Delta-oriented Development of Software Product Lines

Ina Schaefer
Institut für Software Systems Engineering
Technische Universität Braunschweig
i.schaefer@tu-braunschweig.de

Philipps-Universität Marburg
27. Juni 2011
Software Product Lines

- A product line is “a family of products designed to take advantage of their common aspects and predicted variabilities.” [Weiss; 1999]

- A product line is “a set of systems sharing a common set of features that satisfy the specific needs of a particular market segment.” [Clements, Northrop; 2001]
Product Lines in Industry

- Industrial-scale Approach for Reuse in Software Engineering
- Members of Product Line Hall of Fame: [http://www.splc.net/fame.html](http://www.splc.net/fame.html)
  - HP, Ericsson, Nokia, GM Powertrain, Boeing, Bosch, Lucent, Philips, Toshiba
- Commercially successful, e.g., HP Owen Firmware Cooperative:
  - 1/4 of the staff, 1/3 of the time, and 1/25 of the bugs (compared to previous single application engineering)
Product Line Development

- Feature Model
- Family Engineering
- Product Line Artifacts Base
- Application Engineering
- Product

Delta-oriented Development of Software Product Lines
Product Line Development

Feature Model → Family Engineering → Product Line Artifacts Base → Automated Product Derivation → Product

Feature Configuration
Outline

• Delta Modeling
• Delta Modeling and Model-based Development
• Delta-oriented Programming in DeltaJava
• Incremental Product Line Analysis
• Pure Delta-oriented Programming
Delta Modeling of Product Lines

- Product for valid feature configuration.
- Developed with Standard Techniques
- Modifications of Core Product.
- Application conditions over product features.
- Partial ordering for conflict resolution.

Configuration

For a Feature Configuration:

• Determine product deltas with valid application condition.

• Determine linear ordering of product deltas compatible with partial ordering.

• Apply changes specified by product deltas to core product in the linear order.
Feature-based Variability

Example: Trading System Software Product Line
Feature-based Variability

Example: Trading System Software Product Line

Payment Features
Component Modeling

Component Core Model (for Cash Payment):

Component Delta Model (for Card Payment):
Component Modeling (2)

Configured Component Model (for Cash and Card Payment):

Inventory → Cash Desk → Bank
Delta Modeling

- Evolutionary Development by Adding Product Deltas
- Explicit Treatment of Combinations of Features by Complex Application Conditions
- Proactive, Reactive and Extractive SPL Development
- Usable with Different Modelling Formalisms and Implementation Techniques
- Model Refinements are Orthogonal to Variability Modeling.
Model-based Development

Feature Modelling

Modelling Level 1
- Core Model\(_1\)
- Delta Models\(_1\)

Modelling Level 2
- Core Model\(_2\)
- Delta Models\(_2\)

Implementation
- Core Module
- Delta Modules

- Feature Model
- Feature Configuration
- Core Model\(_1\)
- Delta Models\(_1\)
- Product Models\(_1\)
- Core Model\(_2\)
- Delta Models\(_2\)
- Product Models\(_2\)
- Core Module
- Delta Modules
- Product Impl.

Model-based Development

Feature Modelling

- Feature Model
  - Feature Configuration
    - configure
    - create

Modelling Level 1

- Core Model\(_1\)  Delta Models\(_1\)
  - Product Models\(_1\)
    - configure
    - refine

Modelling Level 2

- Core Model\(_2\)  Delta Models\(_2\)
  - Product Models\(_2\)
    - configure
    - refine
    - [...] (refine)

Implementation

- Core Module  Delta Modules
  - Product Impl.
    - configure

A Component Design can be refined to Class Diagrams:

- Each Component in Core Model is realized by a Class Core Diagram.
- Component Delta Models are realized by Class Delta Diagrams.

For each Component:

1 Class Core Diagram

Set of Class Delta Diagrams
Core Class Diagram

Inventory

Cash Desk

Display

+CashDeskDisplay()
+void displayProduct(Product, float)
+void displayTotal(float)
+void displayTotalAndReturn(float)

Cash Desk

-Order

+CashDesk()
+void addProduct(int id)
+void receiveMoney(float)
+void cancelCurrentPayment()
+void startPaymentProcess()
+void endPaymentProcess()
+void selectCashPayment()
-void connectWithInventory()

Keyboard

+Keyboard(CashDesk)
+void terminate()
+void setStatePrePaymentProcess()
+void setStatePaymentProcess()
+void setStatePaymentSelection()
+void setStateCashPayment()
+void enable()
+void disable()
Core Class Diagram

Refinement of Core Cash Desk Component
Class Delta Diagram

Delta-oriented Development of Software Product Lines
Class Delta Diagram

Refinement of Cash Desk Component Delta Model

Delta-oriented Development of Software Product Lines
Commutativity

If Model Refinement and Model Configuration are compatible, it holds that:

\[
\text{refine}(\text{configure}_i((\text{Core}_i, \Delta s_i), fc)) = \text{configure}_{i+1}(\text{refine} (\text{Core}_i, \Delta s_i), fc)
\]
Compatibility

- Additions of Modeling Elements are refined to Additions only.
- Removals of Modeling Elements are refined to Removals only.
- Modifications of Modeling Elements are refined to Additions, Modifications and Removals, such that for each Modification operation \( \text{mod}(e) \) of a Modelling Element \( e \), it holds that

\[
\text{apply(refine}(e), \text{refine}(\text{mod } e)) = \text{refine}(e')
\]
Model-based Development

Feature Modelling

Feature Model \(\xrightarrow{\text{configure}}\) Feature Configuration

create

Modelling Level 1

Core Model\(_1\) Delta Models\(_1\) \(\xrightarrow{\text{refine}}\) Product Models\(_1\)

configure

Modelling Level 2

Core Model\(_2\) Delta Models\(_2\) \(\xrightarrow{\text{refine}}\) Product Models\(_2\)

configure

Implementation

Core Module Delta Modules \(\xrightarrow{\text{configure}}\) Product Impl.

refine

[...]

refine

Delta-oriented Development of Software Product Lines
Model-based Development

Feature Modelling

Feature Model \(\xrightarrow{\text{create}}\) Feature Configuration

Modelling Level 1

Core Model\(_1\) Delta Models\(_1\) \(\xrightarrow{\text{configure}}\) Product Models\(_1\)

Modelling Level 2

Core Model\(_2\) Delta Models\(_2\) \(\xrightarrow{\text{configure}}\) Product Models\(_2\)

Implementation

Core Module Delta Modules \(\xrightarrow{\text{configure}}\) Product Impl.
DeltaJava - A PL for SPL

- Extension of Java with Core and Delta Modules
- Core Product is implemented by Core Module.
- Product Deltas are implemented by Delta Modules.
- A Product Implementation is obtained by Application of Delta Modules to Core Module.
- Type System ensures Safety of Delta Application.

Example Product Line

Feature Model:

Expression Product Line

Data
- Lit
- Add
- Neg

Operations
- Print
- Eval
Core Module

A core module contains a set of Java classes.

core Print, Lit {
    class Lit implements Exp{
        int value;
        Lit(int n) { value=n; }
        void print()
            { System.out.println(value); }
    }
    class Test {
        Exp a;
        void run()
            { a=new Lit(3); a.print(); }
    }
}
Delta Modules

- Modifications on Class Level:
  - Addition, Removal and Modification of Classes

- Modifications of Class Structure:
  - Changing Super Class and Constructor
  - Adding/Removing Fields/Methods
  - Modifying Methods (wrapping with original call)

- Application Condition in when clause: Boolean Constraint on Features in Feature Model

- Partial Ordering of Delta Modules by after clauses
Delta Module for Add

```java
delta DAdd when Add {
  adds class Add {
    Exp expr1;
    Exp expr2;
    Add(Exp a, Exp b) { expr1=a; expr2=b; }
  }
}
```
Delta for Add and Print

delta DAddPrint after DAdd when Add && Print{

  modifies class Add {
    adds void print()
    { expr1.print();
      System.out.print(" + ");
      expr2.print(); } 
  }

  modifies class Test {
    adds Exp e;
    modifies void run()
    { original();
      e=new Add(a, a);
      e.print(); } 
  }
}
Product Generation

For a Feature Configuration:

• Determine product deltas with valid application condition in when clauses.

• Determine linear ordering of product deltas compatible with partial ordering in after clauses.

• Apply changes specified by product deltas to core product in the linear order.
class Lit {
    int value;
    Lit(int n) { value=n; }
    void print() { System.out.print(value); }
}

class Add {
    Exp e1; Exp e2;
    Add(Exp a, Exp b) { e1=a; e2=b; }
    void print() { e1.print(); System.out.print(" + ");
                     e2.print();}
}

class Test {
    Exp a; Exp e;
    void run() { a=new Lit(3); a.print();
                 e=new Add(a,a); e.print(); }
}
Delta-oriented Development of Software Product Lines

DeltaJava Product Lines

Starting from a Simple Core (Proactive Development)

features Lit, Add, Neg, Print, Eval
configurations Lit &amp; Print &amp; (Add | Neg | Eval )
core Lit, Print { [ ... ] }
delta DEval when Eval { [ ... ] }
delta DAdd when Add { [ ... ] }
delta DAddPrint after DAdd when Add &amp; Print { [ ... ] }
delta DAddEval after DAdd when Add &amp; Eval { [ ... ] }
delta DNeg when Neg { [ ... ] }
delta DNegPrint after DNeg when Neg &amp; Print { [ ... ] }
delta DNegEval after DNeg when Neg &amp; Eval { [ ... ] }
DeltaJava Product Lines (2)

Starting from a Complex Core (Extractive/Reactive Development)

---

**features** Lit, Add, Neg, Print, Eval

**configurations** Lit && Print && (Add | Neg | Eval )

**core** Lit, Print, Add, Eval  { [ ... ] }

**delta** DRemEval when !Eval && Add  { [...] }

**delta** DRemAdd when !Add  { [...] }

**delta** DRemAddEval when !Add && !Eval  { [...] }

**delta** DNeg when Neg  { [...] }

**delta** DNegPrint after DNeg when Neg && Print  { [...] }

**delta** DNegEval after DNeg when Neg && Eval  { [...] }

---

Delta-oriented Development of Software Product Lines
## Deltas vs. Feature Modules

[AHEAD, Batory et al. 2004]

<table>
<thead>
<tr>
<th></th>
<th>Feature-oriented Programming</th>
<th>Delta-oriented Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressiveness</strong></td>
<td>Design from Base Module</td>
<td>Design from Any Product</td>
</tr>
<tr>
<td><strong>Domain Features</strong></td>
<td>Bijection between Features and Feature Modules</td>
<td>Delta Modules for Feature Combinations</td>
</tr>
<tr>
<td><strong>Optional Feature Problem</strong></td>
<td>Rearrange Code, Multiple Impl., Derivative Modules</td>
<td>Direct Implementation of Interaction</td>
</tr>
<tr>
<td><strong>Safe Composition</strong></td>
<td>External Tools, Type Systems</td>
<td>Partial Order for Conflict Resolution, Type System</td>
</tr>
<tr>
<td><strong>Evolution</strong></td>
<td>Refactoring</td>
<td>Addition of Delta Modules</td>
</tr>
</tbody>
</table>
## Deltas vs. Feature Modules (2)

<table>
<thead>
<tr>
<th></th>
<th>Jak # feature modules</th>
<th>Jak LOC</th>
<th>DeltaJava Simple Core # delta modules</th>
<th>DeltaJava Simple Core LOC</th>
<th>DeltaJava Complex Core # delta modules</th>
<th>DeltaJava Complex Core LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPL [1]</td>
<td>12</td>
<td>98</td>
<td>7</td>
<td>123</td>
<td>6</td>
<td>144</td>
</tr>
<tr>
<td>EPL [26]</td>
<td>6</td>
<td>98</td>
<td>5</td>
<td>117</td>
<td>5</td>
<td>124</td>
</tr>
<tr>
<td>Calculator [7]</td>
<td>10</td>
<td>75</td>
<td>6</td>
<td>76</td>
<td>6</td>
<td>78</td>
</tr>
<tr>
<td>List [7]</td>
<td>4</td>
<td>48</td>
<td>3</td>
<td>58</td>
<td>2</td>
<td>59</td>
</tr>
<tr>
<td>GraphPL [25]</td>
<td>19</td>
<td>2348</td>
<td>20</td>
<td>1407</td>
<td>19</td>
<td>1373</td>
</tr>
</tbody>
</table>

Table 2: Evaluation Results (LOC is the number of lines of code)
Applicability

A set of delta-modules is applicable to a class table if

- all removed or modified classes exists.
- for every delta modifies-clause,
  - all removed methods and fields exist.
  - all modified methods exists.
- if added classes, methods and fields do not exist.
Well-formedness

1. The core configuration is a valid feature configuration.

2. No delta module has a valid application condition for the core configuration.

3. The partial order ensures that for every feature configuration, the set of delta modules with valid application condition
   a. is applicable to the respective intermediate product
   b. and conflict-free (all conflicting modifications are ordered).
Constraint-based Type System

- Type check and generate constraints for core and delta modules in isolation.

- A product is type checked by
  - generating the product constraints (application of the delta module constraints to the core product constraints)
  - checking the derived product constraints against the product’s class signature table.

Implementation

• Based on Xtext Language Development Framework

• Using XPAND for Code Generation

• Eclipse plugin with syntax highlighting, background parsing with error marking, code completion

• Stand-alone compiler
Implementation

Delta-oriented Development of Software Product Lines

Based on Xtext Language Development Framework
Using XPAND for Code Generation
Ecplise plugin with syntax highlighting, background parsing with error marking, code completion
Stand-alone compiler
Implementation

Available at http://deltaj.sourceforge.net/
Behavioral Specifications

- We consider **design-by-contract specifications** with class invariants and method contracts.
- Core module can be specified as a standard Java program using JML.
- Delta modules are specified by Specification Deltas
  - Addition and removal of invariants
  - Addition and removal of method contracts
  - Modification of method contracts
Verification of DeltaJava SPL

- We use the KeY System for deductive verification of DeltaJava SPL.

**Input to KeY:**

- Java Program with JML Specifications (generated from DeltaJava SPL)

- KeY generates proof obligations in dynamic logic and supports automatic and interactive verification.
Verification with KeY

Delta-oriented Software Product Lines
Verification with KeY

```java
inReachableState
& \forall BaseAccount b_0;
& (self.<created> = TRUE & !self = null)
& inInt(x)
& x > (jint)(0)
-> {_x:=x ||
\for BaseAccount x; balanceAtPre_0(x):=x.balance}
<{
exc=null;try {
    self.update(_x)@BaseAccount;
} catch (java.lang.Throwables e) {
    exc=e;
}
}\> (self.balance >= balanceAtPre_0(self) & exc = null)
```
Verification with KeY

Delta-oriented Software Product Lines
Incremental Analysis

Product P + Spec ✔

Proof ✔

Incremental Analysis

Product P
+ Spec

Proof

Delta Application

Code-Deltas
Spec-Deltas

Product P’
+ Spec’

Incremental Analysis

1. Determine valid proof obligations by Delta-oriented Slicing

Product P + Spec

Code-Deltas

Spec-Deltas

Product P’ + Spec’

Proof

Delta Application

Incremental Analysis

1. Determine valid proof obligations by Delta-oriented Slicing

2. Proof Reuse for changed proof obligations

public class BaseAccount {

/*@
@ public instance invariant balance >= 0;
@*/
int balance;

public BaseAccount(){
balance = 0;
}

/*@                  ← Method Contract
@ public normal_behavior
@ requires x > 0;
@ assignable balance;
@ ensures balance >= \old(balance);
@*/
public void update(int x){ balance += x; }
Specification of Delta Sync

delta DSyncUpdate after Dretirement, Dinvestment when Sync {
  modifies class Account {

  add invariant
  @ public instance invariant lock != null;
  adds Lock lock;

  add contract
  @ public normal_behavior
  @ requires x > 0;
  @ assignable balance, lock;
  @ ensures balance >= \old(balance);
  adds void sync_update(int x) { [... ] } 
}
}
public class BaseAccount {

/*@ */
    @ public instance invariant balance >= 0;
    @ public instance invariant lock != null;
    @*/
    Lock lock;
    int balance;

    public BaseAccount(){ balance = 0;}

   /*@ */
    @ public normal_behavior
    @ requires x > 0;
    @ assignable balance;
    @ ensures balance >= \old(balance);
    @*/
    public void update(int x){
        balance += x; }
}
Delta-oriented Software Product Lines

Example

def deltax DSyncUpdate after Dretirement, Dinvestment when Sync {}
modifies class Account {

add invariant
@ public instance invariant lock != null;
adds Lock lock;

add contract
@ public normal_behavior
@ requires x > 0;
@ assignable balance, lock;
@ ensures balance >= \old(balance);
adds void sync_update(int x) { [...] }
}
Example

```java
delta DSyncUpdate after Dretirement, Dinvestment when Sync {
    modifies class Account {

        add invariant
        @ public instance invariant lock != null;
        adds Lock lock;

        add contract
        @ public normal_behavior
        @ requires x > 0;
        @ assignable balance, lock;
        @ ensures balance >= \old(balance);
        adds void sync_update(int x) { [...] } }
}
Example

```java
delta DSyncUpdate after Dretirement, Dinvestment when Sync {
  modifies class Account {

  add invariant
  @ public instance invariant lock != null;
  adds Lock lock;

  add contract
  @ public normal_behavior
  @ requires x > 0;
  @ assignable balance, lock;
  @ ensures balance >= \old(balance);
  adds void sync_update(int x) { [...] } 
}
}
```

Prove Invariant for methods accessing lock

Prove Contract and Existing Invariants
Core DOP

• **Strengths:**
  
  • Any product can be core product.
  
  • Feature combinations in application conditions.
  
  • Explicit ordering for conflict resolution.
  
  • Usable with different modelling and implementation languages.
Core DOP (2)

- **Weaknesses:**
  - Selection of core product (e.g., for mandatory exclusive features).
  - Only partial generalization of FOP.
  - Limited support for product line evolution (e.g., if core product is no longer valid).
Pure DOP

- Modifications of core product.
- Application conditions over product features in when clauses.
- Partial ordering for conflict resolution in after clauses.

Pure DOP

- Modifications of core product.
- Application conditions over product features in when clauses.
- Partial ordering for conflict resolution in after clauses.

Pure DOP

- Modifications of core product.
- Application conditions over product features in when clauses.
- Partial ordering for conflict resolution in after clauses.

Pure DOP

- Product for valid feature configuration.
- Modifications of core product.

Product line specification contains application conditions and ordering.

Pure DOP Product Line

Feature Model:

Expression Product Line

Data
Lit Add Neg

Operations
Print Eval
Delta Modules

delta DLit{
   adds interface Exp {
   }
   adds class Lit implements Exp {
      int value;
      Lit(int n) { value = n; }
   }
}

delta DLitPrint{
   modifies interface Exp {
      void print();
   }
   modifies class Lit implements Exp {
      adds void print() { System.out.println(value); }
   }
}
features Lit, Add, Neg, Print, Eval
configurations Lit & Print
deltas
[ DLit,
  DAdd when Add,
  DNeg when Neg ]

[ DPrint,
  DEval when Eval,
  DAddPrint when Add,
  DAddEval when (Add & Eval),
  DNegPrint when Neg,
  DNegEval when (Neg & Eval) ]
Product Line Specification

features Lit, Add, Neg, Print, Eval
configurations Lit & Print
deltas
  [ DLit,
    DAdd when Add,
    DNeg when Neg ]

  [ DPrint,
    DEval when Eval,
    DAddPrint when Add,
    DAddEval when (Add & Eval),
    DNegPrint when Neg,
    DNegEval when (Neg & Eval) ]

Application ordering is defined by Sequence of delta sets.

Deltas in the same set must be commutative.
• Lightweight Feature Java [Delaware et al.; FOAL’09/FSE’09]

\[
\text{FMD} ::= \text{feature } \varphi \{ \text{cd rcd} \} \quad \text{feature module}
\]
\[
\text{rcd} ::= \text{refines class } C \text{ extending } C \{ \text{ fd md rmd } \} \quad \text{class refinement}
\]
\[
\text{rmd} ::= \text{refines ms } \{ \overline{s}; \text{ Super()}; \overline{s}; \text{ return y}; \} \quad \text{method refinement}
\]
FOP and PureDOP

• Lightweight Feature Java [Delaware et al.; FOAL’09/FSE’09]

FMD ::= feature φ {cd rcd}               feature module
rcd ::= refines class C extending C { fd; md rmd }   class refinement
rmd ::= refines ms {s; Super(); s; return y;}  method refinement

• Embedding of LFJ into LPΔJ - Defines same set of products:

\[
\begin{align*}
\llbracket \text{feature } \phi \{ \text{cd rcd} \} \rrbracket &= \delta \phi \{ \text{adds cd} \llbracket \text{rcd} \rrbracket \} \\
\llbracket \text{refines class C extending C } \{ \text{fd; md rmd} \} \rrbracket &= \text{modifies C extending C } \{ \text{adds fd adds md} \llbracket \text{rmd} \rrbracket \} \\
\llbracket \text{refines ms } \{ \text{s; Super() ; s; return y;} \} \rrbracket &= \text{modifies ms } \{ \text{s; original(); s; return y;} \}
\end{align*}
\]
Extractive PL Development

features Lit, Add, Neg, Print, Eval
configurations Lit & Print
deltas
[ DLitNegPrint when (!Add & Neg) ]

[ DLitAddPrint when (Add | !Neg) ]

[ DNeg when (Add & Neg),
  DremAdd when (!Add & !Neg) ]

[ DNegPrint when (Add & Neg),
  DLitEval when Eval,
  DAddEval when (Add & Eval),
  DNegEval when (Neg & Eval) ]
Extractive PL Development

<table>
<thead>
<tr>
<th>Features</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lit, Add, Neg, Print, Eval</td>
<td>Lit &amp; Print</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deltas</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ DLitNegPrint when (Add &amp; Neg) ]</td>
</tr>
<tr>
<td>[ DLitAddPrint when (Add</td>
</tr>
<tr>
<td>[ DNeg when (Add &amp; Neg), DremAdd when (!Add &amp; !Neg) ]</td>
</tr>
<tr>
<td>[ DNegPrint when (Add &amp; Neg), DLitEval when Eval, DAddEval when (Add &amp; Eval), DNegEval when (Neg &amp; Eval) ]</td>
</tr>
</tbody>
</table>
Extractive PL Development

<table>
<thead>
<tr>
<th>Existing Products</th>
<th>Feature Removal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>features</strong> Lit, Add, Neg, Print, Eval</td>
</tr>
<tr>
<td></td>
<td><strong>configurations</strong> Lit &amp; Print</td>
</tr>
<tr>
<td></td>
<td><strong>deltas</strong></td>
</tr>
<tr>
<td></td>
<td>[ DLitNegPrint <strong>when</strong> (!Add &amp; Neg) ]</td>
</tr>
<tr>
<td></td>
<td>[ DLitAddPrint <strong>when</strong> (Add</td>
</tr>
<tr>
<td></td>
<td>[ DNeg <strong>when</strong> (Add &amp; Neg), DremAdd <strong>when</strong> (!Add &amp; !Neg) ]</td>
</tr>
<tr>
<td></td>
<td>[ DNegPrint <strong>when</strong> (Add &amp; Neg), DLitEval <strong>when</strong> Eval, DAddEval <strong>when</strong> (Add &amp; Eval), DNegEval <strong>when</strong> (Neg &amp; Eval) ]</td>
</tr>
</tbody>
</table>

Delta-oriented Development of Software Product Lines
A LFJ product line can be described using the feature module definitions. A straightforward embedding as for Pure DOP is not possible. This configuration application condition of the delta module denotes all configurations translated to adds clauses and refinements to modifies clauses. The product line is mapped to a delta module where additions are associated to the features in the feature module in $FMT$ to a delta module with the same name. The delta module table $DMT$ is obtained by translating each feature of the SPL. For a LFJ product line $L$, $\{ \text{feature} \}$ is the total order on the set of features $FMT$. $\text{Super}_{\text{dir}}$ denotes the corresponding LP if and only if the product for the initial product line $L$ and methods are added and class and method refinements are carried out. The following theorem states that the LP $\{ \text{feature} \}$ refines $\{ \text{feature} \}$, extending $\{ \text{feature} \}$ if all $\{ \text{feature} \}$ generated by $\{ \text{feature} \}$, $\{ \text{feature} \}$, and a set of class definitions $C$, $\{ \text{feature} \}$ is the total order on the set of features $FMT$. Theorem 3.1. Product line evolution is type-safe if all $\{ \text{feature} \}$, $\{ \text{feature} \}$, and $\{ \text{feature} \}$, $\{ \text{feature} \}$ are introduced for representing subtraction expressions. In the new application condition of the delta module, additionally, the $EPL$ the new requirements following the reactive approach, feature modules might have to be refactored to remove functionalities. Also, in Core DOP, new requirements following the reactive approach, feature modules of features and to remove features. Moreover, in order to deal with other products of the product line, they are derived. Development starts with the existing products from which the product line is evolved. The extractive approach allows turning a set of existing legacy application into a product line. In reactive product line engineering, development requires a high upfront investment to define the scope of the product line and develop delta modules, while Core DOP requires exactly one core module as a component set of products is developed. When new customer requirements arise, the existing product line is evolved. The extractive approach, since we start from the feature diagram for the evolved product line, we have to add delta modules, while Core DOP requires exactly one core module as a component set of products is developed.
Product Line Evolution

features Lit, Add, Neg, Sub, Print, Eval
configurations Lit & Eval & choose1(Neg, Sub)
deltas
[ DLit,
  DAdd when Add,
  DNeg when Neg,
  DSub when Sub ]

[ DLitPrint when Print,
  DLitEval,
  DAddPrint when (Add & Print),
  DAddEval when Add,
  DNegPrint when (Neg & Print),
  DNegEval when Neg,
  DSubPrint when (Sub & Print),
  DSubEval when Sub ]
**Product Line Evolution**

**features** Lit, Add, Neg, Sub, Print, Eval

**configurations** Lit & Eval & choose1(Neg,Sub)

**deltas**

[ DLit,
  DAdd when Add,
  DNeg when Neg,
  DSub when Sub ]

[ DLitPrint when Print,
  DLitEval,
  DAddPrint when (Add & Print),
  DAddEval when Add,
  DNegPrint when (Neg & Print),
  DNegEval when Neg,
  DSubPrint when (Sub & Print),
  DSubEval when Sub ]
Delta-oriented Development of Software Product Lines

Product Line Evolution

Change in Feature Model

- **features** Lit, Add, Neg, Sub, Print, Eval
- **configurations** Lit & Eval & choose1(Neg,Sub)
- **deltas**
  - [ DLit, DAdd when Add, DNeg when Neg, DSub when Sub ]
  - [ DLitPrint when Print, DLitEval, DAddPrint when (Add & Print), DAddEval when Add, DNegPrint when (Neg & Print), DNegEval when Neg, DSubPrint when (Sub & Print), DSubEval when Sub ]

New and Reconfigured Delta Modules
Pure DOP vs. Core DOP

- Core DOP PL can be transformed to Pure DOP PL by introducing core product in first delta module.
- Pure DOP PL can be transformed to Core DOP PL by adding empty product to Core DOP PL.
Conclusion

Summary:

• Delta Modeling of Software Product Lines
• Delta-oriented Programming with Delta Java
• Pure Delta-oriented Programming
• Incremental Product Line Analysis

Future Work:

• Patching and Transforming DOP Product Lines
• Development Guidelines for DOP Product Lines
• Family-based and Compositional Analyses
Future Work

• Transformation of Core DOP Product Lines (to only additive versions)

• Patching Pure/Core DOP Product Lines

• Development Guidelines for DOP Product Lines

• Family-based and Compositional Analyses for DOP Product Lines