Model weaving
Establishing links between model elements
Outline

Model weaving: state of the art and concepts
Practical work: schema mapping and traceability
Matching and transformation production
Practical work: matching and transformation production
Relationships between model elements

- Transformation are not always enough
  - Precise execution semantics

\[ a_i \equiv b_j \] – How to express? – How to compute? – How to generate \( M_a \cap M_b \) or \( M_a \cup M_b \)?

- Three main aspects

\[ \equiv \]
\[ \neq \]
\[ \equiv (60\%) \]
First aspect: relationship representation

Legend
[] - unique cardinality
<> - shared cardinality
Formalism: feature diagrams
Second aspect: relationship computation

Computation (aka. matching)

- Automatic
  - Heuristics
  - <1..2>

- Manual
  - [0..1]

- Tools
  - GUI
  - Configurable
  - Extensible
  - [1..1]
  - Maintenance
  - [0..1]

(around 138 approaches)
- COMA
- iMAP
- MAFRA
- S-Match
- etc.
Third aspect: relationship utilization

Strongly linked to how relationships are produced
Summary and requirements

- Representation
  - Different formats
  - Model management [Bernstein et al. 2000]

- Computation
  - Interoperability difficult

- Utilization
  - Transformation production
  - Traceability
  - Requirements
  - etc.
Representation of relationships

- Multiple technologies
  - Morphism
  - Value correspondences
  - Auxiliary model
  - Ontology bridges

- MDE solutions
  - QVT relations
  - TGG
  - Model link
  - Model weaving
Approaches for relationship representation (1/2)

- **Morphism**
  - A pair \(<L,R>\), where
    - \(L\) is an identifier for the left element
    - \(R\) is an identifier for the right element
  - Bidirectional
  - Example: House <-> Home, Professor <-> Teacher

- **Value correspondences**
  - A function \(f : S \rightarrow T\).
    - A filter over the source elements from \(S\).
  - Directed relationships
  - 1-to-1 function is the most common format
  - Example: People \(\rightarrow\) Person, First + Last Name \(\rightarrow\) Name
  - Largely applied on DB community
Approaches for relationship representation (2/2)

- Auxiliary model
  - A map *model* plus a pair of morphisms

- Ontology bridges
  - Mappings as first class entities for bridging ontologies
  - Identification using RDF IDs
  - Explicit mapping names
    - `AttributeBridge`, `ConceptBridge`, `RelationBridge`
    - `SubClassOf`, `InstanceOf`
Relationships on MDE: QVT Relations

- QVT Relations: from QVT relations, core and mappings
  - "A declarative specification of the relationships between MOF models." from QVT spec.

- Not only simple correspondences
  - Support to pattern matching, (Bi) directionality, nested relations, traceability
  - Targeted for transformations

```plaintext
relation ClassToTable {
<checkonly/enforce>
domain uml c:Class {
    namespace = p:Package {},
    name=cn
}
domain rdbms t:Table {
    schema = s:Schema {},
    name=cn,
    column = cl:Column {
        name=cn+'_tid',
        type='NUMBER',
        primaryKey = k:PrimaryKey {
            name=cn+'_PK',
            column=cl
        }
    }
when {
    PackageToSchema(p, s);
}
where {
    AttributeToColumn(c, t);
}
}```
Triple Graph Grammars (TGG)

- TGG schema
  - a pair of graphs
  - a correspondence graph
- TGG rules
  - Instance-based approach
- Mix of LHS and RHS
  - Instantiate the three elements (left, link and right)
  - Transformation and weaving
    - Limited pattern and navigation expressions
- Rewriting rules (transformations) over the elements of 3 graphs

- Applications
  - (Bidirectional) transformations
  - Integration
  - Synchronization
Model weaving : an illustrative example
Model weaving

- Capture relationships between model elements
- Relationships are reified in a **weaving model**
  - The model elements represent the relationships and the related elements
  - As any kind of model, the weaving model can be saved, stored, transformed, modified
  - Different kinds of links
    - Equality, concatenation, equivalence, etc.

![Weaving Model Diagram]
Weaving model and metamodel

- **Weaving metamodel:** A weaving metamodel is a model \( MM_W = (G_M, \omega_M, \mu_M) \), that defines link types, such that:
  - \( G_M = (N_M, E_M, \Gamma_M) \),
  - \( N_M = N_L \cup N_{LE} \cup N_O \), \( N_L \) denotes the link types; \( N_{LE} \) denotes the link endpoint types and \( N_O \) denote other auxiliary nodes,
  - \( \Gamma_M : E_M \rightarrow (N_L \times N_{LE}) \cup (N_O \times N_M) \), i.e., a link type refers to multiple link endpoint types and the auxiliary nodes refer to any kind of node.

- **Weaving model:** A weaving model is a model \( M_W = (G_W, \omega_W, \mu_W) \), a graph \( G_W = (N_W, E_W, \Gamma_W) \), such that its reference model is a weaving metamodel \( (\omega_W = MM_W) \).

- The related models are independent
  - 1-to-N models can be related
Dereferencing function

- **Dereferencing function**: Given a weaving model \( M_W = (G_W, \text{MM}_W, \mu_W) \), \( G_W = (N_W, E_W, \Gamma_W) \) and a linked model \( M = (G, \omega, \mu) \), \( G = (N_G, E_G, \Gamma_G) \), a dereferencing function \( \rho \) returns the elements of the linked model:
  
  - \( \rho : N_{WLE} \rightarrow N_G, N_{WLE} \subseteq N_W \), such that \( \mu_W(N_{WLE}) = N_{LE} \).

- This means the elements of the weaving models are "pointers" to the elements of the linked models, and they conform to the link end points.
Weaving metamodel (core)

Identifiers (indirect approach)

Link type et link endpoints
Core weaving metamodel

```java
package mwcore {
    abstract class WElement {
        attribute name : String;
        attribute description : String;
        reference model : WModel oppositeOf ownedElement;
    }

    abstract class WModel extends WElement {
        reference ownedElement[] ordered container : WElement oppositeOf model;
        reference wovenModel[] container : WModelRef;
    }

    abstract class WRef extends WElement {
        attribute ref : String;
    }

    abstract class WModelRef extends WRef {
        reference ownedElementRef[] container : WElementRef oppositeOf modelRef;
    }

    abstract class WElementRef extends WRef {
        reference modelRef : WModelRef oppositeOf ownedElementRef;
    }

    abstract class WLink extends WElement {
        reference child[] ordered container : WLink oppositeOf parent;
        reference parent : WLink oppositeOf child;
        reference end[] container : WLinkEnd oppositeOf link;
    }

    abstract class WLinkEnd extends WElement {
        reference link : WLink oppositeOf end;
        reference element : WElementRef;
    }
}
```
Weaving metamodel extensions

- The core metamodel must be extended for a given application domain
  - **Interoperability**
    - Equality, SourceToTarget.
  - **Data integration**
    - Concatenation, Equality, IntToStr.
  - **Traceability**
    - Origin, Source, Evolution, Modified, Added
  - **Composition**
    - Override, Merge, Delete.
  - **Ontology alignment**
    - Equivalent, Equality, Resemblance, Proximity.
class InheritanceLink extends WLink {
    reference parents[1-*] container : WLinkEnd;
    reference child container : WLinkEnd;
}

\[ MM_R = \text{Extend} (MM_w, MM_E, M_{WD}) \]

**Input:**
- \( MM_w \): the metamodel to be extended
- \( MM_E \): the metamodel extension
- \( M_{WD} \): a weaving model between the elements of \( MM_w \) and \( MM_E \)

**Output:**
- \( MM_R \): an extended \( MM_w \)

/* add all elements and edges from \( MM_E \) into \( MM_w \), if they do not already exist*/
for each \( mme \in MM_E \) and not \( mme \in MM_w \)

\[ MM_w \leftarrow MM_w \cup mme \]

/* addLink gets the elements represented by \( M_{WD} \) and create a link between them*/
\[ MM_w \leftarrow MM_w \ \text{addLink} (M_{WD}) \]

return \( MM_w \)
A simple weaving metamodel extension

```java
package mw_base_ext {

class Model extends WModel {
    -- @subsets wovenModel
    reference leftModel container : WModelRef;
    -- @subsets wovenModel
    reference rightModel container : WModelRef;
}
class ElementRef extends WElementRef {
}
class ModelRef extends WModelRef {
}
class Association extends WAssociation {
}
class AssociationEnd extends WAssociationEnd {
}
class Link extends WLink {
    -- @subsets end
    reference left container : WLinkEnd;
    -- @subsets end
    reference right container : WLinkEnd;
}
class LinkEnd extends WLinkEnd {
}
}
```
Adapts to any weaving metamodel extension
- The user interface is automatically generated according to the metamodel extensions
  - Reflective API of EMF (Eclipse Modeling Framework)

A set of extension points is defined to enable to customize the standard user interface
- Extension points to the panels, to the model elements, and to execute model transformations in ATL (Atlas Transformation Language)
- New interfaces can be easily developed

---

**GUI extensions**

- Weaving panel
- Woven panel
- Menus for MDE
- Operation execution
- Dereferencing mechanism
- Metamodel extension

**Model weaver workbench**

**EMF (model manipulation primitives)**
AMW user interface

- Plugged panels
- Adaptive interface
- Identification mechanism
Summary

- Relationship between model elements
  - Several solutions, specialized for different aspects

- Model weaving
  - Generic representation
    - Based on the core metamodel and extensions
    - ATLAS Model Weaver tool
Traceability

- Data provenance
  "the problem of discovering the origin of data after it was transformed from a source schema into a target schema"

- Requirements traceability
  "keeps track of all the steps of a development process: analysis, design, programming, testing. Some possible kinds of links are developed_by, allocated_to, performed, based_on, modify. The key processes are the identification of the possible kinds of links and the development of new traceability reference models”.

  - Static requirements traceability
    • Requirements to code (several stages)
  - Event-based traceability
    • Subscribes to a service (observer pattern)
  - Reference models
    • Models used just for referring traceable models

Traceability survey [Galvao I, Goknil A. Survey of Traceability Approaches in Model-Driven Engineering. EDOC 2007]
Traceability

- Traceability of model transformations
  "Similar to data provenance scenarios, it is often necessary to store the execution trace of model transformations. The execution trace of a transformation indicates, for a set of generated elements, which transformation rules are executed, and which input elements are used."

  - Loosely coupled
  - Batch execution of model transformation produces a weaving model
  - Embedded traceability → annotation
  - Merges a model and its trace information
Outline

Model weaving: state of the art and concepts

Practical work: schema mapping and traceability

Matching and transformation production

Practical work: matching and transformation production
Relational database to XML schema
Weaving metamodel extension

```java
package mw_base_ext {

    class Model extends WModel {
        -- @subsets wovenModel
        reference leftModel container : WModelRef;
        -- @subsets wovenModel
        reference rightModel container : WModelRef;
        -- reference others[] container subsets
        wovenModel : WModelRef;
    }

    class ElementRefXRMI extends WElementRef {
        -- @wmodelRefType ModelRef
    }

    class ModelRef extends WModelRef {
        -- @welementRefType ElementRef
    }

    class ModelRefXRMI extends WModelRef {
        -- @wmodelRefType ModelRef
    }

    class Association extends WAssociation {
        class AssociationEnd extends WAssociationEnd {
        }
    }

    class Link extends WLink {
        -- @subsets end
        reference left container : WLinkEnd;
        -- @subsets end
        reference right container : WLinkEnd;
    }

    class LinkEnd extends WLinkEnd {
    }
}
```

Slide from: Univ. Rey Juan Carlos
Traceability of model transformations

- Original transformation setting

```
rule Book2Publication {
  from
  s : MMb!Book
  to
  t : MMp!Publication (    title <- s.title + s.subtitle,
                           pubYear <- s.year,
                           authors <- author ),
  author : MMp!Author (    name <- s.author
  )
}
```

- How to store traceability information?
Traceability of model transformations

- Produce $M_t'$ from $M_t$ using a Higher Order Transformation
- $M_t'$ produces an additional weaving model

- Weaving metamodel extension

```java
class TraceLink extends WLink{
    attribute ruleName : String;
    reference sourceElements[*] ordered container : WLinkEnd;
    reference targetElements[*] ordered container : WLinkEnd;
}
class TraceLinkEnd extends WLinkEnd {
}
class ElementRef extends WElementRef {
}
```
Outline

Model weaving: state of the art and concepts
Practical work: traceability and schema mapping

Matching and transformation production
Practical work: matching and transformation production
Matching

- Matching is the process of establishing relationships between elements belonging to different models

- Manual
  - User interface

- Automatic
  - Algorithms

- Semi-automatic
  - Utilization of heuristics
Matching heuristics

- String similarity
  - Date <-> BirthDate

- Dictionaries
  - Car <-> Vehicle

- Structural relations
  - Class.name <-> Table.name
  - Class <-> Table

- Different problems
  - How to express this heuristics?
  - How to support different extensions?
<table>
<thead>
<tr>
<th>Tool</th>
<th>Input (internal and external representation)</th>
<th>Matching techniques</th>
<th>Mapping nature</th>
<th>Application scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUPID</td>
<td>DB and XML schemas (rooted graphs)</td>
<td>Structural and linguistic</td>
<td>1:1 correspondences</td>
<td>Generic matching tool</td>
</tr>
<tr>
<td>GLUE</td>
<td>Unified ontology (rooted graph)</td>
<td>Data instances, probability distribution</td>
<td>1:1 mappings</td>
<td>Generic matching tool</td>
</tr>
<tr>
<td>PROMPT</td>
<td>Ontologies (general knowledge model)</td>
<td>Set of iterative operations</td>
<td>None: merges the ontologies</td>
<td>Ontology merging and alignment</td>
</tr>
<tr>
<td>COMA / COMA++</td>
<td>SQL, XML and OWL schemas (rooted directed graphs)</td>
<td>Library of heuristics</td>
<td>Equivalence 1:1 correspondences</td>
<td>Generic matching tool</td>
</tr>
<tr>
<td>ONION</td>
<td>Ontologies (directed graphs)</td>
<td>Interoperation operators</td>
<td>Articulation ontologies</td>
<td>Ontology integration</td>
</tr>
<tr>
<td>MAFRA</td>
<td>RDF schemas</td>
<td>Multi strategies (lexical and structural)</td>
<td>Semantic bridging ontology (SBO)</td>
<td>Alignment of distributed ontologies</td>
</tr>
<tr>
<td>S-Match</td>
<td>Ontologies (propositional formulas)</td>
<td>Propositional unsatisfiability problem</td>
<td>Logical relations</td>
<td>Generic framework</td>
</tr>
<tr>
<td>API for ontology alignment</td>
<td>RDF graphs</td>
<td>Provides an API</td>
<td>Simple 1:1 correspondences translated into XSLT, C-OWL,RDF</td>
<td>Generic ontology matching</td>
</tr>
<tr>
<td>iMAP</td>
<td>Database schemas (graphs)</td>
<td>Different machine learn searchers. Use of domain knowledge</td>
<td>1:1 mappings and complex functions</td>
<td>Data integration</td>
</tr>
<tr>
<td>Xu et al.</td>
<td>Database schemas</td>
<td>Different matchers and domain ontologies</td>
<td>Complex mapping expressions</td>
<td>Data integration</td>
</tr>
</tbody>
</table>

Matching tools
A matching transformation is a domain-specific transformation that takes two or more models as input, and that transform them into a new weaving model:

\[ T_{\text{MATCHING}} \rightarrow \text{model} \times \ldots \times \text{model} \rightarrow \text{weaving model} \]

```plaintext
rule CreateLink {
  from
    aLeft : MMa!Class, aRight : MMb!Attribute
  to
    aLink : MMw!Equivalent (
      left <- getID(aLeft),
      right <- getID(aRight),
      similarity <- aLeft.calcSim(aRight)
    )
}
```

Execution condition

Similarity computation [0-1]
Simple matching extension

class Element extends WLinkEnd {
}
class Equivalent extends WLink {
    attribute similarity : Double;
    reference source container : Element;
    reference target container : Element;
}
class <Type>Equal extends Equivalent {
}
class AttributeToRef extends Equivalent {
    reference targetAttribute container : Element
}
Cumulative matching
Cumulative matching

- Different kinds of matching transformations
- Element creation
- Similarity calculation and propagation
- Link rewriting
- Link selection
Matching rule for creating simple links

(rule CPClass {  
  from  
    left : Ecore!EClass, right : Ecore!EClass  
  to  
    AMW!ClassEqual  
})

(rule CPAattr {  
  from  
    left : Ecore!EAttribute, right : Ecore!EAttribute  
  to  
    AMW!AttributeEqual  
})
Cumulative matching: similarity + link filtering

![Diagram illustrating cumulative matching with similarity and link filtering](attachment:image.png)

- **Matching transformation 1**
  - **r₁**
  - **Sim = 0.3**
  - **date**
  - **expiry date**

- **Matching transformation 2**
  - **r₂**
  - **Sim = 0.7**
  - **name**
  - **f_name**

**Weaving Model - Mw₁**

**Weaving Model – Mw₂**

SLIDE FROM: Univ. Rey Juan Carlos
Calculating similarity

- **Simple element-to-element similarities**

  ```plaintext
  rule AttributeSimilarity {
    from
      mmw : AMW!AttributeEqual
    to
      alink : AMW!AttributeEqual (
        similarity <- (mmw.similarity + mmw.left.similarityName(mmw.right)) * weight
      )
  }

- **Structural similarity**

  ```plaintext
  rule UpdateStructuralSim {
    from
      mmw : MMw!Equal mmw.source.isTypeOf(KM3!Attribute) and mmw.target.isTypeOf(SQLDDL!Column))
    to
      alink : MMw!Equal {
        similarity <- ( mmw.similarity +  mmw.source.requiredSim( mmw.target )) * weight
      }
  }
  helper context KM3!Attribute def: requiredSim (column : SQLDDL!Column) : Real =
    if (self.lower = 0 and column.canBeNull) then
      1
    else
      0
    endif;
  ```
Similarity flooding (SF) : a generic structural algorithm

- **Input**
  - Two metamodules $M_a$ and $M_b$,
  - Model elements $a, a' \in M_a$ and $b, b' \in M_b$.
    - Elements $a$ and $a'$ are connected by a labeled edge $(a, "containment", a')$.
    - Elements $b$ and $b'$ are connected by a labeled edge $(b, "containment", b')$.

- **Initial setup and execution**
  - Link creation : Cartesian product of $M_a \times M_b$
  - Similarity assignment for every pair of elements.

- **Iterative propagation**
  - **General idea**: consider the pairs $(a, b)$ and $(a', b')$, with similarities $x$ and $y$, respectively.
    The algorithm propagates $x$ to $(b, b')$ and it updates the similarity value $y$.
  - **Propagation formula**
    - $y = y + (p \times x)$.
  - **Calculation of $p$**
    - number of edges connecting a given pair of elements
    - Ex.: if $(a, a')$ has 10 neighbors, then $p = 1/10$.
  - **Propagation graph** : stores the propagation information.
Adaptation of SF for model weaving and matching transformations

- Choose one kind of structural information
  - Containment graph
  - Inheritance tree
  - Relation graph
  - Any other relations

- Define how to calculate $p$

- Create a propagation weaving model

- Write the propagation transformation
Propagation weaving metamodel extension

- Propagation extension: propagation from one link into another

```java
package mmw_propagation {

class PropagationElement extends WAssociation {
    reference incomingLink : Equivalent;
    reference outgoingLink : Equivalent;
    attribute propagation : Double;
}
}
```

- Creation of propagation elements

```java
rule CreatePropagationElement {

    from
        source_link : AMW!Equivalent,
        target_link : AMW!Equivalent ( <semantic guard> )
    to
        out : AMW!PropagationElement {
            propagation <- 1 / <propagation_value>,
            outgoingLink <- source_link,
            incomingLink <- target_link
        }
}
```
Containment propagation model creation

\textbf{from}

\begin{verbatim}
source_link : AMW!ClassEqual,
target_link : AMW!AttributeEqual ( 
    target_link.getReferredLeft.owner = source_link.getReferredLeft 
    and target_link.getReferredRight.owner = source_link.getReferredRight
)
\end{verbatim}

\textbf{to}

\begin{verbatim}
out : AMW!PropagationElement ( 
    outgoingLink <- source_link, 
    incomingLink <- target_link 
    propagation <- 
    \[ 1 / \]
    ( source_link.getReferredLeft.getAttributeCount()->size() * 
    source_link.getReferredRight.getAttributeCount()->size()
)
\end{verbatim}
Propagation rule: valid for any kind of propagation model

```java
rule PropagationClass {
    from
        mmw : AMW!Equivalent
    to
        alink : AMW!Equivalent()
    do {
        thisModule.aTuple <-
            AMW!PropagationElement.allInstances()->
        select ( e | e.incomingLink = mmw)->
        iterate (el; acc : TupleType(value : Real, count : Integer) =
            Tuple {value = 0, count = 0} |
            Tuple {
                value = acc.value + (el.outgoingLink.similarity * el.propagation),
                count = acc.count + 1
            }
        )
        alink.similarity <- mmw.similarity +
        thisModule.aTuple.value / thisModule.aTuple.count;
    }
}
```
Link filtering

- Called-rule for selecting better similarities

```plaintext
rule getMaxLink (aSource : MMa!ModelElement) {
  using {
    newLink : MMw!Equivalent = null;
    maxSim : Real = 0;
  }
  do {
    for (e in MMw!Equivalent.allInstances() -> select(e.source=aSource)) {
      if (e.similarity > maxSim) {
        maxSim <- e.similarity;
        newLink <- e;
      }
    }
  }
}
```
Link rewriting

- The final similarity weaving model may not have the right connections
  - Nested relationships
  - Hierarchy
  - Others

```plaintext
rule NestedRewriting {
  from
    attr_link : MMw!AttributeEqual,
    class_link : MMw!ClassEqual ( attr_link.source.owner = class_link.source and
    attr_link.target.owner = class_link.target )
  to
    link : MMw!AttributeEqual ( parent <- class_link )
}
```
Link rewriting

AUTOSAR 2.0

PortHW

- ShortName: String
- Name: String

Link Rewriting

AUTOSAR 2.1

PortHardWare

- ShortName: String
- Name: String

Autosar_cp_sim_th.amw

<<link>> PortHW_PortHardWare

- <<left>> PortHW
- <<right>> PortHardware

<<link>> ShortName_ShortName

- <<left>> ShortName
- <<right>> ShortName

Autosar_cp_sim_th_lr.amw

<<link>> PortHW_PortHardWare

- <<left>> PortHW
- <<right>> PortHardware

<<link>> ShortName_ShortName

- <<left>> ShortName
- <<right>> ShortName

Slide from: Univ. Rey Juan Carlos

© Copyright IBM France 2010
Configuration model: setting up an execution chain

Finding the good combination of transformations/parameters is fundamental

Configuration metamodel:

- **Transformations**
  - **ParameterSet**
  - **Metamodel**
- **depends**
  - **Transformation**
    - name: String
    - description: String
    - selected: Boolean
- **metamodels**
  - **LinkGeneration**
  - **ElementToElement**
  - **Structural**
  - **Filter**
    - threshold: Double

**Configuration model**

- **Link generation**
  - Name: Restricted cartesian product
  - Execute: Yes
- **Element to element**
  - Name: Name equality
  - Weight: 0.8
  - Execute: Yes
  - Name: Cardinality
  - Weight: 0.2
  - Execute: Yes
  - Name: Type and conformance
  - Weight: 0.0
  - Execute: Yes
- **Structural**
  - Name: Propagation graph
  - Execute: No
  - Name: Similarity flooding
  - Execute: No
- **Filtering**
  - Name: Select from a threshold
  - Threshold: 0.6
  - Execute: Yes
  - Name: Link rewriting
  - Threshold: 0.0
  - Execute: No
  - Name: Not equivalence
  - Threshold: 0.0
  - Execute: No
  - Name: Normalization
  - Threshold: No
  - Execute: Yes

Save intermediate models: Yes
Transformation production : how to use these weaving models?

- Typical situation
  - Weaving model between 2 metamodels (source and target)
  - Transformation between source and target terminal models

- Based on 3 observations
  - Transformations have frequently-used expressions (e.g., equality, concatenation)
  - Metamodel has link types and link endpoints
  - Transformation languages are similar

- Pattern for generating transformations
  - TransfGen : weaving model $\rightarrow$ transformation model
Some solutions in DB community

- Typically called *query discovery*
- Difficult when using complex mappings
- Specific to the corresponding application domain

<table>
<thead>
<tr>
<th></th>
<th>Input</th>
<th>Mapping nature</th>
<th>Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clio</td>
<td>A pair of relational and XML nested schemas (internal nested format)</td>
<td>1:1/n:m value correspondences</td>
<td>Produces logical operational mappings that are translated in SQL or XSLT</td>
</tr>
<tr>
<td>Kedad et al.</td>
<td>Two or more XML schemas</td>
<td>1:1 value correspondences</td>
<td>XQuery</td>
</tr>
<tr>
<td>An et al.</td>
<td>Relational schemas plus a conceptual model</td>
<td>1:1 value correspondences and the mappings between a schema and its conceptual model</td>
<td>Relational mappings</td>
</tr>
<tr>
<td>SMART</td>
<td>XML schemas and conceptual schemas</td>
<td>1:1 value correspondences with inclusion labels</td>
<td>XML transformations</td>
</tr>
</tbody>
</table>
Higher order transformation. A higher-order transformation is a transformation $T_{OUT} : MM_T = T_{HOT} (T_{IN} : MM_T)$, such that the input and/or the output models are transformation models. Higher-order transformations either take a transformation model as input, either produce a transformation model as output, or both.

TransfGen. TransfGen is a higher-order transformation that takes a weaving model $M_w$ as input and that produces a transformation model $M_T$ as output. The weaving model conforms to a data interoperability metamodel extension $MM_w$.
- $M_T : MM_w = TransfGen (M_w : MM_w)$. 
TransfGen: input metamodel extension
TransfGen: output metamodel

Transformation metamodel
(an abstraction of ATL metamodel)

class Module {
    reference rules [1-*] container : Rule;
}
class Rule {
    attribute name : DataType;
    reference input container: InputElement;
    reference output[*] container: OutputElement;
}
class InputElement {
    reference element : ReferredElement;
    reference condition [0-1] : Expression;
}
class OutputElement {
    reference element : ReferredElement;
    reference bindings [*] : Binding;
}
class Binding {
    reference target : ReferredElement;
    reference source : Expression;
}

Transformation model

rule <name> {
    from
    input (condition)
    to
    output1 (target1 <- source1
                target2 <- source2
                targetN <- sourceN
            ),
    outputN ...
}
rule <name2> ...

Transformation metamodel (an abstraction of ATL metamodel)
TransfGen operation template

1 Module TransfGen (C: ωC)
2
3 inputModel: C /* a correspondence model conforming to a correspondence metamodel ωC */
4 outputModel: T /* a transformation model conforming to ωT */
5
6 rule newModule
7 input WModel
8 output Module
9 rules ← ownedElement (ownedElement isA WLinkST)
10
11 rule newRule
12 input WLinkST (parent isA WModel) /*classifiers (classes, references, attributes)*/
13 output Rule
14 input ← source
15 output ← target
16
17 rule newInput
18 input WLinkEnd (link.source = self)
19 output InputElement
20 element ← ρ (element.ref)
21 condition ← /*depends on the WLinkST and WLinkEnd types*/
22
23 rule newOutput
24 input WLinkEnd (link.target = self)
25 output OutputElement
26 element ← ρ (element.ref)
27 bindings ← link.child /*get the sibling WLinkEnd*/
28
29 rule newExpression
30 input WLinkST (parent isA WLinkST)
31 output Binding
32 source ← MapExp (ρ (source.element.ref)) /*mapping expressions here,*/
33 target ← ρ (target.element.ref) /*according to the WLinkST type*/
Outline

Model weaving: state of the art and concepts
Practical work: schema mapping and traceability
Matching and transformation production

Practical work: matching and transformation production
Two major points
- Transformations are models
- Different transformation metamodels (e.g., ATL or XSLT)
Case study: Comparison and migration – putting all together

- Two versions
  - Scade of Esterel Technologies (v1 and v2)
  - Autosar (v2.0 and v2.1)

- Creation of the weaving model
Comparison and migration (cont’d)

- Transformation generation
  - KM3
  - Scade (v1)
  - Scade (v2)
  - MMw
  - ATL
  - Mw
  - TransfGen
  - Mt (v1-v2)

- Model migration
  - KM3
  - Scade (v1)
  - ATL
  - Scade (v2)
  - Source
  - Mt (v1-v2)
  - Target
Scalability

- **Scade**

<table>
<thead>
<tr>
<th></th>
<th>Elements</th>
<th>Classes</th>
<th>Attributes</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version 1</strong></td>
<td>449</td>
<td>106</td>
<td>105</td>
<td>231</td>
</tr>
<tr>
<td><strong>Version 2</strong></td>
<td>381</td>
<td>95</td>
<td>89</td>
<td>190</td>
</tr>
</tbody>
</table>

**Links:** 379  
**Transformation:** 1030 lines

- **Autosar**

<table>
<thead>
<tr>
<th></th>
<th>Elements</th>
<th>Classes</th>
<th>Attributes</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version 2.0</strong></td>
<td>4569</td>
<td>700</td>
<td>2262</td>
<td>1607</td>
</tr>
<tr>
<td><strong>Version 2.1</strong></td>
<td>6360</td>
<td>1020</td>
<td>3254</td>
<td>2086</td>
</tr>
</tbody>
</table>

**Links:** 3411  
**Transformation:** 7990 lines

- **Remarks**
  - Identical executions
  - Optimized transformations
  - Graphical interface essential
Summary

- Matching
  - Several solutions
  - (semi) automatic creation of weaving models
  - Coupling of transformations and weaving models provide a generic framework
  - Necessary for real world model integration/migration scenarios

- Transformation production
  - Uses the result of a matching operation
  - Generates the final model transformation
  - Relies on higher-order transformations: difficult to write, but quite useful
General conclusions

- Relationships between model elements are ubiquitous
- Several solutions, different application domains, implementations, techniques
- Model weaving
  - Generic MDE solution
  - Simple core that is extended to a given application domain
- Others
  - TGG : transformation by example
  - Model link : simple Ecore2Ecore links
- Several use cases
  - Traceability
  - Model integration and comparison
  - Model merging
  - Annotation
  - Others