dipartimentoinformatica
Università degli Studi dell'Aquila

Model Transformations

SFM-12: Model-Driven Engineering – June 19, 2012

Alfonso Pierantonio

Dipartimento di Informatica
Università degli Studi dell'Aquila

alfonso.pierantonio@univaq.it

Rien ne se perd
rien ne se crée
tout se trasforme

(Antoine-Laurent de Lavoisier)

Rien ne se perd
rien ne se crée
tout se trasforme

(Antoine-Laurent de Lavoisier)

Regardless whether two metamodels are isomorphic *or not, any transformation among them should produce not only models but also their* complement, *somewhat adhering to the Lavoisier's law*

Transformations

- Introduction
- Why Model Transformation languages ?
- Dimensions and Classification

A demonstration of ATL

- Objectives
- Metamodels
- Live Demonstration

Bidirectional Transformation for Change Propagation

- Problem
- Requirements
- Janus Transformation Language
- Change Propagation and non-determinism

Higher-Order Transformations for Automating Co-evolution

- Evolution in MDE
- Metamodel Changes Classification
- Metamodel Differences
- Automated Adaptation

Summary

Introduction

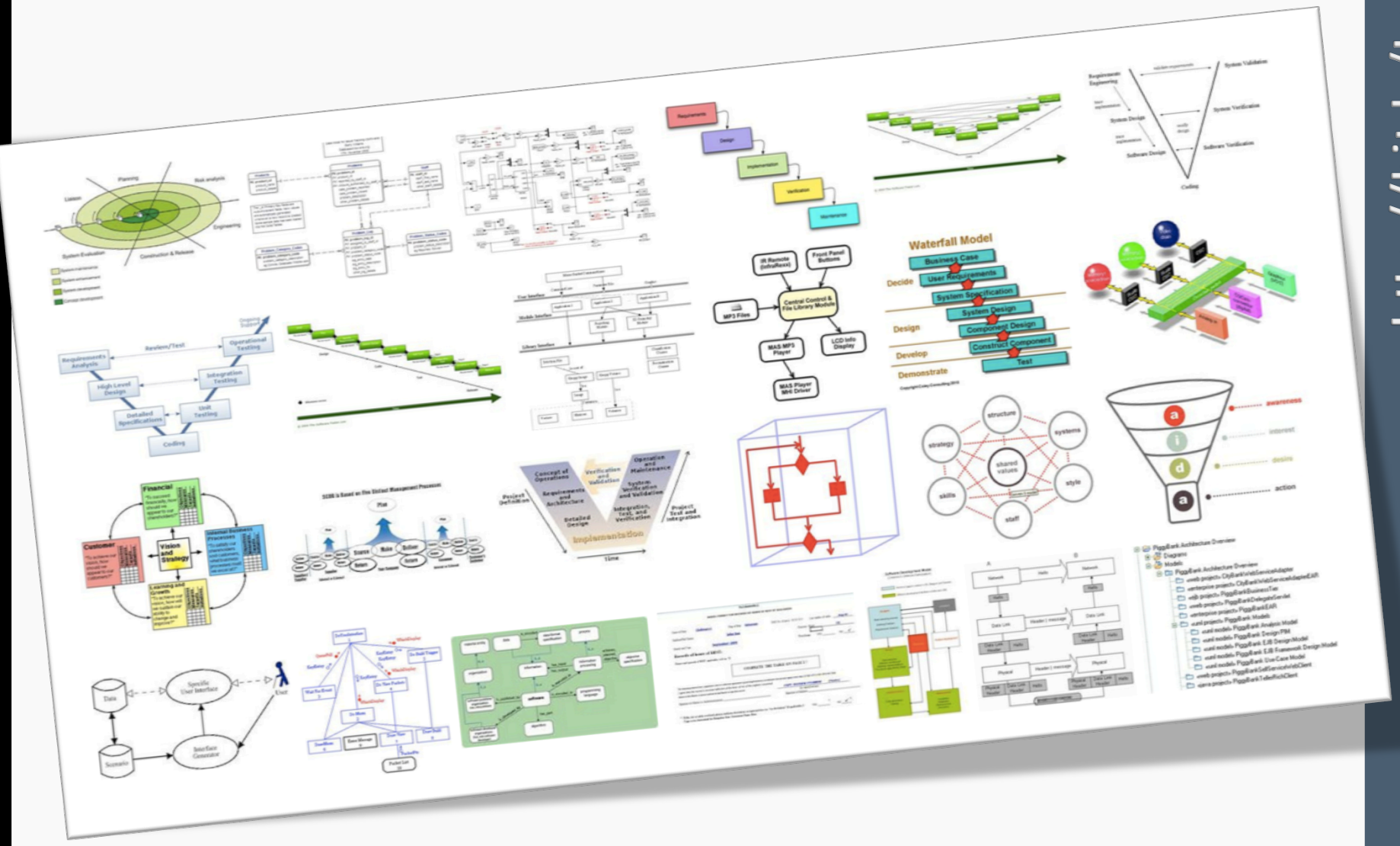
In recent years the movement towards developing software with the use of **models** has increased rapidly

Models are used to design, develop, deploy, and manage technology solutions

A partial list of these models might include

- business cases, use case diagrams, entity relationship models, object models, code, test suites, deployment plans, logical data center models, and exception management plans

Software model in Google images

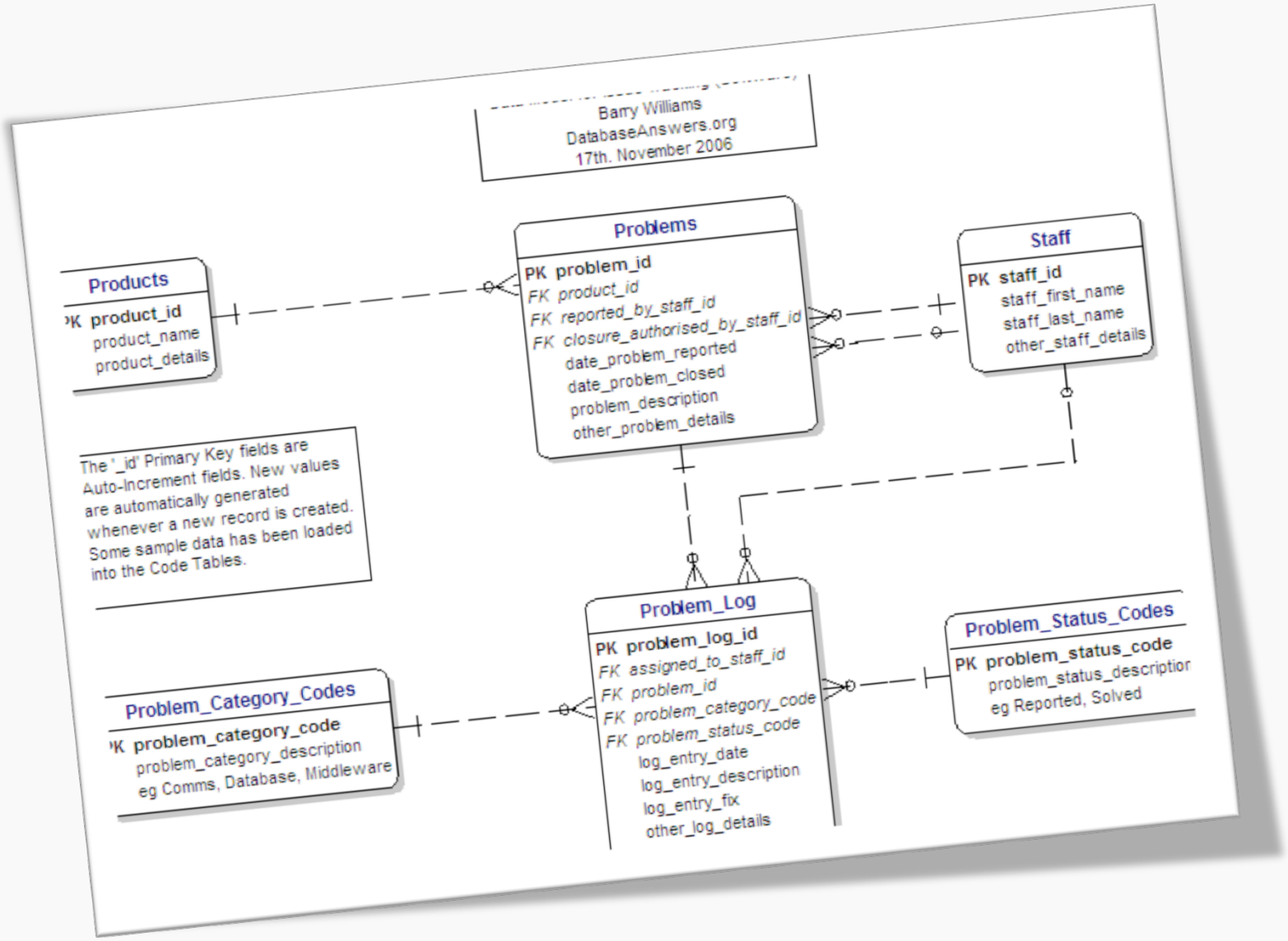


http://bit.ly/LpuIuk

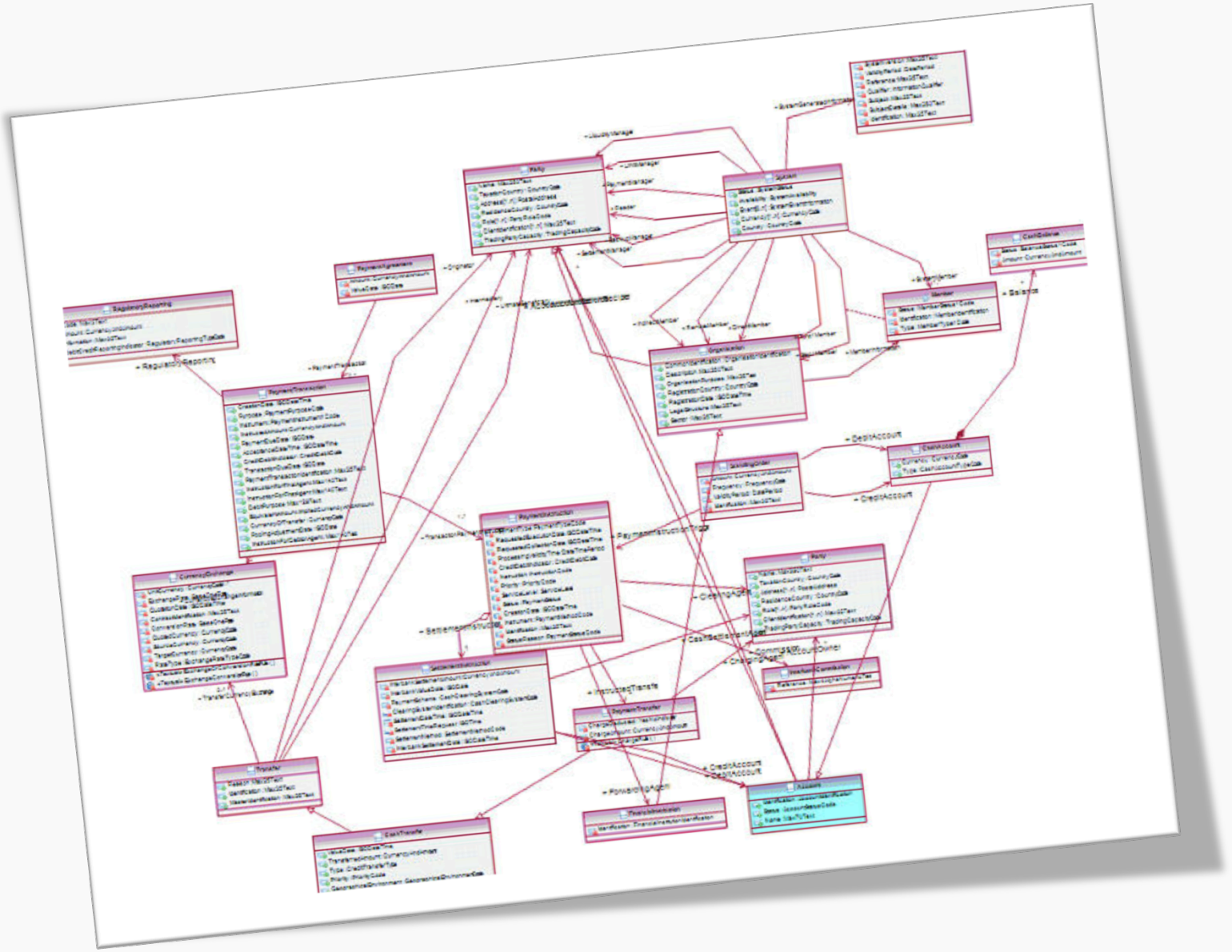
Introduction

Models are employed and characterized according to several dimensions

- domains
  - eg. software systems, embedded software, web apps, etc.
- languages/notations
  - eg. UML, Simulink, WebML, etc.
- concrete syntaxes
  - diagrammatic or textual
- tools and platforms
  - eg. EMF, GME, Kermeta, etc.
- …


Barry Williams
DatabaseAnswers.org
17th. November 2006
Products
PK product_id
product_name
product_details
Problems
PK problem_id
FK product_id
FK reported_by_staff_id
FK closure_authorised_by_staff_id
date_problem_reported
date_problem_closed
problem_description
other_problem_details
Staff
PK staff_id
staff_first_name
staff_last_name
other_staff_details
The '_id' Primary Key fields are Auto-Increment fields. New values are automatically generated whenever a new record is created. Some sample data has been loaded into the Code Tables.
Problem_Log
PK problem_log_id
FK assigned_to_staff_id
FK problem_id
FK problem_category_code
FK problem_status_code
log_entry_date
log_entry_description
log_entry_fix
other_log_details
Problem_Category_Codes
PK problem_category_code
problem_category_description
eg Comms, Database, Middleware
Problem_Status_Codes
PK problem_status_code
problem_status_description
eg Reported, Solved

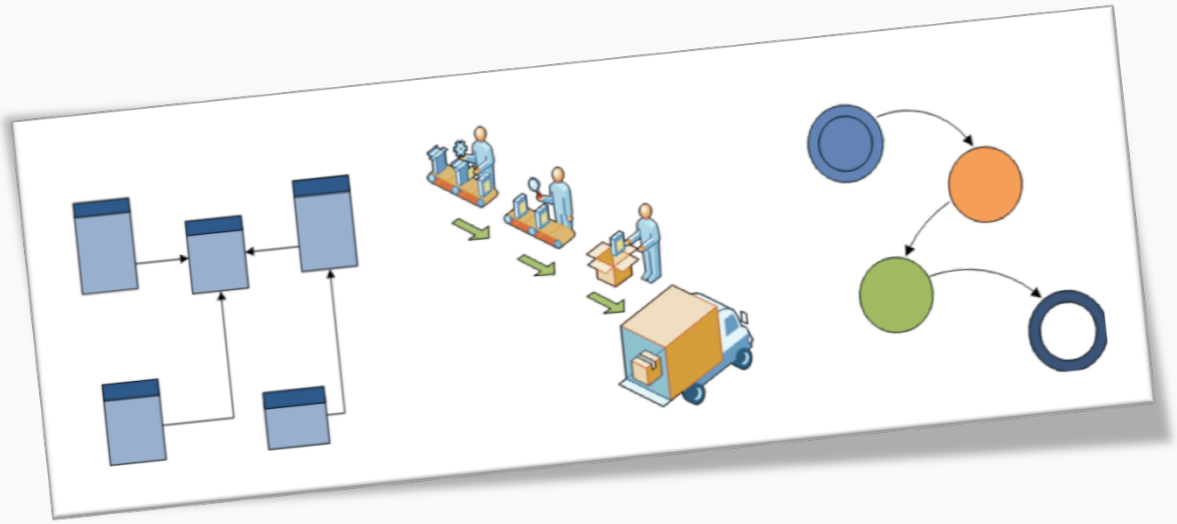
Business object model



Introduction

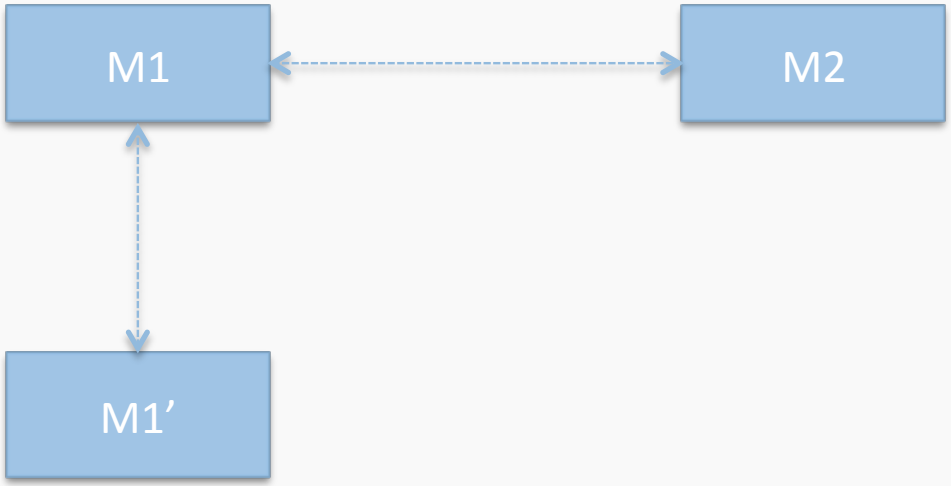
Stakeholders have access to current, accurate, and appropriate representations of the systems, expressed in languages with which each is familiar is relevant

- **Metamodels** permits to describe problems in terms of concepts – and relations among them – proper of the application domain



Transformations

Software lifecycle methodologies have traditionally been making efforts to **automate** the production of *concrete* models from *abstract* ones or even to keep the different system models **synchronized**



Transformations

Software lifecycle methodologies have traditionally been making efforts to **automate** the production of *concrete* models from *abstract* ones or even to keep the different system models **synchronized**

In other words, to leverage automation descriptive models need to be made prescriptive and given a first-class status, ie. models must be formal and processable

Transformations

Model-Driven Engineering (MDE) emphases the use of models not just for documentation and communication purposes, but as first-class artifacts to be **transformed** into other work products

→ e.g., other models, source code, and test scripts

A simple definition of a **model transformation** is that it is a **program** which mutates one model into another

Transformations

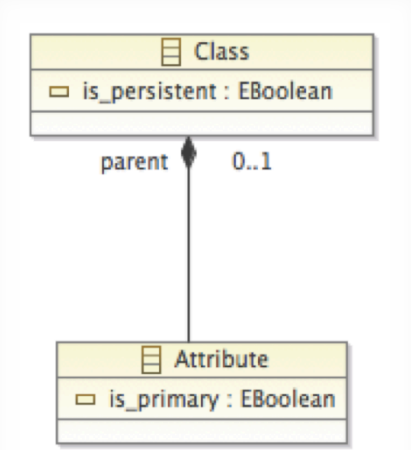
Model transformations are required to perform complex tasks

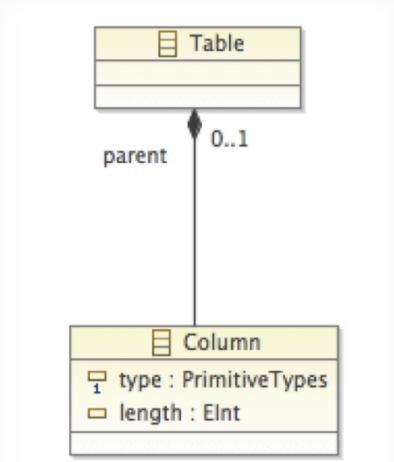
- → e.g., when integrating tools it is frequently required that after an initial transformation of a model from one tool to another, subsequent changes are propagated in a non-destructive manner

Using standard programming languages and libraries to write even simple model transformations is a challenging, tedious and error-prone task

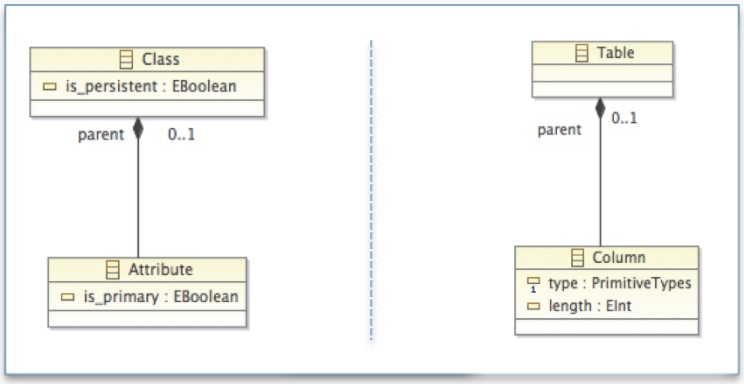
Model transformations need **specialized support** in several aspects in order to realise their full potential

Let us consider a simple transformation for bridging *UML classes* and *RDBMS tables*





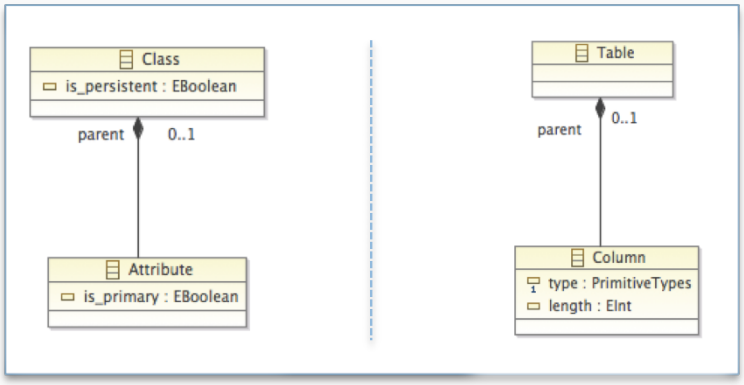
UML Metaclasses		RDBMS Metaclasses
Class	–>	Table
Attribute	–>	Column
“parent”	–>	“parent”



source metamodel

target metamodel

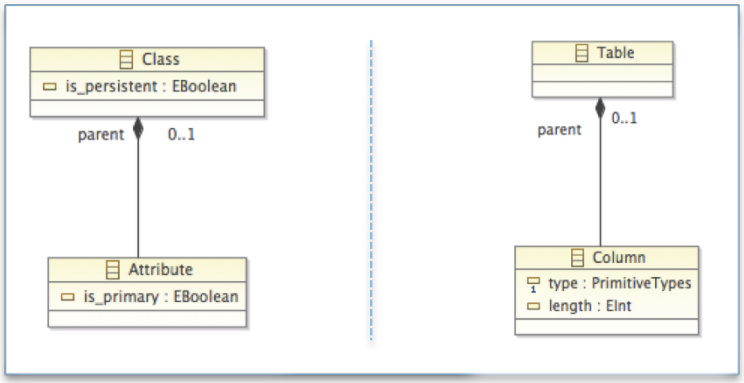
```
rule Class2Table (in source: UML!Class;
                  out target: RDBMS!Table) {

        target.name <- source.name;
}


rule Attribute2Column (in source: UML!Attribute;
                       out target: RBDMS!Column) {

        target.name <- source.name;
        target.type <- source.type;

        target.parent  <-  source.parent;
}
```



source metamodel

target metamodel

```
rule Class2Table (in source: UML!Class;
                  out target: RDBMS!Table) {

        target.name <- source.name;
}


rule Attribute2Column (in source: UML!Attribute;
                       out target: RBDMS!Column) {

        target.name <- source.name;
        target.type <- source.type;

        target.parent  <-  source.parent;
}
```



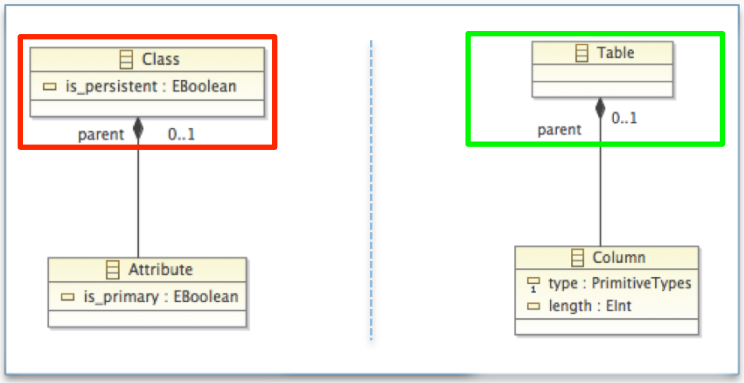
source metamodel

target metamodel

```
rule Class2Table (in sourc
                  out targ

        target.name <- so
}

rule Attribute2Column (in
                       out

        target.name <- source.na
        target.type <- source.ty

        target.parent  <-  source.parent;
}
```

There is a typing issue!

The left-hand side is a table while the right-hand side is a class.



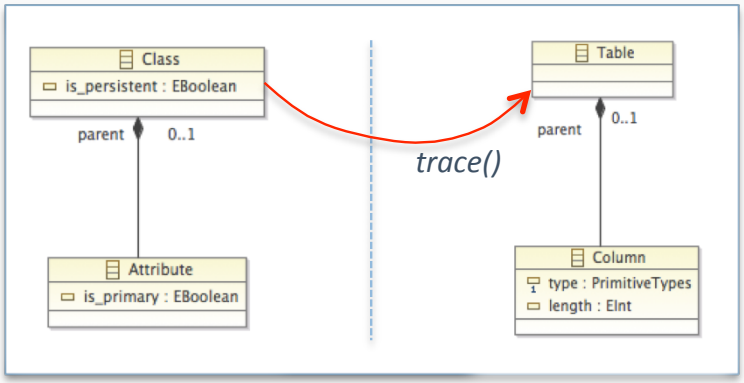
source metamodel

target metamodel

```
rule Class2Table (in sourc
                  out targ

        target.name <- so
}

rule Attribute2Column (in
                       out

        target.name <- source.na
        target.type <- source.ty

        target.parent  <-  source.parent;
}
```

There is a typing issue!

However the intention of the designer is evident: she meant the table, although she wrote the class.



Not just a simple evaluation of the RHS!

Runtime lookup of **tracing information** to find the right element according to the typing of the LHS.

```
Class;
MS!Table) {

me;

 UML!Attribute;
: RBDMS!Column) {

        target.name     rce.name;
        target.type <-  rce.type->;

        target.parent  <-  source.parent; // ~ trace(source.parent);
}
```

Why model transformation languages ?

Model transformations are intrinsically difficult and need specialized linguistic support with

- **accurate pattern matching**, most of the transformation languages are rule-based which means they have an implicit matching algorithm
  - no need to specify control-flow (not completely true, cfr. lazy rules)
- **persistency and change propagation**, information needs to be kept about which elements are related to which by a transformation – this is typically called tracing information
  - a transformation which can record and utilize such information to propagate changes as a persistent transformation

$$T(M_1) \rightarrow M_2$$

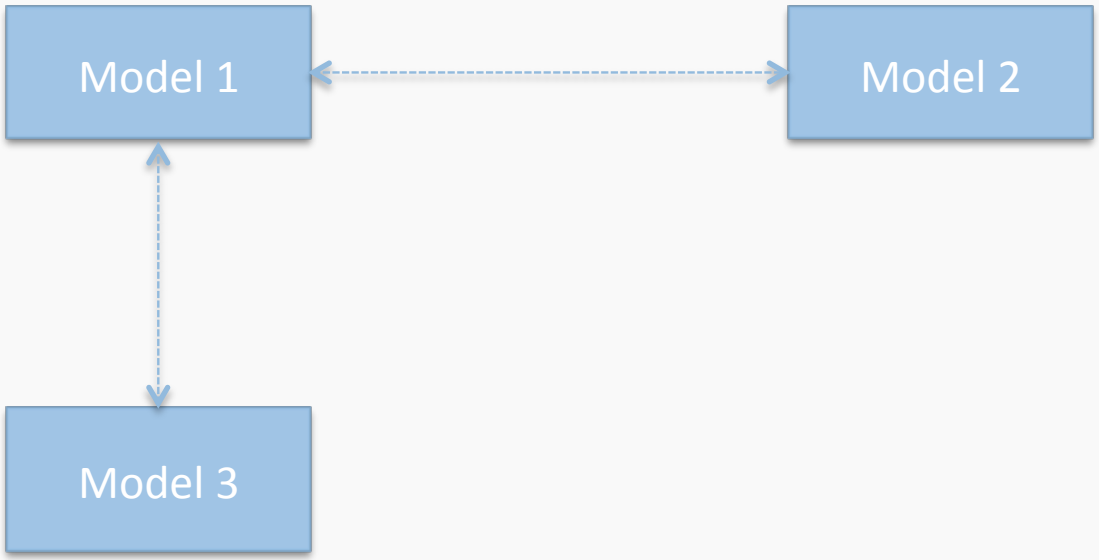
$$T(M_1) \rightarrow \langle M_2, \mathcal{T} \rangle$$

Transformations

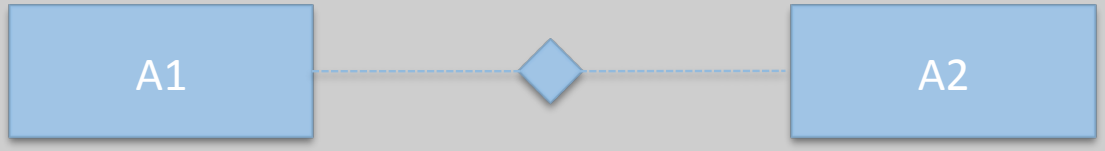
Software methodologies have traditionally endeavoured to **automate** the production of *concrete* models from *abstract* ones or even to keep the different system models **synchronized**

Transformations

Software methodologies have traditionally endeavoured to **automate** the production of *concrete* models from *abstract* ones or even to keep the different system models **synchronized**



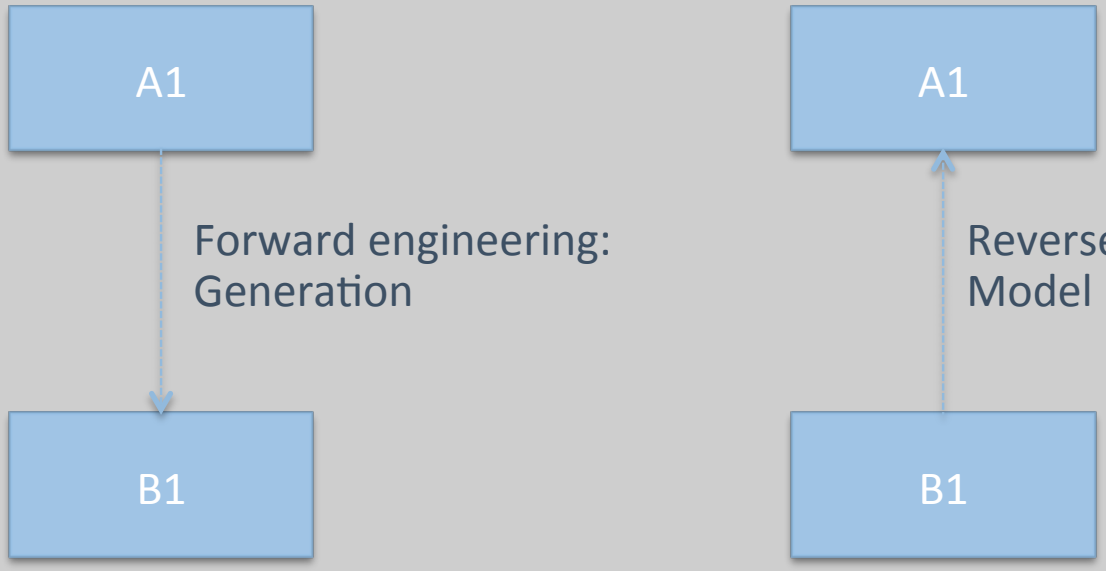
horizontal transformations



Consistency management

Synchronization

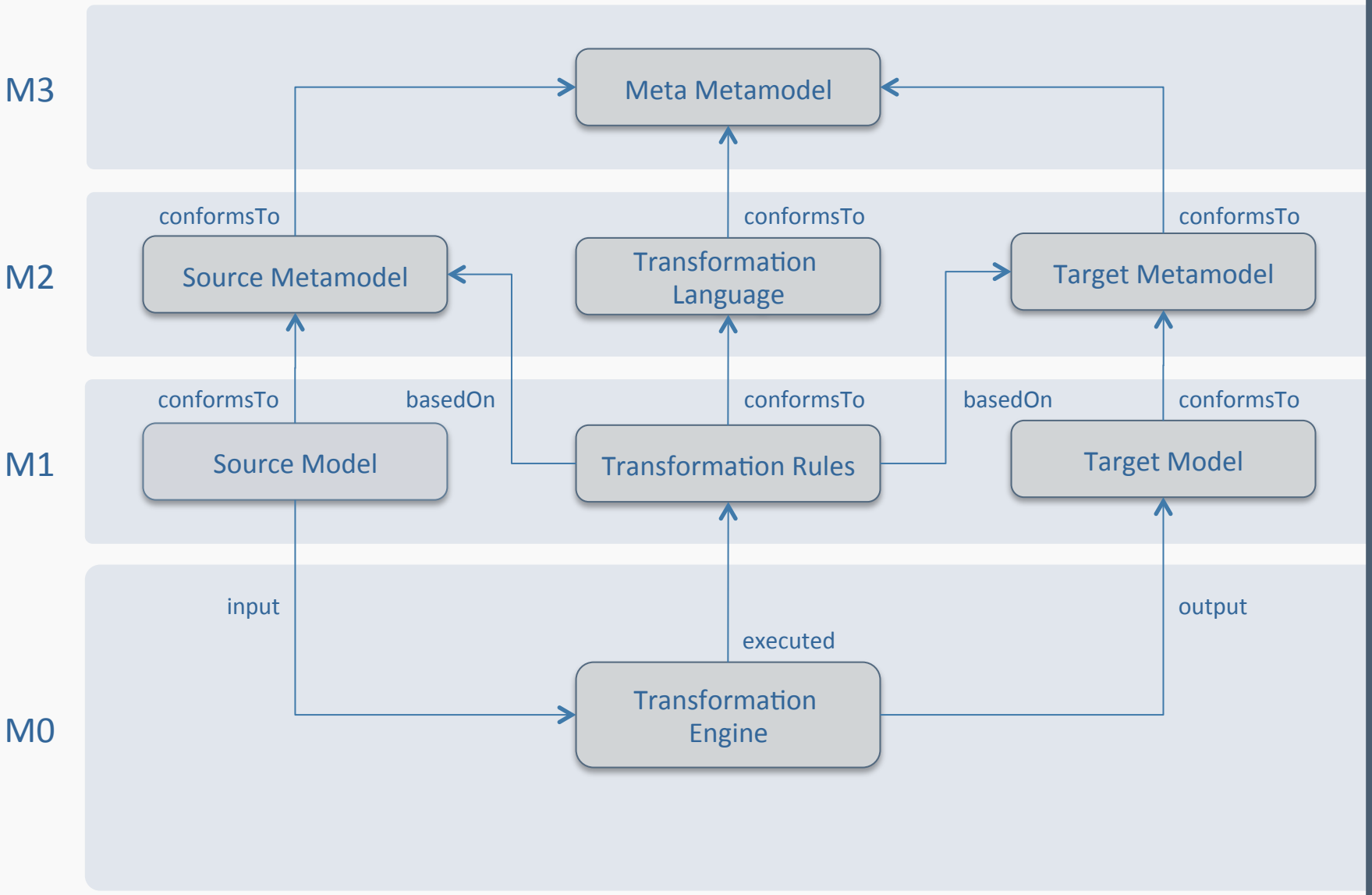
vertical transformations

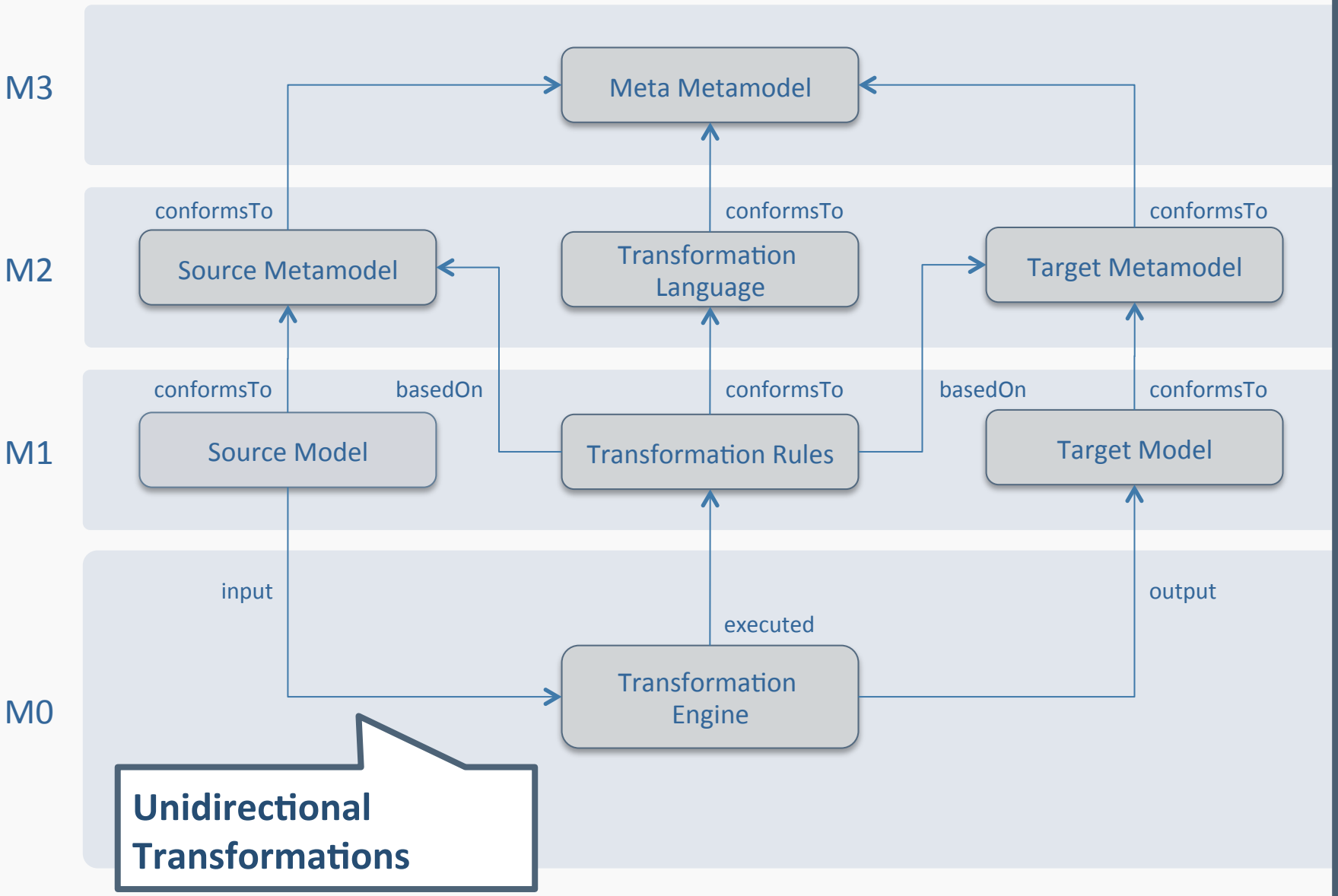


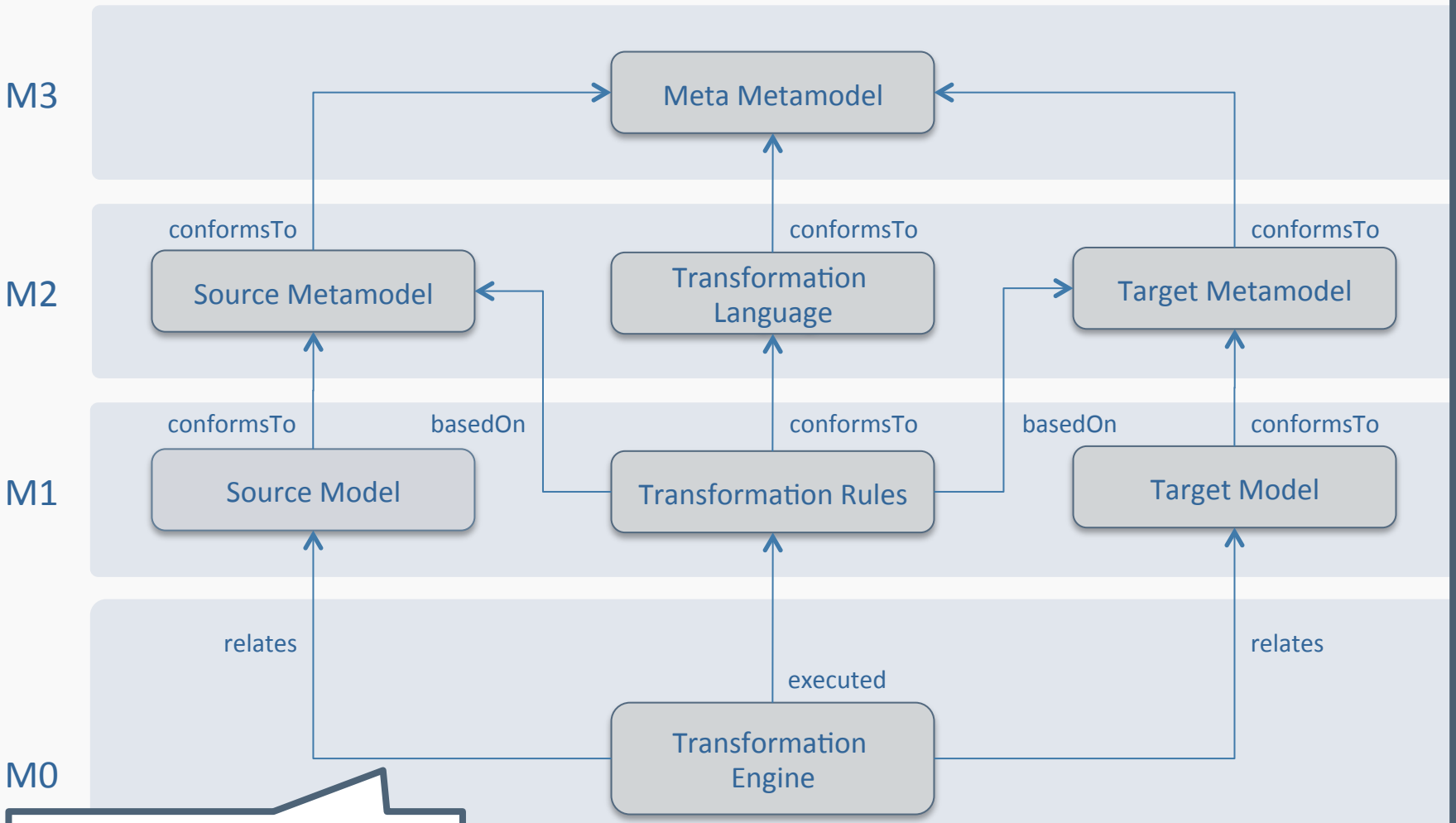
Transformations

Definition. *A* transformation *consists of a set of transformation rules that describe how a model in the source language can be transformed into a model in the target language.*

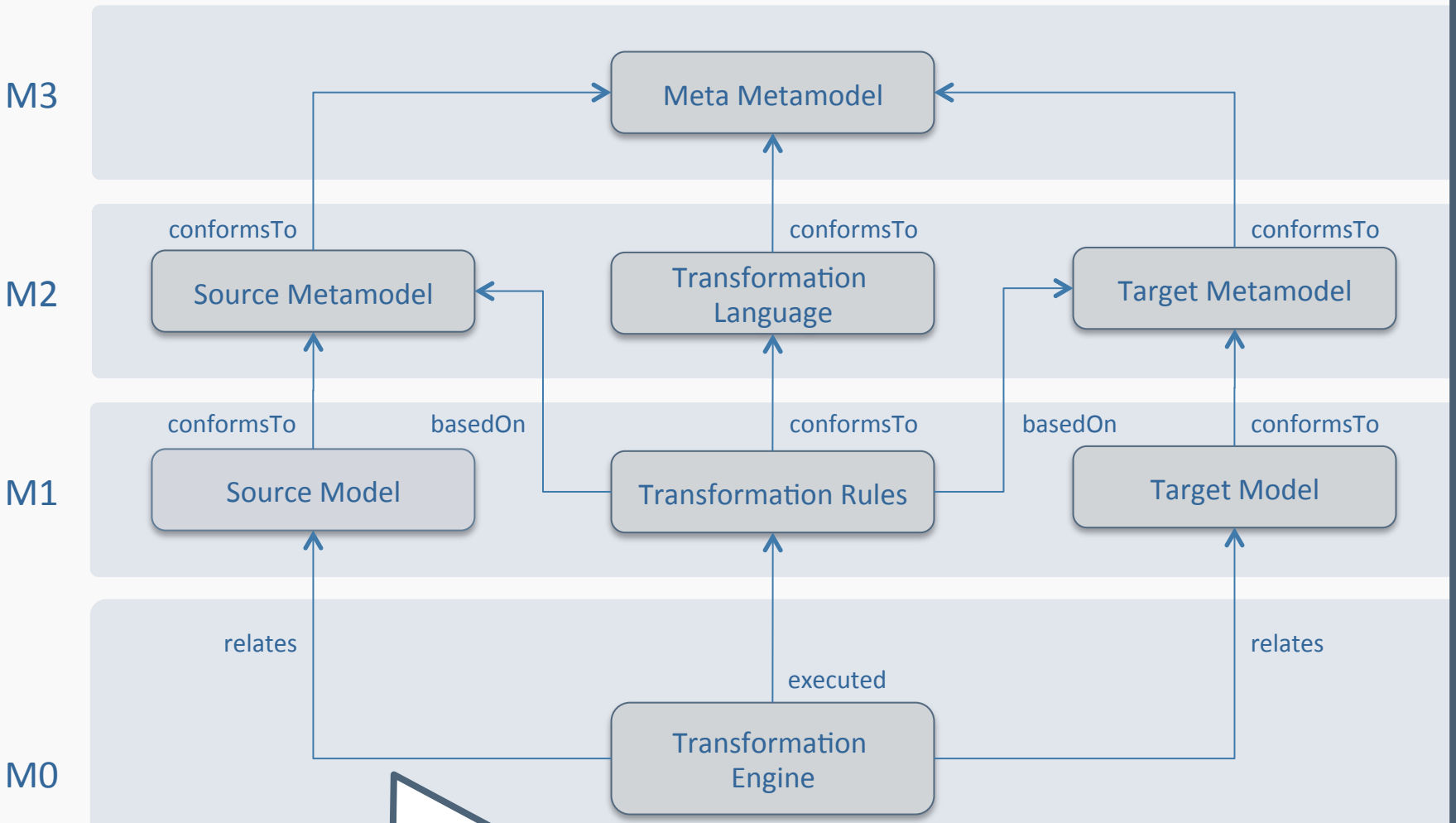
Definition. *A* transformation rule *is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.*


M3
M2
M1
M0
Meta Metamodel
conformsTo
conformsTo
conformsTo
Source Metamodel
Transformation Language
Target Metamodel
conformsTo
basedOn
conformsTo
basedOn
conformsTo
Source Model
Transformation Rules
Target Model
input
executed
output
Transformation Engine

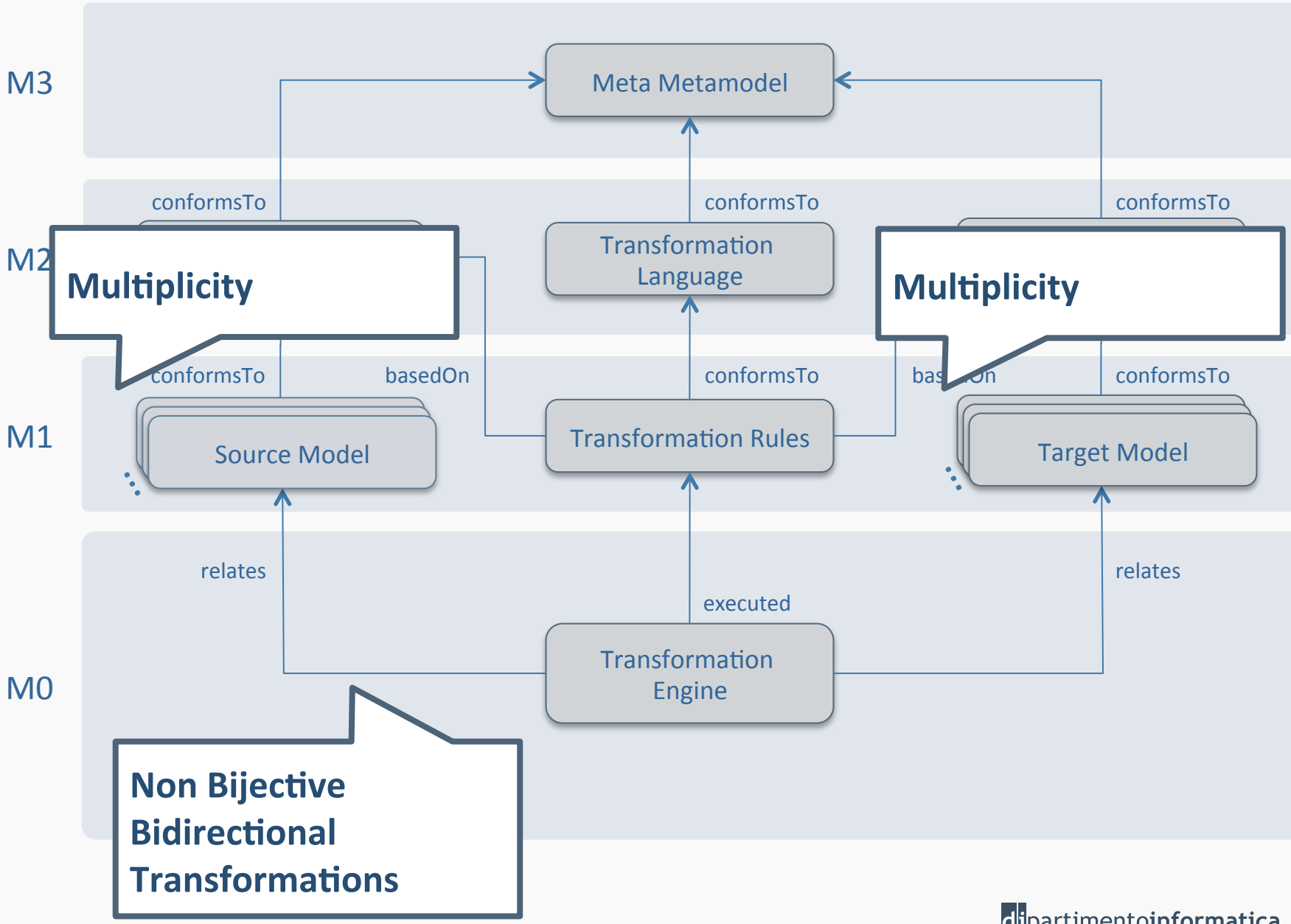

M3
M2
M1
M0
Meta Metamodel
conformsTo
conformsTo
conformsTo
Source Metamodel
Transformation Language
Target Metamodel
conformsTo
basedOn
conformsTo
basedOn
conformsTo
Source Model
Transformation Rules
Target Model
input
executed
output
Transformation Engine
Unidirectional Transformations



Bidirectional Transformations



Bijective Bidirectional Transformations


M3
M2
M1
M0
Meta Metamodel
conformsTo
conformsTo
conformsTo
Multiplicity
Transformation Language
Multiplicity
conformsTo
basedOn
conformsTo
conformsTo
Source Model
Transformation Rules
Target Model
relates
executed
relates
Transformation Engine
Non Bijective Bidirectional Transformations

Classification

A possible classification of model to model transformation languages is the following

- Direct Manipulation
- Operational
- Relational
- Hybrid
- Graph-transformation
- Rule-based

Direct Manipulation

An internal representation of the models is exposed by means of APIs

Users have to typically implement in an object-oriented language from scratch

- → transformation rules
- → scheduling
- → tracing management
- → other facilities

Operational approaches

Similar to the direct manipulation but it offers some dedicated support for model transformations

Typically the metamodeling architecture is extended with a formalism for expressing computation

Languages: QVT operational mappings, XMF, MTL, Kermeta, etc.

Relational Approaches

Declarative approaches based on relations, they can be seen as constraint solving approaches

The relations are specified among source and target element types using constraints

- → sometimes the constraints are executed by external solvers, eg. DLV and Alloy

Side-effect free and implicit target creation, no in-place transformations

Languages: QVT Relations, Tefkat, JTL, etc.

Hybrid Approaches

It combines the characteristics of both relational/ declarative and operation languages

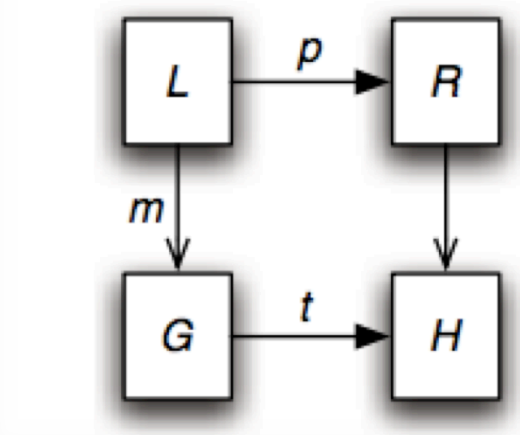
The most prominent languages are ATL and ETL which embodies imperative sections within declarative ones (rules)

Graph Transformations

Graph transformations provide a formal theory and some established formalisms for the automated manipulation of models viewed as graphs

Transformation rules are applied to the source graph in order to obtain the target one

A graph transformation $G \Rightarrow t\, H$ is a pair $\mathrm{t} = (\mathrm{p}, \mathrm{m})$ consisting of a production rule $\mathrm{p} : \mathrm{L} \rightarrow \mathrm{R}$ and a total injective graph morphism (called match) $\mathrm{m} : \mathrm{L} \rightarrow \mathrm{G}$



Languages and systems: AGG, AToM3, VIATRA2, etc

Triple Graph Grammars

TGGs defines correspondences between two different types of models in a declarative way

The correspondences can be made operational by inducing a forward and backward (incremental) transformation

Can be used to synchronize and maintain the correspondences between two models

Languages and Systems: MOFLON, Fujaba

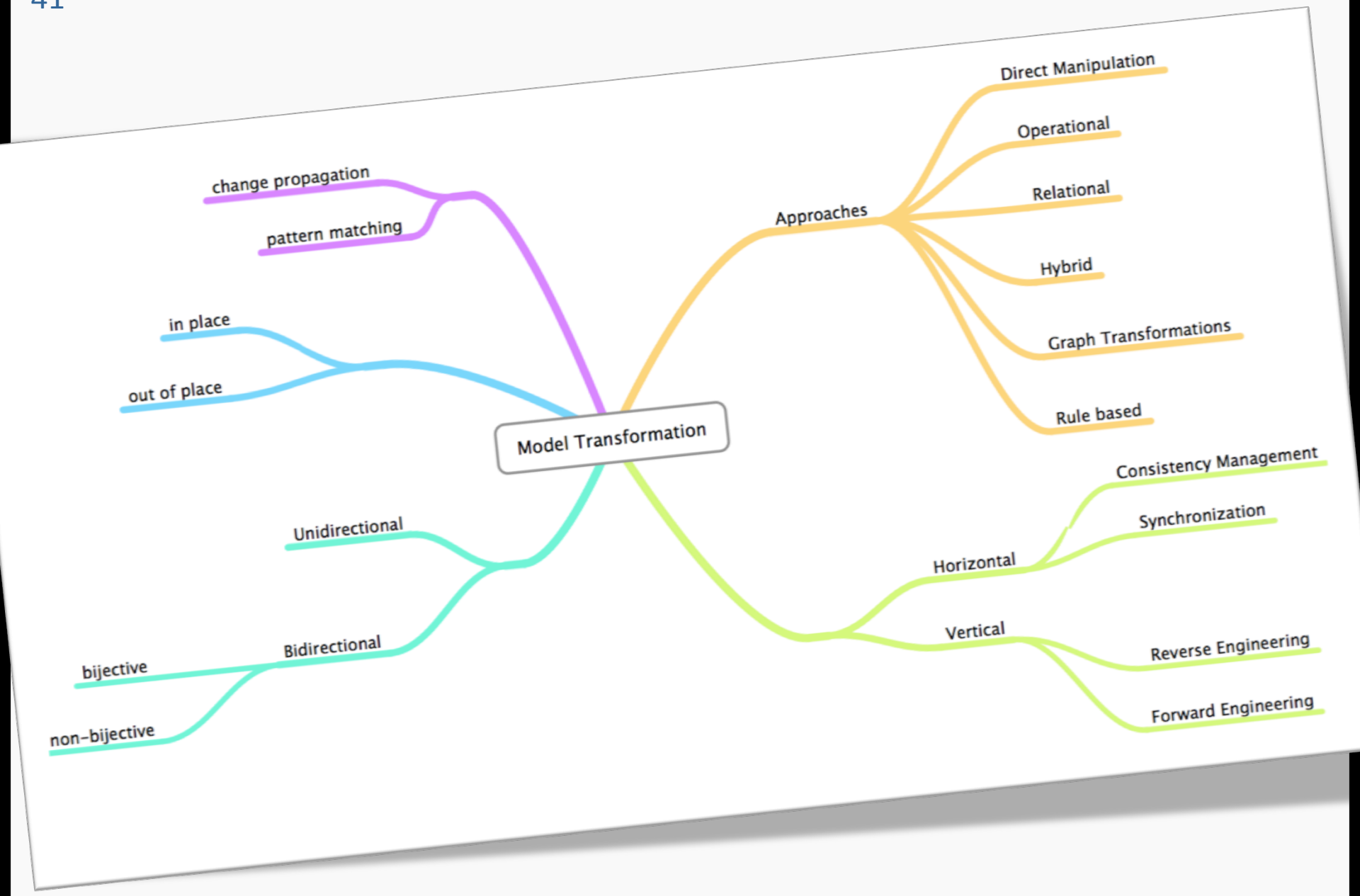
Rule-based Approaches

Some of the mentioned approaches are also rule-based, ie. multiple independent rules

guard => action

The rules are implicitly applied according to the evaluation of the guards in contrast with explicit scheduling of certain approaches

A transformation can be decomposed in rules where some logics is encapsulated within crispy boundaries


Model Transformation
Approaches
Direct Manipulation
Operational
Relational
Hybrid
Graph Transformations
Rule based
Horizontal
Consistency Management
Synchronization
Vertical
Reverse Engineering
Forward Engineering
change propagation
pattern matching
in place
out of place
Unidirectional
Bidirectional
bijective
non-bijective

Transformations

- Introduction
- Why Model Transformation languages ?
- Dimensions and Classification

A demonstration of ATL

- Objectives
- Metamodels
- Live Demonstration

Bidirectional Transformation for Change Propagation

- Problem
- Requirements
- Janus Transformation Language
- Change Propagation and non-determinism

Higher-Order Transformations for Automating Co-evolution

- Evolution in MDE
- Metamodel Changes Classification
- Metamodel Differences
- Automated Adaptation

Summary

A DEMONSTRATION OF ATL

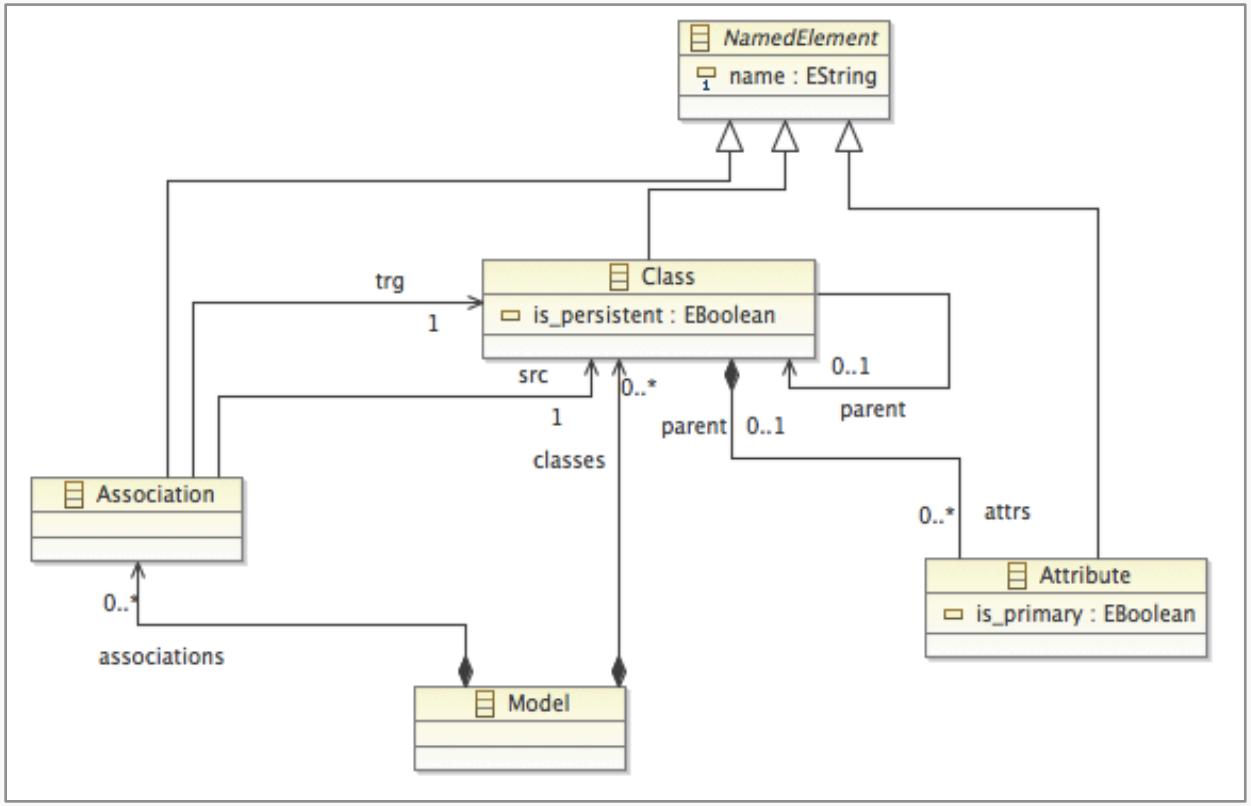
Objectives

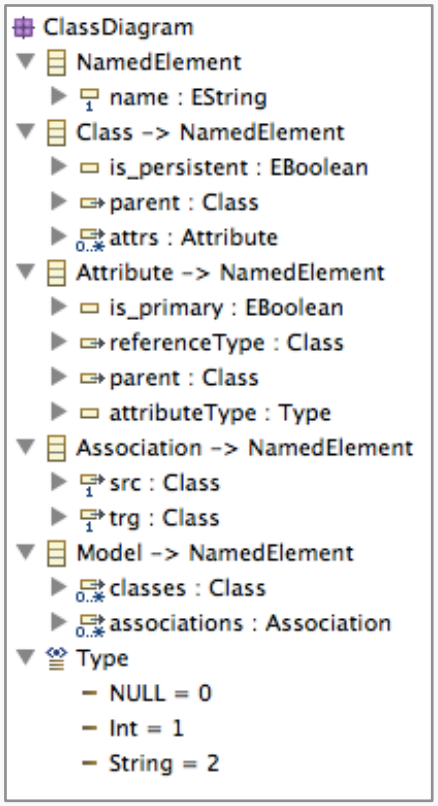
To create a relatively simple transformation for mapping UML to RDBMS (both simplified)

- Metamodels
- Sample UML model
- ATL Transformation
  - outline
  - coding & code illustration

Technology

- Platform: Eclipse EMF
- Meta-metamodel (M3): Ecore



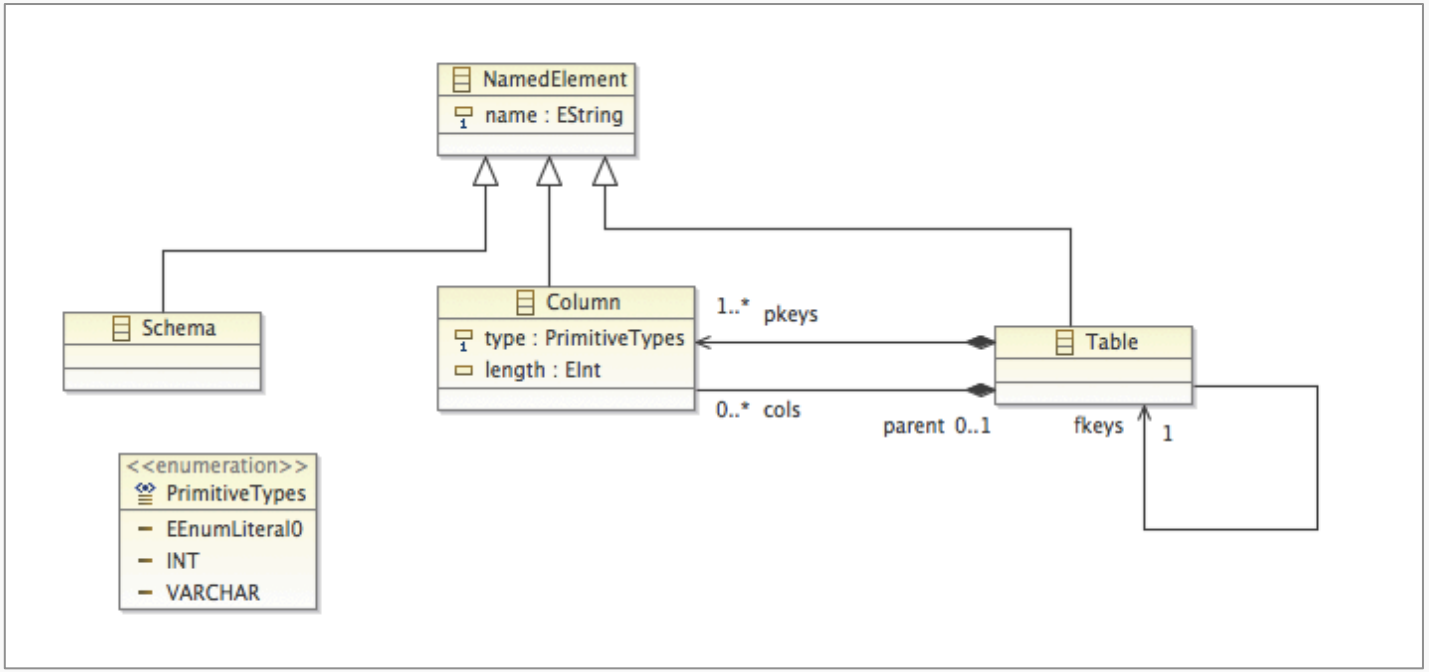


For the sake of simplicity we consider only a simplified version of the class diagrams

Metaclasses

- Model
- Class
- Attribute
- Association

UML Simplified Metamodel



- NamedElement
  - name : EString
- Schema -> NamedElement
  - tables : Table
- Table -> NamedElement
  - pkeys : Column
  - fkeys : Table
  - cols : Column
- Column -> NamedElement
  - type : PrimitiveTypes
  - length : EInt
  - parent : Table
- PrimitiveTypes
  - INT = 0
  - VARCHAR = 1

Metaclasses

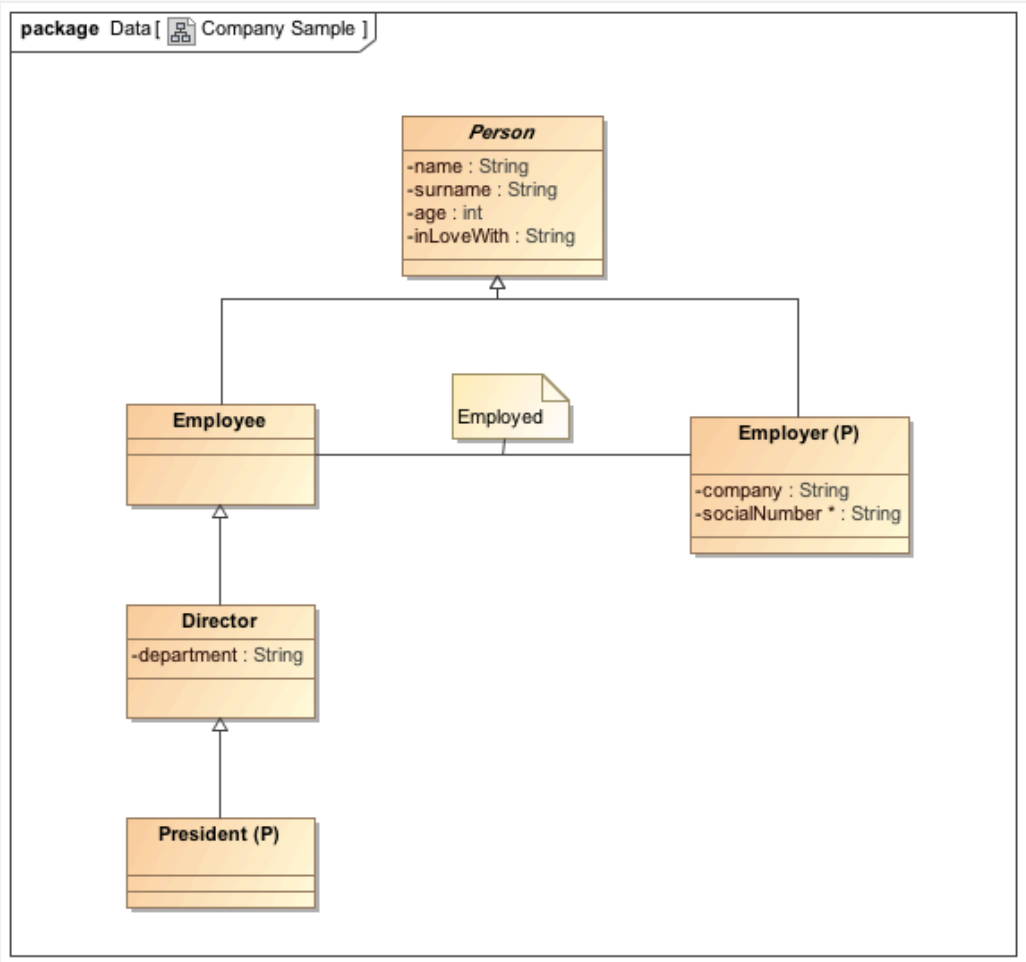
- Schema
- Table
- Column

The ATL program must be written according to the following mapping schema

- The metaclasses are translated according to the following correspondence

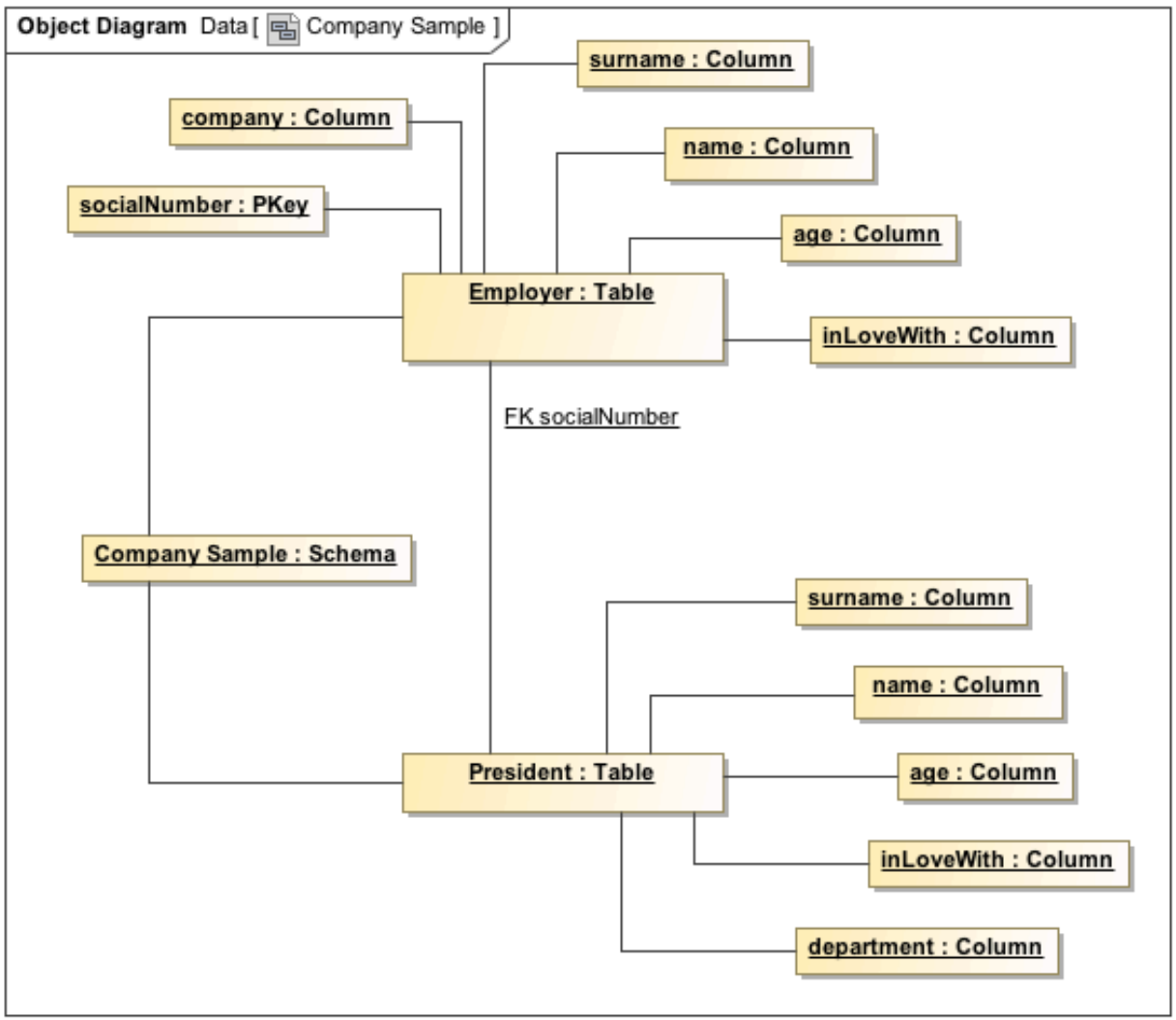
UML Metaclasses		RDBMS Metaclasses
Model	–>	Schema
(persistent) Class	–>	Table
Attribute	–>	Column
Association	–>	*foreign key*

- Columns are collected from all inherited attributes of the superclasses of the persistent class being translated
- Associations source end are assigned to the table corresponding to the highest persistent subclass



EMF tree representation

- Model Company Sample
  - Class Person
    - Attribute name
    - Attribute surname
    - Attribute age
    - Attribute inLoveWith
  - Class Employer
    - Attribute company
    - Attribute socialNumber
  - Class Employee
  - Class Director
    - Attribute department
  - Class President
  - Association Employed



EMF tree representation

- Schema Company Sample
  - Table Employer
    - Column socialNumber
    - Column company
    - Column name
    - Column surname
    - Column age
    - Column inLoveWith
  - Table President
    - Column FK_socialNumber
    - Column department
    - Column name
    - Column surname
    - Column age
    - Column inLoveWith

What is ATL?

ATL is a hybrid model transformation language

It permits to generate a (fixed) number of target models starting from a (fixed) number of source models – no non-determinism!

An ATL transformation program is composed of rules that define how source model elements are **matched** and **navigated** to create and initialize the elements of the target models

- → the creation is implicit as rules are triggered by the matches
- → the initialization is explicit and defined within the rules

The structure of a program

```
Module <name>;
create {<targetModel>: <targetMM>} from {<sourceModel>: <sourceMM>};

{<helper definition>}

{<lazy rules>}

{<matched rules>}
```

Matched rules

A matched rule specifies the way target model elements must be generated from source model elements, ie

- → which source model element(s) must be matched,
- → the number and the type of the generated target model element(s), and
- → the way these target model element(s) must be initialized from the matched source element(s)

A matched rule is invoked implicitly

```
rule Model2Schema {

        from s:UML!Model
        to t:RDBMS!Schema (
                name <- s.name,
                tables <- s.classes
        )
}
```

Matched rules

A matched rule specifies the way target model elements must be generated from source model elements, ie

- which source model element(s) must be matched,
- the number and the type of the generated target model element(s), and
- the way these target model element(s) must be initialized from urce element(s)

A r s invoked implicitly

rule name

```
rule Model2Schema {

        from s:UML!Model
        to t:RDBMS!Schema (
                name <- s.name,
                tables <- s.classes
        )
}
```

Matched rules

A matched rule specifies the way target model elements must be generated from source model elements, ie

- → which source model element(s) must be matched,
- → the number and the type of the generated target model element(s), and
- → the way these target model element(s) must be initialized from the matched source element(s)

A matched rule is invoked implicitly

Match over the source model

```
rule Model2Schema {

        from s:UML!Model
        to t:RDBMS!Schema (
                name <- s.name,
                tables <- s.classes
        )
}
```

Matched rules

A matched rule specifies the way target model elements must be generated from source model elements, ie

- → which source model element(s) must be matched,
- → the number and the type of the generated target model element(s), and
- → the way these target model element(s) must be initialized from the matched source element(s)

A matched rule is invoked implicitly

```
rule Model2Schema {

        from s:UML!Model
        to t:RDBMS!Schema (
                name <- s.name,
                tables <- s.classes
        )
}
```

Target element to be **created**

Matched rules

A matched rule specifies the way target model elements must be generated from source model elements, ie

- which source model element(s) must be matched,
- the number and the type of the generated target model element(s), and
- the way these target model element(s) must be initialized from the matched source element(s)

A matched rule is invoked implicitly

```
rule Model2Schema {

        from s:UML!Model
        to t:RDBMS!Schema (
                name <- s.name,
                tables <- s.classes
        )
}
```

Additional ingredients

Besides matched rules ATL provides

- **lazy rules**: similar to matched rules but do not perform implicit matching, thus they have to be explicitly invoked
- **helpers**: computational "read-only" units which are used when the structure of computation does not align to the metamodel structure, eg. transitive closures, exceptions, etc.

DEMO

Some considerations

Model transformations are at the core of MDE

A partial list of the difficulties

- **scalability**, transformations do not scale well
- **reuse** is not easy, but we are witnessing an old ideas rentrée
- the way a transformation is decomposed is driven by the metamodel structure/granularity
  - a complete alignment between metamodel and transformation structure is limited to trivial cases
- **agility**, EMF conformance relies on the Java strong typing
- a killer app is missing, ATL is an academic product used by industry
- etc.

Eclipse EMF workspace

The workspace of the demonstration with

- Metamodels
  - UML.ecore
  - RDBMS.ecore
- Models
  - Model_UML.xmi
  - DEMO_Model_UML.xmi (the one used during the demo)
  - Model_RDBMS.xmi
- Transformation
  - DEMO_UML2RDBMS.atl (the one build during the demo)
  - UML2RDBMS.atl

can be downloaded from

- **http://bit.ly/KCYe2o**

Transformations

- Introduction
- Why Model Transformation languages ?
- Dimensions and Classification

A demonstration of ATL

- Objectives
- Metamodels
- Live Demonstration

Bidirectional Transformation for Change Propagation

- Problem
- Requirements
- Janus Transformation Language
- Change Propagation and non-determinism

Higher-Order Transformations for Automating Co-evolution

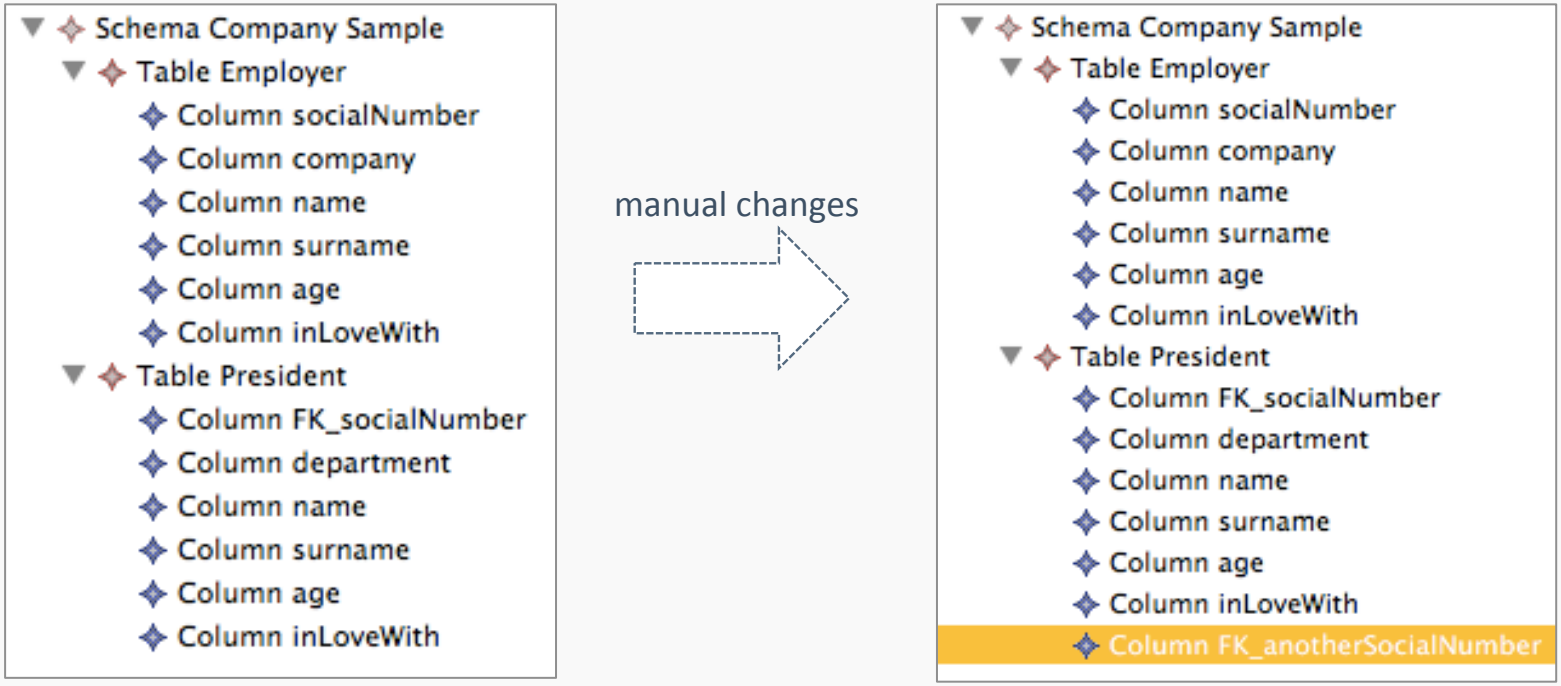
- Evolution in MDE
- Metamodel Changes Classification
- Metamodel Differences
- Automated Adaptation

Summary

BIDIRECTIONALITY AND CHANGE PROPAGATION

Problem

It is possible to back propagate manual changes operated on the outcome of a transformation ?

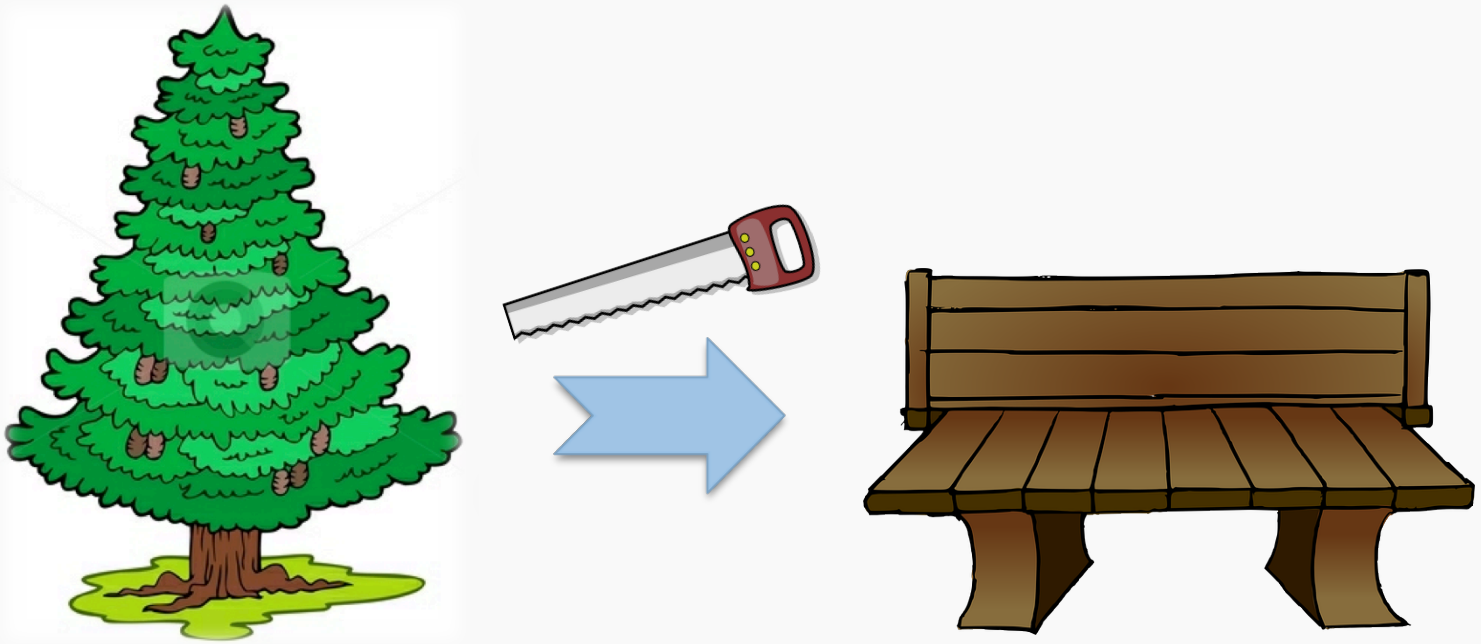


Raising the level of abstraction

Reverse engineering

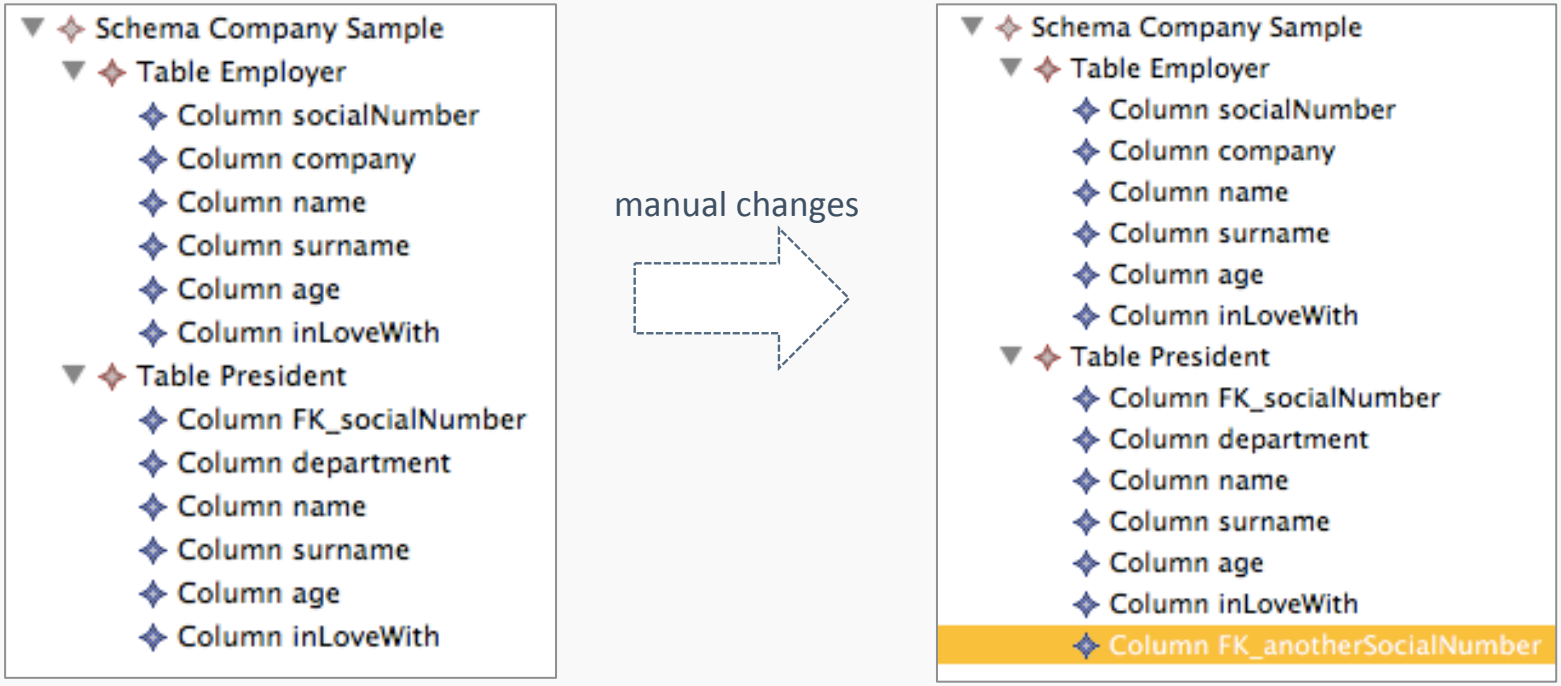
NB: Slide idea borrowed from an **itemis AG** presentation

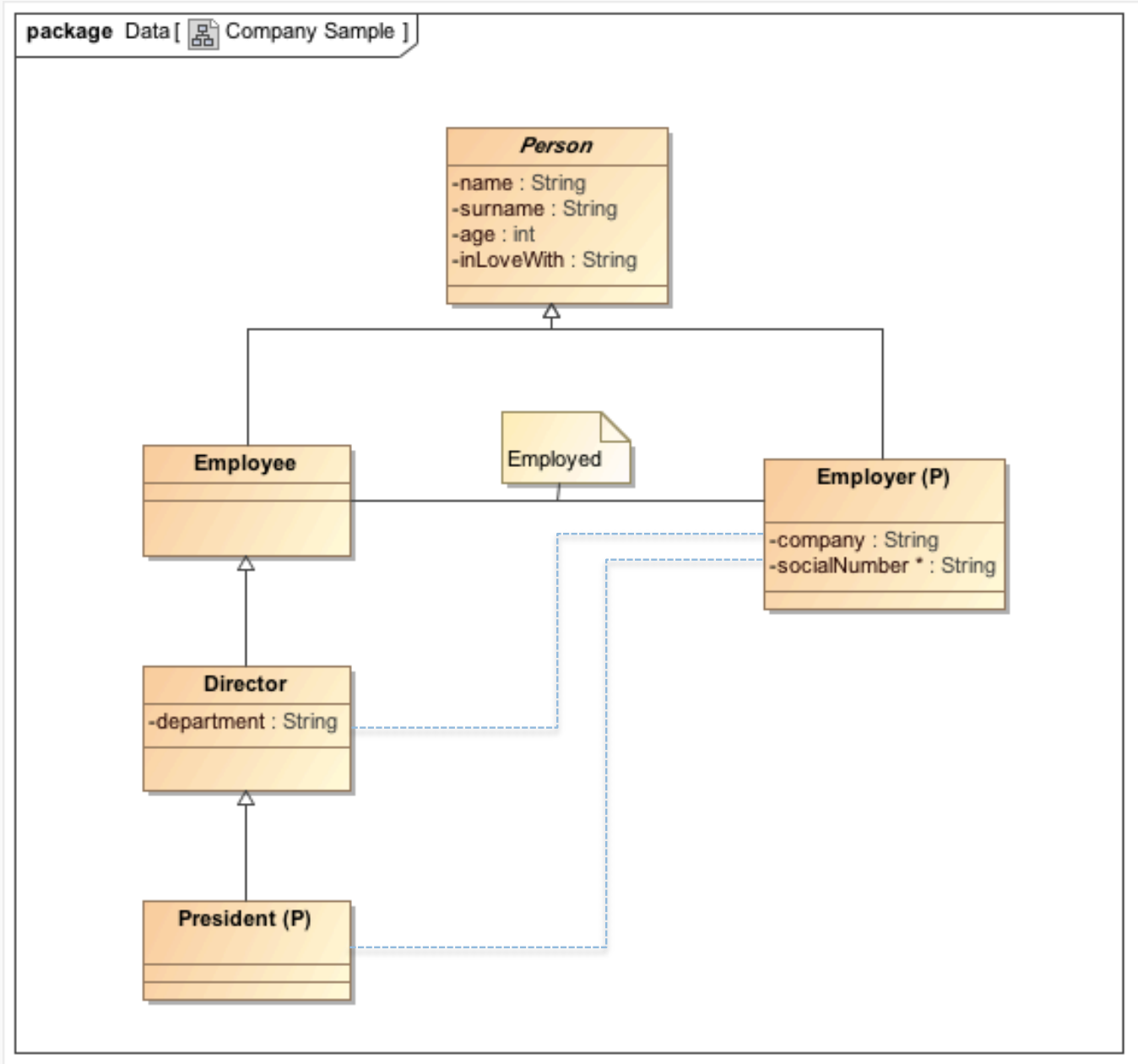
(Thanks Bran!)

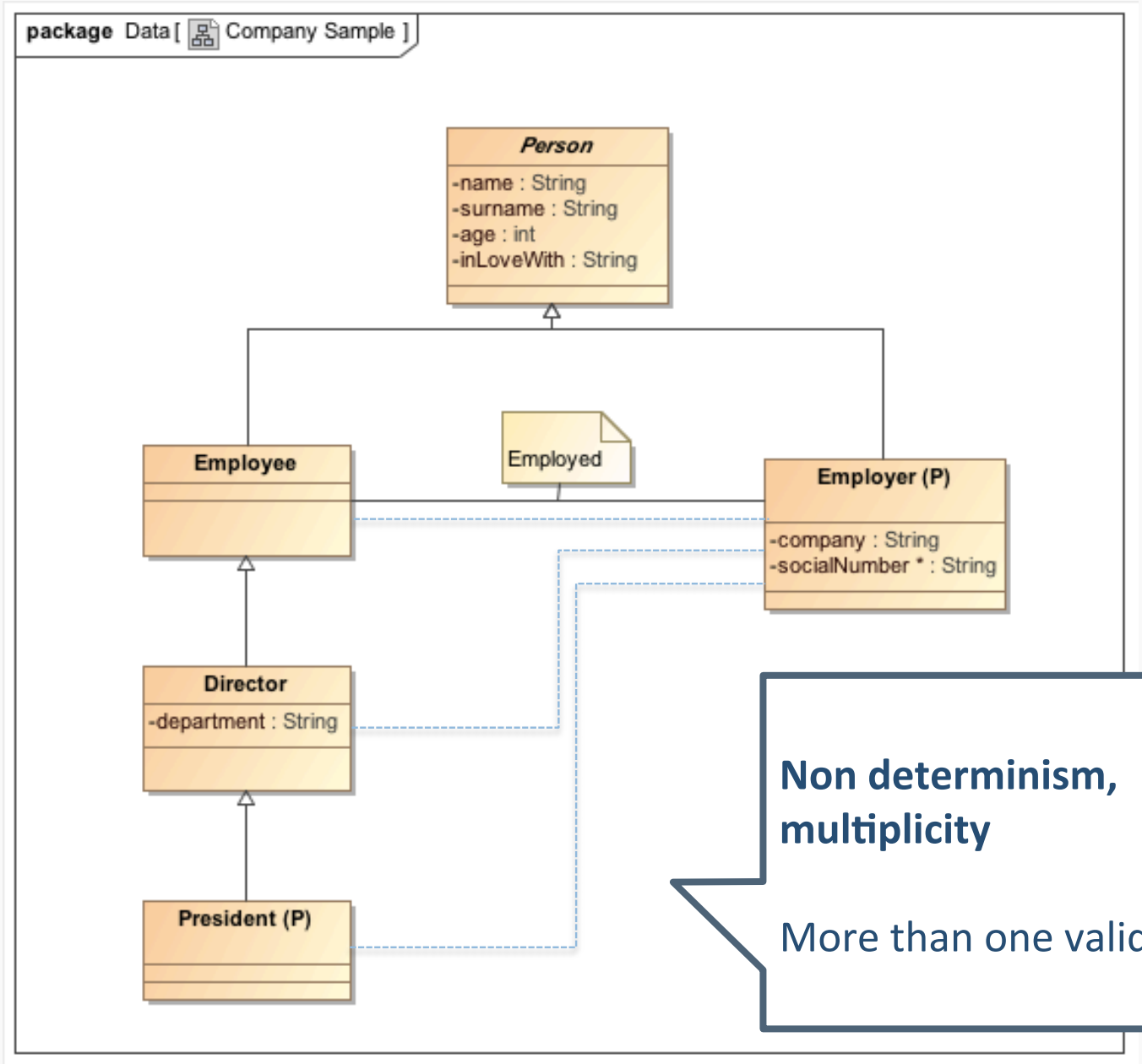


Problem

It is possible to back propagate manual changes operated on the outcome of a transformation ?




package Data [Company Sample]
Person
-name : String
-surname : String
-age : int
-inLoveWith : String
Employee
Employed
Employer (P)
-company : String
-socialNumber * : String
Director
-department : String
President (P)



Non determinism, multiplicity

More than one valid model

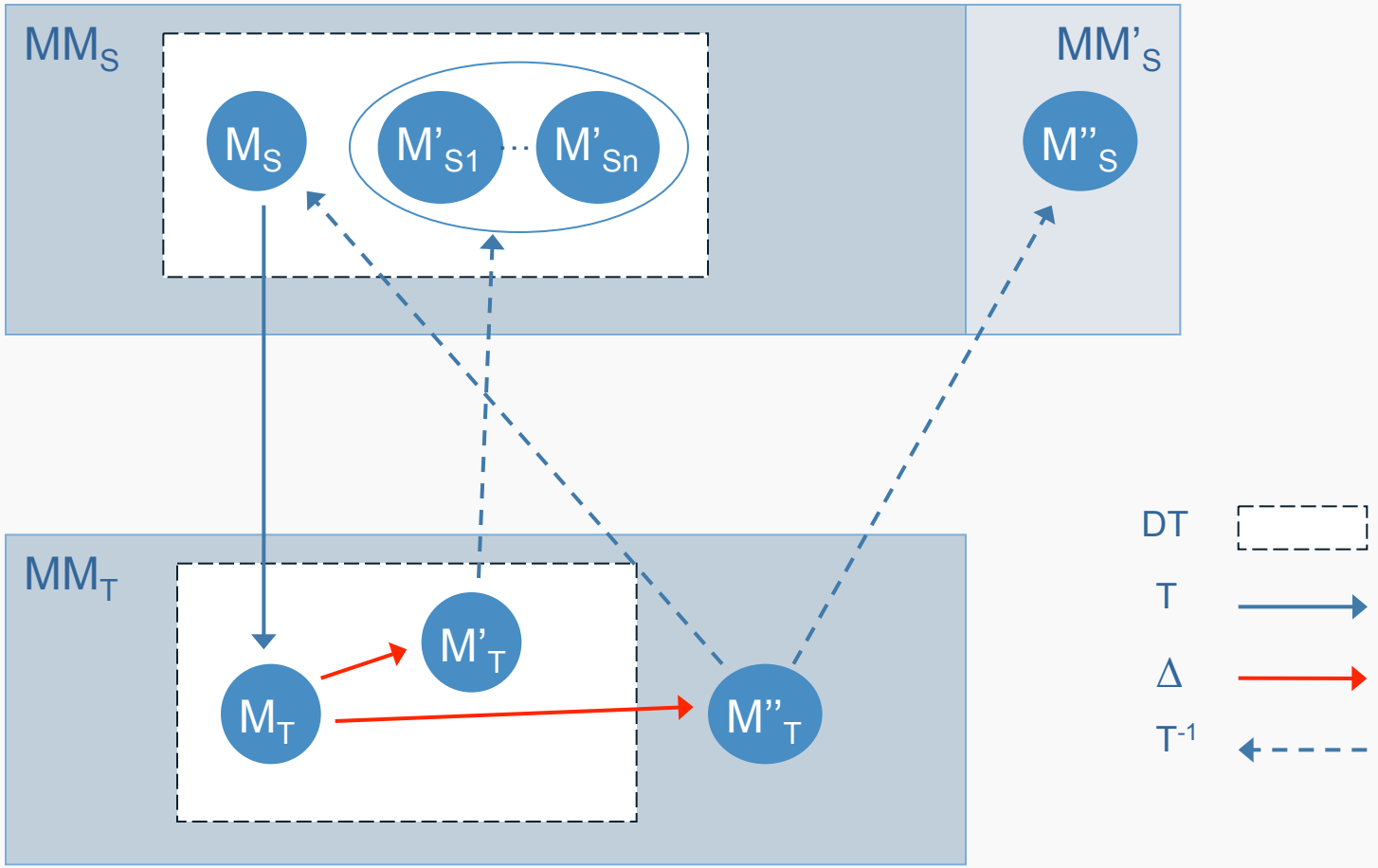
Bidirectionality and Change Propagation

The relevance of bidirectionality in model transformations has been advocated already in 2005 by OMG by including a bidirectional language in QVT

Bidirectional transformations are useful for maintaining the **consistency** of two (or more) related sources of information

- Transformations may be **non-bijective**: given a source model, there could be more than one target model which correctly related to the source
- Transformations may be **not total**: only the relevant concepts of the source models are mapped toward the corresponding target elements

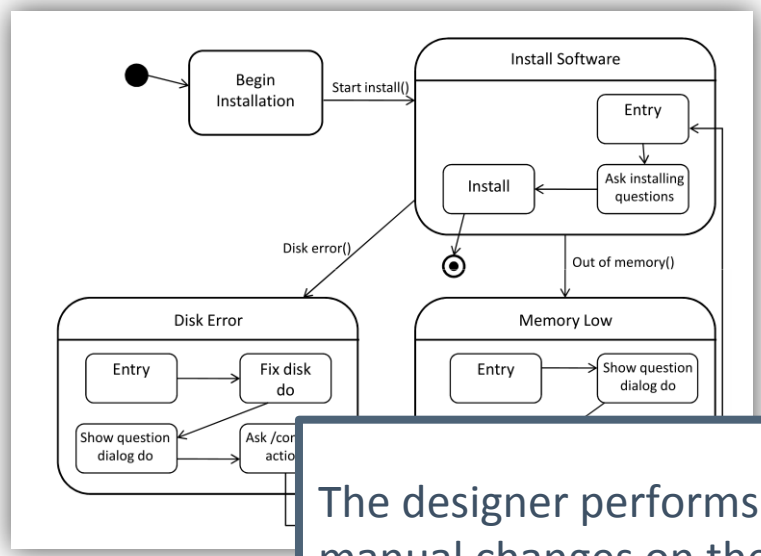
Bidirectionality and Change Propagation



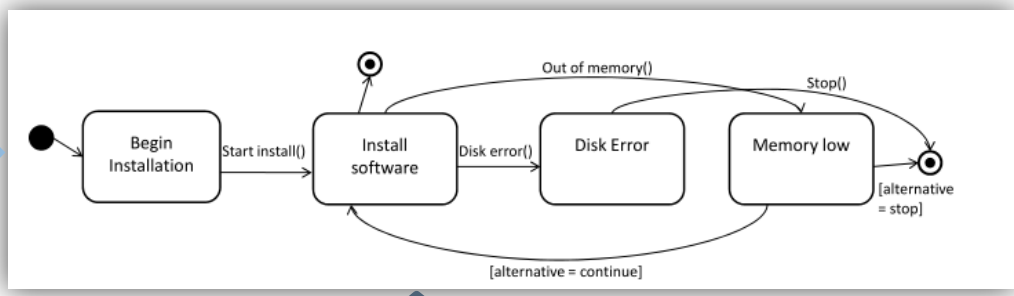
The designer may need to manually modify the generated model to resolve unforeseen requirements or limited expressiveness of the source metamodel

source

target



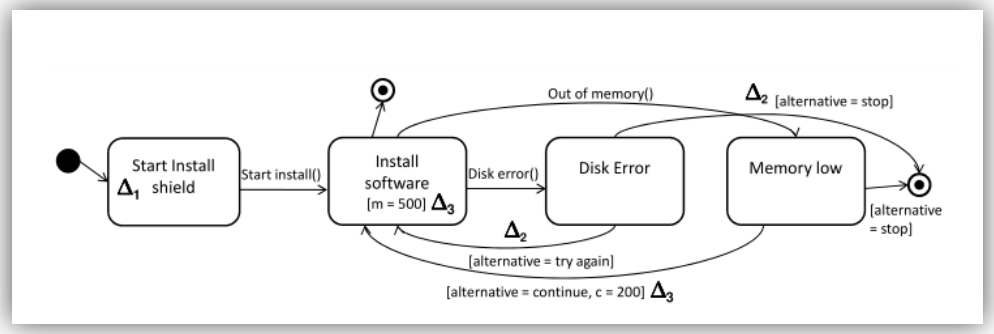
T



The designer performs some manual changes on the generated model

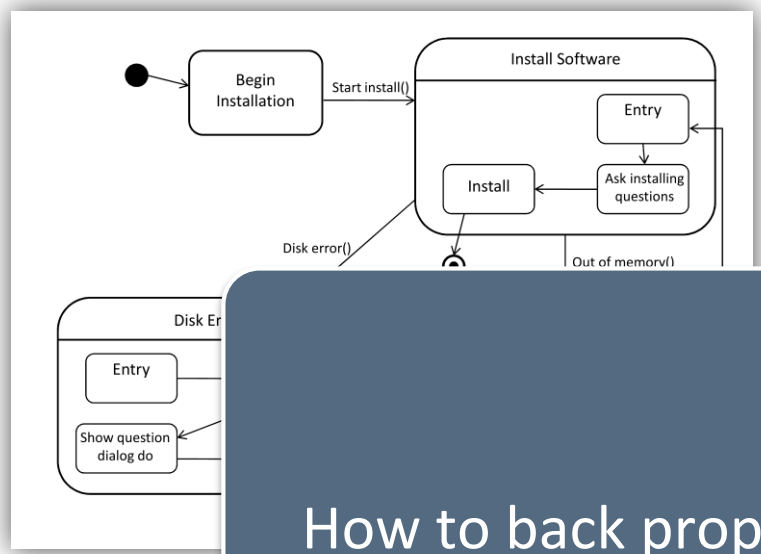
Hiera

Non-hierarchical state machine obtained by flattening the source model

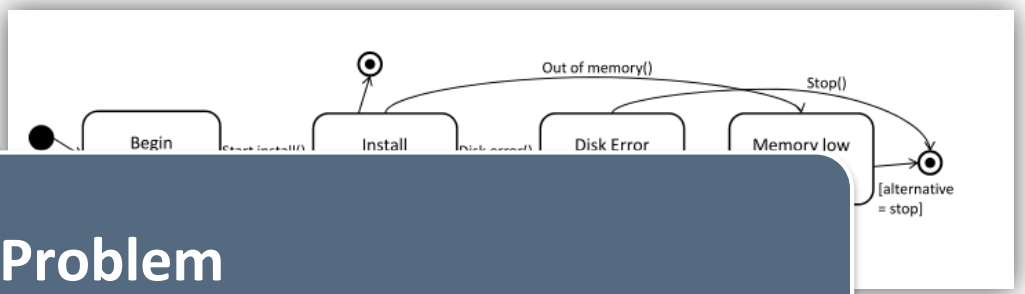


source

target



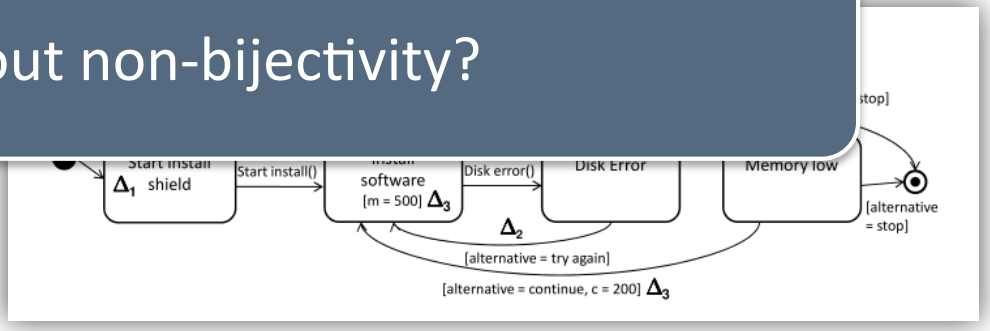
T

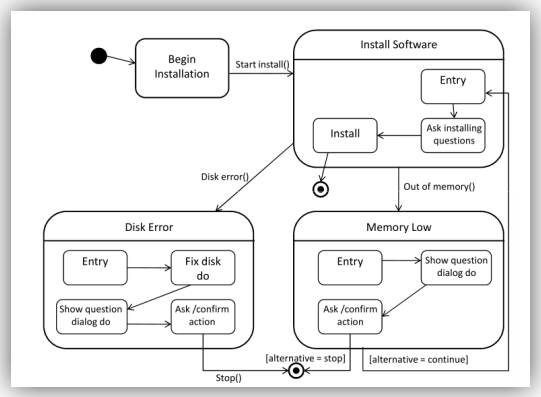


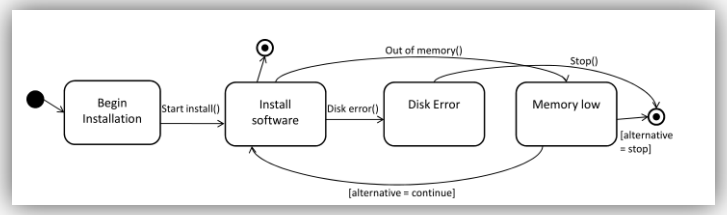
Problem

How to back propagate the manual changes on the target model towards the source models according to the knowledge encoded in T?

What about non-bijectivity?







The HSM and NHSM are non isomorphic metamodels, thus when back propagating changes on the target there are two alternatives

- **human-out-of-the-loop** metamodel are artificially made isomorphic by programmatically selecting one and only one model
- **human-in-the-loop** all the solutions are computed and the designer select one (or more) of them by inspection

Requirements for bidirectional transformations

A non-bijective bidirectional transformation R between M and N, and M more expressive than N, is characterized by

$\overrightarrow{R} : M \times N \rightarrow N$

$\overleftarrow{R} : M \times N \rightarrow M^*$

where $\overrightarrow{R}$ takes a pair of models (m, n) and works out how to modify n so as to enforce the relation R. $\overleftarrow{R}$ propagates changes in the opposite direction: $\overleftarrow{R}$ is a non-bijective function able to map the target model in a set of corresponding source models conforming to M

Reachability

$\overleftarrow{R}(m, n') = m^* \in M^*$

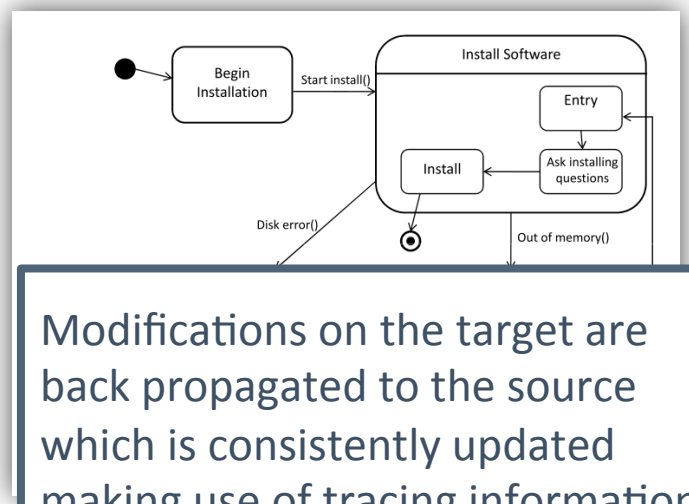
$\overrightarrow{R}(m', n') = n' \in N$ for each $m' \in m^*$

Choice preservation

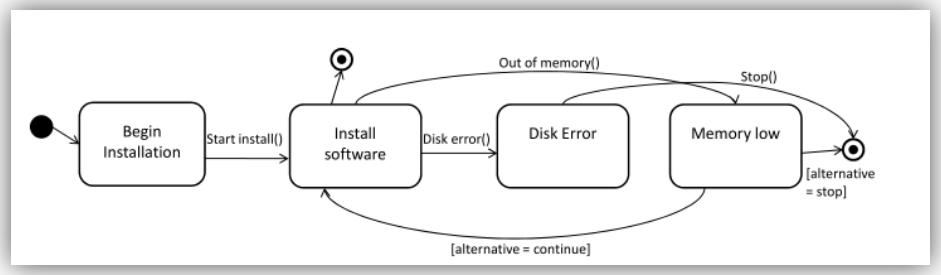
$\overleftarrow{R}(m', \overrightarrow{R}(m', n')) = m'$ for each $m' \in m^*$

source

target



T



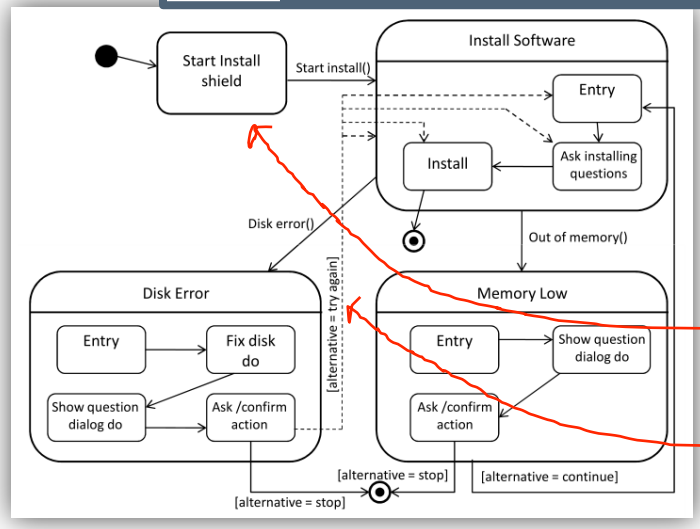
Modifications on the target are back propagated to the source which is consistently updated making use of tracing information

some

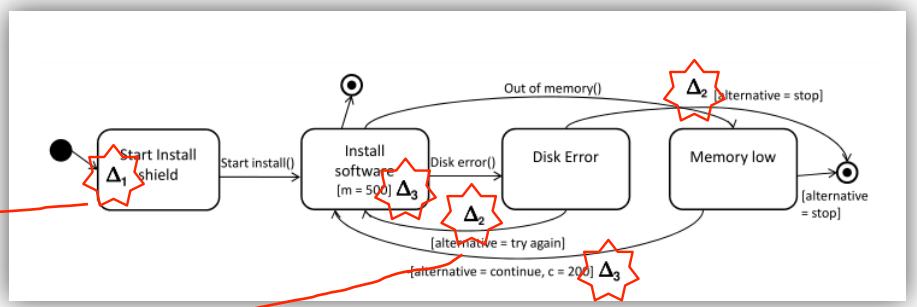
manual changes on the generated model

Hiera

The JTL transformation generates a non-hierarchical state machine by flattening the source model



T



dipartimentoinformatica
Università degli Studi dell'Aquila

Janus Transformation Language (JTL)

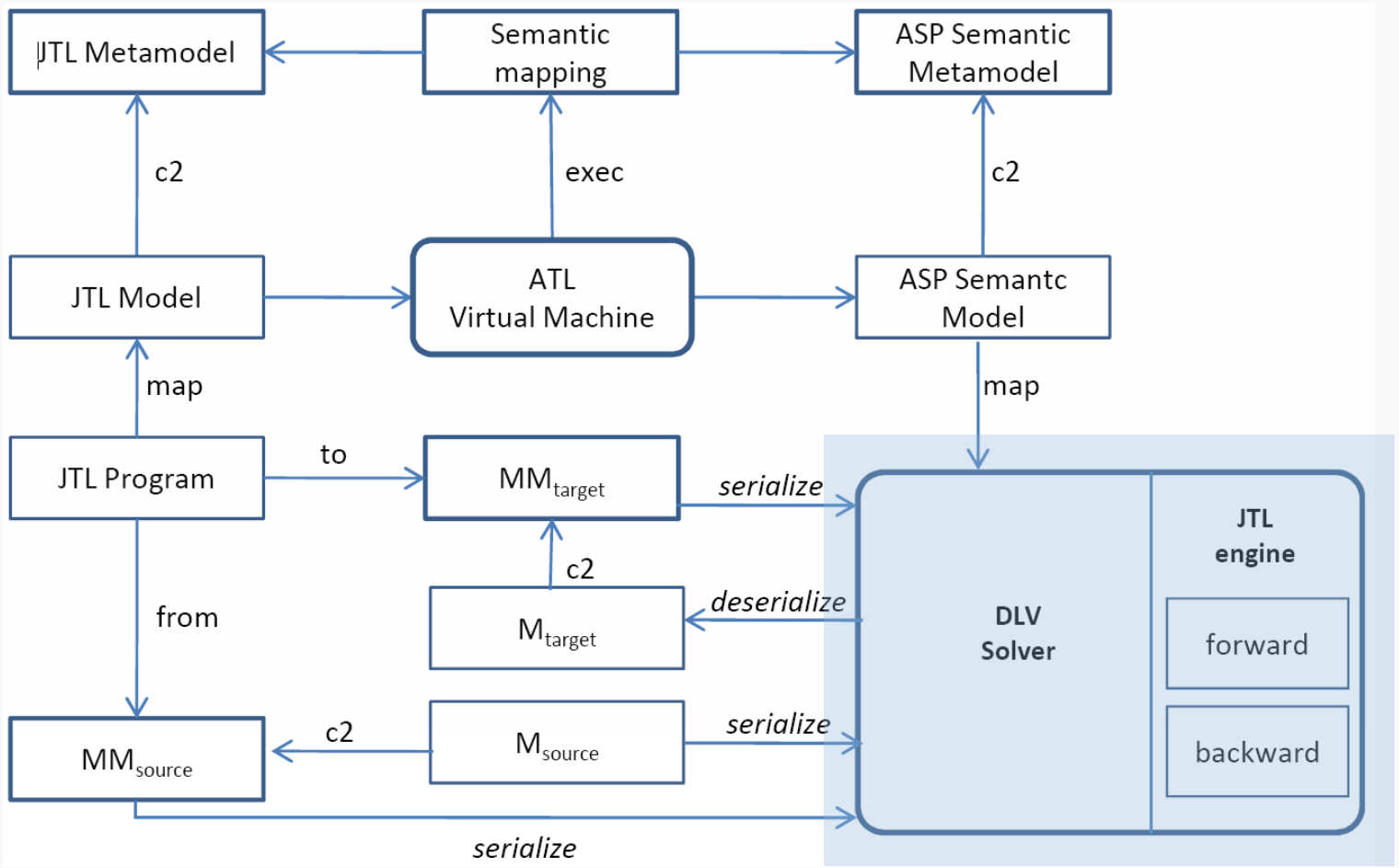
JTL is a bidirectional model transformation language capable of computing all the possible solutions at once

It has formal semantics based on Answer Set Programming and therefore can be considered a constraint-based approach

It back propagates changes occurring on a target model to the corresponding source ones by giving tracing information a first-class status

JTL is embedded in a framework available on the Eclipse platform and can be applied to Ecore meta/models

JTL Environment



JTL Engine

Declarative and relational engine for bidirectional model transformation based on Answer Set Programming (ASP)

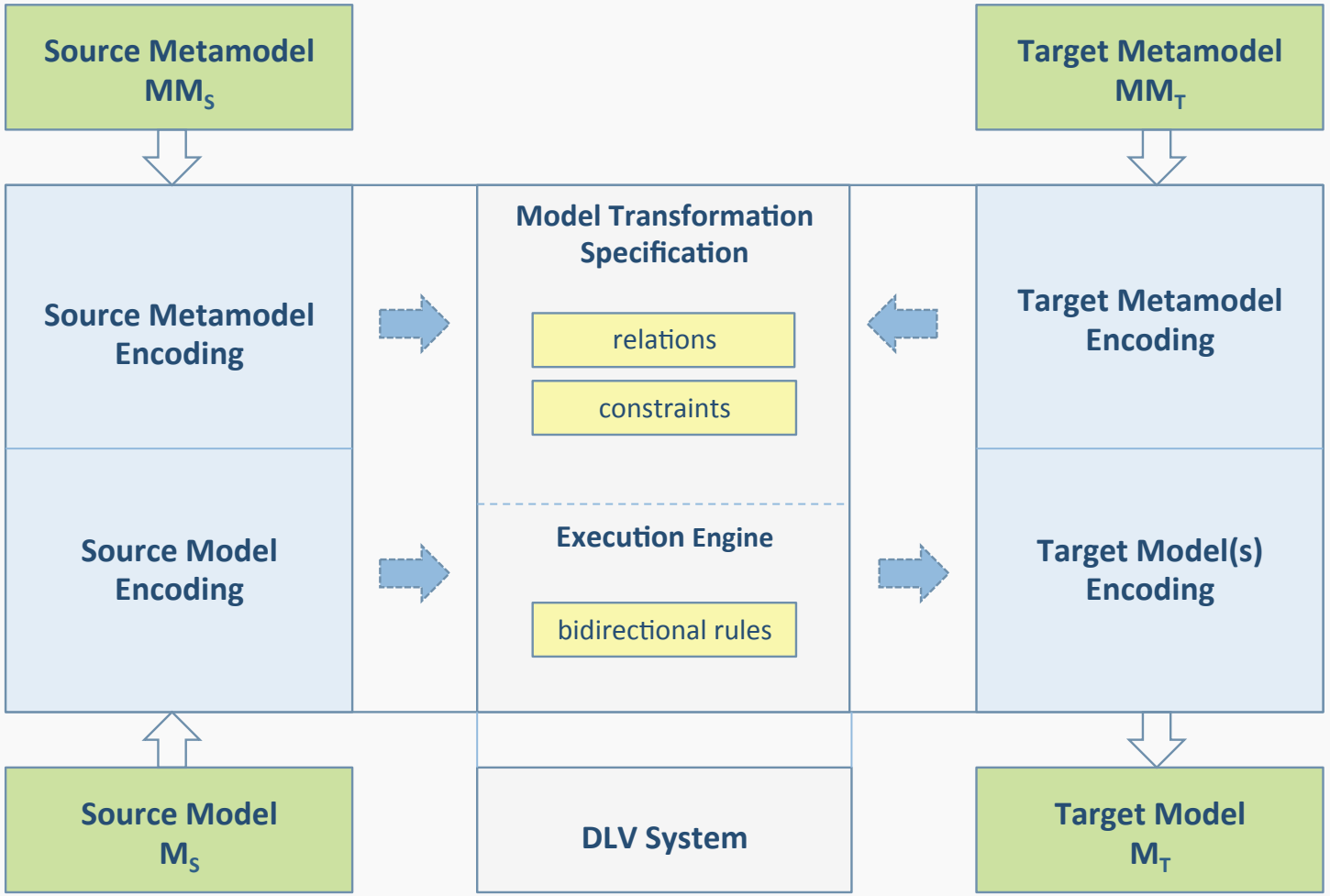
Answer Set Programming:

- declarative paradigm of logic programming that uses logic and proof procedures to define and resolve problems
- based on the stable model (answer set) semantics of logic programming and on the first-order logic
- able to dealing with non-deterministic derivations which represent alternative solutions to a given problem

Each mapping can be read as a transformation in either direction

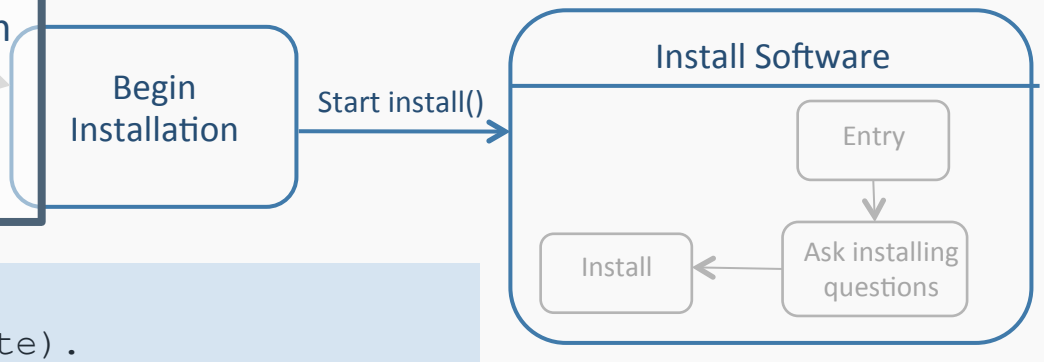
Approximation of the ideal model, from which the modified one can be generated, with respect to the available transformations and metamodels

JTL Engine


Source Metamodel
MM_S
Target Metamodel
MM_T
Source Metamodel
Encoding
Source Model
Encoding
Model Transformation
Specification
relations
constraints
Execution Engine
bidirectional rules
Target Metamodel
Encoding
Target Model(s)
Encoding
Source Model
M_S
DLV System
Target Model
M_T

Models and metamodels encodings

The metamodel encoding is based on a set of terms each characterized by the predicate symbols metanode, metaedge, and metaprop




```
metanode(HSM, state).
metanode(HSM, compositeState).
metanode(HSM, transition).
metaprop(HSM, name, state).
metaprop(HSM, name, compositeState).
metaedge(HSM, association, source, transition, state).
metaedge(HSM, association, target, transition, state).
```

```
node(HSM, "s1", state).
node(HSM, "s2", state).
node(HSM, "t1", transition).
prop(HSM, "s1.1", "s1", name, "begin installation").
prop(HSM, "s2.1", "s2", name, "install software").
prop(HSM, "t1.1", "t1", trigger, "install software").
prop(HSM, "t1.2", "t1", effect, "start install").
edge(HSM, "tr1", association, source, "s1", "t1").
edge(HSM, "tr1", association, target, "s2", "t1").
```

Models are sets of entities (predicate symbol node), each characterized by properties (prop) and related together by relations (edge).

Model Transformation Execution

After the encoding phase, the deduction of the target model is performed according to the model transformation rules defined in the ASP program

ASP model transformations are specified by means of:

- **Relations** which describe correspondences among element types of the source and target metamodels
- **Constraints** which specify restrictions on the given relations that must be satisfied in order to execute the corresponding mappings

The transformation process logically consists of the following steps:

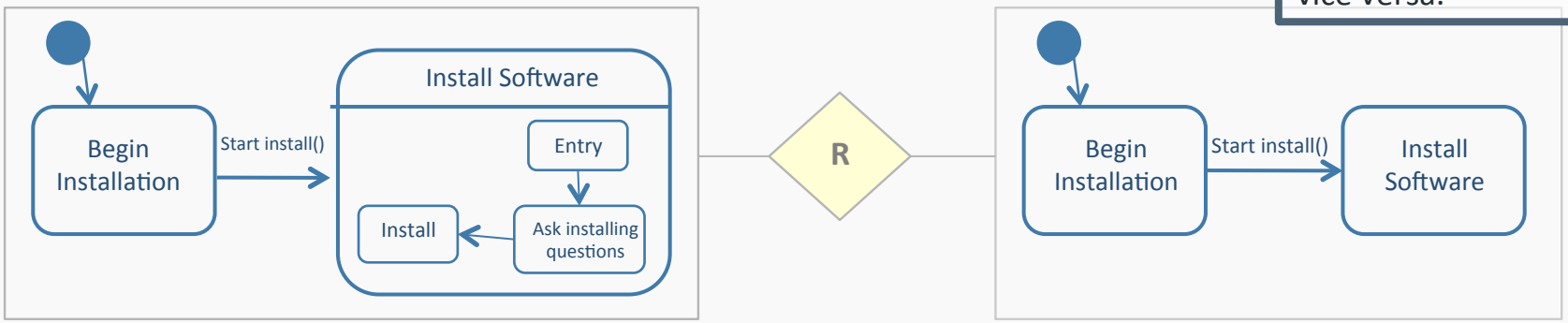
- given the input (meta)models, the execution engine induces all the possible solution candidates according to the specified relations
- the set of candidates is refined by means of constraints

Model Transformation Execution

```
relation("r1",HSM,state).
relation("r1",NHSM,state).
:- node(HSM,ID,state), not edge(HSM,IDe,owningCompositeState,ID,IDc),
   not node'(NHSM,ID,state).
:- node(HSM,ID,state), edge(HSM,IDe,owningCompositeState,ID,IDc),
   node(HSM,IDc,compositeState), node'(NHSM,ID,state).
:- node(NHSM,ID,state), not trace_node(HSM,ID,compositeState),
   not node'(HSM,ID,state).
:- node(NHSM,ID,state), trace_node(HSM,IDc,compositeState),
   node'(HSM,ID,state).

relation("r2",HSM,compositeState).
relation("r2",NHSM,state).
:- node(HSM,ID,compositeState), not node'(NHSM,ID,state).
:- node(NHSM,ID,state), trace_node(HSM,ID,compositeState),
   not node'(HSM, ID,compositeState).
```

"r1" relates the metaclasses *State* and *State*. Each time a state occurs in the HSM model, the correspondent one in the NHSM model is generated only if the source element is not a sub-state, vice versa, each state in the NHSM model is mapped in the HSM model.

"r2" relates the metaclasses Composite state and State. Each time a composite state occurs in the HSM model a correspondent state in the NHSM model is generated, and vice versa.



Execution Engine

The specified transformations are executed by a generic bidirectional engine consisting of ASP transformation rules

Transformation rules may produce more than one target models, which are all the possible combinations of elements that the program is able to create

The invertibility of transformations is obtained by means of trace information that connects source and target elements

- during the transformation process, the relationships between models that are created by the transformation executions can be stored to preserve mapping information in a persistent way
- all the source elements lost during the forward transformation execution (for example, due to the different expressive power of the metamodels) are stored in order to be generated again in the backward transformation execution.

Execution Engine

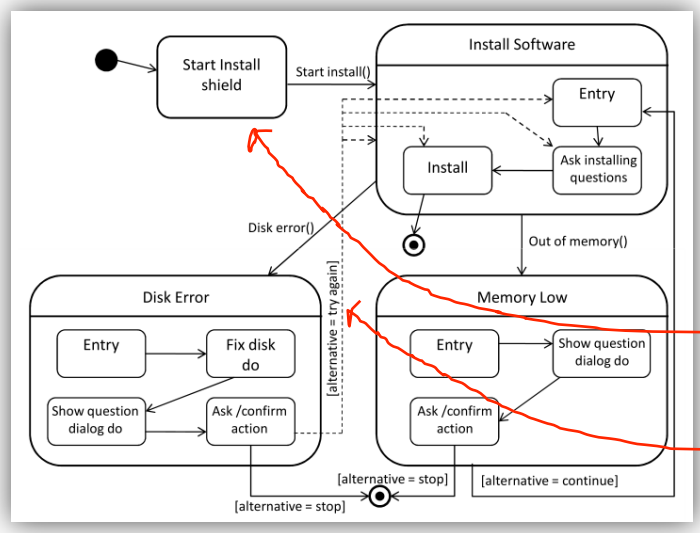
```
is_source_metamodel_conform(MM,ID,MC) :- node(MM,ID,MC), node(MM,MC).
bad_source :- node(MM,ID,MC), not is_source_metamodel_conform(MM,ID,MC).
mapping(MM,ID,MC) :- relation(R,MM,MC), relation(R,MM2,MC2),
                     node(MM2,ID,MC2), MM!=MM2.
is_target_metamodel_conform(MM,MC) :- metanode(MM,MC).
{is_generable(MM,ID,MC)} :- not bad_source, mapping(MM,ID,MC),
                            is_target_metamodel_conform(MM,MC), MM=mmt.
node'(MM,ID,MC) :- is_generable(MM,ID,MC), mapping(MM,ID,MC), MM=mmt.
```

Target elements (node') are created if the following conditions are satisfied:

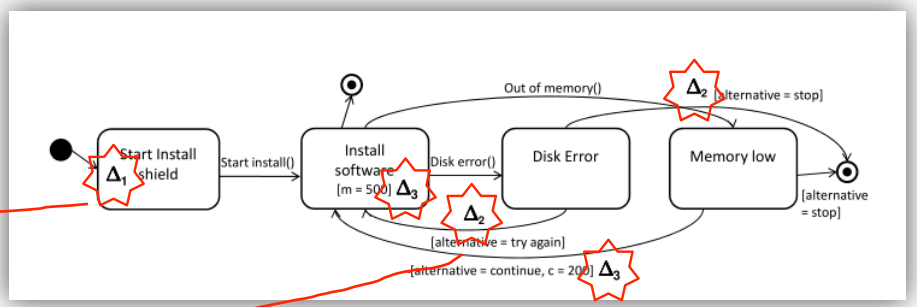
– The considered element is declared in the input source model
– at least a relation exists between a source element and the candidate target element;
– the candidate target element conforms to the target metamodel;
– finally, any constraint defined in the relations is violated.

source

target

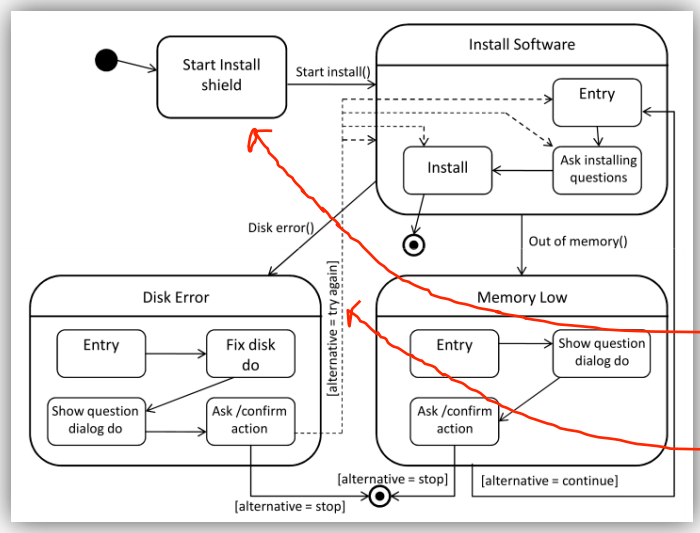


T

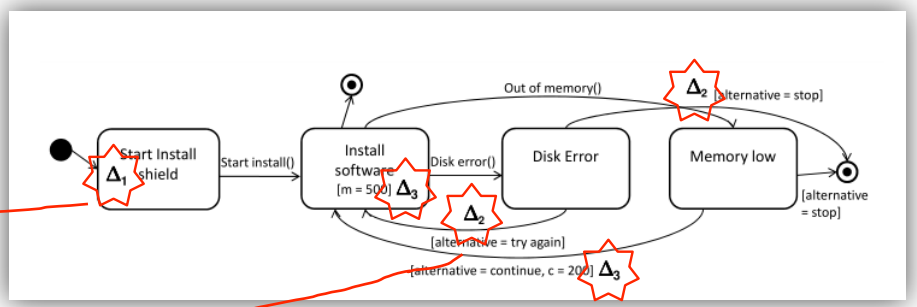


source

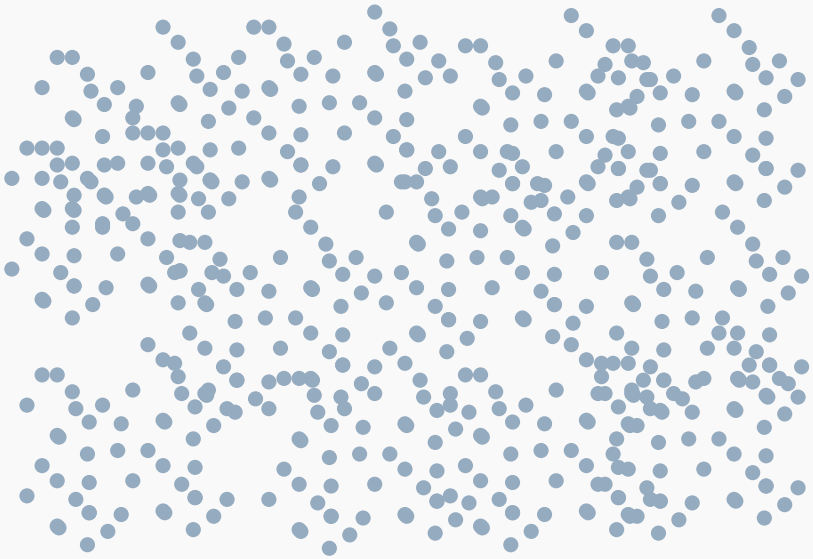
target



T



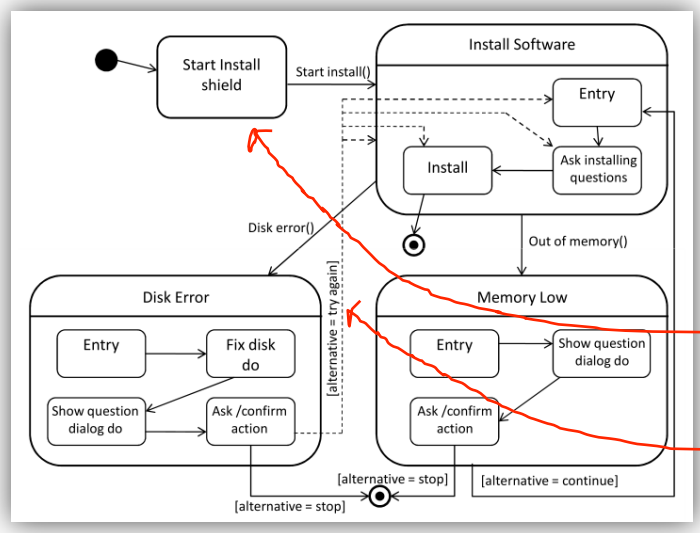
T



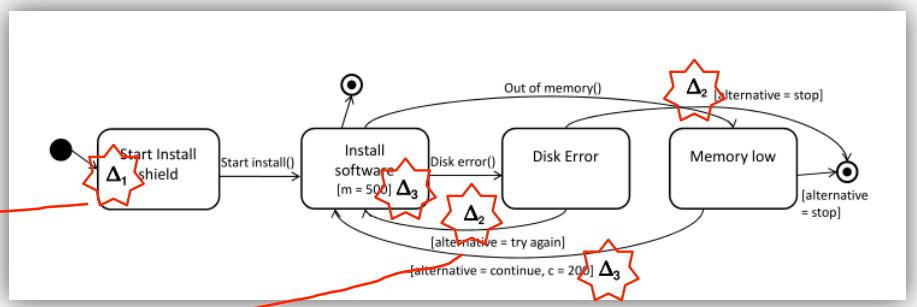
Only the information encoded in the transformation is used

source

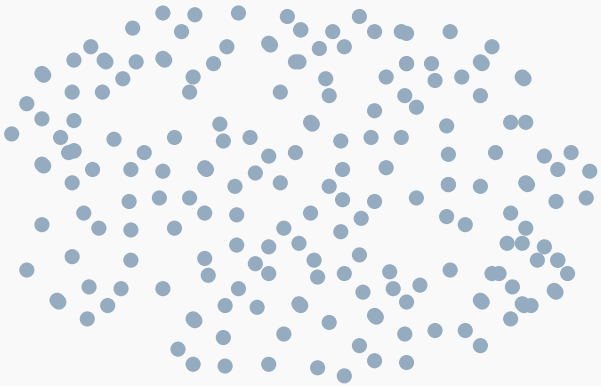
target



T



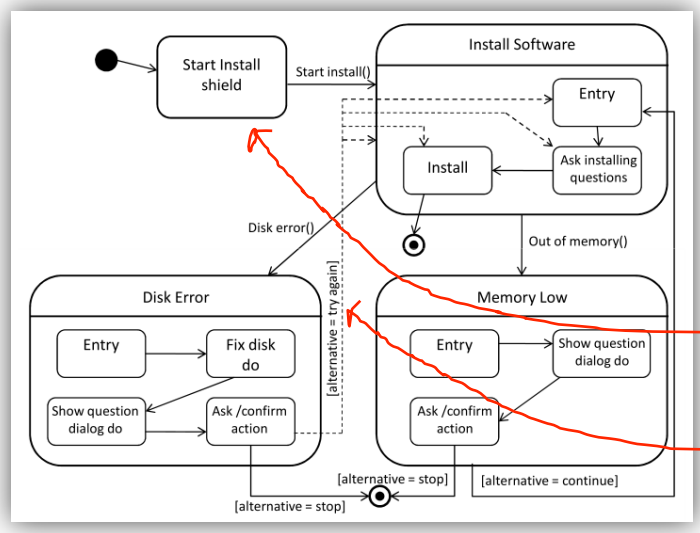
T



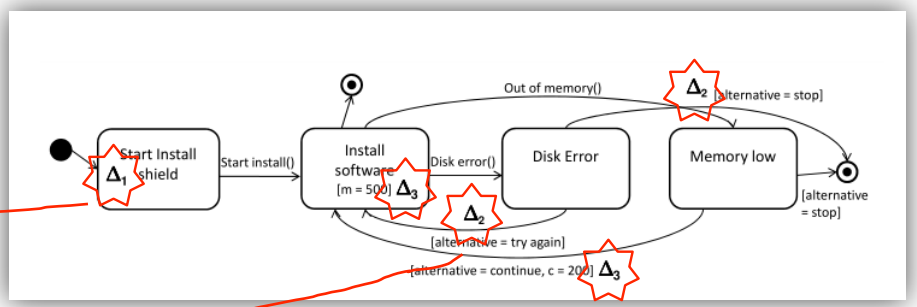
The answer set is filtered according to the constraints induced by the source metamodel

source

target



T

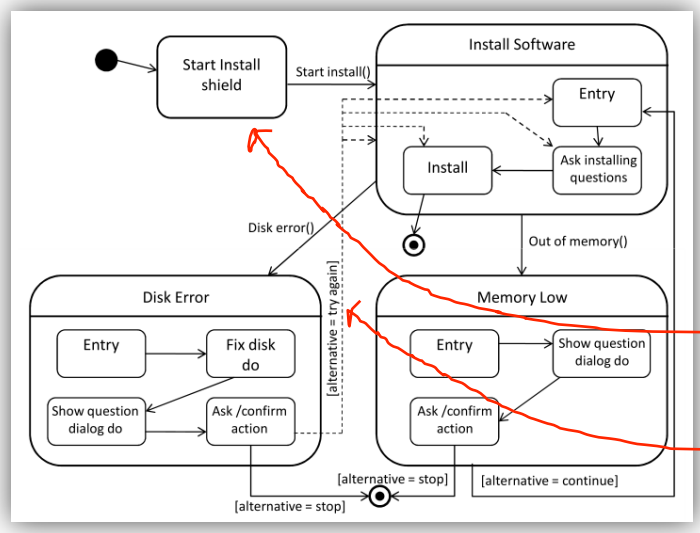


T

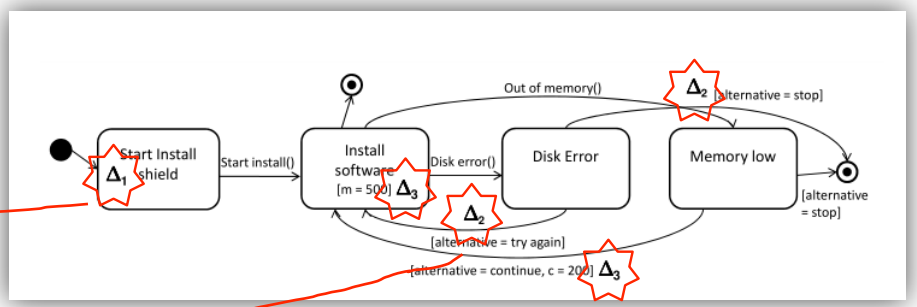
The answer set is further reduced by considering the constraints induced by the tracing information

source

target



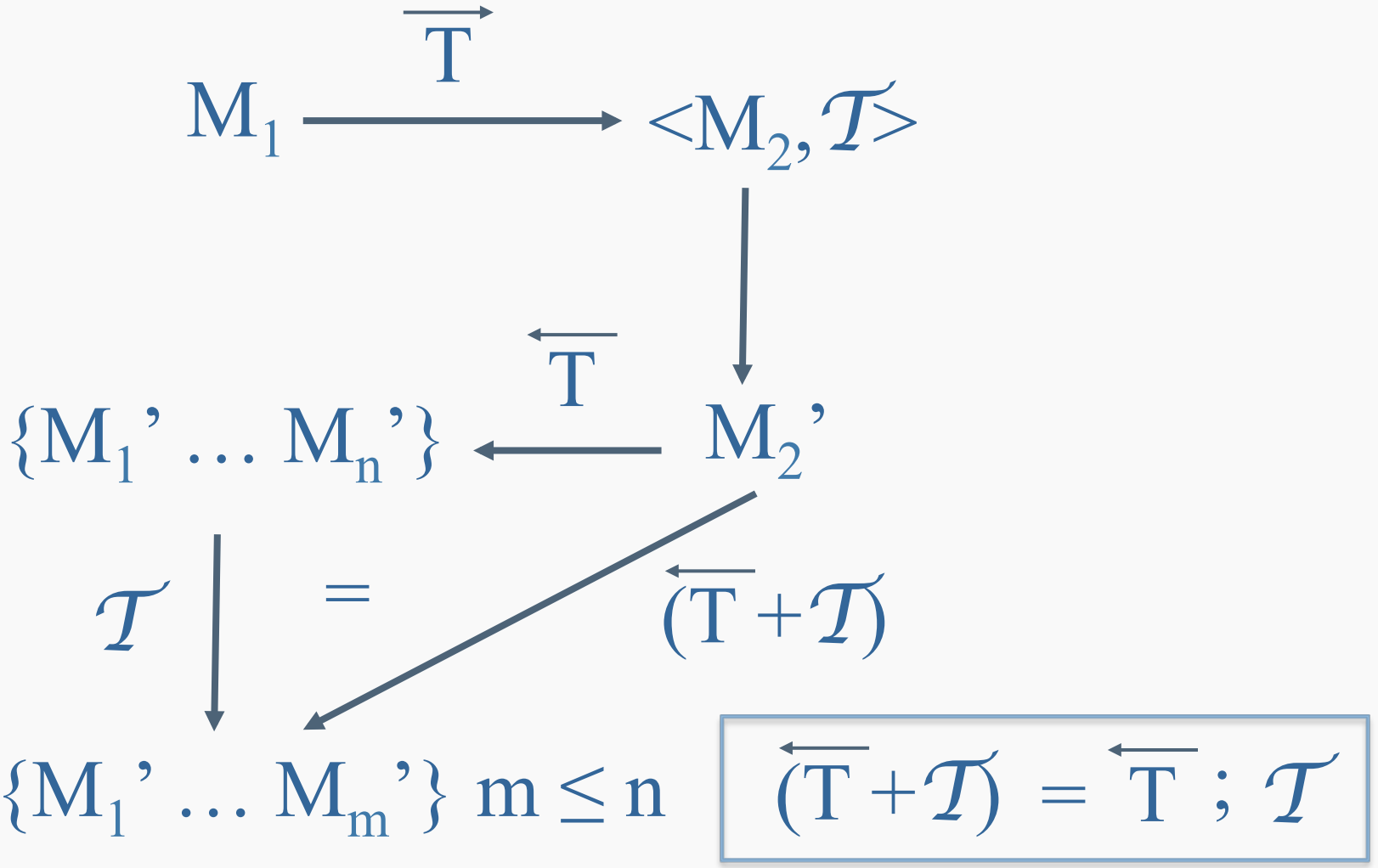
T



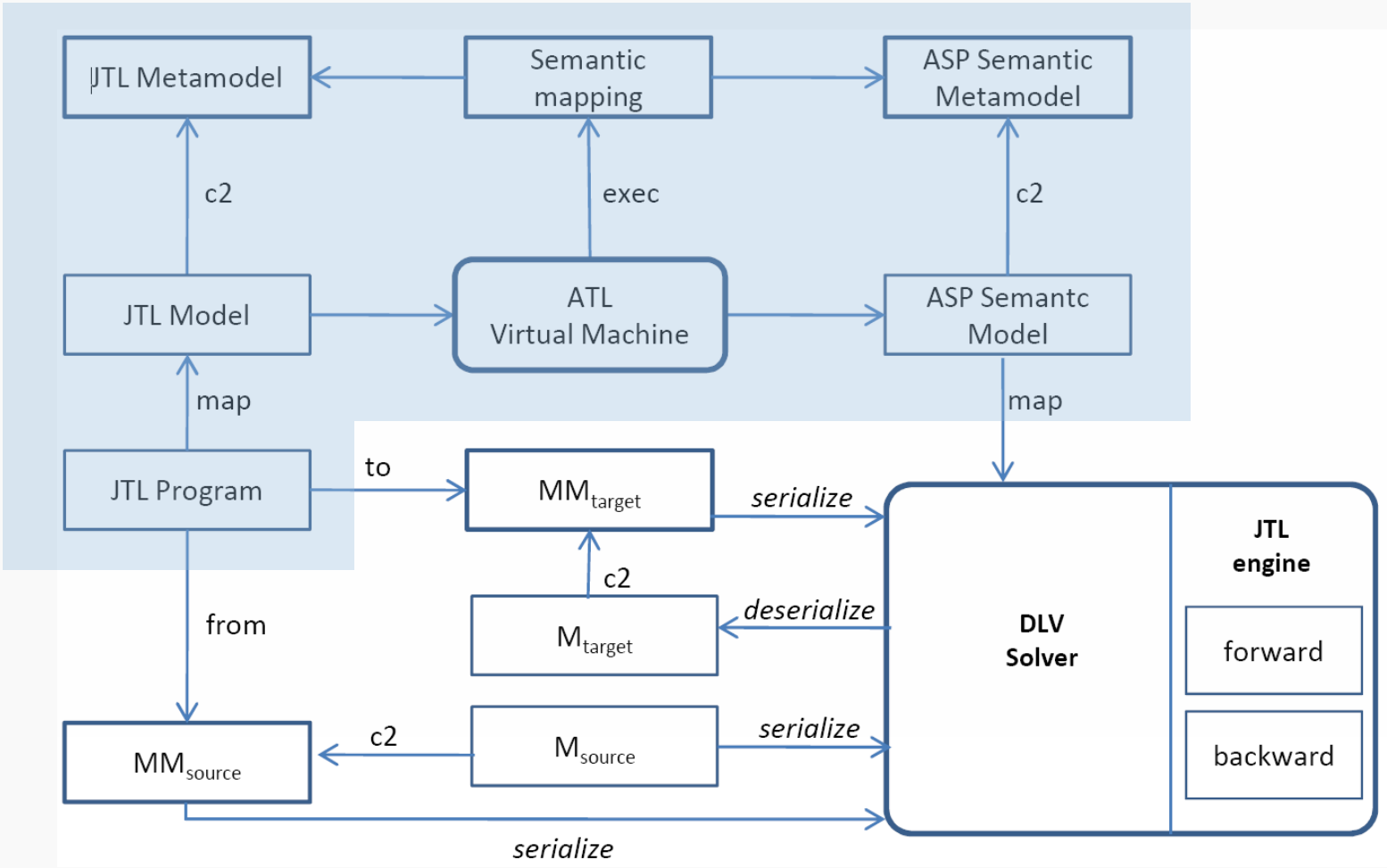
T

Additional user-defined constraints can be added to browse the space of solutions

Theorem (compositionality)



JTL Environment



Specifying transformation with Janus

JTL provides support for specifying transformations by means of a QVT-R like syntax

Relational model transformations in JTL can be applied on Ecore models and metamodels.

JTL has been given formal semantics through a semantic anchoring towards ASP

The constraint based nature of the JTL semantics permits further refinements of the solution space.

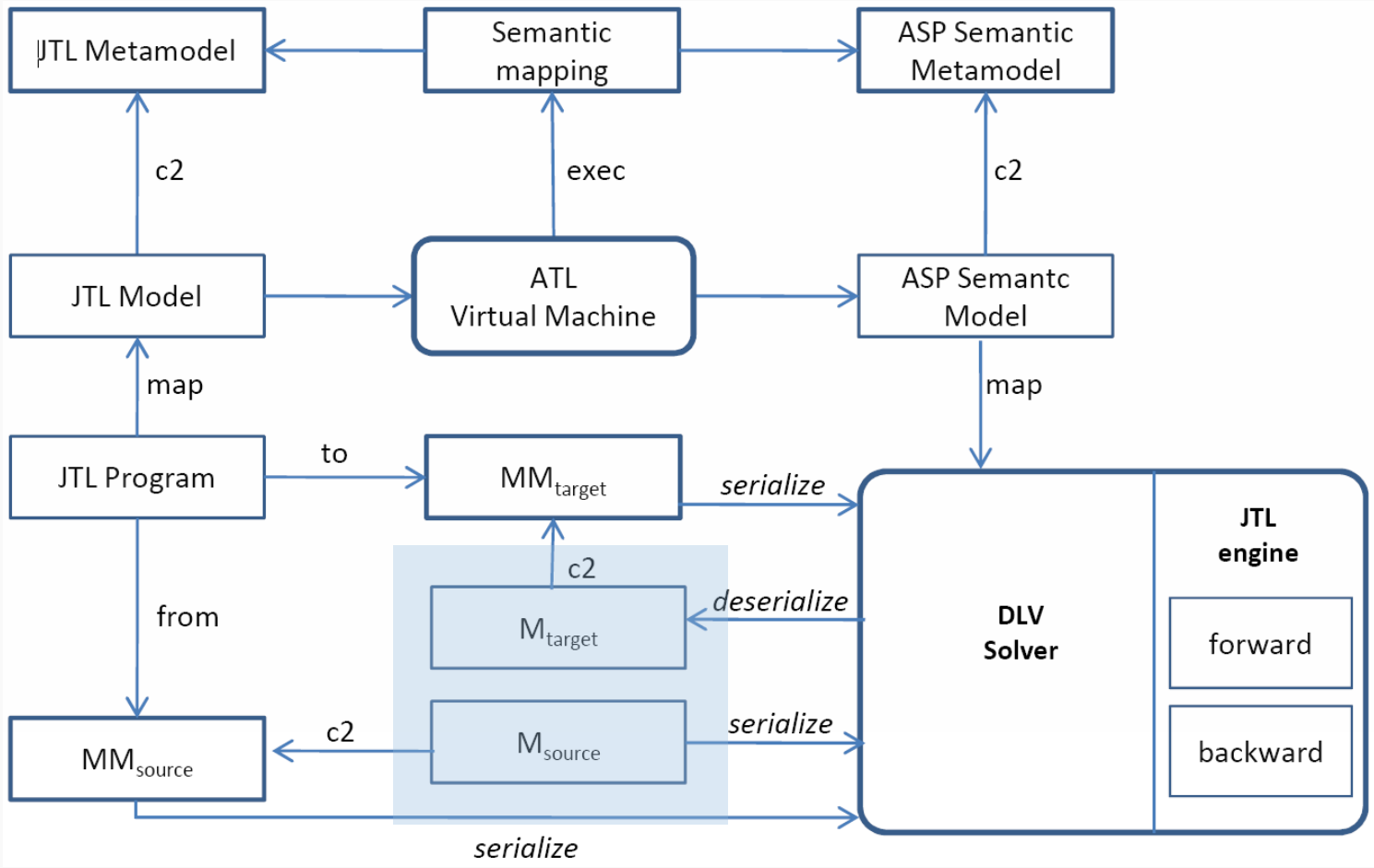
Specifying transformation with Janus

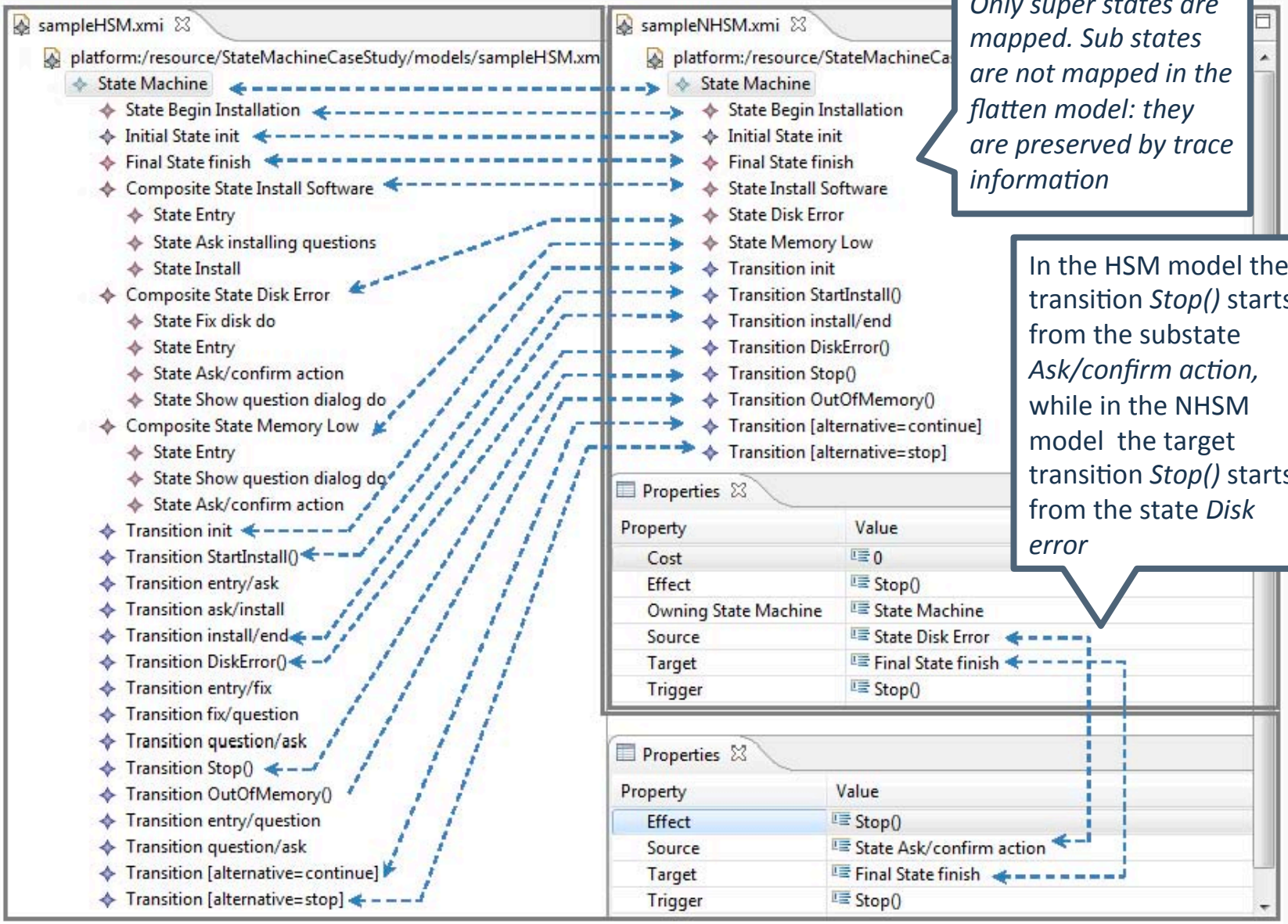
Fragment of the HSM2NHSM transformation specified in JTL .
It transforms hierarchical state machines into flat state machines and the other way round.

```
transformation hsm2nhsm(source : HSM, target : NHSM) {
  top relation StateMachine2StateMachine {
    enforce domain source sSM : HSM::StateMachine;
    enforce domain target tSM : NHSM::StateMachine;
  }

  top relation State2State {
    enforce domain source sourceState : HSM::State;
    enforce domain target targetState : NHSM::State;
    when {
      sourceState.owningCompositeState.oclIsUndefined();
    }
  }

  top relation CompositeState2State {
   enforce domain source sourceState : HSM::CompositeState;
   enforce domain target targetState : NHSM::State;
  }
}
```

The forward transformation is clearly non-injective as many different hierarchical machines can be flattened to the same model and consequently transforming back a modified flat machine can give place to more than one hierarchical machine.

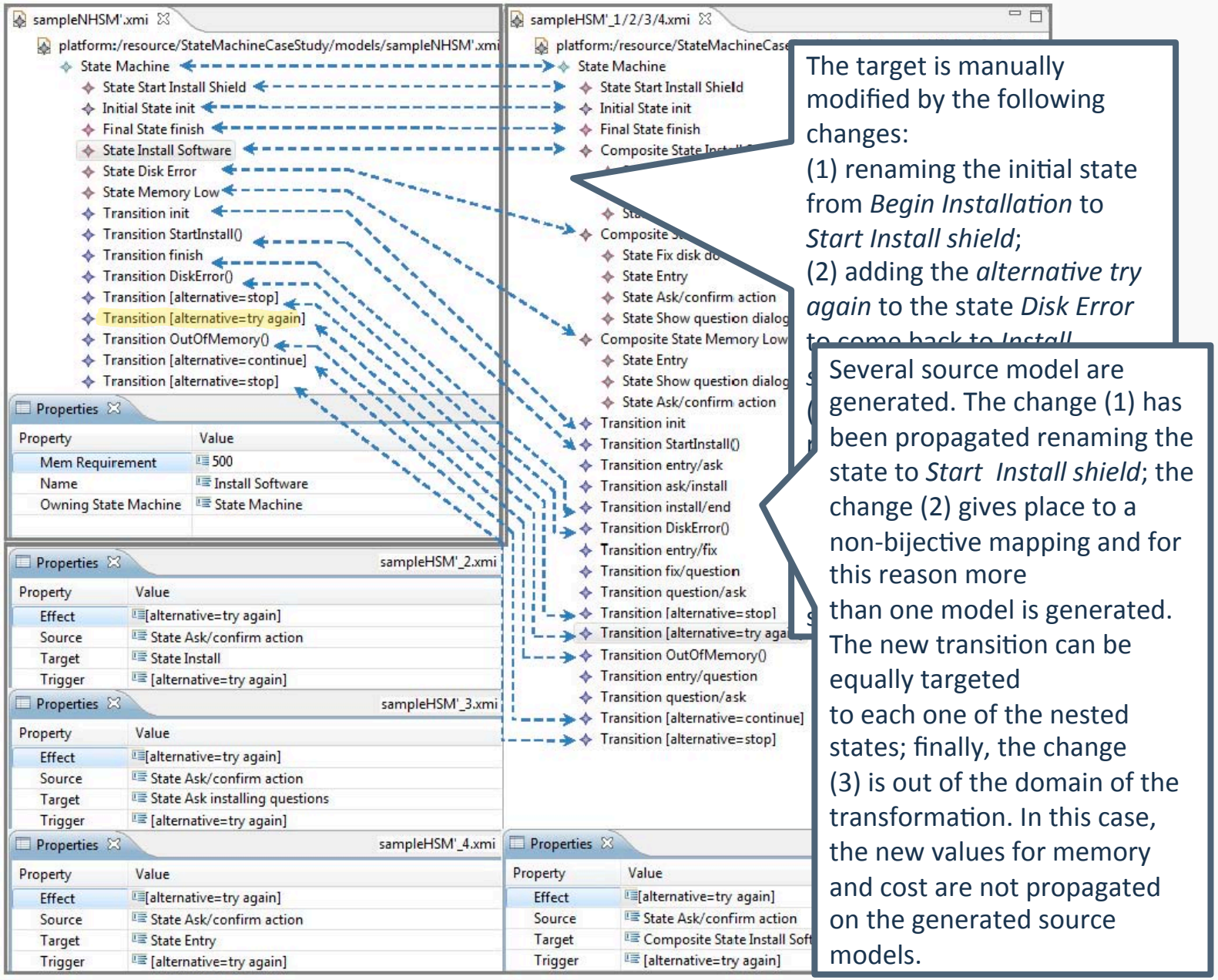
JTL Environment





Only super states are mapped. Sub states are not mapped in the flatten model: they are preserved by trace information

In the HSM model the transition *Stop()* starts from the substate *Ask/confirm action,* while in the NHSM model the target transition *Stop()* starts from the state *Disk error*



The target is manually modified by the following changes:
(1) renaming the initial state from *Begin Installation* to *Start Install shield*;
(2) adding the *alternative try again* to the state *Disk Error*

Several source model are generated. The change (1) has been propagated renaming the state to *Start Install shield*; the change (2) gives place to a non-bijective mapping and for this reason more than one model is generated. The new transition can be equally targeted to each one of the nested states; finally, the change (3) is out of the domain of the transformation. In this case, the new values for memory and cost are not propagated on the generated source models.

Non bijectivity

Non bijectivity is difficult to be handled, the two approaches

- → human-out-of-the-loop
- → human-in-the-loop

are both needed in

- → synchronization
- → consistency management and change propagation

respectively

Difficulties with JTL

JTL is very versatile in its application spectrum, it can be used in both cases, it all depends on the amount of constraints the implementor provides

Visualization of multiple models is critical, if the designer needs to inspect them

- → an intentional definition of the solutions is necessary, eg. variability modeling
- → the overlapping parts can be easily factorized by providing the right constraints without or with little computational overhead

Transformations

- Introduction
- Why Model Transformation languages ?
- Dimensions and Classification

A demonstration of ATL

- Objectives
- Metamodels
- Live Demonstration

Bidirectional Transformation for Change Propagation

- Problem
- Requirements
- Janus Transformation Language
- Change Propagation and non-determinism

Higher-Order Transformations for Automating Co-evolution

- Evolution in MDE
- Metamodel Changes Classification
- Metamodel Differences
- Automated Adaptation

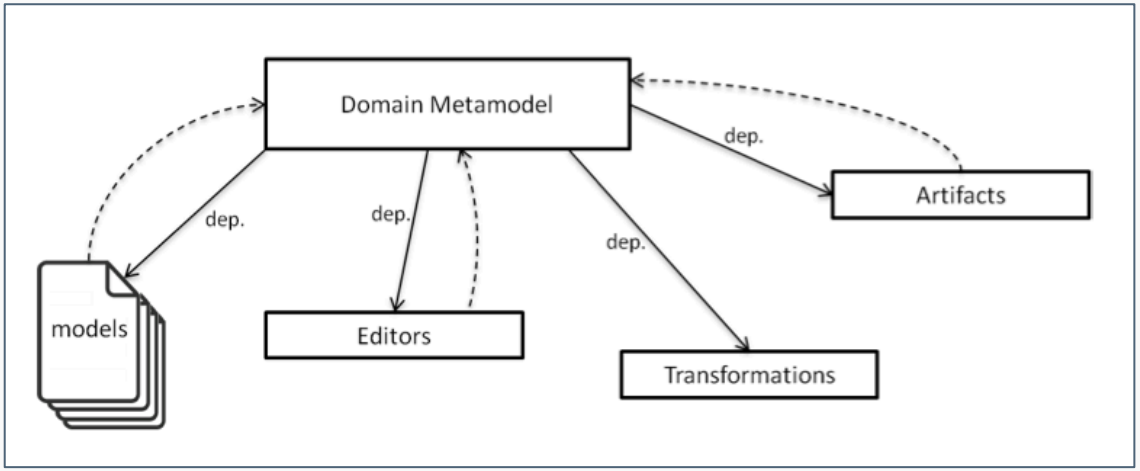
Summary

HIGHER ORDER TRANSFORMATION FOR AUTOMATING CO-EVOLUTION

dipartimentoinformatica
Università degli Studi dell'Aquila

Modeling Ecosystem

Metamodels are a pivotal component of MDE

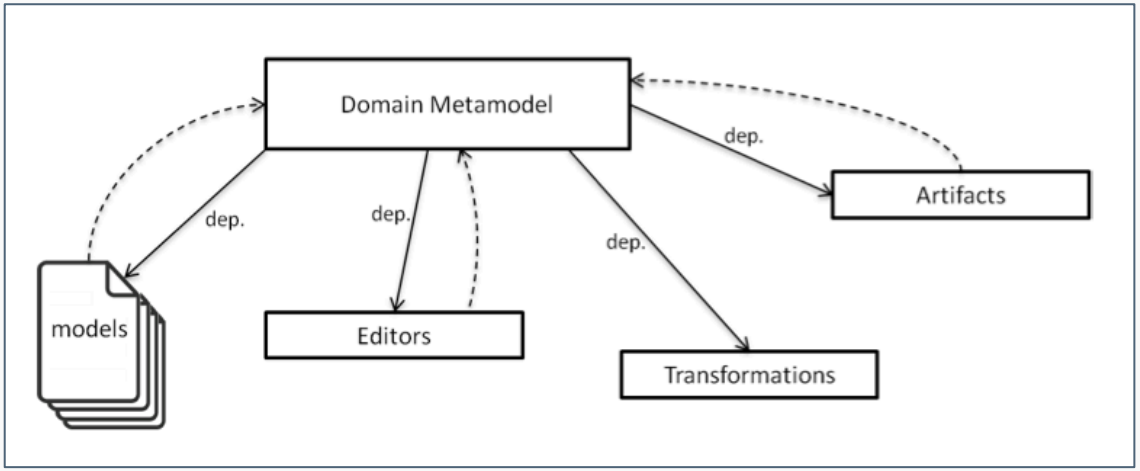


Each time a metamodel undergoes modifications, a number of components might be not valid any longer

Understanding the rationale of the changes to automatically derive transformations for adapting the artifacts

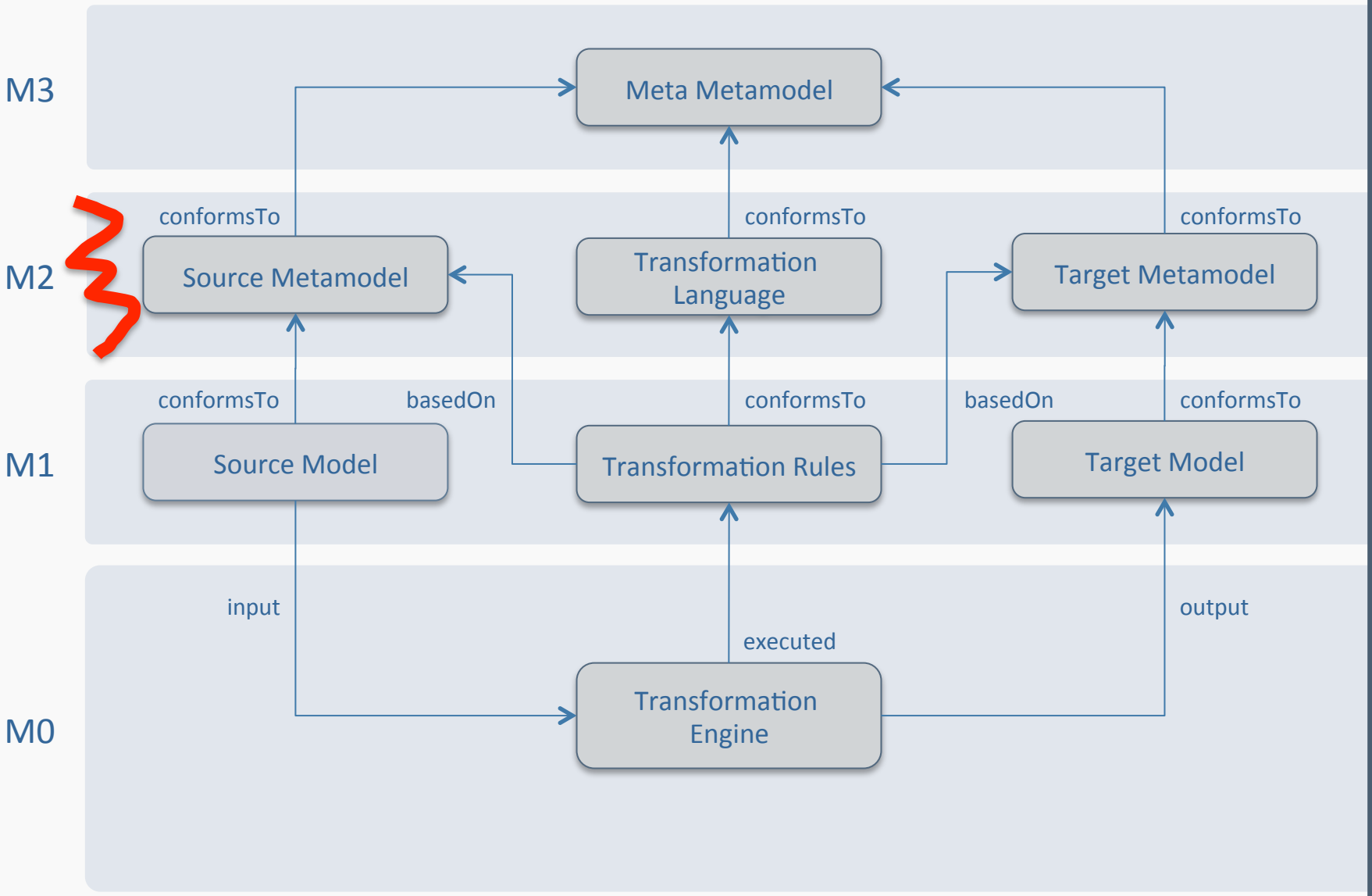
Modeling Ecosystem

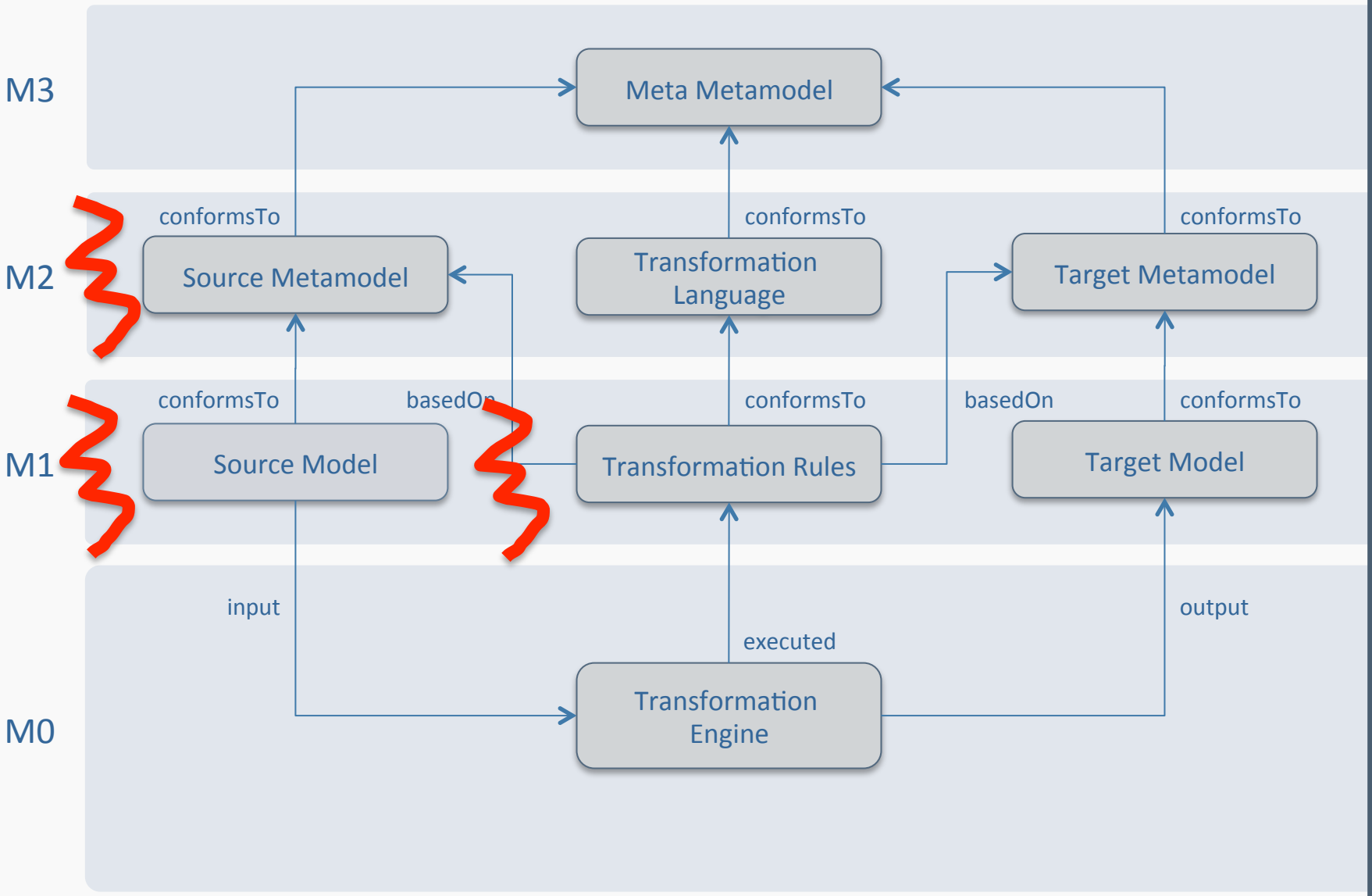
Metamodels are a pivotal component of MDE



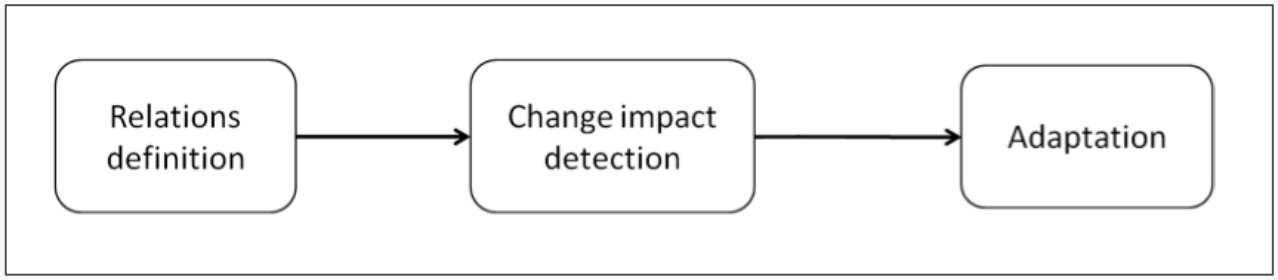
Each time a metamodel undergoes modifications, a number of components might be not valid any longer

Understanding the rationale of the changes to automatically derive transformations for adapting the artifacts


M3
M2
M1
M0
Meta Metamodel
conformsTo
conformsTo
conformsTo
Source Metamodel
Transformation Language
Target Metamodel
conformsTo
basedOn
conformsTo
basedOn
conformsTo
Source Model
Transformation Rules
Target Model
input
executed
output
Transformation Engine


M3
M2
M1
M0
Meta Metamodel
conformsTo
conformsTo
conformsTo
Source Metamodel
Transformation Language
Target Metamodel
conformsTo
basedOn
conformsTo
basedOn
conformsTo
Source Model
Transformation Rules
Target Model
input
executed
output
Transformation Engine

Co-evolution



Relations definition. The relation between the various artifacts and the metamodel must be identified

Change impact detection. In this step the relationships defined in step 1 can be considered in order to assess the impact on the related artifacts of the changes made in the domain metamodel.

Adaptation. In this step the developer apply some adaptation actions on the (corrupted) artifacts. Model Transformations are used here.

Metamodel/model co-evolution

A metamodel can undergo a number of different kinds of modifications which are classified in

- **Non-breaking**
- **Breaking**

The breaking modifications can be divided into

- **Breaking and resolvable**: existing instances need to be co-adapted to conform to the new metamodel version. The co-evolution can be automatically operated
- **Breaking and unresolvable**: the necessary co-adaptation of existing models can not be automatically computed due to the need of further information

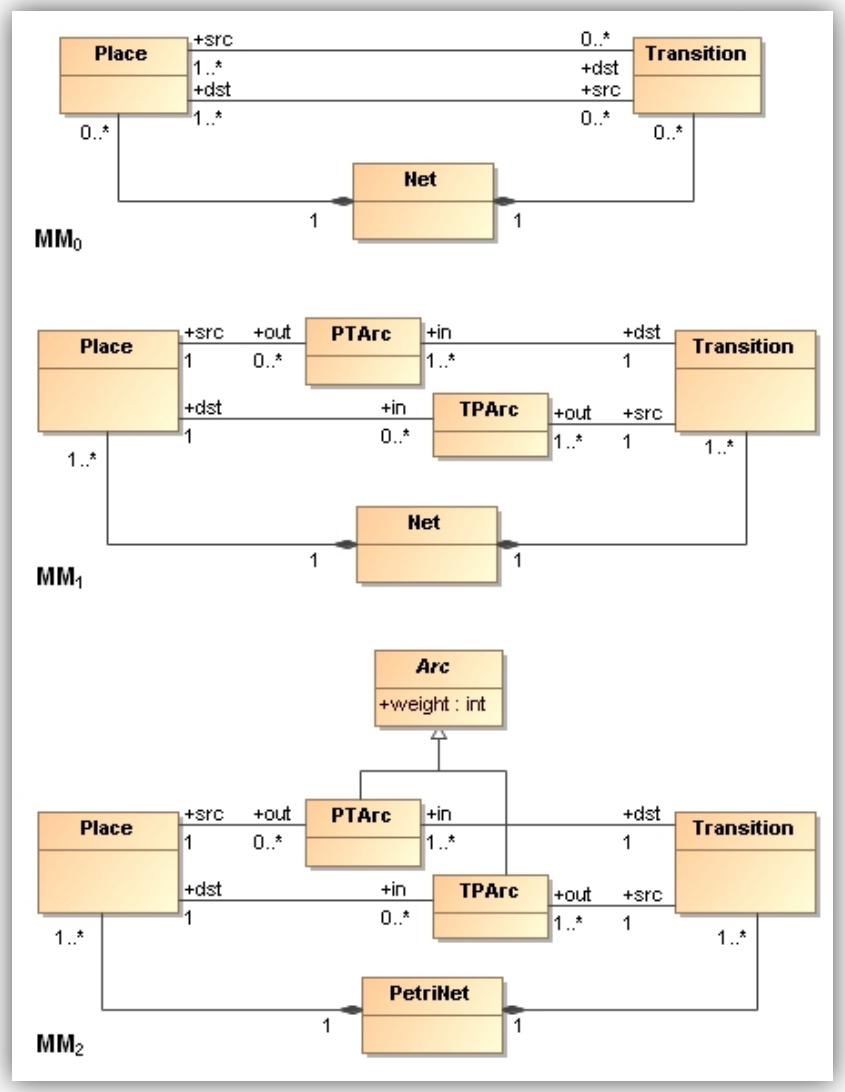
Metamodel changes

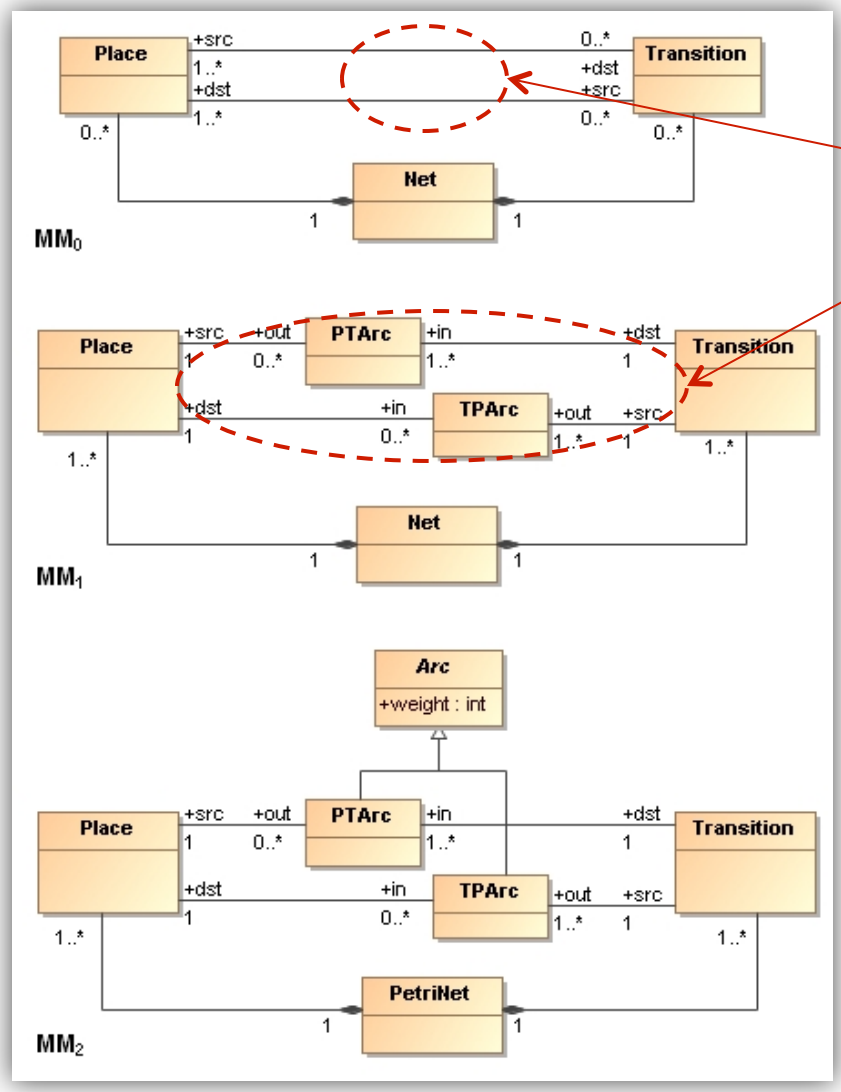
- **A metamodel can undergo a number of different kinds of modifications which are classified in**
  - **Non-breaking**
  - **Breaking**
- **The breaking modifications can be divided into**
  - **Breaking and resolvable**: existing instances need to be co-adapted to conform to the new metamodel version. The co-evolution can be automatically operated
  - **Breaking and unresolvable**: the necessary co-adaptation of existing models can not be automatically computed due to the need of further information

Metamodel changes classification

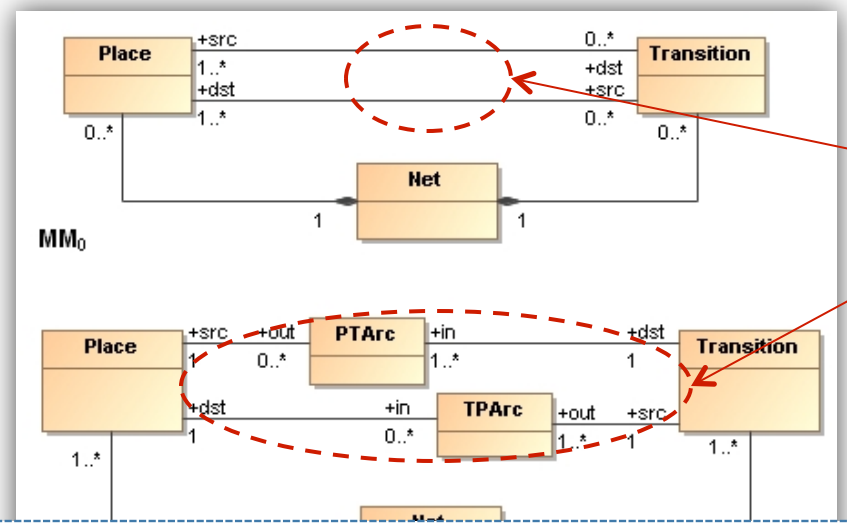
Change type	Change
Non-breaking	Generalize metaproperty Add (non-obligatory) metaclass Add (non-obligatory) metaproperty
Breaking and resolvable	Extract (abstract) superclass Eliminate metaclass Eliminate metaproperty Push metaproperty Flatten hierarchy Rename metaelement Move metaproperty Extract/inline metaclass
Breaking and unresolvable	Add obligatory metaclass Add obligatory metaproperty Pull metaproperty Restrict metaproperty Extract (non-abstract) superclass

Sample Petri Net metamodel changes

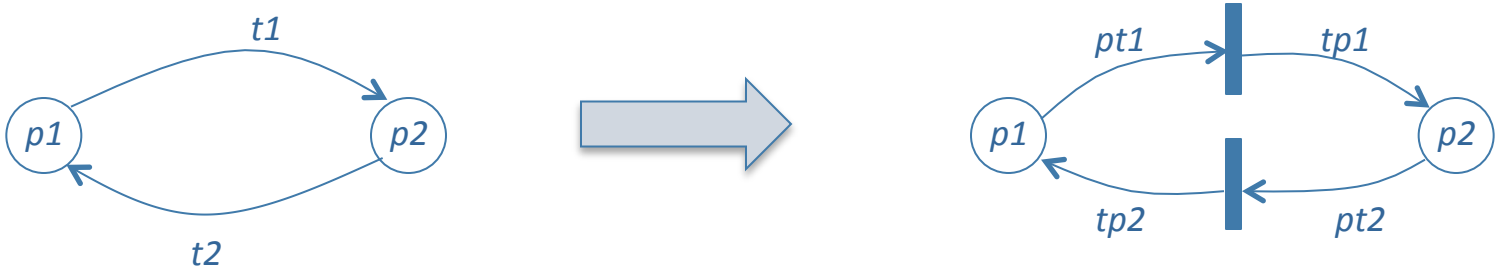




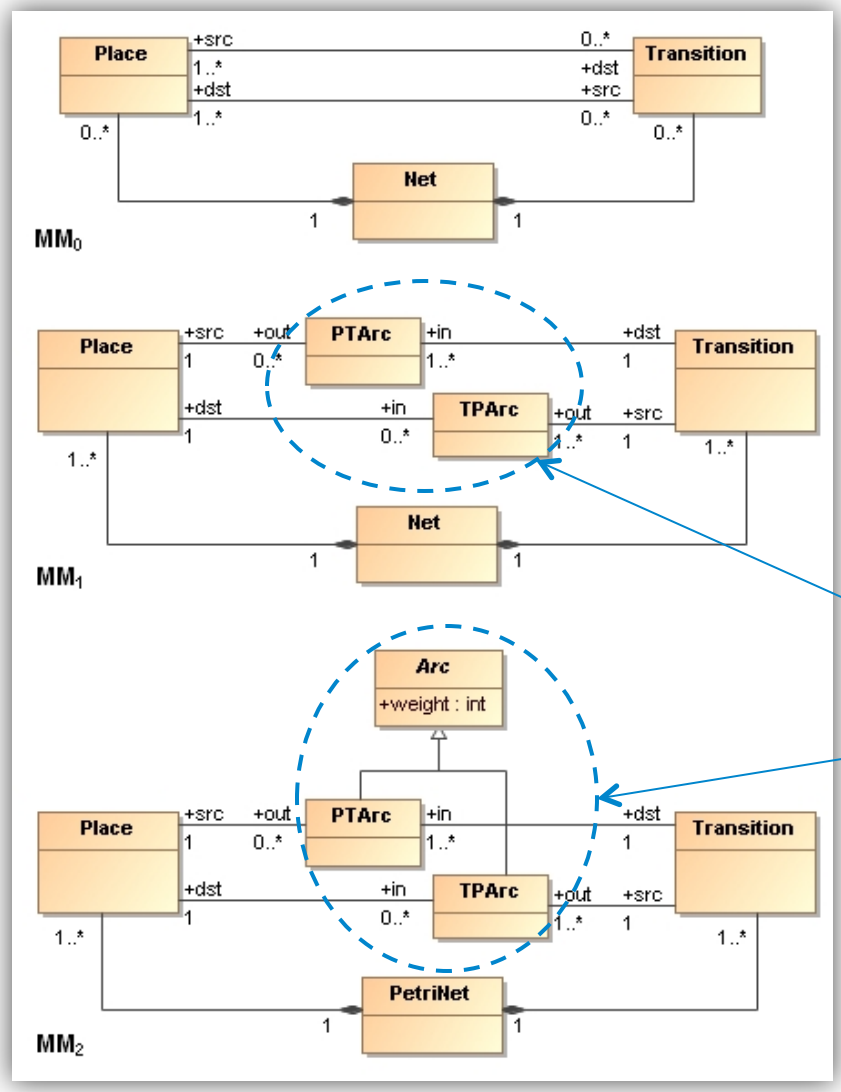
Breaking and resolvable changes
(*extract meta-class*)



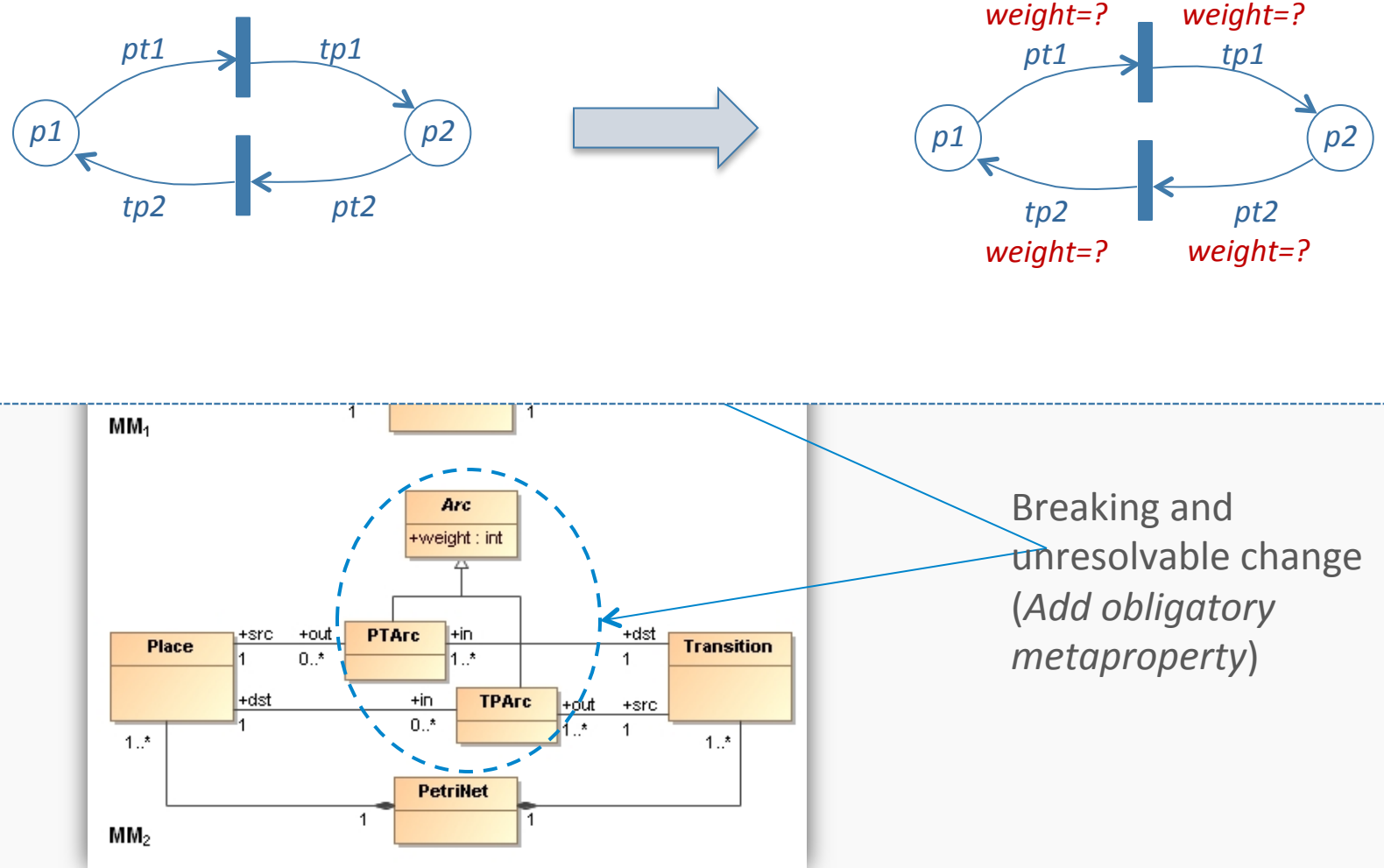
Breaking and resolvable changes
(*extract meta-class*)



Sample Petri Net metamodel changes

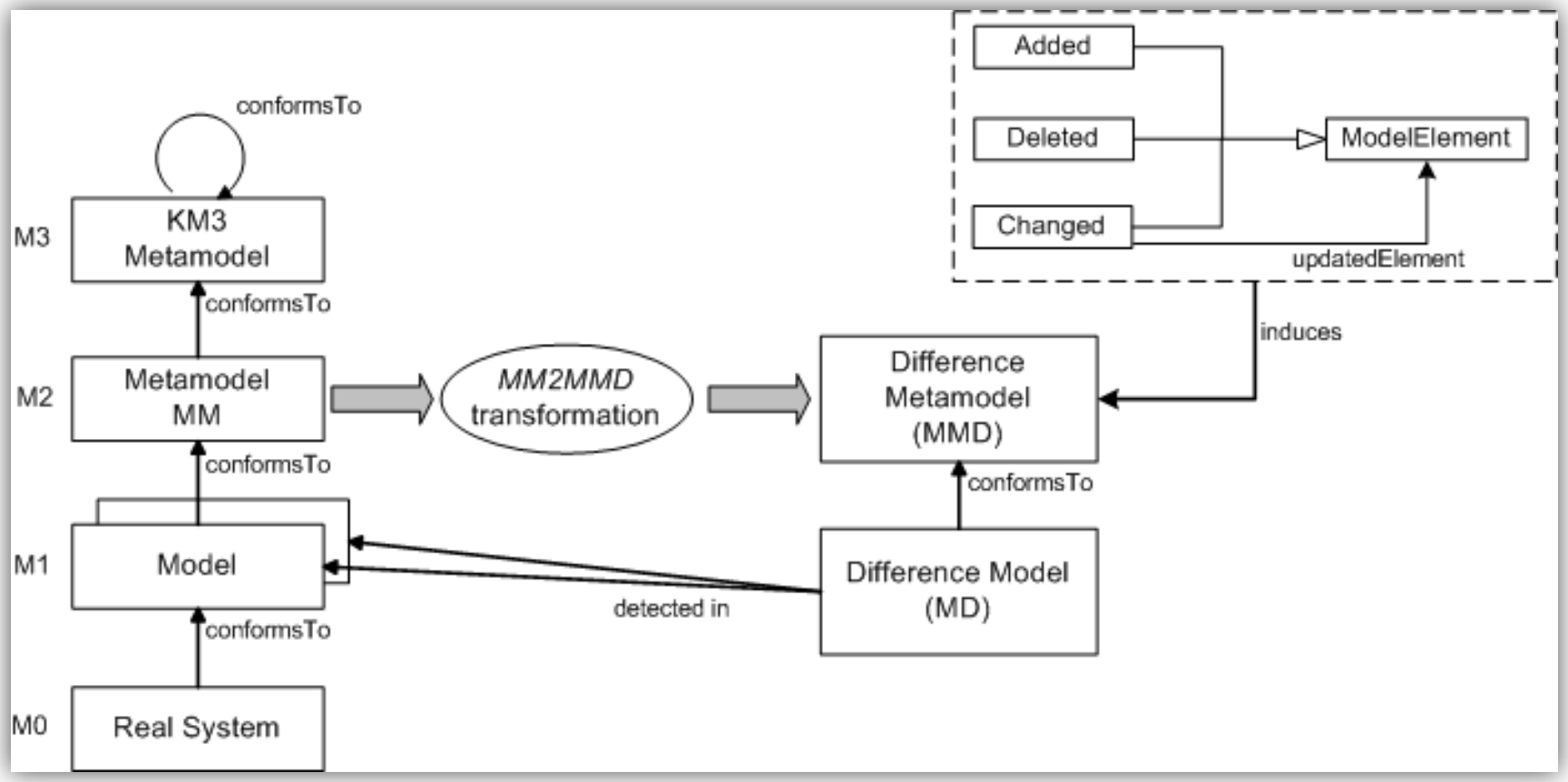


Breaking and unresolvable change (*Add obligatory metaproperty*)

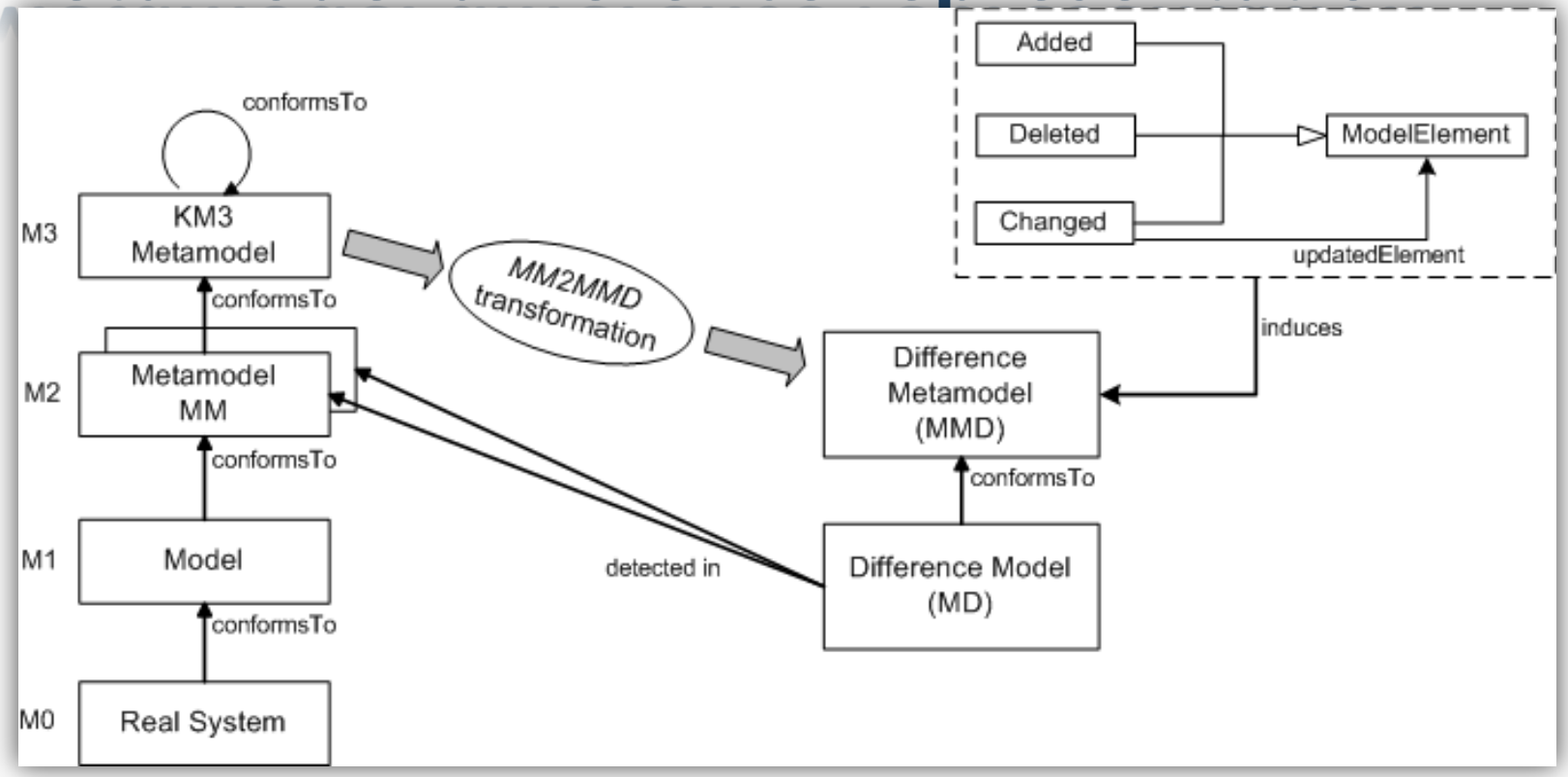


Breaking and unresolvable change (*Add obligatory metaproperty*)

Model difference representation

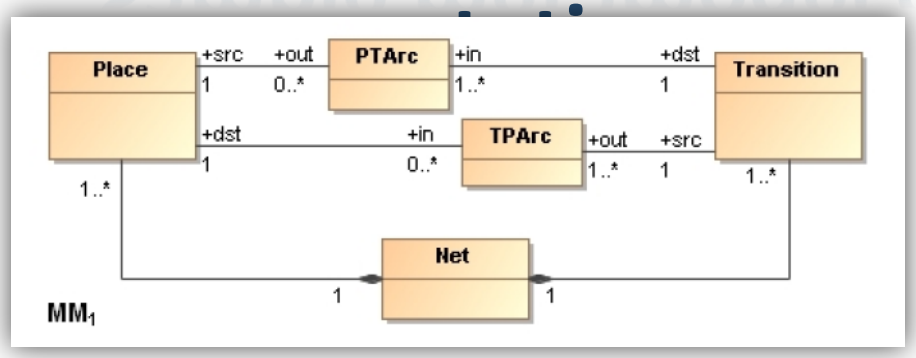


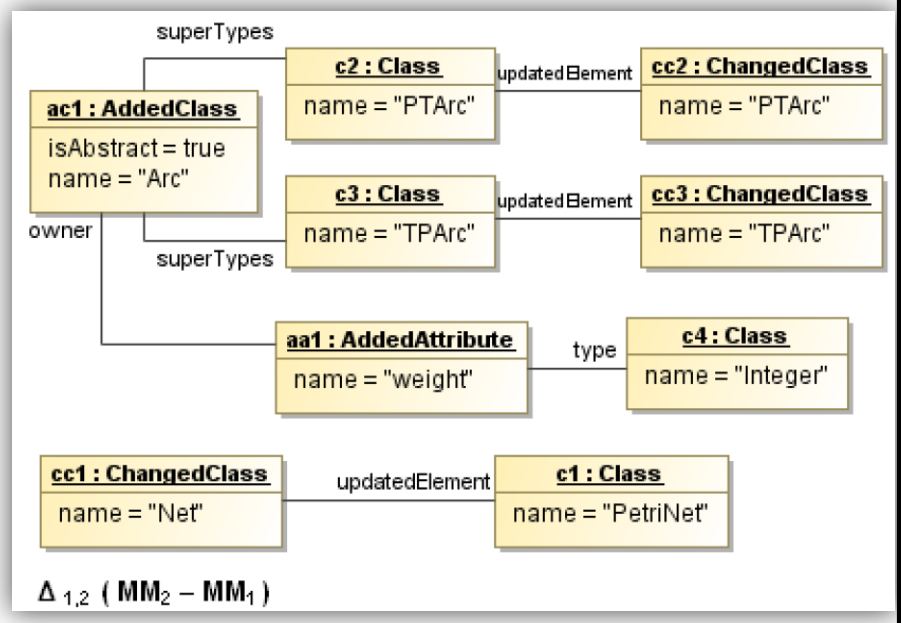
Metamodel difference representation

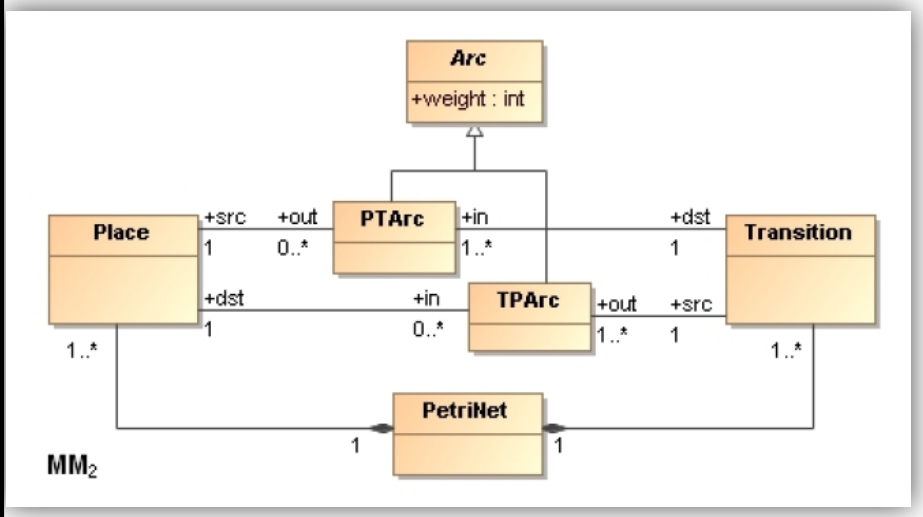


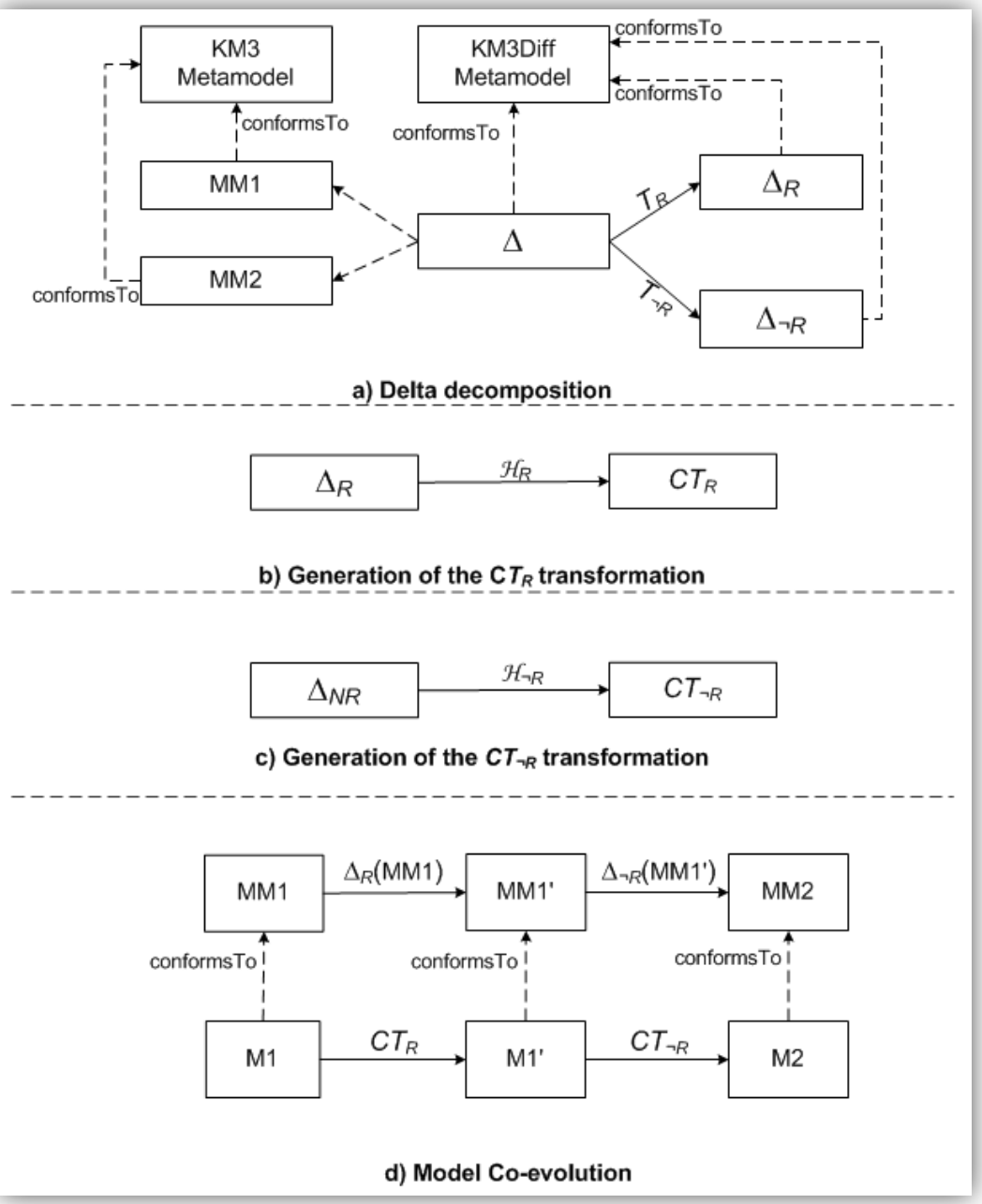
Since a metamodel is a model itself, metamodel differences can be represented by exploiting the previously mentioned approach

Sample metamodel difference







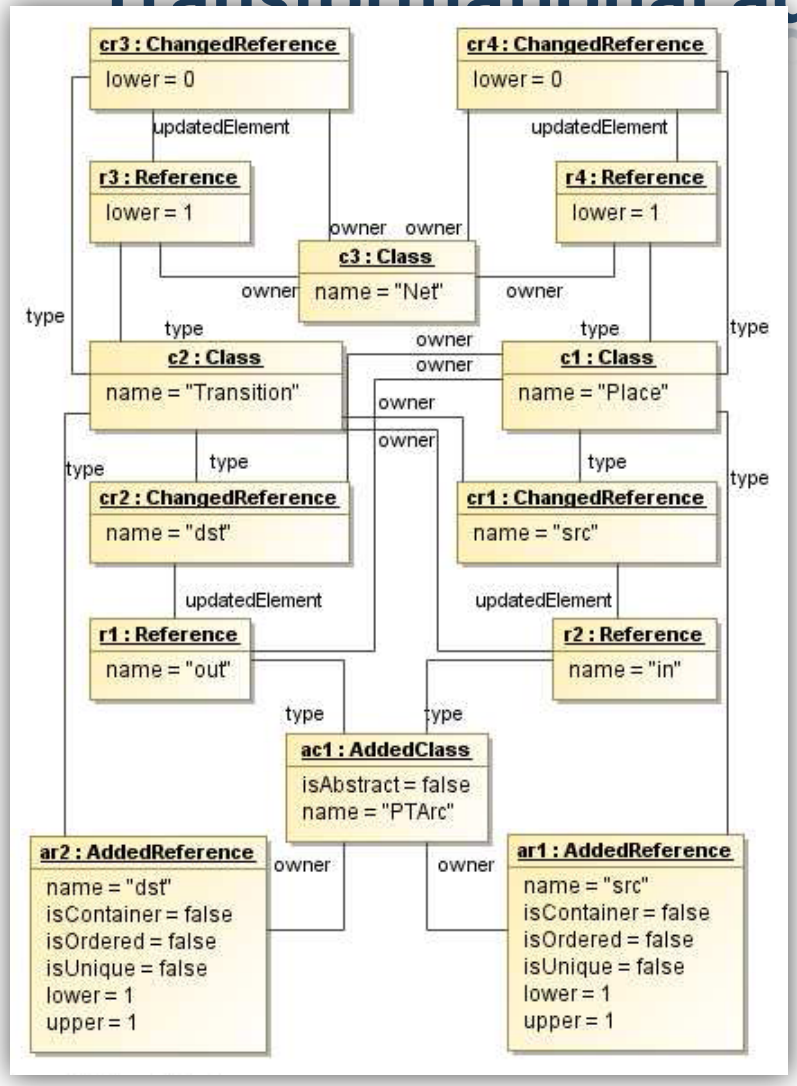


In realistic applications, the modifications in Δ consist of an arbitrary combination of the atomic changes previously summarized

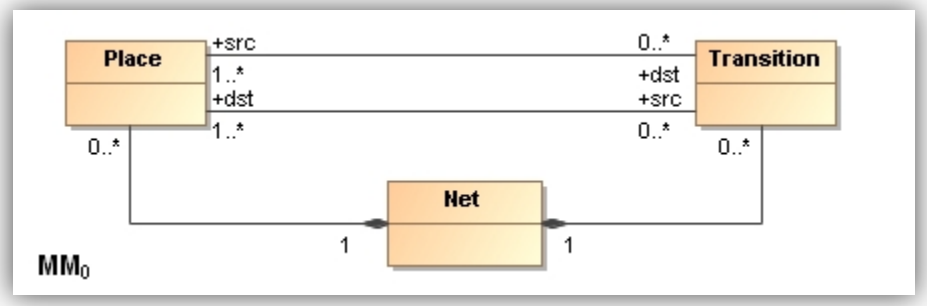
In order to distinguish them the following steps are performed:

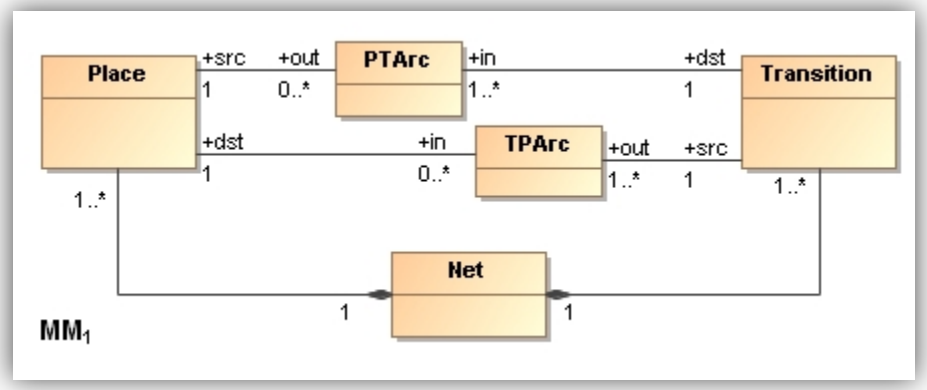
1. automatic decomposition of Δ in two disjoint (sub) models, Δ_R and $\Delta_{\neg R}$, which denote breaking resolvable and unresolvable changes;
2. if Δ_R and $\Delta_{\neg R}$ are parallel independent then we separately generate the corresponding co-evolutions;
3. if Δ_R and $\Delta_{\neg R}$ are parallel dependent, they are further refined to identify and isolate the interdependencies causing the interferences

Transformational adaptation of models



$\Delta_{(0,1)}$

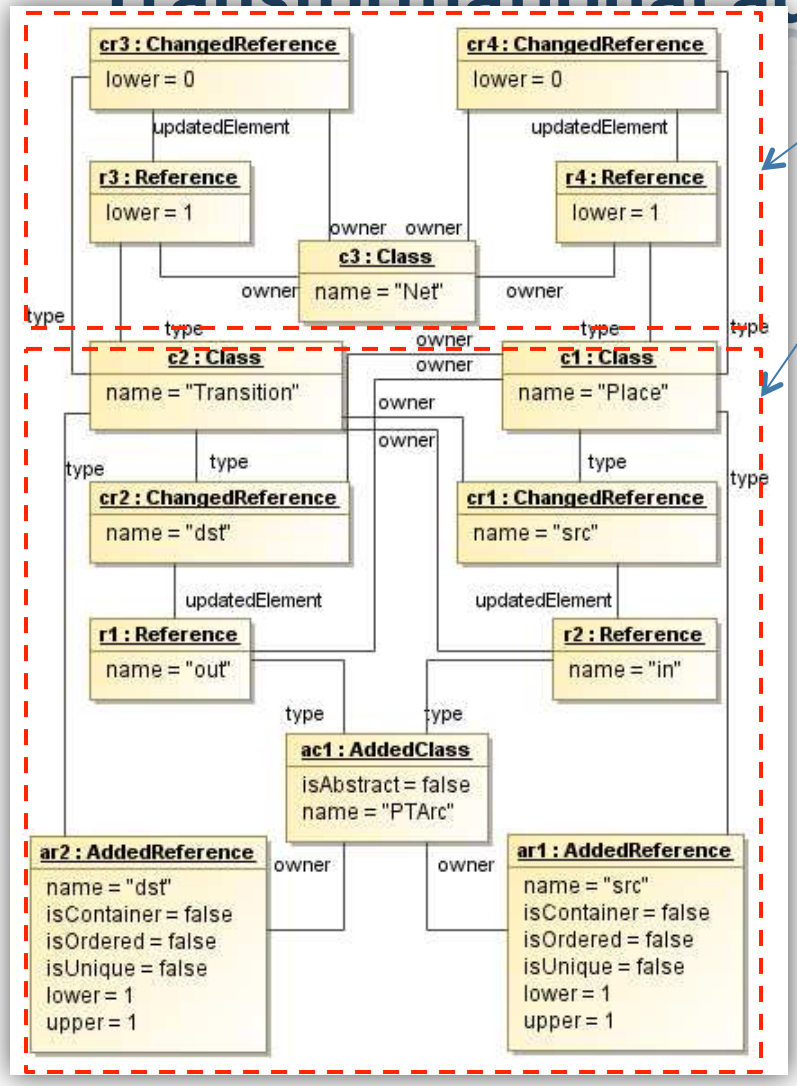




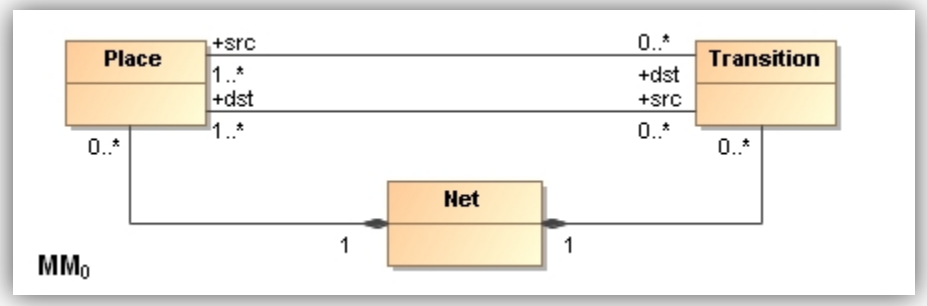
Transformational adaptation of models

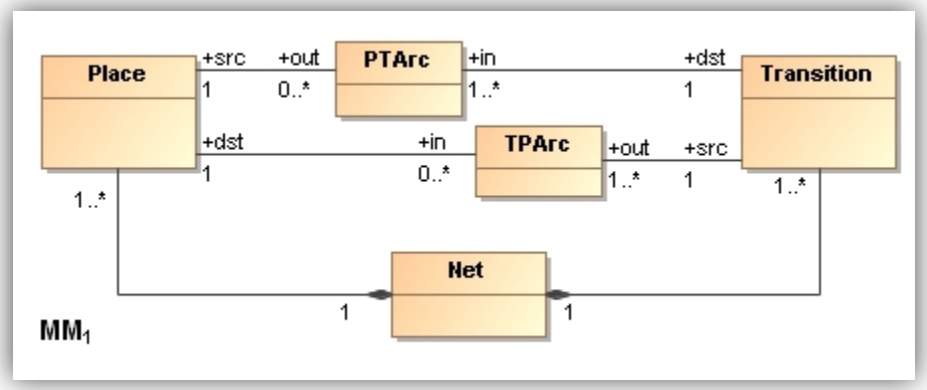
Restrict metaproperty change

Extract metaclass changes

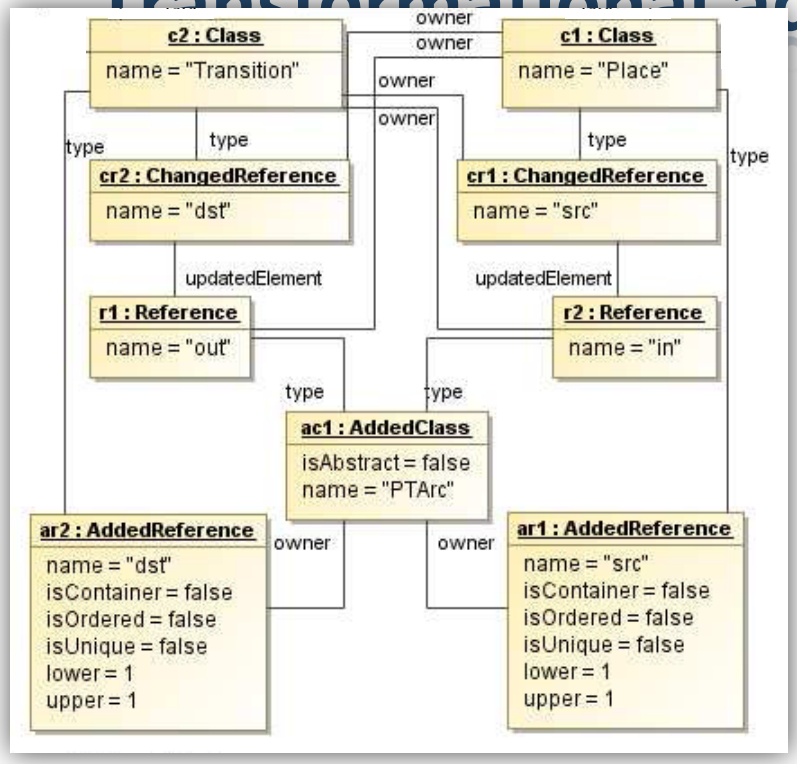


$\Delta_{(0,1)}$





Transformational adaptation of models:

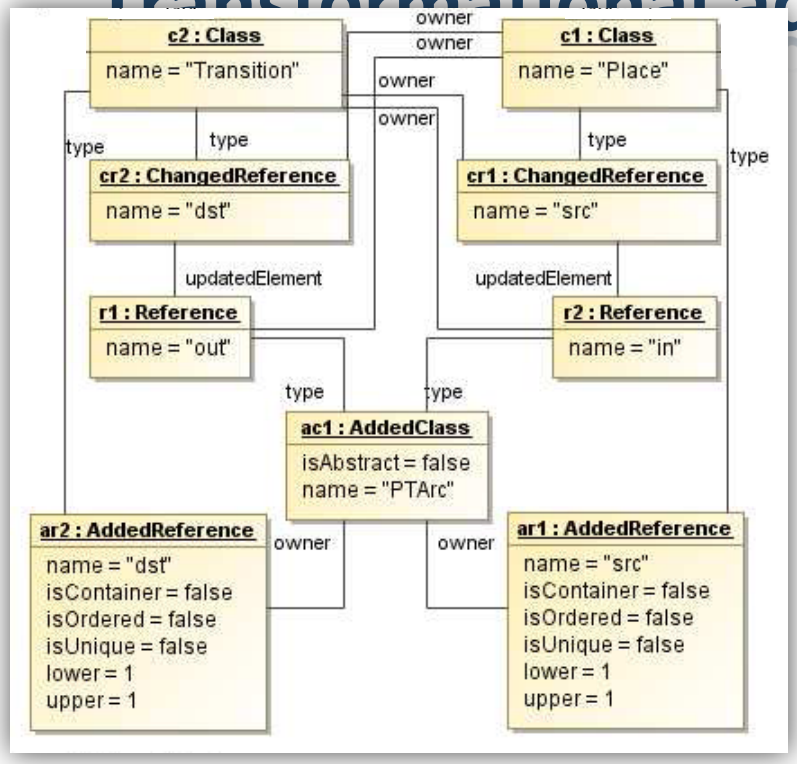


$\Delta_{R(0,1)}$

```
module H_R;
create OUT : ATL from Delta : KM3Diff;

rule CreateRenaming {
…
}
rule CreateExtractMetaClass {
…
}
…
```

H_R

Transformational adaptation of models:



$\Delta_{R(0,1)}$

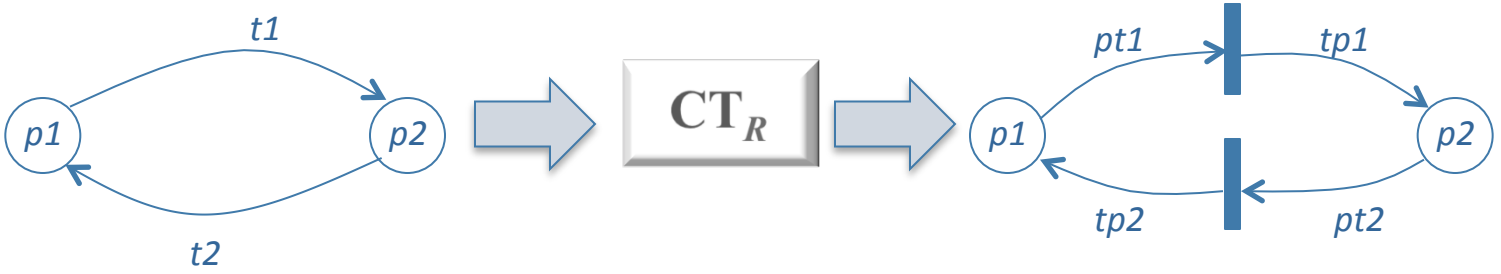
```
module H_R;
create OUT : ATL from Delta : KM3Diff;

rule CreateRenaming {
…
}
rule CreateExtractMetaClass {
…
}
…
```

H_R

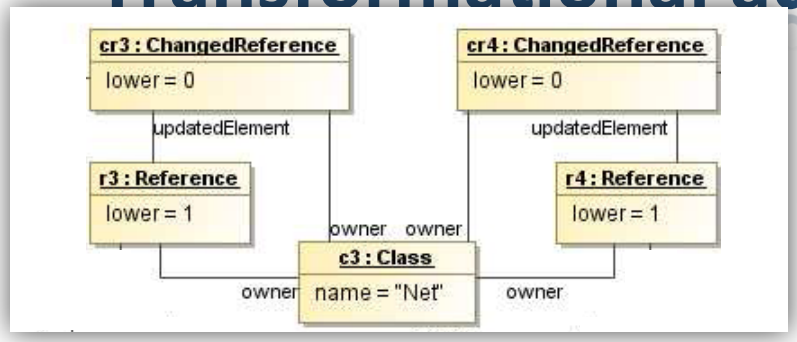
```
module CTR;
create OUT : MM1 from IN : MM0;
…
rule createPTArc(s : OclAny, n : OclAny)
{
…
}
rule createTPArc(s : OclAny, n : OclAny)
{
…
}
```

CT_R

Transformational adaptation of models: example



Transformational adaptation of models:

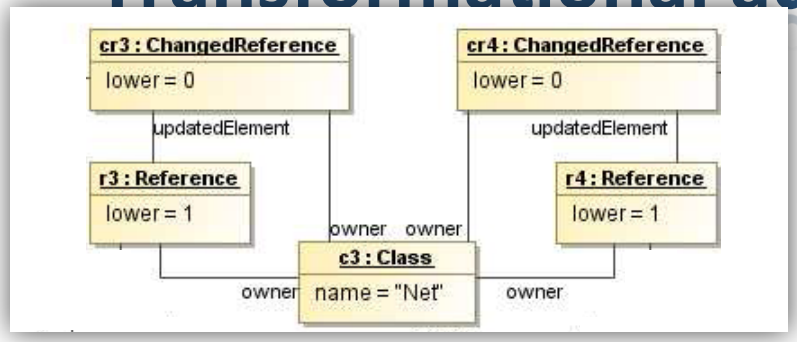


$\Delta_{\neg R(0,1)}$

```
module H_NR;
create OUT : ATL from Delta : KM3Diff;

rule CreateRestrictMetaproperty{
…
}
rule AddObligatoryMetaclass {
…
}
…
```

$H_{\neg R}$

Transformational adaptation of models:



$\Delta_{\neg R(0,1)}$

```
module H_NR;
create OUT : ATL from Delta : KM3Diff;

rule CreateRestrictMetaproperty{
…
}
rule AddObligatoryMetaclass {
…
}
…
```

$H_{\neg R}$

```
module CTR;
create OUT : MM1 from IN : MM0;
…
helper context MM2!Net def:createPlaceInstances() : Sequence (MM2!Place) =
if (thisModule.placeInstances < 1) then
        thisModule.createPlace(self)
        ->asSequence()
        ->union(self.createPlaceInstances())
else
        Sequence {}
endif;
…
```

$CT_{\neg R}$

Parallel dependent modifications

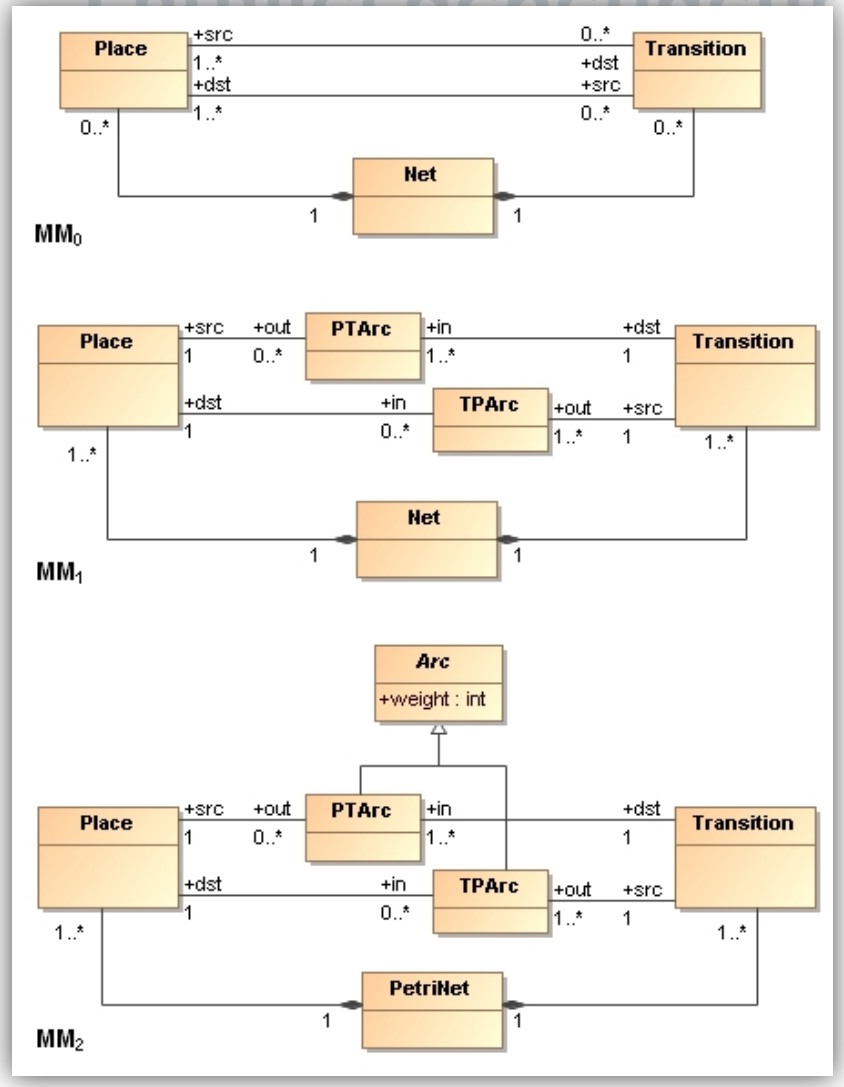
The automatic co-adaptation of models relies on the **parallel independence** of breaking resolvable and unresolvable modifications, or more formally

$$\Delta R | \Delta \neg R = \Delta R;\Delta \neg R + \Delta \neg R;\Delta R$$

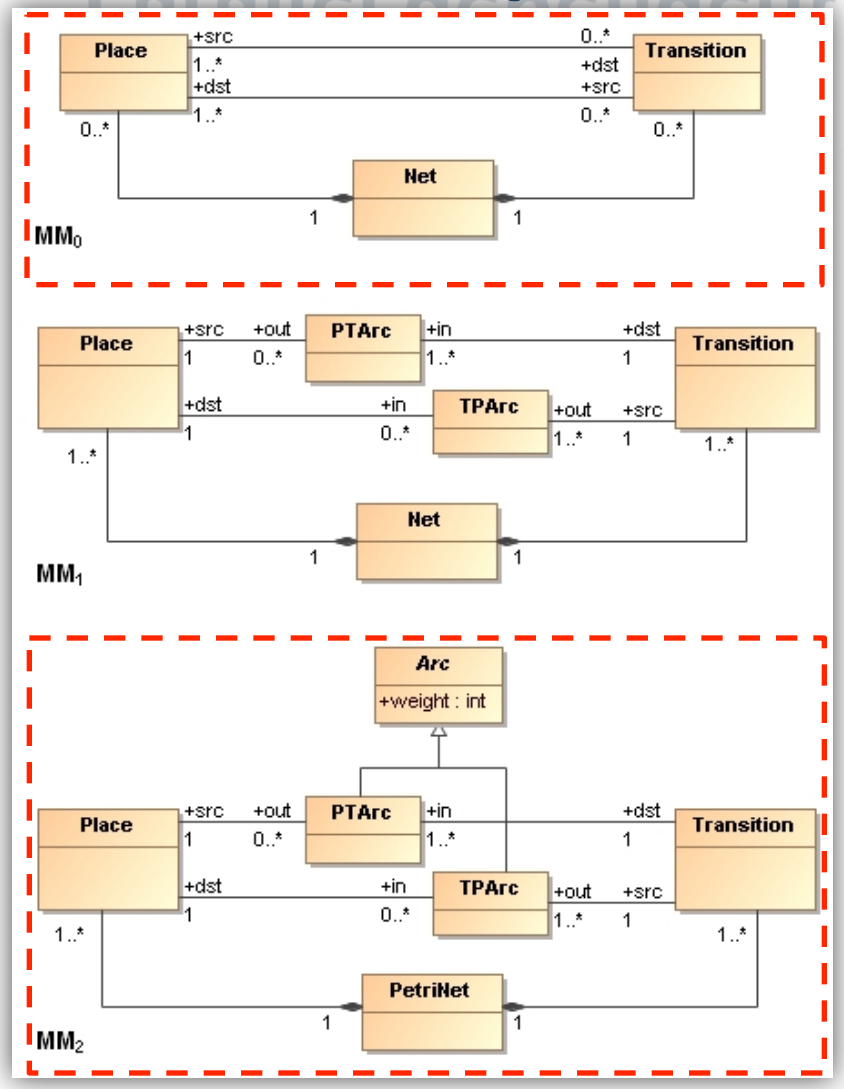
where + denotes the non-deterministic choice

Unfortunately, the distinction between resolvable and unresolvable changes is not always feasible because of **parallel dependent changes**, i.e. situations where multiple changes are mixed and interdependent one another

Parallel dependent modifications



Parallel dependent modifications



The differences between *MM2* and *MM0* are not parallel independent (although the sub steps *MM0–MM1* and *MM1 – MM2* are directly manageable)

The interdependencies between the atomic changes in *MM2 – MM0* have to be isolated (i.e. the attribute *weight* of the *Arc* metaclass of *MM2*)

Priorization of changes

The problem can be solved by sorting out those modifications, say δ, which are interfering each other

$$\Delta = (\Delta' \; R \; | \Delta' \neg R); \; \delta$$

Moreover, δ must be the smallest modification conforming to the difference metamodel

The modification in δ priorized according to a dependency analysis whose outcome does not depend on the metamodel but only on the meta-metamodel, thus it is always possible

Transformations

- Introduction
- Why Model Transformation languages ?
- Dimensions and Classification

A demonstration of ATL

- Objectives
- Metamodels
- Live Demonstration

Bidirectional Transformation for Change Propagation

- Problem
- Requirements
- Janus Transformation Language
- Change Propagation and non-determinism

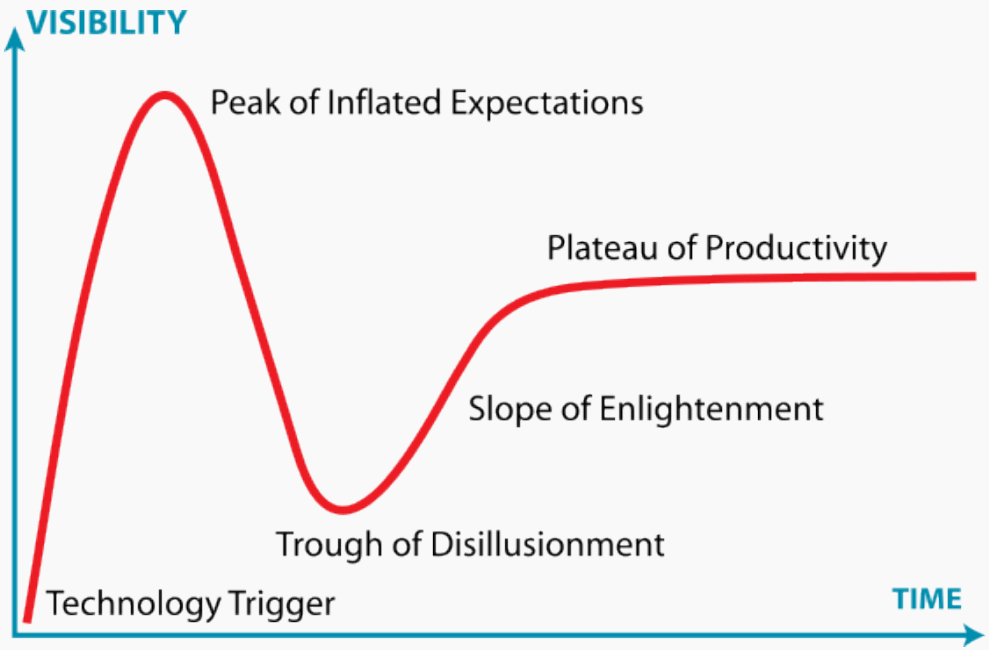
Higher-Order Transformations for Automating Co-evolution

- Evolution in MDE
- Metamodel Changes Classification
- Metamodel Differences
- Automated Adaptation

Summary

Conclusions

Whether MDE will miss the boat or will reach the plateau of productivity very much depends also on the effectiveness of model transformations



Conclusions

It is a critical mistake thinking model transformations useful only for code generation like problems

There are plenty of projects using model transformations and general model management in a productive and innovative way

These applications can increase of a key factor effectiveness or even enlarge the application spectrum

Current research directions

Reuse of Transformations

- → Generic Transformations
- → Model Typing

Verification and Testing of Transformation

- → See Antonio Vallecillo contribution on Saturday

Agility and Model Transformations

- → Test-driven development of transformations
- → Transformations by examples

Contact Information

Prof. Alfonso Pierantonio
Dipartimento di Informatica
Università degli Studi dell'Aquila
I-67100 L'Aquila, Italy

alfonso.pierantonio@univaq.it
http://www.di.univaq.it/alfonso

Research Interests: Coupled Evolution of Models, Tranformations and Tools, Bidirectional Transformations, Model Versioning