On Type Checking Delta-Oriented Product Lines

Ferruccio Damiani Michael Lienhardt

{ferruccio.damiani,michael.lienhardt}@di.unito.it
University of Torino (Italy)

COST Action ARVI meeting

June 5, 2016

http://hyvar-project.eu/

This presentation: mainly about the iFM 2016 paper

- A modular and tunable type system for delta-oriented programming

Structure

- Introduction
  - Software Product Line, delta-oriented programming, core calculus
- Approach
- Type System
- Future Work:
  - We need RV

Introduction

Software Product Line (SPL) **[Clements and Northrop, 2001]**

A family of programs (called variants) generated from a common artifact base

Introduction

Software Product Line (SPL) [Clements and Northrop, 2001]

A family of programs (called variants) generated from a common artifact base

Delta-Oriented Programming (DOP) [Schaefer et al., SPLC 2010]

An approach for implementing SPLs

Introduction

Structure of an SPL [Czarnecki and Eisenecke, 2000]

- **Feature Model** (FM)
  - Features (a feature represents an abstract description of functionality)
  - Products (a product is a set features)
- **Artifact Base** (AB)
  - Set of reusable code artifacts
- **Configuration Knowledge** (CK)
  - Connects FM and AB (induces a mapping from products to variants)

Introduction

Structure of an SPL [Czarnecki and Eisenecke, 2000]

- **Feature Model** (FM)
  - Features (a feature represents an abstract description of functionality)
  - Products (a product is a set features)
- **Artifact Base** (AB)
  - Set of reusable code artifacts
- **Configuration Knowledge** (CK)
  - Connects FM and AB (induces a mapping from products to variants)

DOP**: aimed at flexibility and modularity** [Schaefer and Damiani, FOSD 2010]

- The Artifact Base
  - Base program (a Java program)
  - Deltas (sets of class operations)
    adds, **removes**, **modifies** classes and attributes
- The Configuration Knowledge
  - α : activation conditions of deltas
  - $<$: application order between deltas

Structure of an SPL [Czarnecki and Eisenecke, 2000]

- **Feature Model** (FM)
  - Features (a feature represents an abstract description of functionality)
  - Products (a product is a set features)
- **Artifact Base** (AB)
  - Set of reusable code artifacts
- **Configuration Knowledge** (CK)
  - Connects FM and AB (induces a mapping from products to variants)

DOP: aimed at flexibility and modularity [Schaefer and Damiani, FOSD 2010]

- The Artifact Base
  - Base program (a Java program)
  - Deltas (sets of class operations) **adds**, **removes**, **modifies**
- The Configuration Knowledge
  - α : activation conditions of deltas
  - $<$: application order between deltas

Automated generation:

1. INPUT: selected features (product)
2. OUTPUT: variant generated by applying the activated deltas in the order

Introduction

DOP calculus: Imperative Featherweight Java + DOP constructs

DOP calculus: IMPERATIVE FEATHERWEIGHT JAVA + DOP constructs

The IFJ calculus

P	::=	$\overline{CD}$	Program
CD	::=	**class** C **extends** C { $\overline{AD}$ }	Class
AD	::=	FD | MD	Attribute (Field or Method)
FD	::=	C f	Field
MD	::=	C m($\overline{\mathtt{C}\ \mathtt{x}}$) {**return** e; }	Method
e	::=	x | e.f | e.m($\overline{e}$) | **new** C() | (C)e | e.f = e | **null**	Expression

Introduction

DOP calculus: IMPERATIVE FEATHERWEIGHT JAVA + DOP constructs

The IFJ calculus

P	$::= \overline{CD}$	Program
CD	$::=$ **class** $\mathtt{C}$ **extends** $\mathtt{C}$ $\{\ \overline{AD}\ \}$	Class
AD	$::= FD \mid MD$	Attribute (Field or Method)
FD	$::= \mathtt{C}\ \mathtt{f}$	Field
MD	$::= \mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}\ \mathtt{x}})$ $\{$**return** e; $\}$	Method
e	$::= \mathtt{x} \mid e.\mathtt{f} \mid e.\mathtt{m}(\overline{e}) \mid$ **new** $\mathtt{C}() \mid (\mathtt{C})e \mid e.\mathtt{f} = e \mid$ **null**	Expression

The IFΔJ calculus

L	$::=$ FM CK AB	Product Line
AB	$::=$ FJ $\overline{\Delta}$	Artifact Base
Δ	$::=$ **delta** $\mathtt{d}$ $\{\ \overline{CO}\ \}$	Delta Module
CO	$::=$ **adds** CD $\mid$ **removes** $\mathtt{C}$ $\mid$ **modifies** $\mathtt{C}$ [**extends** $\mathtt{C}'$] $\{\ \overline{AO}\ \}$	Class Operation
AO	$::=$ **adds** AD $\mid$ **modifies** MD $\mid$ **removes** $\mathtt{a}$	Attribute Operation

Example

```
class Int extends Object { ... }

delta dExpression {

  adds class Exp extends Object {
    Int eval() { return 0; }
  }

  adds class Add extends Exp {
    Exp l, r;
    Int eval() { return l.eval() + r.eval(); }
  }

  modifies Int extends Exp {
    adds Int eval() { return this; }
  }
}
```

Base Program

`dExpression` delta

Adds Classes `Exp` and `Add`

Modifies the Super Class of `Int`

Adds the method `eval` to `Int`

Problem and Approach

Type Checking an SPL

- Ensure that all Variants can be Generated
- Ensure that all Variants are well-typed

Problem and Approach

Type Checking an SPL

- Ensure that all Variants can be Generated
- Ensure that all Variants are well-typed

Previous approaches [Bettini et al., Acta Inf. 1013] [Damiani and Shaefer, ISoLA 2012]

- implemented for ABS modeling language [Johnsen et al., 2012] by Radu Muschevici `https://github.com/abstools/abstools/tree/master/frontend/src/abs/frontend/delta`
- require to iterate over every variant ⇒ exponential complexity

Problem and Approach

Type Checking an SPL

- Ensure that all Variants can be Generated
- Ensure that all Variants are well-typed

Previous approaches [Bettini et al., Acta Inf. 1013] [Damiani and Shaefer, ISoLA 2012]

- implemented for ABS modeling language [Johnsen et al., 2012] by Radu Muschevici https://github.com/abstools/abstools/tree/master/frontend/src/abs/frontend/delta
- require to iterate over every variant $\Rightarrow$ exponential complexity

The new approach

- Reduce Type Checking to SAT (as proposed by [Thaker, et al., GPCE 2007] [Delaware et al., ESEC/FSE 2009] for FOP)
  - co-NP hard
  - Euristics in SAT solvers
- Modular: three independent analysis
- Tunable: DOP guidelines

Problem and Approach

Modular type system

- Partial Typing

  ⇒ Checks that *declaration* and *usage* match
  - "Standard" type checking algorithm
- Applicability

  ⇒ Checks that variants can be generated
  - SAT constraint
- Dependency

  ⇒ Checks that there are no missing dependencies in generated variants
  - SAT constraint

Problem and Approach

Tunable type system: DOP guidelines

G1: no useless operations

- Useless operation: declaration in the base program and adds or modifies operations in deltas that introduce code that is never present in any of the variants

Problem and Approach

Tunable type system: DOP guidelines

G1: no useless operations

- Useless operation: declaration in the base program and adds or modifies operations in deltas that introduce code that is never present in any of the variants

G2: type uniformity

- Type uniformity: all versions of an attribute "a" accessible from a class "C" **must** have the same type in the artifact base
  - ⇒ No alternatives
  - ⇒ Simplifies the SPL (and the type system)

Problem and Approach

Tunable type system: DOP guidelines

G1: no useless operations

- Useless operation: declaration in the base program and adds or modifies operations in deltas that introduce code that is never present in any of the variants

G2: type uniformity

- Type uniformity: all versions of an attribute "a" accessible from a class "C" **must** have the same type in the artifact base
  - ⇒ No alternatives
  - ⇒ Simplifies the SPL (and the type system)

Variant type uniformity: all versions of an attribute "a" accessible from a class "C" **must** have the same type in all variants

Problem and Approach

Tunable type system: DOP guidelines

G1: no useless operations

- Useless operation: declaration in the base program and adds or modifies operations in deltas that introduce code that is never present in any of the variants

G2: type uniformity

- Type uniformity: all versions of an attribute "a" accessible from a class "C" **must** have the same type in the artifact base
  - $\Rightarrow$ No alternatives
  - $\Rightarrow$ Simplifies the SPL (and the type system)

Variant type uniformity: all versions of an attribute "a" accessible from a class "C" **must** have the same type in all variants

- G2 $\Rightarrow$ variant uniformity
- (G1 $\wedge$ variant uniformity) $\Rightarrow$ G2

Problem and Approach

Tunable type system: DOP guidelines

G1: no useless operations

- Useless operation: declaration in the base program and adds or modifies operations in deltas that introduce code that is never present in any of the variants

G2: type uniformity

- Type uniformity: all versions of an attribute "a" accessible from a class "C" **must** have the same type in the artifact base
  - ⇒ No alternatives
  - ⇒ Simplifies the SPL (and the type system)

Variant type uniformity: all versions of an attribute "a" accessible from a class "C" **must** have the same type in all variants

- G2 ⇒ variant uniformity
- (G1 ∧ variant uniformity) ⇒ G2

Other guidelines...

1. Partial Typing

Property

⇒ Ensure that Declaration and Usage Match

Example

```
class Int extends Object { ... }

delta dExpression {
  ...
  adds class Add extends Exp {
    Exp l, r;
    Int eval() { return l.eval() + r.eval(); }
  }
  ...
}
```

Must exist (highlighted: Exp)

Must exist and have type () → `Int` (highlighted: eval())

1. Partial Typing

Property

⇒ Ensure that Declaration and Usage Match

Example

```
class Int extends Object { ... }

delta dExpression {
  ...
  adds class Add extends Exp {
    Exp l, r;
    Int eval() { return l.eval() + r.eval(); }
  }
  ...
}
```

Must exist (highlighted: Exp)
Must exist and have type () → `Int` (highlighted: eval())

Algorithm

⇒ Similar to IFJ typing

- Typing Environment is
  all Declarations
  from Base Program and Deltas

2. Applicability

Property

$\Rightarrow$ Ensure that Every Variant can be Generated

Code Generation

1. Keep only Deltas activated by the Selected Features
2. Apply them in order

$$\Rightarrow \text{Application Constraints} = \begin{pmatrix} \textbf{adds}\ \rho \text{ needs } \rho \text{ absent} \\ \textbf{removes}\ \rho \text{ needs } \rho \text{ present} \\ \textbf{modifies}\ \rho \text{ needs } \rho \text{ present} \end{pmatrix}$$

Algorithm

- Generate SAT Constraints for each Operation

2. Applicability

adds

$$\mathsf{appADD}(\rho) \triangleq \bigwedge_{\mathtt{d} \neq \mathtt{d}'} \mathtt{d} \wedge \mathtt{d}' \Rightarrow \bigvee_{\mathtt{d}''} \mathtt{d}'' \quad \text{with} \begin{cases} \mathtt{d}, \mathtt{d}' \in \mathsf{add}(\rho) \text{ , } \mathtt{d}'' \in \mathsf{remove}(\rho) \\ \mathtt{d} <_L \mathtt{d}'' <_L \mathtt{d}' \end{cases}$$

If two deltas add ρ

there must exist a delta
in between that removes it

2. Applicability

there must be a delta adding before ρ

removes

$$\mathrm{appRM}(\rho) \triangleq \bigwedge_{\mathtt{d}} \mathtt{d} \Rightarrow \left(\bigvee_{\mathtt{d}_1} \mathtt{d}_1 \wedge \bigwedge_{\mathtt{d}'} (\mathtt{d}' \Rightarrow \bigvee_{\mathtt{d}_2} \mathtt{d}_2)\right) \quad \text{with} \begin{cases} \mathtt{d}, \mathtt{d}' \in \mathrm{remove}(\rho) \\ \mathtt{d}_1, \mathtt{d}_2 \in \mathrm{add}(\rho) \\ \mathtt{d}_1 <_L \mathtt{d} <_L \mathtt{d}_2 <_L \mathtt{d}' \end{cases}$$

If there is a delta removing ρ

and if there is another delta removing ρ

there must be one in between that adds it

2. Applicability

there must exist a delta before adding ρ

modifies

$$\text{appMOD}(\rho) \triangleq \bigwedge_{\mathtt{d}} \mathtt{d} \Rightarrow \left(\bigvee_{\mathtt{d}'} \mathtt{d}' \wedge \bigwedge_{\mathtt{d}''} \neg \mathtt{d}'' \right) \quad \text{with} \begin{cases} \mathtt{d} \in \text{modify}(\rho) \\ \mathtt{d}' \in \text{add}(\rho) \\ \mathtt{d}'' \in \text{remove}(\rho) \\ \mathtt{d}' <_L \mathtt{d}'' <_L \mathtt{d} \end{cases}$$

If there is a delta modifying ρ

with no **removes** ρ in between

2. Applicability

Applicability: Global Constraint

L is *applicability-consistent* iff Φ is **valid**

$$\Phi \triangleq L.\texttt{FMandCK} \Rightarrow \bigwedge_{\rho \in \mathsf{add}(L)} \mathsf{appADD}(\rho) \wedge \mathsf{appRM}(\rho) \wedge \mathsf{appMOD}(\rho)$$

All the products of L

3. Dependency

Property

⇒ Ensure that Variant's code have no **Missing Dependencies**

Example

```
class Int extends Object { ... }

delta dExpression {
  ...
  adds class Add extends Exp {
    Exp l, r;
    Int eval() { return l.eval() + r.eval(); }
  }
  ...
}
```

(Highlighted: Exp, Exp, Int, eval(), eval()) Must be present when `dExpression` is activated

3. Dependency

Property

⇒ Ensure that Variant's code have no **Missing Dependencies**

Example

```
class Int extends Object { ... }

delta dExpression {
  ...
  adds class Add extends Exp {
    Exp l, r;
    Int eval() { return l.eval() + r.eval(); }
  }
  ...
}
```

Must be present when dExpression is activated (highlighted: Exp, Exp, Int, eval(), eval())

Algorithm

- Generate SAT Constraint for each Dependency
- Induction over all the declarations

3. Dependency

Induction Rules: for Expressions

$$\Gamma \vdash e : T \mid \Phi$$

Typing Environment:
for Dependencies on C.a

Generated Constraint

Analyzed Expression

Type of e

3. Dependency

Examples:

(E:Field)

$$\frac{\Gamma \vdash e : \texttt{C} \mid \Phi \qquad \mathsf{type}(\texttt{C.f}) = \texttt{C}'}{\Gamma \vdash e.\texttt{f} : \texttt{C}' \mid \Phi \wedge \mathsf{decl}(\texttt{C.f})}$$

(E:Call)

$$\frac{\Gamma \vdash e : \texttt{C} \mid \Phi \qquad \mathsf{type}(\texttt{C.m}) = (\texttt{C}_1, \ldots, \texttt{C}_n) \rightarrow \texttt{C}' \qquad \Gamma \vdash e_i : T_i \mid \Phi_i \qquad \Phi'_i = \mathsf{sub}(T_i, \texttt{C}_i)}{\Gamma \vdash e.\texttt{m}(e_1, \ldots, e_n) : \texttt{C}' \mid \bigwedge_i (\Phi_i \wedge \Phi'_i) \wedge \Phi \wedge \mathsf{decl}(\texttt{C.m})}$$

Constraint encoding when C.f is present

Arguments must be a subtype of Formal Parameter

C.m must be present

3. Dependency

Induction Rules: for Declaration

$$\Phi \vdash D : \Phi$$

Declaration Environment:
Contains

- `d`:Analyzed Delta
- `C`: analyzed Class

Analyzed Declaration

Generated Constraint

3. Dependency

Examples:

If the Field is not removed

$\mathtt{C}'$ must be present

(D:FIELD)

$$\mathtt{d}, \mathtt{C} \vdash \mathtt{C}'\ \mathtt{f} : \neg \mathsf{rm}(\mathtt{d}, \mathtt{C.f}) \Rightarrow \mathsf{decl}(\mathtt{C}')$$

3. Dependency

Examples:

If the Field is not removed

C′ must be present

(D:FIELD)

$$\mathtt{d}, \mathtt{C} \vdash \mathtt{C'}\ \mathtt{f} : \neg\mathsf{rm}(\mathtt{d}, \mathtt{C.f}) \Rightarrow \mathsf{decl}(\mathtt{C'})$$

If the Class is not removed

then its inner dependencies must hold

and if its super is not changed

C′ must exist and is not a subtype of C

(D:CLASS)

$$\frac{\mathtt{d}, \mathtt{C} \vdash AD_i : \Phi_i}{\mathtt{d} \vdash \textbf{class}\ \mathtt{C}\ \textbf{extends}\ \mathtt{C'}\ \{AD_1 \ldots FD_n\}}$$

$$: \neg\mathsf{rm}(\mathtt{d}, \mathtt{C}) \Rightarrow \bigwedge_i \Phi_i \wedge (\neg\mathsf{modifyEC}(\mathtt{d}, \mathtt{C}) \Rightarrow \mathsf{decl}(\mathtt{C'}) \wedge \neg\mathsf{sub}(\mathtt{C'}, \mathtt{C}))$$

Theorem

$$\text{Partially Typed} + \text{Applicability} + \text{Dependency}$$
$$\Leftrightarrow$$
$$\text{SPL type checks}$$

Theorem

Partially Typed + Applicability + Dependency
$\Leftrightarrow$
SPL type checks

- Prototype Implementation at https://github.com/gzoumix/IFDJTS

Future Work

- A better implementation.
  E.g., for ABS, for DOP on full Java [Koscielny et al., PPPJ 2014]
- Case studies
- Delta-oriented Multi SPLs

Future Work

Starting from:

- Delta-oriented dynamic SPLs [Damiani et al, GPCE 2012]
  - Extends DOP with a dynamic reconfiguration graph (runtime reconfiguration of code and heap)
  - Ensures type safety
- Formal verification of delta-oriented SPLs (ongoing work [Hähnle and Schaefer, ISoLA 2012] [Damiani et al, FMSPLE 2012 @ SPLC],...)

use RV techniques to

- Monitor the behavior of a variant
- Decide how and when trigger a reconfiguration

Thanks