Introduction to Model Versioning

SFM-12: MDE

June 22, 2012

Petra Brosch, Gerti Kappel,
Philip Langer, Martina Seidl,
Konrad Wieland, and Manuel Wimmer

Business Informatics Group
Institute of Software Technology and Interactive Systems
Vienna University of Technology
Favoritenstraße 9-11/188-3, 1040 Vienna, Austria
phone: +43 (1) 58801-18804 (secretary), fax: +43 (1) 58801-18896
office@big.tuwien.ac.at, www.big.tuwien.ac.at
Outline

- Context of Model Versioning
  - Foundations of Model Versioning
  - Conflict Categorization
  - Adaptable Model Versioning
- Future Challenges
Evolution

Definition

  - A process of change in a certain direction
  - A process of continuous change from a lower, simpler, or worse to a higher, more complex, or better state

- Software Evolution
  - The process of changing a software system from its creation until its shutdown.
Programs, like people, get old. We can’t prevent aging, but we can understand its causes, take steps to limits its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable. A sign that the Software Engineering profession has matured will be that we lose our preoccupation with the first release and focus on the long term health of our products. Researchers and practitioners must change their perception of the problems of software development. Only then will Software Engineering deserve to be called Engineering.

Software Evolution aka Software Maintenance

Cost Distribution in the Software-Life-Cycle

Source: Principles of Software Engineering and Design, Zelkovits, Shaw, Gannon 1979
Evolution

Is Software resistant to change? Of course, not!

- Reasons for evolution
  - Technology Switch
  - Restructuring
  - Bug Fixing
  - Functionality Extension
  - Optimization
  - …

- Bad news: A computer program that is used will be modified, thus its entropy increases.

- Silver Bullet?: Model-driven Software Engineering
  - Platform independent models
  - Generate code automatically
  - Are models resistant to change?
Everything changes…

Evolution in Model-driven Software Engineering
Evolution in Model-driven Software Engineering

Modeling Languages evolve too!
Modeling Language Evolution

Metamodels are the central artefacts!

- Textual Concrete Syntax
- Graphical Concrete Syntax
- Model 2 Model Transformations
- Model 2 Code Transformations
- Inplace Transformations
- OCL Constraints
- Models
Co-Evolution
Consequence of Metamodel Evolution

- Term “Co-Evolution” borrowed from Biology
  - Biological co-evolution is the change of a biological entity triggered by the change of a related entity
    - One-to-one Relationships: Predator/Prey, Host/Symbiont, Host/Parasite, ...
    - Diffuse Relationships: An entity evolves in response to a number of other entities, each of which is also evolving in response to a set of entities

- Co-Evolution in MDE
  - Co-evolution is the change of a model triggered by the change of a related model
  - Current View
    - Relationship: r(a,b)
      - a → a'  
      - b → b' | r(a',b')
    - Challenge: Relationship Reconciliation
  - Current research focus is on one-to-one relationships
Metamodel Evolution

Mainly models, i.e., instances of metamodels, are currently co-evolved!
Evolution in Model-driven Software Engineering

Platforms (Software and Hardware) evolve rapidly!
Platforms evolve rapidly!

Platforms are represented by metamodels

- Modeling Language → Source Metamodel \[ \text{MM}a \] \[ \text{t}_1 \] \[ \text{MM}b \] Target Metamodel → Platform

Deployment

\[ \text{t}_1 \ldots \text{Forward Transformation} \]
Metamodel/Transformation (Co-)Evolution

Platform evolution is reflected by target metamodel evolution

Source Metamodel \[ \text{MMa} \] \[ t_1 \] \[ \text{MMb} \] \[ t_2 \] \[ \text{MMb}' \] \[ t_3 \] \[ \text{MMb}'' \]

Target Metamodel

Platform Evolution

v1.0

v2.0

v3.0

\[ t_1 \] \[ \ldots \] Forward Transformation

\[ t_2, t_3 \] \[ \ldots \] Migration Transformation
Metamodel/Transformation (Co-)Evolution

Transformation chains for tackling platform evolution

Source Metamodel

\[ \text{MM}_a \]

\[ t_1 \rightarrow \text{MM}_b \]

\[ t_4 = t_3 \cdot t_2 \cdot t_1 \]

Target Metamodel

\[ \text{MM}_b' \rightarrow \text{MM}_b'' \]

Platform Evolution

\[ v1.0 \]

\[ v2.0 \]

\[ v3.0 \]

\[ t_1 \quad \text{... Forward Transformation} \]

\[ t_2, t_3 \quad \text{... Migration Transformation} \]
Evolution in Model-driven Software Engineering

Is modeling a one woman-show?

[Diagram showing the relationships between Metamodel, Model, Requirements, and Platform with arrows indicating changes.]
Evolution in Model-driven Software Engineering

No, support for team-based development is needed!
Support for Team-based Development of Models is Needed!

- Recall some definitions of Software Engineering (SE)
  - SE is defined as the **multi-person construction** of multi-version software
    
    
    \[ \text{– David Lorge Parnas, 1975} \]
  
    - SE deals with the building of software systems that are so large or so complex that they are **built by teams** of engineers
    
    \[ \text{– Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli, 2002} \]

- Assume we have four modelers (a, b, c, d)

  \[
  \begin{align*}
  m_1 &:= m_a \oplus m_b \oplus m_c \oplus m_d \\
  \text{pre} &:= \text{parallelIndependent}(\Delta_d, \Delta_c, \Delta_b, \Delta_a)
  \end{align*}
  \]
Evolution Scenarios at a Glance

- **Evolution**
  - $a \rightarrow a'$
  - Challenge: Realization & Understanding

- **Co-Evolution**
  - Consistency Relationship: $c(a, b)$
  - $a \rightarrow a'$
  - $b \rightarrow b' \mid c(a', b')$
  - Challenge: Consistency Reconciliation

- **Parallel Evolution**
  - $a \rightarrow a_1, a \rightarrow a_2$
  - $a_1 \oplus a_2 \rightarrow a'$
  - Challenge: Conflict Detection & Resolution
Outline

- Context of Model Versioning
- Foundations of Model Versioning
- Conflict Categorization
- Adaptable Model Versioning
- Future Challenges
Versioning

- **Software Configuration Management (SCM)**
  - Originated from aerospace industries in the 1950s
    - Initiated by issues coming from inadequately documented engineering changes
  - Managing and controlling
    - Corrections
    - Extensions
    - Adaptations
  - Traceability throughout the system lifecycle
  - Complex software systems pose similar challenges as other systems
    → Configuration Management for software in late 1970s
    → Software Configuration Management was born (with SCCS)

- **Versioning**
  - “Maintaining a historical archive of a set of artifacts as they undergo a series of changes”
  - Fundamental building block of SCM

---

History of Versioning Systems

Versioning Paradigms

- **Pessimistic Versioning**
  - Central repository for storing shared artifacts
  - No concurrent write access allowed (file locking)

- **Optimistic Versioning**
  - Concurrent teamwork allowed
  - Version Merging necessary
    - Conflicts might occur when same unit of comparison is changed in different ways
      - Update-update, Delete-update
    - Conflicts have to be resolved manually
Model Versioning
Overview on Optimistic Model Versioning
Model Versioning

Is getting more and more important

- Empirical study on versioning habits in practice

The Model Versioning Process Revised
Why not SVN with Unix diff?
Text-based Merging

Example

Grammar

```
Class:= "class" name=ID "{" 
  (references+=Reference)* 
  (attribute+=Attribute)* 
    
Reference:= target=[Class] "[" lower=BOUND 
  ".." upper=BOUND 
    
Attribute:= type=ID "[" lower=BOUND 
  ".." upper=CARD 
    
terminal ID:= ('a'..'z'|'A'..'Z'|'_')+; 
terminal BOUND:= (('0'..'9')+)|{"*"};
```

Version 0

```
1: class Human { 
2:   string[1..1] name  
3: } 
4: class Vehicle { 
5:   integer[0..1] carNo 
6: }
```

Version 1

```
1: class Person { 
2:   string[1..1] name 
3:   Vehicle[0..*] owns 
4: } 
5: class Vehicle { 
6:   integer[1..1] carNo 
7: }
```

Version 2

```
1: class Human { 
2:   string[1..1] name 
3: } 
4: class Car { 
5:   integer[0..1] regId 
6: }
```
Text-based and State-based Merging

Version 0

1: class Human {
2:  string[1..1] name
3: }
4: class Vehicle {
5:  integer[0..1] carNo
6: }

Version 1

1: class Person {
2:  string[1..1] name
3:  Vehicle[0..*] owns
4: }
5: class Vehicle {
6:  integer[1..1] carNo
7: }

Version 2

1: class Human {
2:  string[1..1] name
3: }
4: class Car {
5:  integer[0..1] regId
6: }

Version 3

1: class Person {
2:  string[1..1] name
3:  Vehicle[0..*] owns
4: }
5: class Car {
6:  <UP/UP> a: integer[1..1] carNo
7:  b: integer[0..1] regId
8:  c: integer[1..1] regId
9: }

1: class Person {
2:  string[1..1] name
3:  Vehicle[0..*] owns
4: }
5: class Vehicle {
6:  integer[1..1] carNo
7: }
Text-based and Operation-based Merging

Version 0

1: class Human {
2:  string[1..1] name
3: }
4: class Vehicle {
5:  integer[0..1] carNo
6: }

Version 1

1: class Person {
2:  string[1..1] name
3:  Vehicle[0..*] owns
4: }
5: class Vehicle {
6:  integer[1..1] carNo
7: }

Version 2

1: class Human {
2:  string[1..1] name
3: }
4: class Car {
5:  integer[0..1] regId
6: }

Version 3

1: class Person {
2:  string[1..1] name
3:  Car[0..*] owns
4: }
5: class Car {
6:  <<UP/UP>>
7: }

Rename-Op:
change Class.name;
update Property.type
pre@Class.name with
post@Class.name;
Graph-based Merging

Example

![Diagram showing graph-based merging of classes and attributes]

Legend:
- `<NodeName> : <Type>`
- `<attributeName> = <value>`
- Containment Edge
- Edge

Version 1
- **Human : Class**
  - `name : Attribute`
    - type = string
    - lower = 1
    - upper = 1
- **Vehicle : Class**
  - `carNo : Attribute`
    - type = integer
    - lower = 0
    - upper = 1

Version 2
- **Person : Class**
  - `name : Attribute`
    - type = string
    - lower = 1
    - upper = 1
  - `owns : Reference`
    - lower = 0
    - upper = *
- **Vehicle : Class**
  - `carNo : Attribute`
    - type = integer
    - lower = 1
    - upper = 1

**Car : Class**
- `regId : Attribute`
  - type = integer
  - lower = 0
  - upper = 1
Graph-based and State-based Merging

Heuristic-based Matching
Graph-based and State-based Merging

Match Model based on Heuristic (Name + Structure Equivalence)
Graph-based and State-based Merging

Match Model + Diff Model

Version 0:
- **Human**: Class
  - **name**: Attribute
    - type = string
    - lower = 1
    - upper = 1
- **Vehicle**: Class
  - **carNo**: Attribute
    - type = integer
    - lower = 0
    - upper = 1

Version 1:
- **Person**: Class
  - **name**: Attribute
    - type = string
    - lower = 1
    - upper = 1
  - **owns**: Reference
    - lower = 0
    - upper = *

Version 2:
- **Human**: Class
  - **name**: Attribute
    - type = string
    - lower = 1
    - upper = 1
- **Car**: Class
  - **regId**: Attribute
    - type = integer
    - lower = 0
    - upper = 1

remove(Vehicle:Class, carNo:Attribute)
Graph-based and State-based Merging

ID-based Matching

Version 0

1. Human : Class
   - name : Attribute
     - type = string
     - lower = 1
     - upper = 1

2. Car : Class
   - carNo : Attribute
     - type = integer
     - lower = 0
     - upper = 1

Version 1

1. Person : Class
   - name : Attribute
     - type = string
     - lower = 1
     - upper = 1
   - owns : Reference
     - lower = 0
     - upper = *

2. Car : Class
   - regId : Attribute
     - type = integer
     - lower = 0
     - upper = 1

Version 2

1. Person : Class
   - name : Attribute
     - type = string
     - lower = 1
     - upper = 1

2. Car : Class
   - regId : Attribute
     - type = integer
     - lower = 0
     - upper = 1

Version 3

1. Person : Class
   - name : Attribute
     - type = string
     - lower = 1
     - upper = 1
   - owns : Reference
     - lower = 0
     - upper = *

2. Car : Class
   - regId : Attribute
     - type = integer
     - lower = 0
     - upper = 1
State-based Merge Algorithm (agnostic of representation & match and diff realization)

Design Decisions

- **Matching** (hasMatch)
  - By Equivalence
  - By ID
  - By Heuristics
- **Comparison** (diff)
  - Element granularity
  - Feature granularity
- **Consolidation** (remove/add)
  - Take element from either V1 or V2
  - Evolve original element (V0) → towards operation-based merging

```plaintext
for each n0 ∈ V0
    n1 := match(n0 in V1)
    n2 := match(n0 in V2)
    if hasMatch(n0 in V1) && hasMatch(n0 in V2)
        if diff(n0, n1) && not diff(n0, n2) -> use n1
        if not diff(n0, n1) && diff(n0, n2) -> use n2
        if diff(n0, n1) && diff(n0, n2)
            -> raise update/update conflict
            if not diff(n0, n1) && not diff(n0, n2) -> use n0
        end if
    if hasMatch(n0 in V1) && not hasMatch(n0 in V2)
        if diff(n0, n1) -> raise delete/update conflict
        if not diff(n0, n1) -> remove n0
    end if
    if not hasMatch(n0 in V1) && hasMatch(n0 in V2)
        if diff(n0, n2) -> raise delete/update conflict
        if not diff(n0, n2) -> remove n0
    end if
    if not hasMatch(n0 in V1) && not hasMatch(n0 in V2)
        -> remove n0
    end if
end for
for each m1 ∈ V1 | not hasMatch(m1, V0)
    -> add m1 to merged version
for each m2 ∈ V2 | not hasMatch(m2, V0)
    -> add m2 to merged version
```
Outline

- Context of Model Versioning
- Foundations of Model Versioning
- Conflict Categorization
- Adaptable Model Versioning
- Future Challenges
Conflict Examples
Contradicting Change

Person

Person

Person

getName()
Conflict Examples

Equivalent Change
Conflict Examples

Equivalent Change

UML Metamodel V1

Class

name: String

superClasses

0..*

UML Metamodel V2

Class

name: String

superClass

1..1

subClasses

1..*

Generalization

01, 02 | 01.id = 02.id
01 ∈ Class, 02 ∈ Class | 01.name = 02.name

Model (AS)

c1: Class

name = “Person“

c2: Class

name = “Employee“

Sally

Harry

c3: Class

name = “Employee“

Model (CS)

Person

Employee

Sally

Harry

Person

Employee

Person

Employee
Conflict Examples

Syntactic Inconsistency

Car Engine

Car Engine

Car Engine

Car Engine

Car Engine

Car Engine

Car Engine
Conflict Categorization

- What is the reason for a conflict?
  - Contradicting operations
    - One operation cannot be applied after the other
    - The order in which the operations are applied affect the merge result
    → **Operation-based conflicts**
    → The set of applied operations and a specification of the characteristics of the operations is required, but no language specific knowledge

- Inconsistent resulting state
  - Inconsistent regarding abstract syntax rules
  - Inconsistent regarding the semantics
  → **State-based conflicts**
  → A merged state and consistency rules are required
Conflict Categorization

Which information/knowledge is required for detection?

- **Generic information on operations**
- **Extra Knowledge necessary**
- **Modeling Language Dependent**
Contradicting Operations: Update/Update

\[ V_0 \]
- Person \rightarrow Driving License

\[ V_{r1} \]
- Person \rightarrow \text{1} \rightarrow \text{Driving License}

\[ V_{r2} \]
- Person \rightarrow * \rightarrow \text{Driving License}

<table>
<thead>
<tr>
<th>Overlapping Operations</th>
<th>Equivalent Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contradicting Operations</td>
<td>Equivalent Operations</td>
</tr>
<tr>
<td>Update/Update</td>
<td>Add/Add</td>
</tr>
<tr>
<td>Delete/Update</td>
<td>Delete/Delete</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Language Knowledge</th>
<th>Domain Knowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metamodel</td>
<td>Common Knowledge</td>
</tr>
<tr>
<td></td>
<td>User-defined Knowledge</td>
</tr>
<tr>
<td></td>
<td>Redundancy</td>
</tr>
<tr>
<td>Different language constructs equivalent semantics</td>
<td>Different words with equivalent semantics</td>
</tr>
<tr>
<td>Well-formedness Rule</td>
<td>Upper Ontology</td>
</tr>
<tr>
<td>Abstract Syntax</td>
<td>Thesaurus</td>
</tr>
<tr>
<td>Operation Contract</td>
<td>...</td>
</tr>
<tr>
<td>Pre-/Postconditions</td>
<td>Use Case Description</td>
</tr>
<tr>
<td></td>
<td>Requirement Specification</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Equivalent Operations: Add/Add

- **V_o**
  - Employee
  - Person

- **V_{rl}**
  - Employee
  - Person

- **V_{r2}**
  - Employee
  - Person

### Overlapping Operations
- **Contradicting Operations**
  - Update/Update
  - Delete/Update
- **Equivalent Operations**
  - Add/Add
  - Delete/Delete

### Language Knowledge
- **Metamodel**
  - Well-formedness Rule
  - Abstract Syntax

### Domain Knowledge
- **Common Knowledge**
  - Upper Ontology
  - Thesaurus
  - ...
- **User-defined Knowledge**
  - Use Case Description
  - Requirement Specification
  - ...

### Violations
- **Relevancy**
  - Different words with equivalent semantics
- **Redundancy**
  - Different language constructs equivalent semantics
Equivalent Operations: Delete/Delete

\[ V_0 \]

\[ \text{Employee} \]

\[ \text{contract} \]

\[ V_{r1} \]

\[ \text{Employee} \]

\[ V_{r2} \]

\[ \text{Employee} \]
Redundancy: Language Knowledge

$V_0$

- **Developer**
- **Manager**

$V_{r1}$

- **Developer** ➔ **Manager**
  - * managedBy

$V_{r2}$

- **Developer** ➔ **Manager**
  - * managedBy

---

### Overlapping Operations

- **Contradicting Operations**
  - Pre-/Postconditions
- **Equivalent Operations**
  - Add/Add
  - Delete/Delete

---

### Language Knowledge

- **Metamodel**
  - Well-formedness Rule
  - Abstract Syntax
- **Operation Contract**
  - Pre-/Postconditions

### Domain Knowledge

- **Common Knowledge**
  - Upper Ontology
  - Thesaurus
  - ...
- **User-defined Knowledge**
  - Use Case Description
  - Requirement Specification
  - ...

---

**Violations**

- Redundancy

---

**Atomic**

- Composite / Atomic

---

**Developer**

- Manager

---

**Manager**

- * managedBy

---

**Redundancy**

- Different language constructs equivalent semantics

---

**Different words with equivalent semantics**

---

**48**
Redundancy: Language Knowledge

<table>
<thead>
<tr>
<th>Violations</th>
<th></th>
<th>Overlapping Operations</th>
<th>Equivalent Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metamodel</td>
<td>Different language constructs equivalent semantics</td>
<td>Redundancy</td>
<td>Different words with equivalent semantics</td>
</tr>
<tr>
<td>Operation Contract</td>
<td>Pre-/Postconditions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>User-defined Knowledge</td>
<td>Use Case Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Common Knowledge</td>
<td>Upper Ontology</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thesaurus</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Domain Knowledge**

**Composite / Atomic**

**Atomic**

**Language Knowledge**

- Well-formedness Rule
- Abstract Syntax

- Update/Update
- Delete/Update
- Add/Add
- Delete/Delete

Employee contract
Company

$V_o$

$V_{r1}$

Employee
Contract

$V_{r2}$

Employee
Contract
Company

Nonunique

Employee
Contract
Company

Vo

Vr1

Vr2
## Redundancy: Domain Knowledge

<table>
<thead>
<tr>
<th>Overlapping Operations</th>
<th>Equivalent Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contradicting Operations</td>
<td></td>
</tr>
</tbody>
</table>
- Update/Update
- Delete/Update |
| Equivalent Operations | 
- Add/Add
- Delete/Delete |

### Language Knowledge
- Metamodel
  - Well-formedness Rule
  - Abstract Syntax

### Domain Knowledge
- Common Knowledge
  - Upper Ontology
  - Thesaurus
  - ...

### Violations
- Redundancy

### Composite / Atomic
- Redundancy: Domain Knowledge

<table>
<thead>
<tr>
<th>V₀</th>
<th>V₁</th>
<th>V₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>Person</td>
<td>Person</td>
</tr>
<tr>
<td></td>
<td>lastname: String</td>
<td>surname: String</td>
</tr>
</tbody>
</table>
Metamodel Violation

\[ V_0 \]

\[ V_{r1} \]

\[ V_{r2} \]

- Overlapping Operations
  - Contradicting Operations
    - Pre-/Postconditions
  - Equivalent Operations
    - Upper Ontology
    - Thesaurus
    - Use Case Description
    - Requirement Specification

- Domain Knowledge
  - Common Knowledge
    - Upper Ontology
    - Thesaurus
    - Requirement Specification
  - User-defined Knowledge
    - Use Case Description
    - Requirement Specification

- Atomic
  - Different language constructs equivalent semantics
  - Different words with equivalent semantics

- Composite / Atomic
  - Metamodel
    - Well-formedness Rule
    - Abstract Syntax
  - Operation Contract
    - Pre-/Postconditions
Operation Contract Violation

\[ V_0 \]

\[ V_{r1} \]

\[ V_{r2} \]

Contradicting Operations
- Update/Update
- Delete/Update

Equivalent Operations
- Add/Add
- Delete/Delete

Language Knowledge
- Metamodel
  - Well-formedness Rule
  - Abstract Syntax

Domain Knowledge
- Common Knowledge
  - Upper Ontology
  - Thesaurus
  - ...

- User-defined Knowledge
  - Use Case Description
  - Requirement Specification
  - ...

Operation Contract Violation
- Pre-/Postconditions

Composite / Atomic Operations

- Update/Update
- Delete/Update
- Add/Add
- Delete/Delete
Common Knowledge Violation

- Contradicting Operations
  - Update/Update
  - Delete/Update

- Equivalent Operations
  - Add/Add
  - Delete/Delete

- Overlapping Operations
  - Atomic
  - Composite

- Language Knowledge
  - Different language constructs equivalent semantics
  - Redundancy
  - Different words with equivalent semantics

- Domain Knowledge
  - Metamodel
    - Well-formedness Rule
    - Abstract Syntax
  - Operation Contract
    - Pre-/Postconditions
  - Common Knowledge
    - Upper Ontology
    - Thesaurus
  - User-defined Knowledge
    - Use Case Description
    - Requirement Specification

- Violations
  - Circle
  - Square
  - Rectangle
  - Rhomboid
  - Quadrangle
User-defined Knowledge Violation

\[ \begin{align*}
V_o & \quad \text{turnOn()}\quad \text{makeCoffee()} \\
V_r & \quad \text{turnOn()}\quad \text{makeCoffee()}\quad \text{turnOff()} \\
V_{r2} & \quad \text{turnOn()}\quad \text{makeCoffee()}\quad \text{makeTea()}
\end{align*} \]

<table>
<thead>
<tr>
<th>Violations</th>
<th>Language Knowledge</th>
<th>Domain Knowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overlapping Operations</td>
<td>Contradicting Operations</td>
<td>Equivalent Operations</td>
</tr>
<tr>
<td>• Update/Update</td>
<td>• Add/Add</td>
<td>• Delete/Delete</td>
</tr>
</tbody>
</table>
| • Delete/Update
| Redundancy | Different words with equivalent semantics |
| Different language constructs equivalent semantics |

| Metamodel | Common Knowledge |
| Well-formedness Rule | Upper Ontology |
| Abstract Syntax | Thesaurus |

| Operation Contract | User-defined Knowledge |
| Pre-/Postconditions | Use Case Description |
| Requirement Specification | ... |
Outline

- Context of Model Versioning
- Foundations of Model Versioning
- Conflict Categorization
- Adaptable Model Versioning
- Future Challenges
AMOR: Adaptable Model Versioning

Overview

- **FFG FIT-IT Semantic Systems Project**
  - 2009 – 2011

- **Collaborating parties**
  - Vienna University of Technology
  - Johannes Kepler University, Linz
  - SparxSystems, the vendor of Enterprise Architect

- Three resulting dissertations
  - Petra Brosch, "Conflict Resolution in Model Versioning“, TU Wien, 2012.
Adaptable Model Versioning

Overview

Diagram showing the process of versioning with check-in and check-out at different times and versions.
Adaptable Model Versioning

Versioning Process

- Operation Detection
- Operation-based Conflict Detection
- Conflict-tolerant Merge
- State-based Conflict Detection
- Resolution

$V_o$

$V_{r1} \quad V_{r2}$
Adaptable Model Versioning

Versioning Process

Operation Detection

Operation-based Conflict Detection

Conflict-tolerant Merge

State-based Conflict Detection

Resolution

V₀

MV₀ • Vr₁

MV₀ • Vr₂

Vr₁

DV₀ • Vr₁

DV₀ • Vr₂

Vr₂
Adaptable Model Versioning

Versioning Process

Operation Detection
Operation-based Conflict Detection
Conflict-tolerant Merge
State-based Conflict Detection
Resolution

Vo

MVo•vr1 MVo•vr2

Vr1 DVo•vr1 DVo•vr2 Vr2

CV1•vr2
Adaptable Model Versioning

Versioning Process

Operation Detection → Operation-based Conflict Detection → Conflict-tolerant Merge → State-based Conflict Detection → Resolution

Vo → MVo•vr1 → Vr1
Vo → MVo•vr2 → Vr2

DVo•vr1 → Vr2
DVo•vr2 → Vr2

CV1•vr2 → Vm+oc
Adaptable Model Versioning

Versioning Process

Operation Detection
Operation-based Conflict Detection
Conflict-tolerant Merge
State-based Conflict Detection
Resolution

Vo

MVo•vr1

MVo•vr2

Vr1

DVo•vr1

DVo•vr2

Vr2

CV1•vr2

Vm+oc+sc
Adaptable Model Versioning

Versioning Process

Operation Detection
Operation-based Conflict Detection
Conflict-tolerant Merge
State-based Conflict Detection
Resolution

Vo

MVo•vr1

Vr1

DVo•vr1

Vr2

MVo•vr2

CVo•vr2

Vm

Vo

MVo•vr1

Vr1

DVo•vr1

Vr2

MVo•vr2

CVo•vr2

Vm
Adaptable Model Versioning

Versioning Process: Step by step
Operation Detection

Two approaches for obtaining applied operations

- **Model Differencing**
  - **Comparison** of states of an artifact
  - **Match function** to find correspondence of two elements in compared artifacts necessary
  - Differences are converted in atomic operations
  - Editor and language independent
  - **Composite operation** detection difficult, but possible in many cases

- **Operation Recording**
  - **Records** operation sequences
  - Recording of atomic operations (*add*, *update*, *delete*)
  - Recording of composite operations possible if available in the editor
  - Cleansing to remove obsolete operations for more efficient merging
  - Strong editor dependence
Operation Detection

Overview

- **Two-phase process**
  - **Phase 1**: Matching for finding correspondences between objects
  - **Phase 2**: Fine-grained comparison of corresponding objects
  - **Phase 3**: Detection of composite operation applications

![Diagram](Image)
Operation Detection: Phase 1 – Matching

Goal

- Match function is needed
  1. Universally Unique ID (UUID)
  2. Heuristics to compute similarity measures based on features of objects

- Quality of operation detection depends on quality of the match
Operation Detection: Phase 1 – Matching

Metamodel and Example Model

- Correspondences are described by a match model (weaving model)
  - Links corresponding elements
  - Marks unmatched elements
Operation Detection: Phase 1 – Matching

Metamodel and Example Model

- Correspondences are described by a match model (weaving model)
  - Links corresponding elements
  - Marks unmatched elements

Match Metammodel

State Machine Model \( V_o \)
- \( \text{StateMachine} \)
  - \( \text{SingleState} \) name = "Cooling"
  - \( \text{SingleState} \) name = "Heating"
  - \( \text{SingleState} \) name = "Idle"
  - \( \text{SingleState} \) name = "Active"
  - \( \text{Transition} \) name = "switch"

State Machine Model \( V_r \)
- \( \text{StateMachine} \)
  - \( \text{CompositeState} \)
  - \( \text{SingleState} \) name = "Cooling"
  - \( \text{SingleState} \) name = "Heating"
  - \( \text{SingleState} \) name = "Idle"
  - \( \text{Transition} \) name = "switch"

Match Model
- \( \text{MatchModel} \)
  - \( \text{Match} \)
  - \( \text{Unmatch} \)
  - \( \text{Original} \) side:Side
  - \( \text{Revised} \) side:Side

EObject (from Ecore)

original
revised

object

69
Operation Detection: Phase 2 – Comparison

- Deleted and inserted objects are explicitly marked by match model
  - But what is with structural feature changes?
- Fine-grained operation types depend on the metamodeling features
  - E.g., Ecore offers ordered features, etc.
- Supported operations
  - Insert Object
  - Delete Object
  - Feature Update
  - Insert Feature Value
  - Delete Feature Value
  - Insert Ordered Feature Value
  - Delete Ordered Feature Value
  - Feature Order Change
  - Move
Operation Detection: Phase 2 – Comparison

Difference Metamodel

- **FeatureChange**: Represents changes in features. It has a source, target, and value. It can represent insertions, deletions, or updates.
  - **FeatureUpdate**: Indicates an update to a feature.
  - **InsertFeatureValue**: Represents the insertion of a feature value.
  - **DeleteFeatureValue**: Represents the deletion of a feature value.
  - **InsertOrderedFeatureValue**: Represents the insertion of an ordered feature value.
  - **DeleteOrderedFeatureValue**: Represents the deletion of an ordered feature value.

- **ValueOrderChange**: Indicates changes in the order of feature values.
  - **InsertOrderedFeatureValue**: Represents the insertion of an ordered feature value.
  - **DeleteOrderedFeatureValue**: Represents the deletion of an ordered feature value.

- **ObjectChange**: Represents changes in objects, including insertions and deletions.
  - **InsertObject**: Represents the insertion of an object.
  - **DeleteObject**: Represents the deletion of an object.

- **EStructuralFeature**: Represents structural features of the model.

- **EObject**: Represents objects in the model, which can be changed or contained within other objects.

The diagram is a visual representation of the relationships and operations between these models, showing how changes in one model can affect others.
Operation Detection: Phase 2 – Comparison

Difference Model Example

State Machine Model $V_o$

- Active
  - Heating
  - Idle
- Cooling

State Machine Model $V_r$

- Active
  - Heating
- Cooling
Operation Detection: Phase 2 – Comparison

Difference Model Example

State Machine Model $V_o$

- : StateMachine
  - : CompositeState
    - name = "Active"
  - : SingleState
    - name = "Cooling"
  - : SingleState
    - name = "Heating"
  - : SingleState
    - name = "Idle"
  - : Transition
    - name = "switch"

State Machine Model $V_r$

- : StateMachine
  - : CompositeState
    - name = "Active"
  - : SingleState
    - name = "Cooling"
  - : SingleState
    - name = "Heating"
  - : Transition
    - name = "switch"

Difference Model $D_{V_o \cdot V_r}$

- : EReference
  - name = "states"
  - containment = true
  - ordered = false
  - lowerBound = 0
  - upperBound = -1
- : EClass
  - name = "SingleState"
- : EReference
  - name = "transitions"
  - containment = true
  - ordered = false
  - lowerBound = 0
  - upperBound = -1
- : EClass
  - name = "StateMachine"

State Machine Metamodel
Operation Detection: Phase 3
Composite operation detection process

1. **Diff Model Preprocessing**
   - **Input Signature**
   - **Operation Signature**
   - **Preselection**
     - [no diff match]
     - [diff matches]
   - **Potential Operation Occurrence**

2. **Derive Precondition Binding**
   - **Precondition Binding**
   - **Evaluate Binding**
     - [invalid]
     - [valid]
   - **Valid Precondition Binding**

3. **Derive Postcondition Binding**
   - **Postcondition Binding**
   - **Evaluate Binding**
     - [invalid]
     - [valid]
   - **Operation Occurrence**

**Operation Specifications**

\[ D_{V_0 \cdot V_{r1/2}} \]
Operation Detection: Phase 3

Composite operation detection process

Where do operation specifications come from?

Diff Model Preprocessing

Input Signature

Operation Signature

Preselection

[no diff match] [diff matches]

Potential Operation Occurrence

Derive Precondition Binding

Evaluate Binding

Precondition Binding

Valid Precondition Binding

[invalid] [valid]

Derive Postcondition Binding

Evaluate Binding

Postcondition Binding

[invalid] [valid]

Operation Occurrence

Operation Occurrence

$D_{V_0 \cdot V_{r1/2}}$
Operation Specifications

How to specify operation specifications

- **Operation Specifications**
  - Define composite operations

- **Composite operations**
  - Set of cohesive atomic operations
  - Applied in a transaction
  - To fulfill a common goal
  → Common goal should be respected during merge
  → Thus, they should be detected

- **Operation specifications are language-specific**
  - A plethora of modeling languages exists
  - Each has its own composite operations
  - Infeasable for language engineers to pre-specify all of them
  → Allow modelers themselves to specify them

- **Operation specifications are model transformations**
  - In particular, endogeneous in-place transformations
Operation Specifications

How to specify operation specifications

- Prerequisites for model transformation development
  - Experience in model transformation languages
  - Deep understanding of the involved metamodels
    - Common modelers are only aware of the concrete syntax
    - Model transformation languages are based on the abstract syntax

![Diagram showing concrete and abstract syntax for a university student exam scenario](image-url)
Model Transformation by Demonstration (MTBD)

- **Goal of MTBD**
  - Easing the specification of model transformations
  - Specification using the concrete syntax
  - Derive the general transformation automatically from a **demonstration** of these changes applied to an example model

**Operation Specification**
- Preconditions
- Actions
- Postconditions

- Can also be used as selectors.
- Can also be used for computing target values.
MTBD Example: Introduce Composite State

- Introduce Active
- Change target of lift
- Introduce new initial state
- Introduce transition initial to DialTone
- Change source of one hang_up transition
- Remove all other hang_up transitions
- Move states into Active
MTBD Process

Overview

1. Create initial model
2. Copy initial model
3. Perform updates
4. State-based comparison
5. Imply conditions
6. Edit conditions
7. Generate OSM
8. Generate specific artifacts

Legend
- automatic
- manual
MTBD Process

Step 1: Create Initial Model

Model elements, which are…
Required for execution
Handled differently

Legend
- automatic
- manual
MTBD Process

Step 2: Copy Initial Model

1. Create initial model
2. Copy initial model
3. Perform updates
4. State-based comparison
5. Imply conditions
6. Edit conditions
7. Generate OSM
8. Generate specific artifacts

Unique IDs preserve relationship
MTBD Process

Step 3: Perform updates

1. Create initial model
2. Copy initial model
3. Perform updates
4. Generate OSM Model
5. Generate specific artifacts
6. Refactoring Wizard
7. Operation Specification Model
8. Collaborate & Generation

Illustrate composite operation

Legend
- automatic
- manual
MTBD Process

Step 4: State-based Comparison

**Legend**
- automatic
- manual

**State-based Comparison**
- Inputs: Initial and Revised Model
- No editor-specific change tracking
- Any editor applicable
MTBD Process

Step 5: Imply Conditions

**Generic Condition Generation**
- Initial model \(\rightarrow\) preconditions
- Revised model \(\rightarrow\) postconditions
- Act as a basis for the next step

**Legend**
- automatic
- manual

Outside : SingleState_0
- name = “Outside”
- incoming->includes(#{Transition_1})
- ...

IntoStart : SingleState_1
- toFold : Transition_1
- event = “toFold”
- target = #{SingleState_0}
- ...

Outside : SingleState_0
Outside : SingleState_0
Outside : SingleState_0
MTBD Process

Step 5: Imply Conditions

Generic Condition Generation
- Initial model \(\rightarrow\) preconditions
- Revised model \(\rightarrow\) postconditions
- Act as a basis for the next step

...
MTBD Process

Step 6: Edit Conditions

1. Create initial model
2. Copy initial model
3. Perform updates
4. Revised model

<table>
<thead>
<tr>
<th>Modeling</th>
<th>Wizard</th>
<th>Configuration &amp; Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create initial model</td>
<td>Relax (deactivate conditions)</td>
<td>Manual settings</td>
</tr>
<tr>
<td>Initial model</td>
<td>Enforce (activate conditions)</td>
<td>Automatic</td>
</tr>
<tr>
<td>Copy initial model</td>
<td>Modify (change conditions)</td>
<td></td>
</tr>
<tr>
<td>Working model</td>
<td>Augment iterations and user inputs</td>
<td></td>
</tr>
</tbody>
</table>

Legend:
- **automatic**
- **manual**

**Configuration**
- Relax (deactivate conditions)
- Enforce (activate conditions)
- Modify (change conditions)
- Augment iterations and user inputs

**Default configuration: Relaxing**
- String features and
- Values which are empty and null
MTBD Process

Step 7 and Step 8

1. Create initial model
2. Copy initial model
3. Perform updates
4. State-based comparison
5. Imply conditions
6. Edit conditions
7. Generate OSM
8. Generate specific artifacts

Legend:
- **automatic**
- **manual**
MTBD: Tooling

- Eclipse Modeling Operations (EMO)
Operation-based Conflict Detection

Overview

- **Step 1: Atomic operation conflict detection**
  - Does not regard composite operation

- **Step 2: Composite operation conflict detection**
  - Checks for violated preconditions of applied composite operations

- Input: Two difference models
- Output: Conflict model

![Diagram](attachment:diagram.png)
Atomic Operation-based Conflict Detection

Atomic Conflict Metamodel

- Operation-based conflicts are described by a metamodel
  - Each conflict type is represented by a dedicated metaclass
- Conflict patterns for detecting operation-based conflicts
  - Each concrete conflict type has conflict patterns attached
Atomic Operation-based Conflict Detection

Conflict Pattern 1: Delete/Use Conflict

- Delete/Use conflict occurs if
  - Modeler A deletes object o
  - Modeler B uses object o in a featureChange operation as value

Example

Pattern

du: DeleteUse

```
S1 -> S2
S1
S1
S2
```

do: DeleteObject

```
object

value
```

fc : FeatureChange

```
delete
use
```
Atomic Operation-based Conflict Detection

Conflict Pattern 2: Delete/Update Conflict

- Delete/Update conflict occurs if
  - Modeler A deletes object o
  - Modeler B updates a feature f of object o

There are several more patterns!

Example

Pattern


P. Langer: Adaptable Model Versioning based on Model Transformation By Demonstration; Reviewer: G. Kappel, J. Gray; E188 Institut für Softwaretechnik und Interaktive Systeme, 2011.
For each composite operation application,
- check whether it is *applicable* at the *opposite side*
Example

Composite Operation Conflict Detection

- Example

V₀

Introduce Composite State

V₁

Rename hangup

V₂

- Example

- Example
Composite Operation Conflict Detection

- Example

Transition_1.name = Transition_2.name

Precondition of composite operation is violated!
Conflict-tolerant Merge
How to support developers when merging different versions?

- **Motivation**
  - Only one modeler is responsible for conflict resolution
    → Final version does not reflect all intentions of the modelers
  - Conflicts are considered harmful
    → They should be subject of discussion
  - Merging conflicts is rejecting one of the conflicting operations
    → However, a merged model is helpful for deciding how to resolve conflicts

- **Goal**
  - Conflict-tolerant model versioning system supporting a collaborative conflict resolution process

- **Challenge**
  - Merge is not possible in case of conflicts!
  - How can a merge be accomplished in case of conflicts?
  - How can the conflicts be visualized for any modeling language?
Conflict-tolerant Merge

Solution

- General Approach
  1. Conflict-tolerant merge rules
     - To avoid conflict resolution during check-in
     - To find a merge result irrespectively of conflicts
  2. Conflict annotations
     - To avoid information loss
     - To enable distributing the responsibility of conflict resolution
     - To provide a basis for discussion
  3. Conflict resolution model
     - To control the lifecycle of a conflict
     - To manage conflicts and resolutions
Conflict-tolerant Merge Rules

Prioritize one change or omit both changes

- To enable a merge irrespectively of conflicts

- Omit both conflicting changes in case of …
  - Update-update: Use original value
  - Move-move: Use original container

- Omit deletions out of two conflicting changes
  - Delete-update: Prioritize Update
  - Delete-use: Prioritize Use
  - Delete-move: Prioritize Move
Conflict-tolerant Merge Rules

Example: Update-update Conflict

- Update-update Conflict

Origin Version created by USER_A

- o1:Object
  - f1 = x

Update by USER_B

- o1:Object
  - f1 = y

Update by USER_C

- o1:Object
  - f1 = z

Merged Version

- <<UpdateUpdate>>
  --Involved Elements
  Upd_feature= {f1}
  Upd_value ={<USER_B : y>,<USER_C : z>}

--User-related Metadata
User_upd= {USER_B, USER_C}
Owner= USER_A

--Time related Metadata
...
Conflict-tolerant Merge Rules

Example: Delete-update Conflict

- **Delete-update Conflict**

---

**Origin Version created by USER_A**

- o1:Object
  - f1 = x

**Update by USER_B**

- o1:Object
  - f1 = y

**Delete by USER_C**

- o1:Object
  - f1 = y

**Merged Version**

- **<<DeleteUpdate>>**
  --Involved Elements
  Upd_feature = {f1}
  Upd_value = {y}
  Old_value = {x}

--User-related Metadata
User_del = USER_C
User_upd = USER_B
Owner = USER_A

--Time related Metadata
...

**Update by USER_B**

- o1:Object
  - f1 = y

**Delete by USER_C**

- o1:Object
  - f1 = y

**Merged Version**

- **<<DeleteUpdate>>**
  --Involved Elements
  Upd_feature = {f1}
  Upd_value = {y}
  Old_value = {x}

--User-related Metadata
User_del = USER_C
User_upd = USER_B
Owner = USER_A

--Time related Metadata
...
Conflict Annotations
Using Annotations in the Merged Model for Marking Conflicts

- Annotate conflicts in the merged model
  - For visualizing conflicts
  - For providing a basis for resolution
    → On top of the concrete syntax of the merged model

- Annotations can be stored across revisions
  - Allows to tolerate conflicts for a while
  - Enables distributing them among team members

- Lightweight annotations using profiles (cf. UML Profiles)
  - Model elements are annotated by stereotypes in concrete syntax
  - Tagged values comprise additional conflict information
Conflict Annotations

Profile for Merge Conflicts
Conflict Annotations

Example

Apply conflict-tolerant merge rules

Annotate conflicts (i.e., apply conflict profile)
Conflicts Annotations

Applied Conflict Profile
How can profiles be applied to non-UML models?

→ EMF Profiles
Overall Goal of EMF Profiles

“Adopt the notion of UML Profiles to DSMLs residing in EMF”

- UML Profiles: A Short Reminder
  - Lightweight Language Extension Mechanism of the UML
  - A Profile consists of Stereotypes
  - Stereotypes extend base classes
    - And may introduce “tagged values”

- Profiles are applied to UML models
  - By applying stereotypes to instances of their base classes
  - Specifying concrete values for their defined “tagged values”
UML Profiles: A Short Introduction

Profile specification

- «profile» EJB
- «metaclass» Component
- «enumeration» StateKind
  - stateless
  - stateful
- «stereotype» Bean
- «stereotype» EntityBean
  - state: StateKind
- «stereotype» SessionBean
- «apply»

Profile application

- «SessionBean» Customer
  - state=stateless
Motivation

- Overall goal
  - “Adopt the notion of *UML Profiles* to *DSMLs* residing in *EMF*”

- Is combination of profiles with *DSMLs* a contradiction?
  - Many debates on Pros and Cons of adopting *UML Profiles or DSMLs*

**UML Profiles vs DSMLs**
UML Profiles vs DSMLs

Reuse UML’s language concepts and existing UML editors!

Create a lean language that is straight to the point!

You’ll have to create the whole infrastructure by yourself!

You’ll end up with an overloaded and imprecise language!
UML Profiles vs DSMLs

These debates concern adopting either UML Profiles or DSMLs for creating new languages.

What about extending existing languages?

It’s your language... just extend your metamodel.
**Motivating Scenario**

Data Modeling Language

- **Meta model**
- **Concrete Syntax**
- **Editor**

... generates Ruby on Rails.

I want to additionally specify "Finder SQL" statements!

... generates JavaServer Faces.

I want to additionally specify the bean scope!

... generates DB Schema.

Leave it as it is! If you introduce every imaginable feature that I don't need, I could have used UML in the first place.
Motivating Scenario

I can’t address all your requirements!

Data Modeling Language

- Metamodel
- Concrete Syntax
- Editor

... generates Ruby on Rails.

I want to additionally specify “Finder SQL” statements!

... generates JavaServer Faces.

I want to additionally specify the bean scope!

... generates DB Schema.

Leave it as it is!
If you introduce every imaginable feature that I don’t need, I could have used UML in the first place.

You need a lightweight extension mechanism: Profiles!

Other scenarios
- No influence on the metamodel
- “Concern-specific” annotations
Benefits of Profiles for DSMLs

- Lightweight language extension
  - Introduce additional features
  - No need for changing the metamodel and the modeling infrastructure

- Dynamic model extension
  - Existing models can be extended; even by multiple profiles and stereotypes
  - Profile applications are separated from the models (→ no model pollution)

- Preventing metamodel pollution
  - Metamodels represent only information coming from the modeling domain
  - Concern-specific information is defined in a profile

- Model-based representation
  - Profile applications are well-defined (→ they can be validated)
  - Profile applications are just additional models (→ reuse existing frameworks)
The Challenge

Okay, so let’s use profiles!

But wait, that’s not so easy with EMF!

<table>
<thead>
<tr>
<th>UML</th>
<th>EMF</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3</td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>«import»</td>
</tr>
<tr>
<td>M2</td>
<td>Profiles</td>
</tr>
<tr>
<td></td>
<td>«instanceOf»</td>
</tr>
<tr>
<td></td>
<td>UML</td>
</tr>
<tr>
<td></td>
<td>«extend»</td>
</tr>
<tr>
<td>M1</td>
<td>aProfile Application</td>
</tr>
<tr>
<td></td>
<td>«instanceOf»</td>
</tr>
<tr>
<td></td>
<td>aUML Model</td>
</tr>
</tbody>
</table>
Solution: Metalevel Lifting by Inheritance

<table>
<thead>
<tr>
<th>Profile Definition</th>
<th>Metalevel Lifting by Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>M3</strong></td>
<td></td>
</tr>
<tr>
<td>Ecore</td>
<td></td>
</tr>
<tr>
<td>«instanceOf»</td>
<td></td>
</tr>
<tr>
<td><strong>M2</strong></td>
<td></td>
</tr>
<tr>
<td>Profile MM</td>
<td></td>
</tr>
<tr>
<td>«instanceOf»</td>
<td></td>
</tr>
<tr>
<td><strong>M1</strong></td>
<td></td>
</tr>
<tr>
<td>aProfile</td>
<td></td>
</tr>
</tbody>
</table>
Solution: Metalevel Lifting by Inheritance

<table>
<thead>
<tr>
<th>Profile Definition</th>
<th>Metalevel Lifting by Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3</td>
<td>Ecore</td>
</tr>
<tr>
<td></td>
<td>«instanceOf»</td>
</tr>
<tr>
<td>M2</td>
<td>Profile MM</td>
</tr>
<tr>
<td></td>
<td>«instanceOf»</td>
</tr>
<tr>
<td></td>
<td>«inheritsFrom»</td>
</tr>
<tr>
<td>M1</td>
<td>aProfile</td>
</tr>
</tbody>
</table>
Solution*: Metalevel Lifting by Inheritance

<table>
<thead>
<tr>
<th>Profile Definition</th>
<th>Metalevel Lifting by Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3</td>
<td>Ecore</td>
</tr>
<tr>
<td>M2</td>
<td>Profile MM</td>
</tr>
<tr>
<td></td>
<td>«instanceOf»</td>
</tr>
<tr>
<td></td>
<td>«inheritsFrom»</td>
</tr>
<tr>
<td>M1</td>
<td>aProfile</td>
</tr>
<tr>
<td></td>
<td>«instanceOf»</td>
</tr>
<tr>
<td></td>
<td>aProfile Application</td>
</tr>
</tbody>
</table>

* There is an alternative solution as well (i.e., transformation to a model at M2).
EMF Profile Metamodel

\[\text{Profile} \rightarrow \text{iconPath : EString}\]

\[\text{Standard EMF Profile}\]

\[\text{Ecore}\]

\[\text{EClass}\]
  
  abstract: EBoolean

  eSuperTypes : EClass

  ...

\[\text{EPackage}\]
  
  nsURI : EString

  eClassifiers : EClassifier

  ...

\[\text{Stereotype}\]

\[\text{Profile Application}\]

appliedTo : EObject

StereotypeApplication

stereotype

Applications

«import»
EMF Profile Metamodel and its Instantiation

### Standard EMF Profile

**Ecore**

- **EClass**
  - abstract: EBoolean
  - eSuperTypes : EClass
  - ...

- **EPackage**
  - nsURI : EString
  - eClassifiers : EClassifier
  - ...

**Profile**

- **Stereotype**
  - iconPath : EString

**Profile Application**

- **Profile**

**A Profile Specification**

- **Profile**
  - eClassifiers

- **EClass**
  - name="Entity"

- **Stereotype**
  - base
  - name="SessionBean"

- **EAttribute**
  - name="isStateful"
EMF Profile Metamodel and its Instantiation

**Standard EMF Profile**

**Ecore**

- **EClass**
  - abstract: EBoolean
  - eSuperTypes : EClass
  - ...

- **EPackage**
  - nsURI : EString
  - eClassifiers : EClassifier
  - ...

**Profile**

- **Stereotype**
  - iconPath : EString

**ProfileApplication**

- **ProfileApplication**
  - stereotype
  - Applications
  - appliedTo : EObject

**A Profile Specification**

- : Profile
  - eClassifiers
  - eSuperTypes

- : EClass
  - name="Entity"

- : Stereotype
  - base
  - name="SessionBean"
  - eAttributes
    - : EAttribute
      - name="isStateful"
**Standard EMF Profile**

**Ecore**

- **EClass**
  - abstract: EBoolean
  - eSuperTypes : EClass

- **EPackage**
  - nsURI : EString
  - eClassifiers : EClassifier

**Profile**

- **Stereotype**
  - iconPath : EString

**ProfileApplication**

- **ProfileApplication**
  - stereotype Applications : ProfileApplication
  - appliedTo : EObject

**A Profile Specification**

- **Profile**
  - eClassifiers
  - eSuperTypes

- **EClass**
  - name="Entity"

- **Stereotype**
  - name="SessionBean"
  - eAttributes
    - name="isStateful"

**A Profile Application**

- **Entity**
  - isStateful=true
  - appliedTo

- **SessionBean**
  - stereotype Applications

- **ProfileApplication**
EMF Profile Metamodel and its Instantiation

Profile
- iconPath : EString

ProfileApplication
- appliedTo : EObject

Ecore
- EClass
  - abstract: EBoolean
  - eSuperTypes : EClass
    - 0..*
  - eClassifiers : EClassifier
    - ...

EPackage
- nsURI : EString
- eClassifiers : EClassifier
  - ...

Profile
- Profile

ProfileSpecification
- Profile
- eClassifiers
- eSuperTypes
  - base
- 0..*

ProfileApplication
- ProfileApplication
  - stereotype Applications
  - 0..*

Stereotype
- Stereotype
  - eAttributes
    - base
  - name="SessionBean"

A Profile Specification
- : Profile
  - eClassifiers
  - eSuperTypes
    - base
    - name=“Entity”

A Profile Application
- : Entity
  - appliedTo
  - : SessionBean
    - isStateful=true
Example: EJB Profile Specification
Example: EJB Profile Application
From UML Profiles to EMF Profiles and Beyond

- UML Profiles can be specified for UML only
  - No need for further reuse of profiles for other languages

- EMF Profiles can be specified for every Ecore-based language
  - Reuse a profile for more than one language

- Generic Profiles
  - Reuse a profile for several “user-selected” DSMLs

- Meta Profiles
  - Reuse a profile for all DSMLs
Generic Profiles

- Reuse a profile for several “user-selected” DSMLs
  - Extend generic types instead of concrete types
  - Bind generic types to concrete types to apply a profile
  - Use OCL to further restrict valid bindings

```
Generic Profiles

- Reuse a profile for several “user-selected” DSMLs
  - Extend generic types instead of concrete types
  - Bind generic types to concrete types to apply a profile
  - Use OCL to further restrict valid bindings

<<profile>> EJB

<<generic type>> Container

<<generic type>> Property
  name : EString

<<stereotype>> SessionBean
isStateful : EBoolean

<<stereotype>> EntityBean
isUserManaged : EBoolean

<<stereotype>> IDAttribute

<<bind>> Container->Entity, Property->Attribute

ER

Entity
  name : String

Attribute
  name : String
```
Meta Profiles

- Reuse a profile for all DSMLs (at once)
  - Each “Meta Stereotype” may be applied to every base type
  - Useful for general annotations
    - E.g., Conflict Profile

![Image of meta-profile and conflict profile application](image-url)
Implementation

- Based on the Eclipse Modeling Framework
- Supports extending every Ecore-based DSML
- Uses the Decoration Service to show icons in GMF editors

- Open Source (EPL 1.0)
  - http://code.google.com/a/eclipselabs.org/p/emf-profiles/

- Try EMF Profiles!
  - Eclipse Update Site
    http://www.modelversioning.org/emf-profiles-updatesite/

- Contact us and get involved!
  - http://groups.google.com/group/emf-profiles
State-based Conflict Detection

Validate of the Result from Conflict-tolerant Merging

- General well-formedness rules are covered by conflict patterns already
  - Unique container
  - No cyclic containments

- But, language-specific validation rules need to be checked
  - Lower/upper bound of multi-valued features
  - Additional OCL Constraints

- Therefore, after conflict-tolerant merging, the resulting state is validated
  - If a constraint is invalid
    → New state-based conflict annotation

- Reuse EMF validation framework
  - Annotate context elements of reported errors/warnings
Resolution

- Dedicated conflict resolution view
  - Based on the conflict-tolerant merge result and the conflicts annotations
  - Allows to…
    - … resolve conflicts manually (e.g., use this change or that change)
    - … resolve conflicts by predefined resolution rules
  - Resolutions can be attached to a conflict annotation

- Lifecycle of a conflict
  - Conflicts have a state (e.g., open or resolved)
  - Cf. next slide
Resolution

Conflict Resolution Model: The Lifecycle of Conflicts
Example: Conflict-tolerant Merging Improves the Result

Issues of Current Versioning/Resolution Protocol

Car as part of one Employee

Model stored in the Repository
Example: Conflict-tolerant Merging Improves the Result

Issues of Current Versioning/Resolution Protocol

Delete/Update Conflict!
Update/Update Conflict!

Model stored in the Repository
Example: Conflict-tolerant Merging Improves the Result

Result of Conflict-tolerant Merge
Example: Conflict-tolerant Merging Improves the Result

Collaborative Resolution based on Conflict-tolerant Merge

Result of Standard Versioning Process

<table>
<thead>
<tr>
<th>Operation Detection</th>
<th>Operation-based Conflict Detection</th>
<th>Conflict-tolerant Merge</th>
<th>Blend-based Conflict Detection</th>
<th>Resolution</th>
</tr>
</thead>
</table>

Alice

Harry

Sally

Joe
Semi-Automated Resolution

- Conflict Resolution Recommender
  - For increasing efficiency and avoiding errors

- Conflict Resolution Patterns
  - Preconditions defined for
    - the merged model
    - the applied changes
    - the annotated conflicts
  - Actions
    - Operations to be applied to resolve the conflict
    - Formally defined using graph transformation systems
Example: Resolution Recommender

The preconditions of the refactoring “Pull Up Field” are violated! AddForbid(SoccerM)
Example: Resolution Recommender

The preconditions of the refactoring “Pull Up Field” are violated! AddForbid(SoccerM)

Resolve AddForbid(SoccerM) by introducing intermediate class?
Outline

- Context of Model Versioning
- Foundations of Model Versioning
- Conflict Categorization
- Adaptable Model Versioning
- Future Challenges
Future Challenges

Consistency-aware Versioning

- Consistency rules, e.g.
  - Metamodel
  - OCL Constraints
  - Requirement documents
  - Additional models

- Merged model violates consistency rules even if both versions are consistent

→ Trace back to the relevant changes causing the violation

Future Challenges

Intention-aware Versioning

- Merged version should incorporate all intentions of each modeler

  → Treat composite operations such as *model refactorings* as first-class entities

  → Going beyond composite operations: E.g., a set of operations has been applied in order to fix a bug… is the bug still fixed after merging?

  → Other ways to capture and respect the intention?

---

Future Challenges
Semantics-aware Versioning

- Model differencing only on syntactic level

→ Formal definition of the modeling language’s semantics
→ Mapping between modeling language and semantic domain
→ Including intra-model dependencies

Different syntax, equal semantics


Future Challenges

Conflict Dependencies

- Dependencies between a sequence of changes
- Leading to dependencies between conflicts

→ Efficient detection between more complex changes like refactorings or violations
→ Optimal order for resolving conflicts


Future Challenges

Diagram Versioning

- Co-evolution of diagram layout information with the model
- Existing approaches neglect the nature of 2D diagram layout

→ Which concurrently performed layout change is in fact a contradicting change of the layout (more fuzziness required than for models)?
→ Respect the preservation of mental map when merging diagram changes!


Future Challenges

Avoiding Conflicts

- Pessimistic versioning and Synchronous modeling not always desirable

- Avoid conflicts through awareness
  - Notification when modelers work on the same model fragment(!)

- Model partitioning
  - How to separate a model?
  - Find appropriate mechanisms for separation of concerns techniques
Future Challenges
Model/Code Versioning

- **Roundtrip Engineering**
  - Synchronizing models and code
  - Problem of inconsistencies when changing models and code in parallel
  - Especially when changes are not on the same granularity level

- **Using models for versioning large code repositories**
  - In case several conflicts between different versions occur

Jim, could you come to me for a moment?

Of course.

What's up?

Nothing, thanks! Just wanted to make sure that I'm the first to check-in the files we've both edited.
The AMOR Team

- Gerti Kappel
- Martina Seidl
- Manuel Wimmer
- Petra Brosch
- Philip Langer
- Konrad Wieland
These Slides are Based on the Following Papers (1/2)

- P. Brosch: 
  *Conflict Resolution in Model Versioning*; Reviewer: G. Kappel, A. Pierantonio; E188 Institut für Softwaretechnik und Interaktive Systeme, to appear 2012.

- P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel: 

- P. Brosch, G. Kappel, M. Seidl, K. Wieland, M. Wimmer, H. Kargl, P. Langer: 
  *Adaptable Model Versioning in Action*. 
  Proc. of the Modellierung, GI, 221-236.

- P. Brosch, H. Kargl, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel: 

- P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel: 
  *Concurrent Modeling in Early Phases of the Software Development Life Cycle*. 

- G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, M. Wimmer: 

- P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger, W. Schwinger: 
  *An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example*; 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09), Denver, USA; in: "Proc. of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)", Springer, LNCS 5795 (2009), ISBN: 978-3-642-04424-3; 271 - 285.
These Slides are Based on the Following Papers (2/2)


References and Further Reading (1/4)

References and Further Reading (2/4)

References and Further Reading (3/4)

References and Further Reading (4/4)


