Commonalities for Preserving Consistency of Multiple Models

Heiko Klare, Joshua Gleitze
VoSE Workshop @ MODELS 2019, 15.09.2019
A Simple Consistency Scenario

A consistency-preserving transformation:

```java
relation Class2ADLComponent {
    componentName : String;
    domain java class : Class {
        name = componentName + "Impl"
        packageName = componentName
    }
    domain uml component : Component {
        name = componentName
    }
}
```

Java

- **Class**
  - name
  - packageName

UML

- **Component**
  - name

«maps to»

Class.name = Component.name + "Impl"
Class.packageName = Component.name
Multi-Model Consistency: Dense Graphs

Option 1: Define relations between all pairs of metamodels

Java

Class
- name
- packageName

«maps to»

Component
- name

Class.name = Component.name + "Impl"
Class.packageName = Component.name

UML

Component
- name

«maps to»

ADL

Component
- name

Class.name = Component.name
Class.packageName = Component.name

«maps to»

Bad compatibility

i.e. consistency relations/constraints may be contradicting

[Klare2018, Gleitze2017]
Multi-Model Consistency: Trees

**Scenario:** Develop a system only with the ADL and Java

**Option 2:** Define relations between metamodels to form a tree

```
Java
Class
   name
   packageName
```

```
UML
Component
   name
```

```
ADL
Component
   name
```

```
Option 2: Define relations between metamodels to form a tree
```

```
Bad modularity
i.e. no arbitrary metamodel selection without loosing consistency relations
[Klare2018, Gleitze2017]
```

```
UML
Component
   name
```

```
Class «maps to» Component
Class.name = Component.name + "Impl"
Class.packageName = Component.name
```

```
ADL
Component «maps to» Component
Component.name = Component.name
```

```
Java
Class
   name
   packageName
```

```
Java
Class
   name
   packageName
```

```
UML
Component
   name
```

```
ADL
Component
   name
```

```
Bad modularity
i.e. no arbitrary metamodel selection without loosing consistency relations
[Klare2018, Gleitze2017]
```

```
Java
Class
   name
   packageName
```

```
UML
Component
   name
```

```
ADL
Component
   name
```

```
Option 2: Define relations between metamodels to form a tree
```

```
Bad modularity
i.e. no arbitrary metamodel selection without loosing consistency relations
[Klare2018, Gleitze2017]
```
Trade-off: Compatibility vs. Modularity

Network topologies

Dense Graph

Tree

Compatibility
(Relations/transformations do not contradict each other)
Maximize: only one path between each metamodel pair

Modularity
(Possibility to select arbitrary metamodel subset without losing consistency relations)

Maximize: $\theta(n^2)$ paths between each metamodel pair

[Klare2018, Gleitze2017]
## Contributions and Expected Benefits

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trade-off between <em>compatibility</em> and <em>modularity</em> in bidirectional transformation networks.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Idea</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resolve the trade-off between <em>compatibility</em> and <em>modularity</em> for consistency of multiple models by making common concepts of metamodels explicit.</td>
</tr>
</tbody>
</table>

### Contributions

- Commonalities Approach
- Commonalities Language
- Proof of Concept

### Expected Benefits

- Improved comprehensibility
- Reduced specification effort
- Improved compatibility and modularity
From Relations to Explicit Common Concepts

Object-oriented Design

Class
name

Explicit specification of common concept

«manifests»

«manifests»

Java

Class
name

UML

Class
name

«maps to»

Implicit specification of common concept in consistency relation
The Commonalities Approach

- Encode commonalities in metaclasses of a conceptual metamodel
- Represent common information in features of the conceptual metamodel

```
<table>
<thead>
<tr>
<th>Concept metamodel</th>
<th>Commonality</th>
<th>Object-oriented Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commonality</td>
<td>Bidirectional transformation</td>
<td></td>
</tr>
<tr>
<td>Concrete metamodel</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“Java is a *manifestation* of Object-oriented Design”

«manifests»

Java

```
<table>
<thead>
<tr>
<th>Class</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td></td>
</tr>
</tbody>
</table>
```

UML

```
<table>
<thead>
<tr>
<th>Class</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML</td>
<td></td>
</tr>
</tbody>
</table>
```
Hierarchic Composition of Commonalities

Naïve: One monolithic concept metamodel
Better: Hierarchy of Commonalities
Goal: Tree of Commonalities (no cycles)

Class.name = Component.name + “Impl”
Package.name = Component.name

Object-oriented Design

Component-based Design

Java

Class
packageName
name

UML

Class
name

Component
name

ADL

Component
name
Design Decisions

Artifact Generation (transparent to user)

Concept metamodels as additional metamodels

Concept metamodel $\rightarrow$ Metamodel
Commonality $\rightarrow$ Metaclass
Manifestation specification $\rightarrow$ Transformation

Alternative: Derive direct transformations between concrete metamodels

Benefits:
- Easy to achieve
- High expressiveness ($n$-ary relations)

Drawback:
- Management of additional artifacts

Commonalities Specification (visible to user)

Internal specification

Integrated definition of concept metamodels with manifestations

Decomposition dimension: Commonalities

Alternative: External specification
(Decomposition dimension: Transformations)

Benefits:
- Easy to add Commonalities
- Improved locality / conciseness

Drawback:
- More difficult to add metamodels
Commonalities Language

```plaintext
concept Components

commonality Component {
  with UML:Component
  with ObjectOrientation:(Class in Package)

  has name {
    = UML:Component.name
    = ObjectOrientation:Package.name
    = \textit{suffix}(ObjectOrientation:Class.name, "Impl")
  }

  has subcomponent \texttt{referencing} Components:Component {
    = UML:Component.packagedElement
    = ObjectOrientation:Package.subpackages
  }
}
```
Proof of Concept

Case Study *(schematic)*

Feasibility

- Test cases performing all possible types of model modifications
- Correct propagation of all changes $\rightarrow$ indicator for functional correctness
Benefit: Comprehensibility

concept Components

commonality Component {
    with uml:Component
    with adl:Component
    with java:Class

    has name {
        = uml:Component.name
        = adl:Component.name
        = java:Class.packageName
        = suffix(java:Class.name, "Impl")
    }
}

relation UMLComponent2ADLComponent {
    componentName : String;
    domain uml component : Component {
        name = componentName;
    }
    domain adl component : Component {
        name = componentName;
    }
}

relation Class2ADLComponent {
    componentName : String;
    domain java class : Class {
        name = componentName + "Impl";
        packageName = componentName;
    }
    domain adl component : Component {
        name = componentName;
    }
}

relation Class2UMLComponent {
    componentName : String;
    domain java class : Class {
        name = componentName + "Impl";
        packageName = componentName;
    }
    domain uml component : Component {
        name = componentName;
    }
}
Benefit: Compatibility and Modularity

**Network topologies**

With Commonalities:
High modularity because inner nodes are concept metamodels

Dense Graph

Tree

Compatibility
(Relations/transformations do not contradict each other)

Maximize: \( \theta(n^2) \) paths between each metamodel pair

Modularity
(Possibility to select arbitrary metamodel subset without losing consistency relations)

Maximize: only one path between each metamodel pair

[KLare2018, Gleitze2017]
Benefit: Specification Effort

Ordinary Transformations

Commonalities

Number of relations grows linearly with Commonalities but quadratically with transformations
Comparison with the SUM(M) Approach

**Commonalities Approach**

*Concept metamodel is union of pairwise intersections of concepts*

- Concept Metamodel
- Concept Metamodel
- Concept Metamodel

**SUM(M) Approach**

*SUM is union of concepts of all metamodels*

- SUM
Related Work

Commonalities Approaches

Practical approaches
- Sophisticated commonalities language [Gleitze2017]
- Role-oriented SUM [Werner2018]
- Domain-specific: DUALLy [Malavolta2010, Eramo2012]

Theoretic considerations
- Multiary Delta Lenses [Diskin2018]
- Commonalities for n-ary constraints [Stünkeli2018]

Multidirectional Transformations and Networks of Bidirectional Transformations
- Dagstuhl Seminar [Cleve2019]
- Constraint decomposition problems [Stevens2017]
- Language-specific: QVT-R [Macedo2014], TGG [Trollmann2016]
# Conclusion and Future Work

## Goal

Resolve the trade-off between *compatibility* and *modularity* for multi-model consistency.

## Contributions

**Commonalities Approach**
- *Concept metamodels of Commonalities*
- Manifestation relations
- Hierarchic composition of Commonalities

**Commonalities Language**
- Design options
  - Artifact generation
  - Commonalities specification
- Proof of concept implementation

## Expected Benefits

<table>
<thead>
<tr>
<th>General</th>
<th>Comprehensibility ↑</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-model case</td>
<td>Effort ↓</td>
</tr>
<tr>
<td></td>
<td>Errors ↓</td>
</tr>
<tr>
<td></td>
<td>Modularity ↑</td>
</tr>
</tbody>
</table>

## Future Work

- Extend language capabilities
- Evaluate benefits
  - *Applicability*: case study
  - *Comprehensibility*: experiment
- Validate practicality of hierarchic composition
## Bib: Commonalities

<table>
<thead>
<tr>
<th>Reference</th>
<th>Abstract</th>
</tr>
</thead>
</table>
Bib: Multidirectional Transformations/Networks


### Bib: Other Related Topics

<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
</table>
Bib: Vitruvius Approach


Design Decision: Artifact Generation

Artifact generation is transparent to user

Concept metamodels as additional metamodels

Concept metamodel → Metamodel
Commonality → Metaclass
Relation specification → Transformation

Transformations between concrete metamodels

Indirect relations across concept metamodels → Transformations between pairs of concrete metamodels

Benefits:
- Easy to achieve
- High expressiveness (n-ary relations)

Benefits:
- No management of additional artifacts
- Easier to understand direct relations
Design Decision: Commonalities Specification

Commonalities specification is visible to user

**External specification**

- Independent definition of concept metamodels and transformations

**Benefits:**
- Easy to add concrete metamodels
- Reuse existing tooling

**Internal specification**

- Integrated definition of concept metamodels with manifestations

**Benefits:**
- Easy to add Commonalities
- Improved locality / conciseness

- Decomposition dimension: Transformations
- Decomposition dimension: Commonalities
Operators in the Commonalities Language

- The Commonalities language can be extended by operators that allow bidirectional propagation of information, e.g. `in` and `suffix`.

- To have well-defined bidirectional transformations, operators must be:
  - Correct
  - Hippocratic
  - Undoable