- JavaFile : (ClassDecl)*;
- ClassDecl : "class" Type "extends" ExtType "{" (VarDecl)* (ClassConstr)* (MethodDecl)* "}";
- MethodDeclaration : Type <IDENTIFIER> "(" (Params)? ")" "{" "return" Expression ";" "}";
- VarDecl : Type <IDENTIFIER> ";" ;
An Idea

- Automate the integration of a new language on the basis of the language's grammar

- Use annotations/attributes to define…
  - which production rules map to non-terminals,
  - which production rules map to terminals, and
  - how the contents of terminals of a certain type are composed

- Generate a parser and pretty printer automatically on the basis of an annotated grammar
Annotating a Simplified Java Grammar

Grammar

JavaFile : (ClassDecl)*;

ClassDecl : "class" Type "extends" ExtType "{"
   (VarDecl)* (ClassConstr)* (MethodDecl)*
   "}";

MethodDeclaration :
    Type <IDENTIFIER> "(" (Params)? ")" "{"  
    "return" Expression "," 
    "}";

VarDecl : Type <IDENTIFIER> ",";
Annotating a Simplified Java Grammar

Grammar

JavaFile : (ClassDecl)*;

ClassDecl : "class" Type "extends" ExtType "{" (VarDecl)* (ClassConstr)* (MethodDecl)* "}";

MethodDeclaration :
    Type <IDENTIFIER> "(" (Params)? ")" "{" "return" Expression "," "}";

VarDecl : Type <IDENTIFIER> ";";

Example

```java
class Counter {
    int val = 0;
    void inc() { val++; }
    int get() { return val; }
}
```

Generated Parser

Counter.java
Annotating a Simplified Java Grammar

Grammar

@FSTNonTerminal()
JavaFile : (ClassDecl)*;

ClassDecl : "class" Type "extends" ExtType "{" (VarDecl)* (ClassConstr)* (MethodDecl)* "}";

MethodDeclaration :
    Type <IDENTIFIER> "(" (Params)? ")" "{" "return" Expression ";"," "}";

VarDecl : Type <IDENTIFIER> ";";

Example

class Counter {
    int val = 0;
    void inc() { val++; }
    int get() { return val; }
}

generated
Parser

Counter.java

Counter
Annotating a Simplified Java Grammar

**Grammar**

```plaintext
@FSTNonTerminal()
JavaFile : (ClassDecl)*;

@FSTNonTerminal(name="{Type}")
ClassDecl : "class" Type "extends" ExtType "{
   (VarDecl)* (ClassConstr)* (MethodDecl)*
   "}",&quot;

MethodDeclaration :
   Type <IDENTIFIER> "(" (Params)? ")" "{" "return" Expression "," "}",";

VarDecl : Type <IDENTIFIER> ",",";
```

**Example**

```plaintext
class Counter {
    int val = 0;
    void inc() { val++; }
    int get() { return val; }
}
```

**Generated Parser**

```
Counter.java

Counter

val inc get
```
Annotating a Simplified Java Grammar

Grammar

@FSTNonTerminal()
JavaFile : (ClassDecl)*;

@FSTNonTerminal(name="{Type}")
ClassDecl : "class" Type "extends" ExtType "{" (VarDecl)* (ClassConstr)* (MethodDecl)* "}";

@FSTTerminal(name="{<IDENTIFIER>}({Params})",
compose="MethodOverriding")
MethodDeclaration :
    Type <IDENTIFIER> "(" (Params)? ")" "{" "return" Expression ";"," "}";

@FSTTerminal(name="{<IDENTIFIER>}",
compose="FieldSpecialization")
VarDecl : Type <IDENTIFIER> ",";

Example

```java
class Counter {
    int val = 0;
    void inc() { val++; }
    int get() {
        return val;
    }
}
```

Generated Parser

Counter.java

Counter

val inc get

How to compose?
FeatureHouse: Language-Independent, Automated Software Composition

FeatureHouse

library of composition rules
## Integrating Languages

- Moderate effort for annotating grammars
- Only a few composition rules ➔ library and reuse

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C#</th>
<th>C</th>
<th>Haskell</th>
<th>JavaCC</th>
<th>XML</th>
</tr>
</thead>
<tbody>
<tr>
<td># rules</td>
<td>135</td>
<td>229</td>
<td>45</td>
<td>78</td>
<td>170</td>
<td>14</td>
</tr>
<tr>
<td># non-terminals</td>
<td>10</td>
<td>17</td>
<td>2</td>
<td>13</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td># terminals</td>
<td>13</td>
<td>18</td>
<td>9</td>
<td>9</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td># attributes</td>
<td>42</td>
<td>53</td>
<td>21</td>
<td>24</td>
<td>61</td>
<td>15</td>
</tr>
</tbody>
</table>
# Case Studies

<table>
<thead>
<tr>
<th>Features</th>
<th>LOC</th>
<th>Artifact Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFJ</td>
<td>2</td>
<td>JavaCC</td>
<td>Grammar of the FFJ language</td>
</tr>
<tr>
<td>Arith</td>
<td>27</td>
<td>Haskell</td>
<td>Arithmetic expression evaluator</td>
</tr>
<tr>
<td>GraphLib</td>
<td>13</td>
<td>C</td>
<td>Low level graph library</td>
</tr>
<tr>
<td>Phone</td>
<td>2</td>
<td>UML</td>
<td>Phone system</td>
</tr>
<tr>
<td>ACS</td>
<td>4</td>
<td>UML</td>
<td>Audio control system</td>
</tr>
<tr>
<td>CMS</td>
<td>10</td>
<td>UML</td>
<td>Conference management system</td>
</tr>
<tr>
<td>GPL (C#)</td>
<td>20</td>
<td>C#</td>
<td>Graph product line (C# version)</td>
</tr>
<tr>
<td>GBS</td>
<td>29</td>
<td>UML</td>
<td>Gas boiler control system (IKERLAN)</td>
</tr>
<tr>
<td>GPL (Java)</td>
<td>26</td>
<td>Java, XHTML</td>
<td>Graph product line (Java version)</td>
</tr>
<tr>
<td>FGL</td>
<td>20</td>
<td>Haskell</td>
<td>Functional graph library</td>
</tr>
<tr>
<td>Violet</td>
<td>88</td>
<td>Java, Text</td>
<td>Visual UML editor</td>
</tr>
<tr>
<td>GUIDSL</td>
<td>26</td>
<td>Java</td>
<td>Product line configuration tool</td>
</tr>
<tr>
<td>Berkeley DB</td>
<td>99</td>
<td>Java</td>
<td>Oracle's embedded DBMS</td>
</tr>
<tr>
<td></td>
<td>289</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>532</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>934</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.004</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.080</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.037</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.148</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.380</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.439</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.730</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9.660</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>13.457</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>84.030</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problems

- Lexical order:
Problems

- Lexical order:

- Unnamed elements:
Problems

- Lexical order:
  \[
  \begin{array}{c}
  X \\
  A & B \\
  \end{array} \quad \neq \quad \begin{array}{c}
  X \\
  B & A \\
  \end{array}
  \]

- Unnamed elements:
  \[
  \begin{array}{c}
  X \\
  ? & ? \\
  \end{array}
  \]

- Ambiguous names:
  \[
  \begin{array}{c}
  X \\
  A & A \\
  \end{array}
  \]
Conclusion

- **Superimposition** is a general mechanism to compose software artifacts (features)
- **Language independent** model captures the essence of superimposition (formal model: feature algebra)
- **FeatureHouse**: Languages can be integrated almost automatically
  - A wide variety of very different languages are integrated
- **Substantial cases studies** demonstrate practicality and reveal open issues
- Problems with lexical order, unnamed elements, and ambiguous names
Questions?

