A Formal Foundation for Dynamic Delta-Oriented Software Product Lines

Ferruccio Damiani\(^{(a)}\), Luca Padovani\(^{(a)}\) and Ina Schaefer\(^{(b)}\)

\(^{(a)}\): University of Torino, Italy
\(^{(b)}\): TU Braunschweig, Germany

CINA - WP1 Meeting

July 23-24, 2014

Genova, Italy

\(^{1}\)Talk based on the paper presented at GPCE 2012 — EAPLS GPCE/SLE 2012 Best Paper Award
Motivation

- DOP has been used for modelling variability at compile time
- This paper extends it for
  1. modelling variability at run time
  2. providing a framework for unanticipated software evolution
Outline

- Delta-oriented Programming
- Dynamic DOP
- Evolving Dynamic DOP SPLs
- Dynamic DOP (cont.)
- Formalization
- Related/Future Work
Delta-oriented Programming (DOP): concepts

A DOP SPL consists of:

- Product Line Declaration
  - Connection between Delta Modules and Product Features
  - Order of Delta Module Application

- Code Base
  - Delta Module \(_1\)
  - [...]
  - Delta Module \(_n\)

[Schaefer et al., SPLC 2010] [Schaefer and Damiani, FOSD 2010] ... [Koscielny et al., PPPJ 2014]
Product Generation in Delta-oriented Product Lines

Given a given feature configuration:

1. determine delta modules with valid application condition

2. apply the changes specified by delta modules
   - to the empty program
   - according to a suitable delta module application ordering
Product Generation in Delta-oriented Product Lines

Given a given feature configuration:

1. determine delta modules with valid application condition

2. apply the changes specified by delta modules
   - to the empty program
   - according to a suitable delta module application ordering

Type-safe SPL

A SPL is type safe if all its products are well-typed programs
Product Generation in Delta-oriented Product Lines

Given a given feature configuration:

1. determine delta modules with valid application condition
2. apply the changes specified by delta modules
   - to the empty program
   - according to a suitable delta module application ordering

Type-safe SPL

A SPL is type safe if all its products are well-typed programs

- A type system that ensures type safety of DOP SPLs of
  Imperative Featherweight Java programs [Bettini et al., Acta Inf., 2013]
Dynamic DOP: concepts

A Dynamic DOP SPL consists of:

1. Product-line code base
2. Product-line declaration
3. Product-line dynamic reconfiguration graph
Example: (Dynamic) Expression Product Line

Exp ::= Lit | Add
Lit ::= <non-negative integers>
Add ::= Exp "+" Exp

Feature Model of (Dynamic) EPL:

There are 2 valid feature configurations:

**LAP** = Lit, Add, Print

**LAPE** = Lit, Add, Print, Eval
1. Product-Line Code Base for (Dynamic) EPL

delta DLitAddPrint{
    adds class Exp extends Object { ... }
    adds class Lit extends Exp { ... }
    adds class Add extends Exp { ... }
    adds class Printer { ... }
    adds class Main { ... }
    adds class ExpScanner ...
}

delta DLitEval {
    modifies Exp { ... }
    modifies Lit { ... }
}

delta DAddEval {
    modifies Add { ... }
}

delta DPrinterEval {
    modifies Printer { ... }
}
Product-Line Code Base for EPL (1/2)

delta DLitAddPrint{
  adds class Exp extends Object { // Exp is only used as a type
    int counter; // number of operations invoked on the expression
    String toString() { return ""; } }

  adds class Lit extends Exp {
    int value; // value >= 0
    Lit setLit(int n) { value = n; return this; }
    String toString() { counter++; return value + ""; }
  }

  adds class Add extends Exp {
    Exp expr1;
    Exp expr2;
    Add setAdd(Exp a, Exp b) { expr1 = a; expr2 = b; return this; }
    String toString() { counter++; return expr1.toString() + "+" + expr2.toString(); }
  }

  adds class Printer {
    String compute(Exp a) { return a.toString(); }
  }

  adds class Main {
    void main() {
      Scanner in = new ExpScanner(System.in);
      Printer pr = new Printer();
      while (true) {
        Exp expr = in.nextExp();
        System.out.println(pr.compute(expr));
      }
    }
  }
  adds class ExpScanner ... // parses data of Exp type}
Product-Line Code Base for EPL (2/2)

delta DLitEval {
    modifies Exp {
        adds int eval() { return 0; }
    }
    modifies Lit {
        adds int eval() { counter++; return value; }
    }
}

delta DAddEval {
    modifies Add {
        adds int eval() { counter++; return expr1.eval() + expr2.eval(); }
    }
}

delta DPrinterEval {
    modifies Printer {
        modifies String compute(Exp a) { return original(a) + "=" + a.eval(); }
    }
}
2. Product-Line Declaration for (Dynamic) EPL

features
   Lit, Add, Print, Eval
configurations
   Lit & Add & Print
deltas
   { DLitAddPrint }
   { DLitEval when Eval, DAddEval when Eval, DPrinterEval when Eval }
3. Dynamic Reconfiguration Graph for Dynamic EPL

Graphical representation:

![Graphical representation of LAP and LAPE nodes with an arrow between them](image)

Textual representation:

```plaintext
nodes
    LAP = Lit, Add, Print;
    LAPE = Lit, Add, Print, Eval;
edges
    LAP => LAPE { }
    LAPE => LAP { }
```
Runtime reconfiguration: changing the configuration of the running product

- A class $C$ is **affected** by a reconfiguration if
  - either $C$ is **recoded** (the code of $C$ or of a superclass of $C$ is modified)
  - or $C$ is **reallocated** (the instances of $C$ are changed)

- A reconfiguration is **enabled** if
  - classes of receivers of methods on the call stack are not affected by the reconfiguration.
class Exp extends Object { // Exp is only used as a type
    int counter; // number of operations invoked on the expression
    String toString() { return ""; }
}

class Lit extends Exp {
    int value; // value >= 0
    Lit setLit(int n) { value = n; return this; }
    String toString() { counter++; return value + ""; }
}

class Add extends Exp {
    Exp expr1;
    Exp expr2;
    Add setAdd(Exp a, Exp b) { expr1 = a; expr2 = b; return this; }
    String toString() { counter++; return expr1.toString() + "+" + expr2.toString(); }
}

class Printer {
    String compute(Exp a) { return a.toString(); }
}

class Main {
    void main() {
        Scanner in = new ExpScanner(System.in);
        Printer pr = new Printer();
        while (true) {
            Exp expr = in.nextExp();
            System.out.println(pr.compute(expr));
        }
    }
}

class ExpScanner ... // parses data of Exp type
Product LAPE of the Dynamic EPL

class Exp extends Object { // Exp is only used as a type
    int counter; // number of operations invoked on the expression
    String toString() { return ""; }
    int eval() { return 0; } // ***ADDED***
}
class Lit extends Exp {
    int value; // value >= 0
    Lit setLit(int n) { value = n; return this; }
    String toString() { counter++; return value + ""; }
    int eval() { counter++; return value; } // ***ADDED***
}
class Add extends Exp {
    Exp expr1;
    Exp expr2;
    Add setAdd(Exp a, Exp b) { expr1 = a; expr2 = b; return this; }
    String toString() { counter++; return expr1.toString() + "+" + expr2.toString(); }
    int eval() { counter++; return expr1.eval() + expr2.eval(); } // ***ADDED***
}
class Printer {
    String compute$DPrinterEval(Exp a) { return a.toString(); } // ***ADDED***
    String compute(Exp a) { return compute$DPrinterEval(a) + "=" + a.eval(); } // ***MODIFIED***
}
class Main { // ***UNCHANGED***
    void main() {
        Scanner in = new ExpScanner(System.in);
        Printer pr = new Printer();
        while (true) {
            Exp expr = in.nextExp();
            System.out.println(pr.compute(expr));
        }
    }
class ExpScanner ... // parses data of Exp type // ***CHANGED***
Unanticipated Evolution

WHAT:

- New products are added
- Existing products (different from the currently running product) are modified or removed
Unanticipated Evolution

WHAT:
- New products are added
- Existing products (different from the currently running product) are modified or removed

HOW:
Changing the DOP product-line code base, declaration, dynamic reconfiguration graph by preserving the currently running product.
Unanticipated Evolution

WHAT:

- New products are added
- Existing products (different from the currently running product) are modified or removed

HOW:

Changing the DOP product-line code base, declaration, dynamic reconfiguration graph by preserving the currently running product.

Preserved product

A product is preserved by an evolution if in both the SPLs:

- the feature configuration of the product is valid, and
- the feature configuration selects the same delta modules (in the same order).\(^a\)

\(^a\)This implies that code of the selected delta modules is unchanged.
Example: Evolving the Feature Model of the Dynamic EPL

There are 4 valid feature configurations:

- **LAP**: Lit / Add / Print
- **LAPE**: Lit, Add, Print, Eval
- **LASPE**: Lit, Add, Sub, Print, Eval
- **LANPE**: Lit, Add, Neg, Print, Eval
- **LSPE**: Lit, Sub, Print, Eval
Example: Evolving the Dynamic Reconfiguration Graph of the Dynamic EPL

There are four (valid) feature configurations:

- **LAP** ≠ Lit, Add, Print
- **LAPE** = Lit, Add, Print, Eval
- **LASPE** = Lit, Add, Sub, Print, Eval
- **LANPE** = Lit, Add, Neg, Print, Eval
- **LSPE** = Lit, Sub, Print, Eval
1. Product-Line Code Base of the evolved Dynamic EPL
(added delta modules)

```java
delta DNeg {
    adds class Neg extends Exp {
        Exp expr;
        Neg setNeg(Exp a) { expr = a; return this; }
        String toString() { counter++; return "(" + "-" + expr.toString() + ")"; }
        int eval() { counter++; return (-1) * expr.eval(); }
    }
    modifies class ExpScanner ... // parses Neg expressions
}

delta DSub {
    adds class Sub extends Exp {
        Exp expr1;
        Exp expr2;
        Add setSub(Exp a, Exp b) { expr1 = a; expr2 = b; return this; }
        String toString() { counter++; return expr1.toString() + "-" + expr2.toString(); }
        int eval() { counter++; return expr1.eval() - expr2.eval(); }
    }
    modifies class ExpScanner ... // parses Sub expressions
}

delta DremAdd {
    removes Add
    modifies class ExpScanner ... // removes code for parsing Add expressions
}
```
features
   Lit, Add, Neg, Sub, Print, Eval
configurations
   Lit & Print & Eval & (Add|Sub) & (Neg->Add) & !(Neg&Sub)
deltas
   { DLitAddPrint }
   { DLitEval, DAddEval when Add, DPrinterEval }
   { DSub when Sub }
   { DNeg when Neg }
   { DremAdd when !Add }
3. Dynamic Reconfiguration Graph of the evolved Dynamic EPL

nodes

LAPE = Lit, Add, Print, Eval;
LASPE = Lit, Add, Sub, Print, Eval;
LANPE = Lit, Add, Neg, Print, Eval;
LSPE = Lit, Sub, Print, Eval;

edges

LAPE => LASPE { }
LASPE => LANPE { // e.g.: (7-8) becomes (7+(-8))
    Sub->Add { pre:
        Exp y1 = this.expr1; Exp y2 = this.expr2;
        post:
        Exp z1 = y1; Exp z2 = y2; Exp z3 = new Neg(z2);
        this.counter = 0; this.expr1 = z1;
        this.expr2 = z3; }
}
LANPE => LSPE {
    Neg->Sub { pre:
        Exp y = this.expr;
        post:
        Exp z1 = new Lit(0); Exp z2 = y;
        this.counter = 0; this.expr1 = z1; this.expr2 = z2; }
    Add->Sub { pre:
        Exp y1 = this.expr1; Exp y2 = this.expr2;
        post:
        Exp z1 = y1; Exp z2 = y2;
        Exp z3 = new Lit(0); Exp z4 = new Sub(z4,z3);
        this.counter = 0; this.expr1 = z1;
        this.expr2 = z4; }
}
LSPE => LASPE { }
Example: reconfiguration LASPE => LANPE in the evolved Dynamic EPL

\[(3 + 4) + ((7 - 8) + 9)\] becomes \[(3 + 4) + ((7 + (-8)) + 9)\]

Affects classes

- **ExpScanner** (without reallocating its object)
- **Sub** (by dropping it and reallocating its object to class **Add**)

```
Add
  expr1
  counter=...

Add
  expr1
  counter=...

Add
  expr1
  counter=...

Add
  expr1
  counter=...

Lit
  value=3
  counter=...

Lit
  value=4
  counter=...

Lit
  value=7
  counter=...

Lit
  value=8
  counter=...

Lit
  value=9
  counter=...
```

```
expr2
```

```
expr2
```

```
expr2
```

```
expr2
```

```
expr2
```

```
expr2
```

```
expr2
```

```
expr2
```

```
expr2
```

```
expr2
```

Runtime reconfiguration: changing the configuration of the running product

- A class C is **affected** by a reconfiguration if
  - either C is **recoded** (the code of C or of a superclass of C is modified)
  - or C is **reallocated** (the instances of C are changed)

- A reconfiguration is **enabled** if
  - classes of receivers of methods on the call stack are not affected by the reconfiguration.
Lazy Heap Update

Each object is reallocated when the running product accesses it.

- The queue of the \((n \geq 0)\) pending reconfigurations is maintained.

- The heap is partitioned in \(n+1\) regions.
  - 0-st region contains non-reallocated objects,
  - 1-nd region contains objects reallocated by the 1-st pending reconfiguration,
  - \(\vdots\)
  - \(i\)-th region \((0 \leq i \leq n)\) contains objects updated by pending reconfigurations \(1, \ldots, i-1\) (the \(n\)-th region contains fully reallocated objects).
STARTING POINT: Imperative Featherweight Delta Java (IFΔJ): a calculus for SPLs of IMPERATIVE FEATHERWEIGHT JAVA programs [Bettini et al., Acta. Inf., 2013]

- Product generation
- Type system ensuring type safety

THIS PAPER: Extends IFΔJ to dynamic DOP:
  1. Dynamic reconfiguration graph
  2. Typing rules for the dynamic reconfiguration graph
  3. Operational semantics with lazy heap (small step reduction)
  4. Type soundness (subject reduction + progress)
IFJD: Syntax of Classes and Delta Modules

Imperative Featherweight Java (IFJ)

```plaintext
CD ::= class C extends C { FD; MD }    classes
FD ::= C f                           fields
MD ::= C m (C x){return e;}           methods
e ::= x | e.f | e.m(e) | new C() | (C)e | expressions
e.f = e | null                         
```

Imperative Featherweight Delta Java (IFJD)

```plaintext
DM ::= delta δ { CO }    delta modules
CO ::= adds CD | modifies C [extending C] { AO }    class operations
AO ::= adds FD | adds MD | attribute operations
modifies MD | removes a                          
```
IFΔJ: Syntax of Classes and Delta Modules

Imperative Featherweight Java (IFJ)

\[
\begin{align*}
\text{CD} & ::= \text{class } C \text{ extends } C \{ \overline{\text{FD}}; \overline{\text{MD}} \} & \text{classes} \\
\text{FD} & ::= C f & \text{fields} \\
\text{MD} & ::= C m (\overline{C} \overline{x}) \{ \text{return } e; \} & \text{methods} \\
\text{e} & ::= x | \text{e.f} | \text{e.m(\overline{e})} | \text{new } C() | (C)e & \text{expressions} \\
& \hspace{1cm} \text{e.f = e} | \text{null} | \text{original} \\
\end{align*}
\]

Imperative Featherweight Delta Java (IFΔJ)

\[
\begin{align*}
\text{DM} & ::= \text{delta } \delta \{ \overline{\text{CO}} \} & \text{delta modules} \\
\text{CO} & ::= \text{adds } \text{CD} | \text{adds } \text{MD} | \text{attribute operations} \\
& \hspace{1cm} \text{modifies } C [\text{extending } C] \{ \overline{\text{AO}} \} | \text{removes } C \\
\text{AO} & ::= \text{adds } \text{FD} | \text{adds } \text{MD} | \text{attribute operations} \\
& \hspace{1cm} \text{modifies } \text{MD} | \text{removes } a
\end{align*}
\]
IFDΔJ: Dynamic Reconfiguration Graph

Syntax

\[ R ::= \psi \Rightarrow \psi' \{ \overline{OR} \} \quad \text{reconfiguration declarations} \]

\[ OR ::= C \rightarrow C' \{ \overline{pre: Ay = p;} \]

\[ \overline{\text{post: Bz = q;} \text{ this.f = z;}} \} \quad \text{object reallocations} \]

\[ p ::= \text{this} | p.f \quad \text{pre-reconfiguration expressions} \]

\[ q ::= y \mid \text{null} \mid \text{new C(z)} \quad \text{post-reconfiguration expressions} \]
IFDΔJ: Dynamic Reconfiguration Graph

Syntax

\[
\begin{align*}
R & ::= \bar{\psi} \Rightarrow \bar{\psi}' \{\text{OR}\} \\
\text{OR} & ::= C \rightarrow C' \{\text{pre}: A y = p; \text{post}: B z = q; \text{this}.f = z;\} \\
p & ::= \text{this} | \text{p}.f \\
q & ::= y | \text{null} | \text{new } C(\bar{z})
\end{align*}
\]

reconfiguration declarations
object reallocations
pre-reconfiguration expressions
post-reconfiguration expressions

Typing

1. If (in \(\bar{\psi}\)) \(C\) is a subclass of \(C_0\) and \(C_0\) is not removed by \(R\), then (in \(\bar{\psi}'\)) \(R(C)\) is a subclass of \(C_0\).

2. If \(y\) of type \(A\) is assigned to \(z\) of type \(B\), then the objects of every subclass \(D\) of \(A\) (in \(\bar{\psi}\)) are reallocated to a class \(D'\) that is a subclass of \(B\) (in \(\bar{\psi}'\)).
Addresses, ranged over by the metavariable $\iota$, are the elements of the denumerable set $\mathbb{I}$.

Values, ranged over by the metavariable $v$, are either addresses or null.

Objects are denoted by $\langle \mathcal{C}, \vec{f} = \vec{v} \rangle$, where $\mathcal{C}$ is the class of the object, $\vec{f}$ are the names of the fields and $\vec{v}$ are the values of the fields.

A stack $\vec{\iota}$ is a possibly empty sequence of addresses (possibly containing duplicates). The empty stack is denoted by $\bullet$.

A heap $\mathcal{H}$ is a mapping from addresses to objects. The empty heap is denoted by $\emptyset$. 
Lazy heaps are defined as follows:

\[ L ::= \mathcal{H} \mid \mathcal{H} : R(L) \]

That is, a lazy heap is either a heap \( \mathcal{H} \) or a partially reconfigured heap of the form \( \mathcal{H}_n : R_n(\mathcal{H}_{n-1} : R_{n-1}(\cdots \mathcal{H}_1 : R_1(\mathcal{H}_0) \cdots )) \), for some \( n \geq 1 \), where

- \( \mathcal{H}_n \) is the part of heap that has been reconfigured by \( R_n, \ldots, R_1 \) and that may have been subsequently modified by the execution of non-reconfiguration operations.

- Each \( \mathcal{H}_i \) (\( 1 \leq i \leq n-1 \)) is the part of heap that has been reconfigured by \( R_i, \ldots, R_1 \) and that may have been subsequently modified by the execution of non-reconfiguration operations before the invocation of \( R_{i+1} \).

- \( \mathcal{H}_0 \) is the heap before the invocation of \( R_1 \).
IFD\(\Delta J\): semantics: reduction rules

\[(R\text{-EVAL})\]
\[
\bar{\psi}, \mathcal{L}, \bar{\imath}, e \rightarrow \bar{\psi}, \mathcal{L}', \bar{\imath}', e'
\]

\[
\bar{\psi}, \mathcal{L}, \bar{\imath}, e \rightarrow \bar{\psi}, \mathcal{L}', \bar{\imath}', e'
\]

\[(R\text{-RECONF})\]
\[
R = \bar{\psi} \Rightarrow \bar{\psi}'\{\cdots\} \quad \text{Enabled}(R, \mathcal{L}, \bar{\imath}, e)
\]

\[
\bar{\psi}, \mathcal{L}, \bar{\imath}, e \rightarrow \bar{\psi}', \emptyset : R(\mathcal{L}), \bar{\imath}, e
\]
IFDΔJ: semantics: computation rules

(C-NEW)

\[
\begin{align*}
\text{I fresh} & \quad \text{fields}_{\psi}(C) = \overline{C} \ \overline{\bar{f}} \\
\_ , L, \bar{i} , \text{new} \ C(\_ ) & \rightarrow \_ , L \cup \{ \bar{i} \mapsto \langle C, \bar{f} = \text{null} \rangle \} , \bar{i} , \bar{l}
\end{align*}
\]

(C-FIELD)

\[
\begin{align*}
olookupup(\bar{i}, L) &= \langle C, \bar{f} = \bar{v} \rangle , L' \\
\_ , L, \bar{i} , \bar{l} . f_i & \rightarrow \_ , L' , \bar{i} , v_i
\end{align*}
\]

(C-ASSIGN)

\[
\begin{align*}
olookupup(\bar{i}, L) &= \langle C, \bar{f} = \bar{v} \rangle , L' \\
\_ , L, \bar{i} , \bar{l} . f_i = v & \rightarrow \_ , L'[\bar{i} \mapsto \langle C, \ldots , f_i = v , \ldots \rangle] , \bar{i} , v
\end{align*}
\]

(C-INVK)

\[
\begin{align*}
olookupup(\bar{i}, L) &= \langle C, \ldots \rangle , L' \\
meth_{\psi}(m, C) &= \_ m(\_ x)\{ \text{return} e_0; \} \\
\_ , L, \bar{i} , \bar{l} . m(\bar{v}) & \rightarrow \_ , L' , \bar{i} l , \text{return}([\bar{x} / \bar{v} , \text{this} / \bar{i}] e_0)
\end{align*}
\]

(C-RET)

\[
\begin{align*}
\_ , L, \bar{i} l , \text{return} (v) & \rightarrow \_ , L, \bar{i} , v
\end{align*}
\]
IFDΔJ: type soundness

Type soundness

Let $L = (\psi, \Phi, DMT, \Delta, \prec, RG)$ be a well-typed IFDΔJ dynamic SPL. If $\psi, \emptyset, \bullet, e$ is the initial state for a valid product $CT_\psi$ and $\psi, \emptyset, \bullet, e \xrightarrow{*} \psi', L', i, e' \xrightarrow{}$, then $e'$ is either

- null, or
- an address $i$ such that $L(i) = \langle C, \bar{f} = \bar{v} \rangle$ with $C <: \psi' C_{Main}$, or
- an expression containing either $null.f$ or $null.f = v$ or $null.m(\bar{v})$ for some $f, v, m,$ and $\bar{v}$. 
Some Recent Related Work

- Dynamic classes: runtime updates in distribute OO systems [Johnsen et al., FM 2009]

- Direct semantics for FOP SPL [Apel et al., J. of Automated Software Engineering, 2010]

- Aspect-based dynamic software update extraction [Cech Previtali and Gross, AOSD 2011]

- JAVAdaptor [Pukall et al. 2013]
Conclusion

Summary:

- Dynamic Delta-Oriented Programming

Future Work:

- Case studies (application to autonomic systems?)
- A *direct semantics* for Dynamic DOP [Apel et al., J. ASE, 2010]
- Extension to *distribute systems* [Johnsen et al., FM 2009]
- Developing a proof system for Dynamic DOP [Damiani et al., FMSPLS@SPLC 2012]
- A general paradigm for Unanticipated Software Evolution
  - Programming languages where code update is a first class construct