Architecture, Design (and a little Verification) for BX

Richard Paige

Dept of Computer Science, University of York

@richpaige

Migrating Uber from MySQL to PostgreSQL

Evan Klitzke

Uber, Inc.

March 13, 2013

ARCHITECTURE

WHY UBER ENGINEERING SWITCHED FROM POSTGRES TO MYSQL

JULY 26, 2016

BY EVAN KLITZKE

Contents

- What is architecture and design for transformations and BX?
- Architecture specification for BX.
- Detailed design specification for BX.
- Design patterns for BX.
- (just a little on…) Verification for BX.

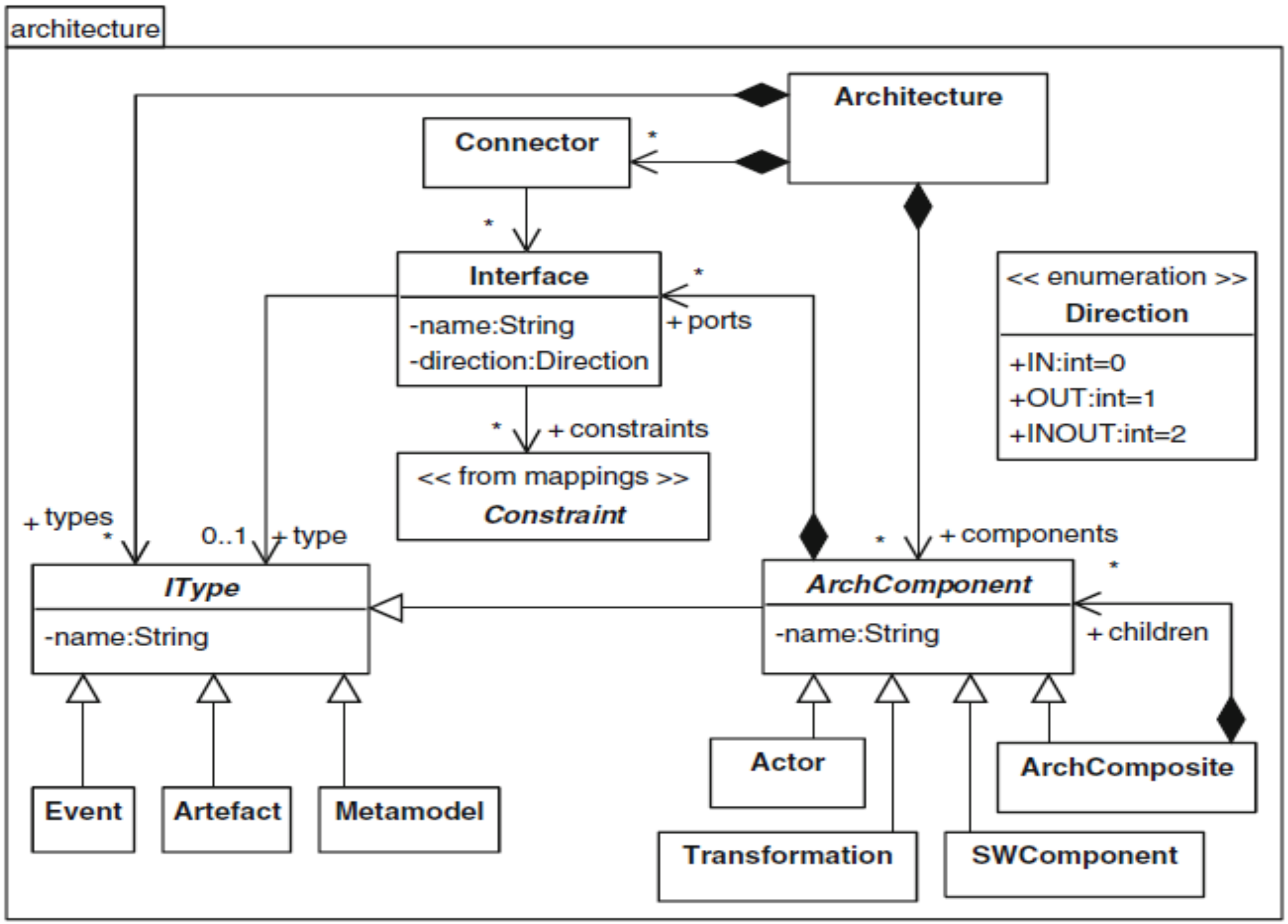
Motivation

- Large and complicated BX are like large and complicated software systems:
  - Many parts
  - Complex interrelationships and dependencies
  - Sophisticated behaviour (often implicit)
  - Difficult to get right, difficult to verify.
- Large software is seldom monolithic.
  - Decomposed into interdependent components

Motivation

- Architecture for BX and transformations is complicated:
  - What are the constituent blocks?
  - How can they be related? (ports, protocols, buffers)
  - How can a transformation architecture be integrated with other components
    - e.g., code generators, visualisations (e.g., non-MDE).

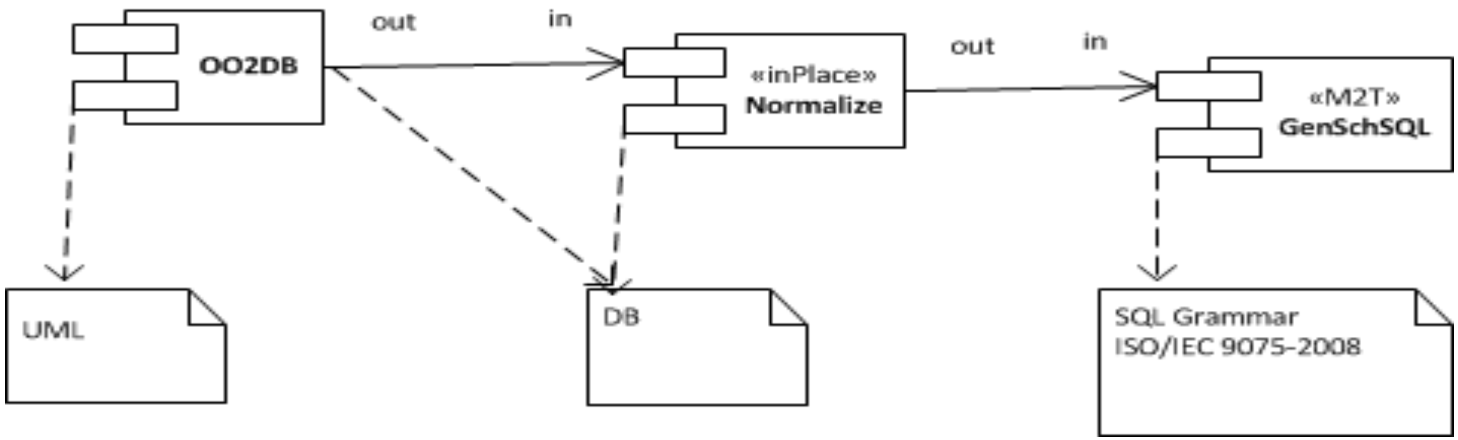
Architecture in *trans*ML



Architecture in *trans*ML

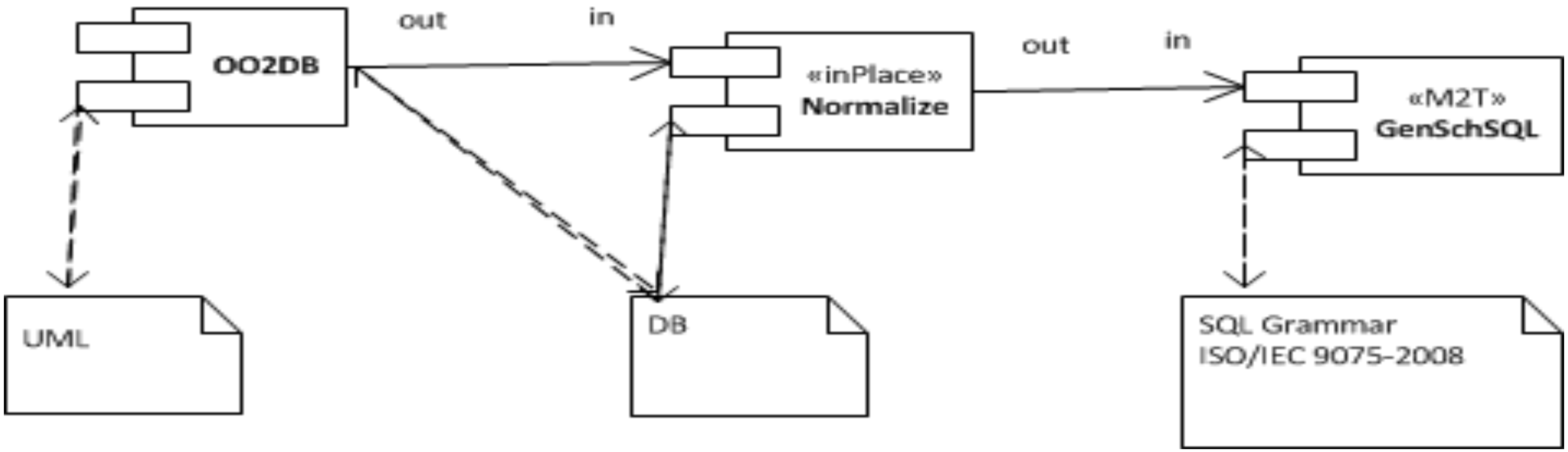
- Components and connectors that interact via directional interfaces.
  - Architectural components can be transformations, software (black box), actors (human intervention), or composites
    - BX do not exist in a vacuum!
  - Types (of interfaces, ports, components) given by metamodels, event types, artefacts or architectural components.
- Contracts can be imposed to restrict expected inputs and outputs, and to enable conformance checking.

Architecture Example



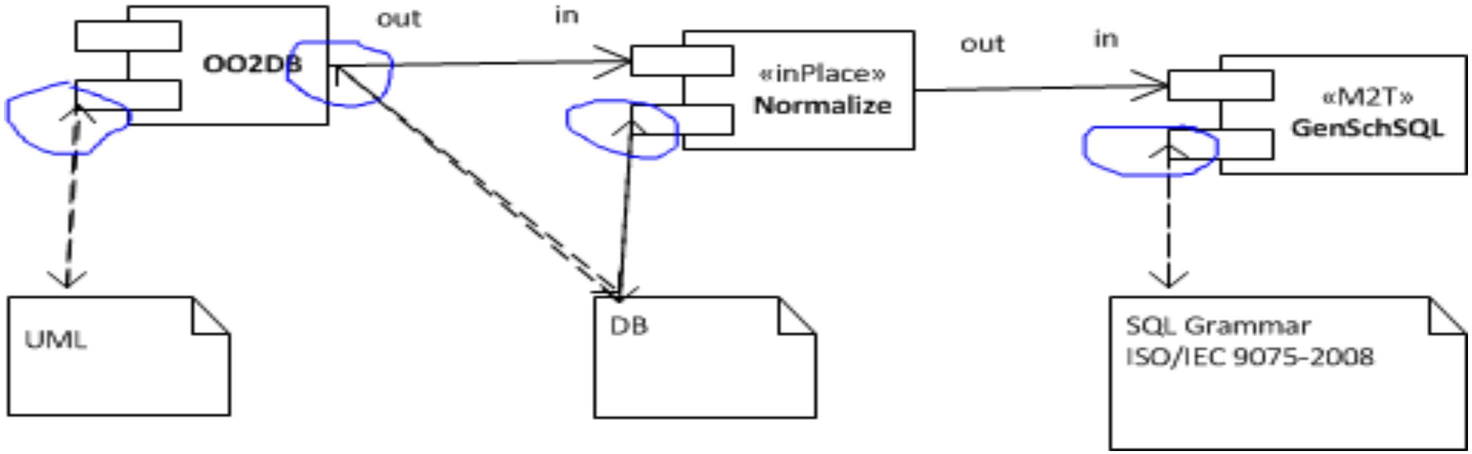
- Transformation centric view
- Unidirectional: OO->DB->optimise->SQL

Architecture Example



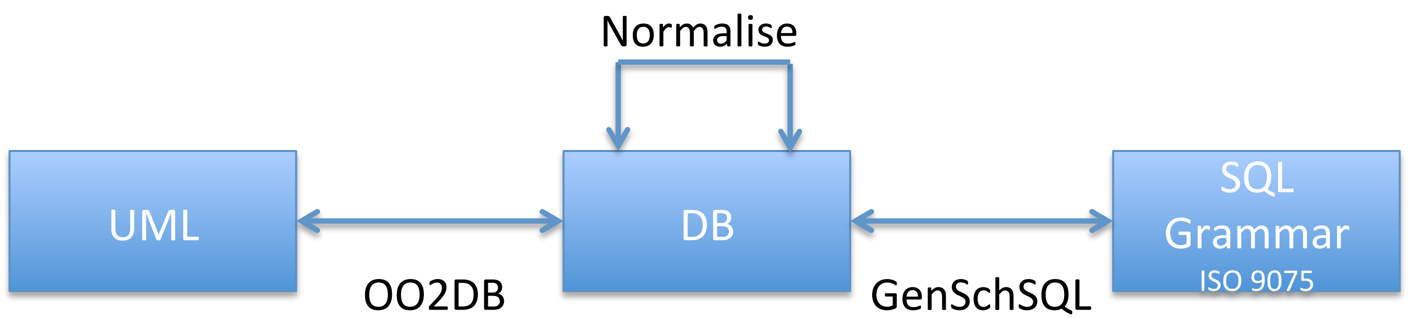
- Transformation centric view
- Bidirectional components: OO->DB->optimise->SQL

Architecture Example



- Transformation centric view
- Bidirectional components: OO->DB->optimise->SQL

Architecture Example



- Type centric view (is this a slightly different architecture?)
- BX: OO2DB, Normalise, GenSchSQL could be run individually in either direction.
- Similar to a megamodel, where components are visualised as arrows connecting interfaces.
  - Useful for bridging grammars and models.

Architecture Styles?

- Are there BX equivalents to typical software architectural styles?
  - Pipe-and-Filter
  - Model-View-Controller
  - Layered
  - Pub-Sub
  - Data-Centric

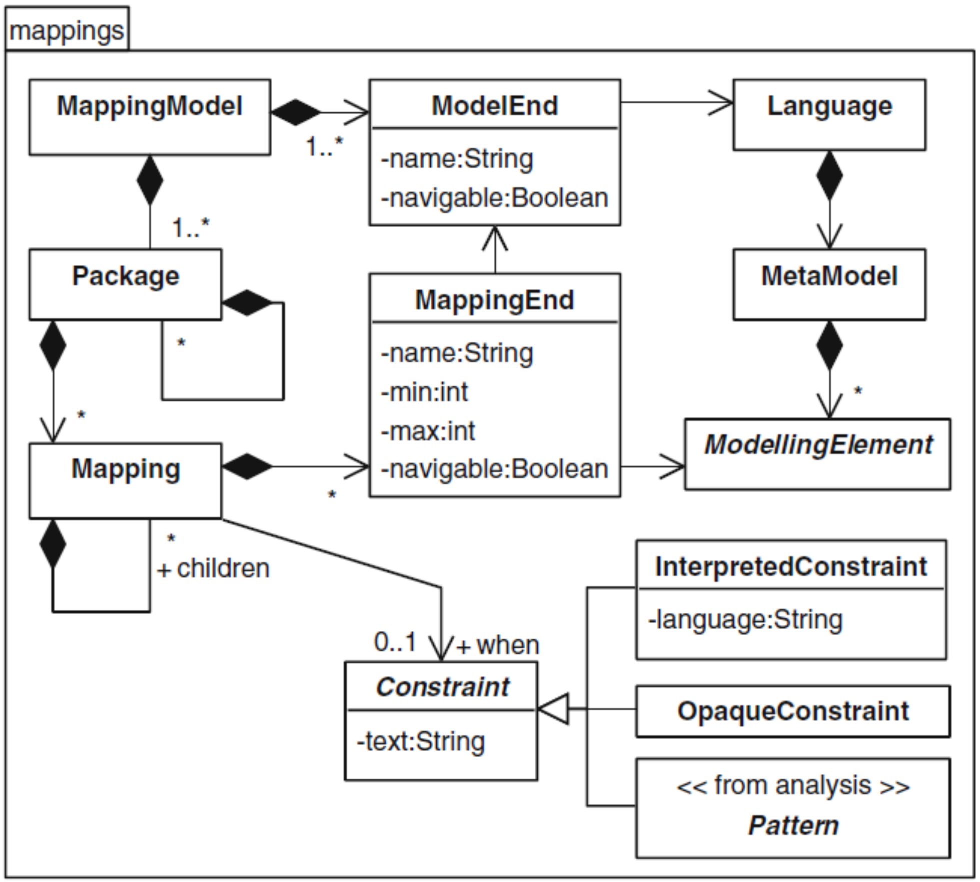
BX Design

- The architecture of a transformation indicates the key components and their connectors.
- Engineering of BX continues with design.
  - High-level design: what is transformed into what?
  - Low-level design: how is the transformation carried out?
- Take each in turn

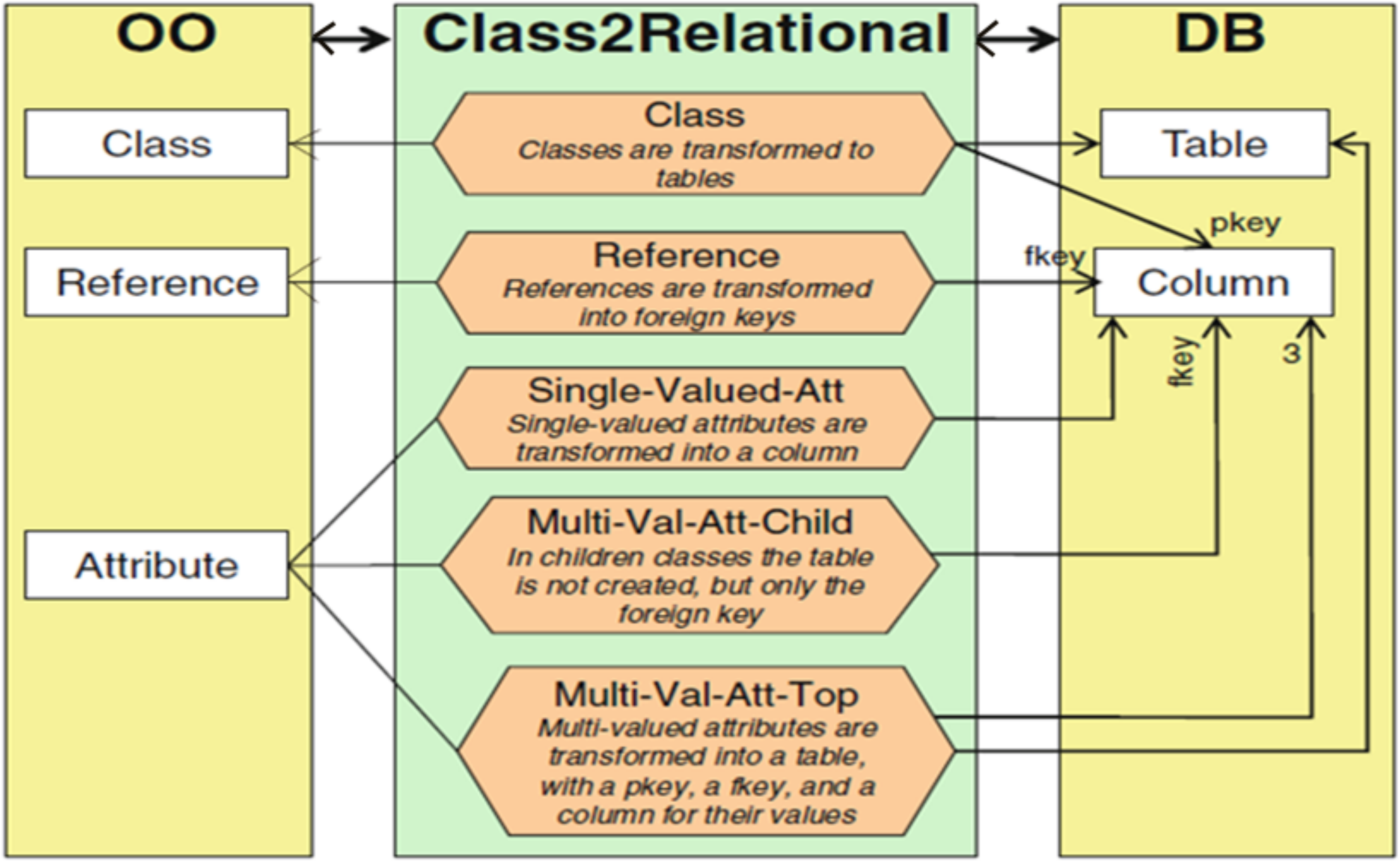
BX High-Level Design

- Mapping diagram.
- Captures the mappings between arbitrary elements in the transformation.
- *trans*ML uses a concrete syntax inspired by TGGs.
  - However, mappings are not meant to be used as a tracing mechanism to guide execution of code.
    - Don’t address, e.g., execution flow.

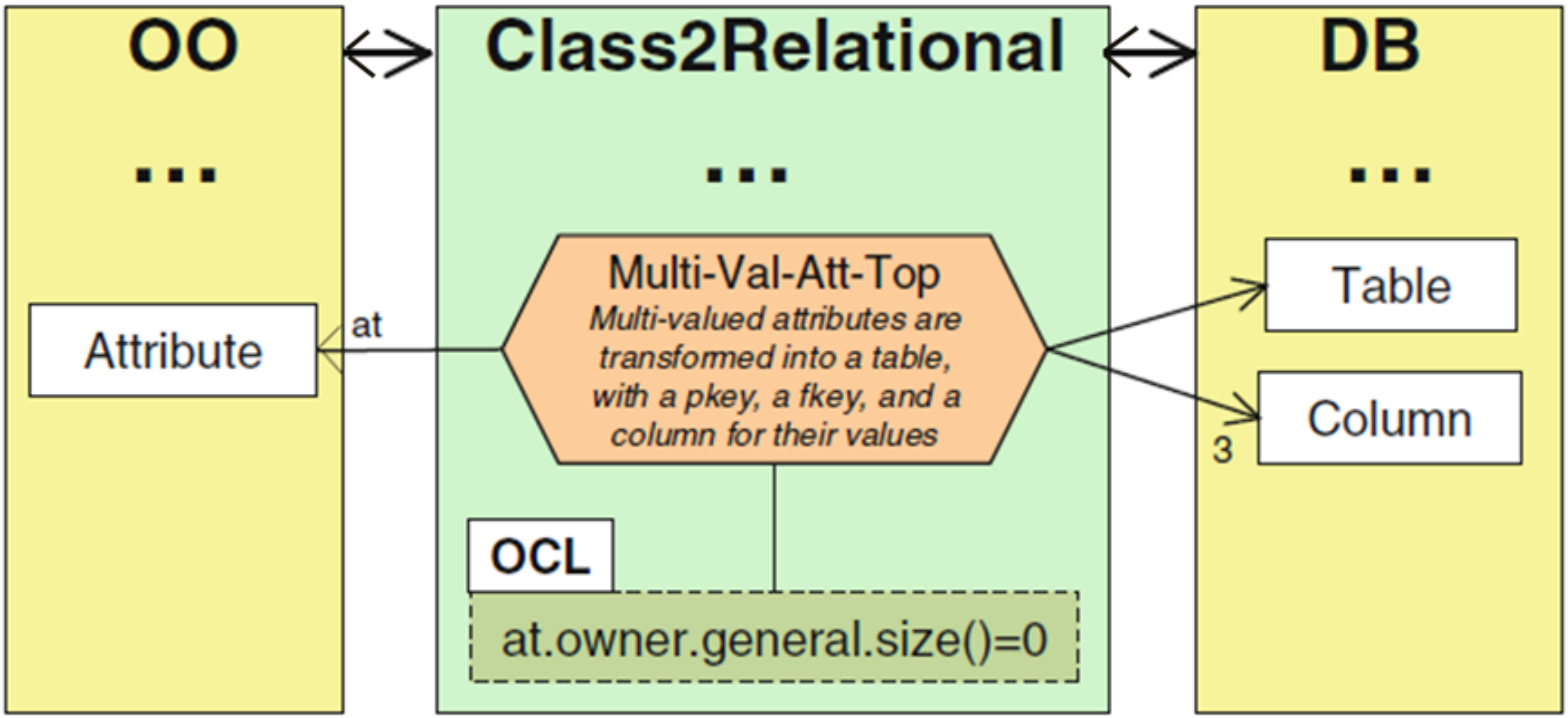
BX Mapping Metamodel



BX Mapping Example



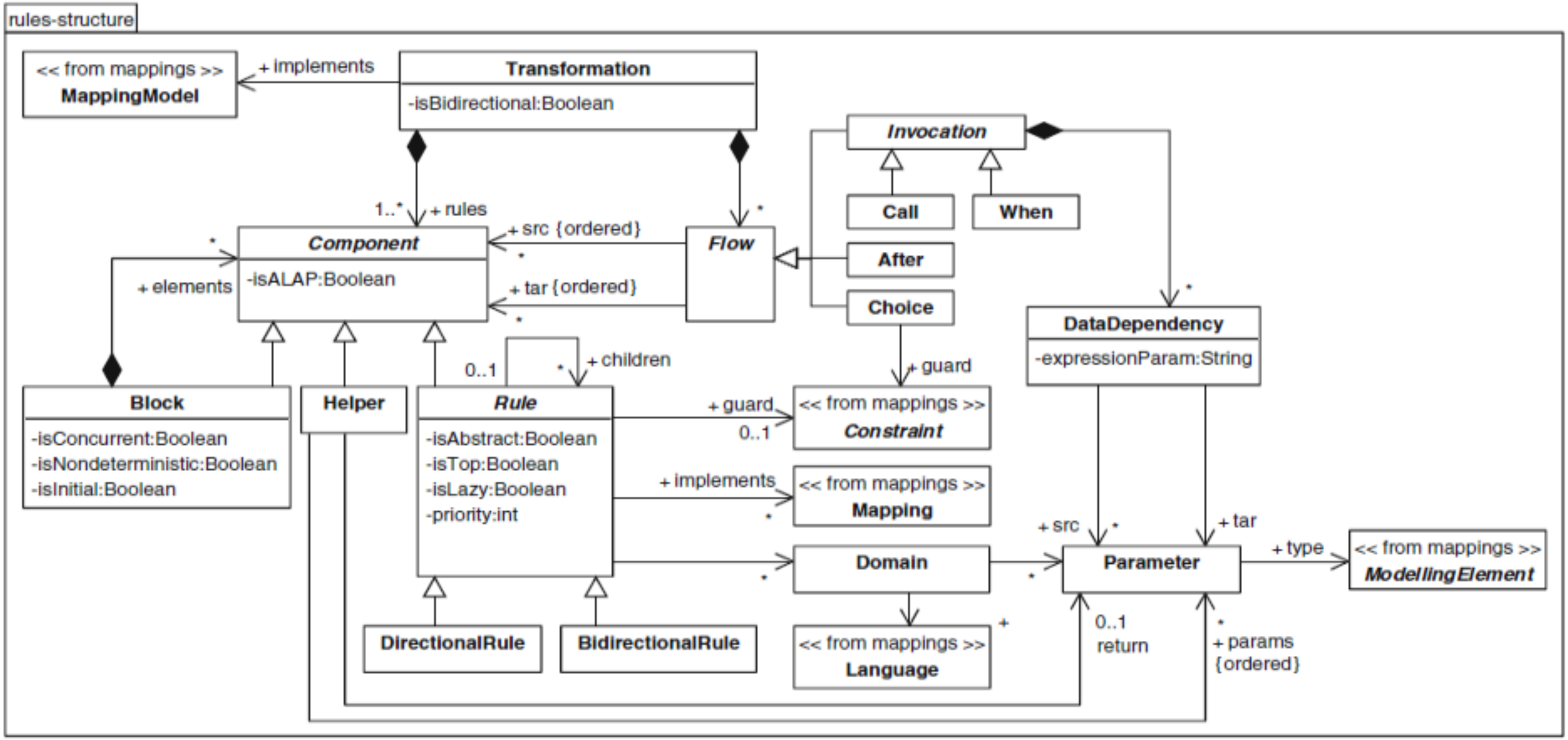
BX Mapping Example – Adding Constraints



BX Low-Level Design

- Indicates how the BX is to be implemented.
- Could use a BX programming language here.
  - But *trans*ML provides low-level design languages to try to support platform independence, focus on essentials
  - Essentials: rule structure, control flow, blocks (some not present in programming languages).
- *trans*ML: rule structure model and rule behaviour model.

Rule Structure Metamodel

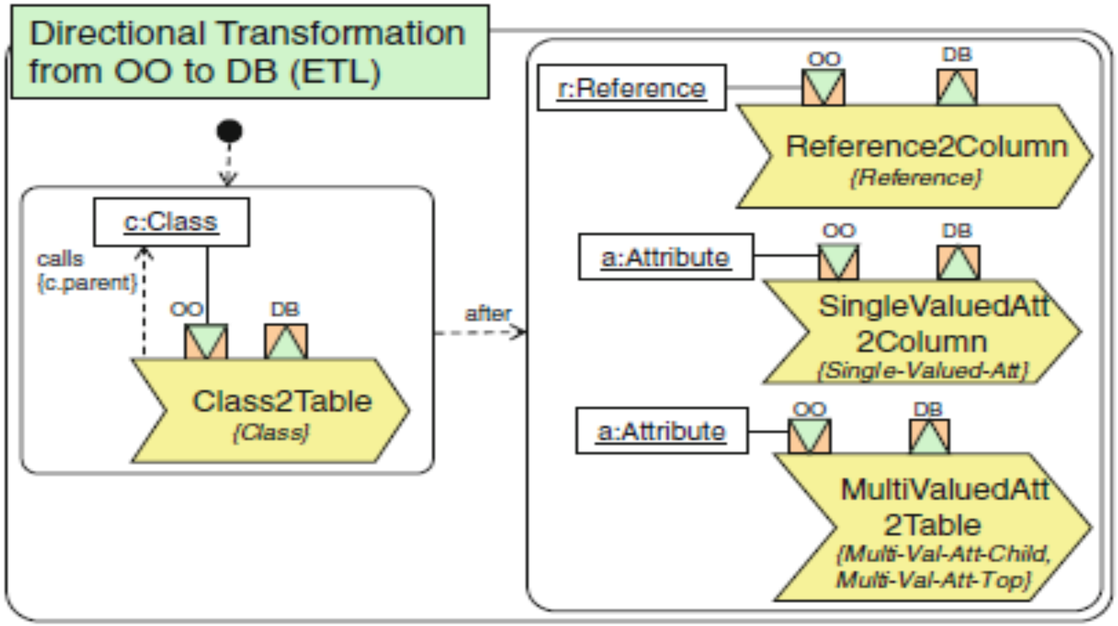


- Describes structure of rules (input, output), execution flow, and data dependencies

Rule Structure Models

- These refine mapping diagrams.
- A rule can contribute to the implementation of several mappings.
- Rules may be uni- or bidirectional.
- Execution flow may be explicit (e.g., a subclass of *Flow*) or non-deterministic:
  - A set of rules can be placed inside a non-deterministic block

Example


Directional Transformation from OO to DB (ETL)
c:Class
calls {c.parent}
OO
DB
Class2Table
{Class}
after
r:Reference
OO
DB
Reference2Column
{Reference}
a:Attribute
OO
DB
SingleValuedAtt 2Column
{Single-Valued-Att}
a:Attribute
OO
DB
MultiValuedAtt 2Table
{Multi-Val-Att-Child, Multi-Val-Att-Top}

Example


```
transformation Tree2Graph {
    nondeterministic RuleBlockForward {
        bidirectional Tree2Node {..};
        bidirectional TreeEdge2GraphEdge {..} ;
    }
    nondeterministic RuleBlockBackward {
        bidirectional TreeLabelsfromNodeLabels {..};
        bidirectional TreeEdgesfromGraphEdges{..};
    }
}
```

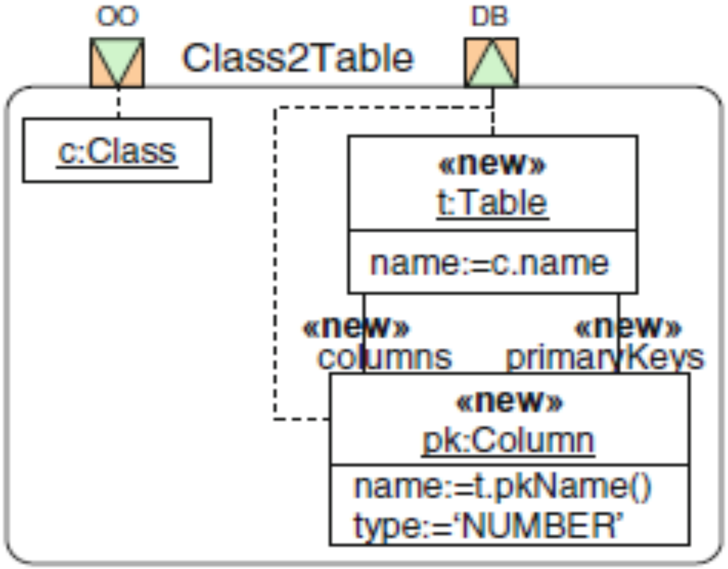
Rule Structure Model

- With rule structure, the particular implementation language of choice needs to be considered.
- This is because these models capture the rules and their execution flow (which is language semantics-specific).
  - For example, execution flow in ETL: each rule is executed once at each instance of input; for graph transformation it's "as long as possible".

Rule Behaviour Diagram

- The rule structure models treat rules as black-boxes, ignoring behaviour:
  - Attribute computation, object and link creation.
- Specified using rule behaviour diagrams:
  - Action language
  - Declarative graphical pre/post
  - Object diagrams annotated with operations (similar to Catalysis snapshots)

Example



transform c: OO!Class
to t:DB!Table, pk: DB!Column

t.name:=c.name;
pk.name:=t.pkName();
pk.type:='NUMBER';
t.columns.add(pk);
t.primaryKeys.add(pk);

Design Patterns for BX

Design Patterns

- Capture recurring design problems and their solutions (which must be instantiated).
- Many different patterns in the literature, including some for model transformation design.
  - Some of these patterns are applicable to the design of uni- or bidirectional transformations.
  - Some specific for BX.
  - Several examples.

Auxiliary Correspondence Model

- Weaving tools (such as AMW, EML) can be used to propagate changes from/to models in a BX.
  - They do or can make use of an auxiliary correspondence (weaving) model.
- Pattern: defines auxiliary model elements and associations that link source and target elements.
- Why: used to record mappings performed by a BX, and to propagate modifications when one model changes.
- Benefits: separation of concerns, helps to ensure correctness
- Disadvantages: must maintain an additional model.

Unique Instantiation

- Why: Avoids creation of unnecessary elements of models and helps to resolve nondeterministic choice in reverse mappings.
  - *E.g.,* in check-before-enforce in QVT-R: new elements are not created if there are elements that satisfy the relations.
- Benefits: helps to establish hippocraticness
- Disadvantages: must test for existence, adds to cost (but other patterns like indexing can help).

Map Objects Before Links

- Why: Separates the relation between elements in target and source models from the relation between links in the models.
  - That is, first map "nodes", then map "edges" (largely useful for models with self-associations or circular dependencies)
- Benefits: modular specification, e.g., if new association is added to languages, new relation can be added more easily.
- Disadvantages: edges modular, features may not be!
  - We've seen this type of trade-off before!

Verification of BX

Verification of BX

- Many approaches, including correctness by construction, unit testing, etc.
  - *transML* includes a model-based testing approach where tests can be automatically generated from transformation scenarios
- Will talk about one specific and different approach.

BX: is there another way?

if a framework existed in which it were possible to write the directions of a transformation separately and then check, easily, that they were coherent, we might be able to have the best of both worlds

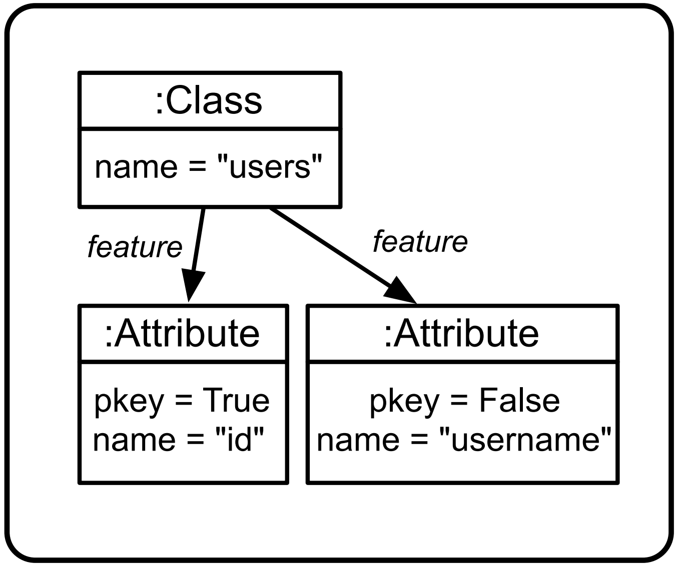
Stevens, P.: A landscape of bidirectional model transformations. In: GTTSE 2007.

"Faking" BX in Epsilon

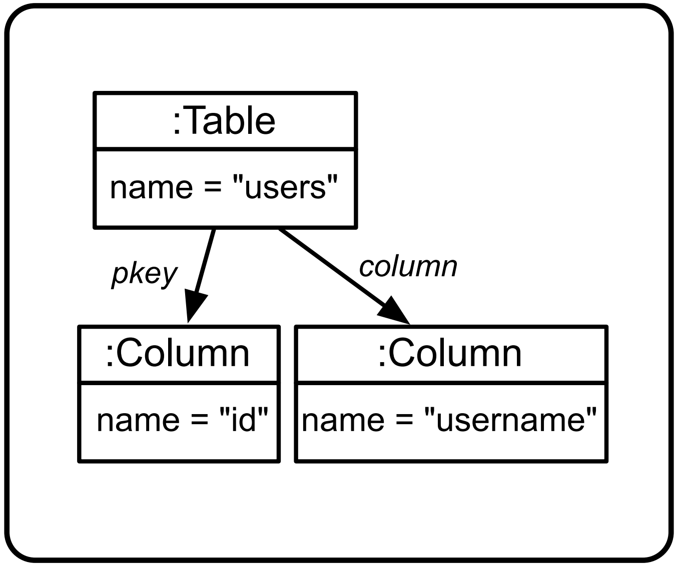
- Epsilon is a platform of interoperable model management languages
- No direct support for BX, but:
  => languages for unidirectional transformations (ETL, EWL, EOL)
  => an inter-model consistency language (EVL)
- BX can be faked in Epsilon by:
  (1) defining pairs of unidirectional transformations
  (2) defining consistency via inter-model constraints

OO2RDBMS

- two metamodels: class diagram and relational DB
- consistency defined in terms of a correspondence between the data (attributes) in the models



class diagram



relational DB

Example BX "faked" in Epsilon

- users of the models should be able to create new classes (or tables) whilst maintaining consistency
- first, we specify a pair of unidirectional transformations in Epsilon's update-in-place language

```
wizard AddClass {
 do {
   var c: new Class;
   c.name = newName;
   self.Class.all.first().contents.add(
        c);
 }}
```

```
wizard AddTable {
 do {
   var table: new Table;
   table.name = newName;
   self.Table.all.first().contents.add(
        table);
 }}
```

Example BX "faked" in Epsilon

- then, we specify and monitor inter-model constraints that express what it means to be consistent

```
context OO!Class {
 constraint TableExists {
   check : DB!Table.all.select(t|t.name
         = self.name).size() > 0
}}
```

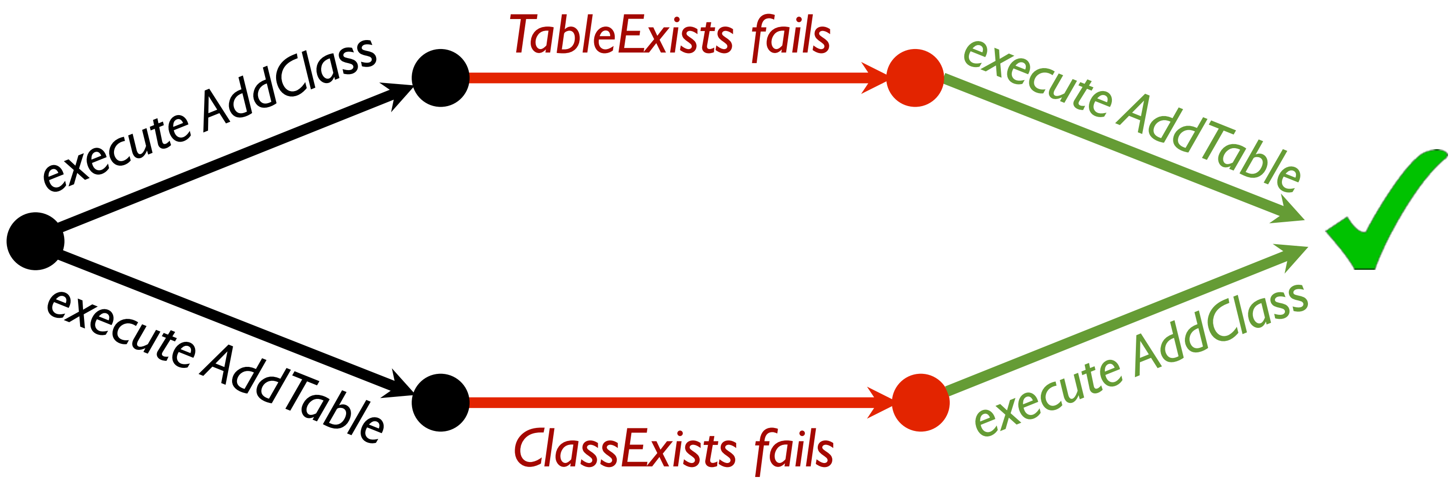
```
context DB!Table {
 constraint ClassExists {
   check : OO!Class.all.select(c|c.name
         = self.name).size() > 0
}}
```

Example BX "faked" in Epsilon

- then, we specify and monitor inter-model constraints that express what it means to be consistent

```
context OO!Class {
 constraint TableExists {
   check : DB!Table.all.select(t|t.name
         = self.name).size() > 0
}}
```

```
context DB!Table {
 constraint ClassExists {
   check : OO!Class.all.select(c|c.name
         = self.name).size() > 0
}}
```



More needs to be "faked"

- fake BX lack the consistency guarantees that true BX have by construction

- what does this mean?

  => compatibility of the directions might not be maintained (e.g., discovered when checking consistency)

  => repair transformations might not actually restore consistency

- our example is obviously compatible, but we should be able to check this easily and automatically

Our proposal

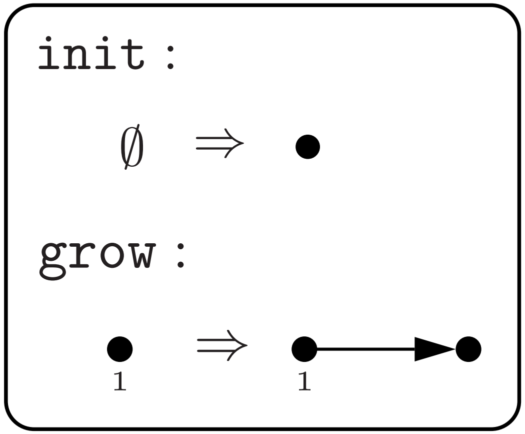
Exploit graph transformation verification techniques to check compatibility

- graph transformation (GT) is a computation abstraction
  => state is represented as a graph
  => computational steps represented as GT rule applications

Our proposal

Exploit graph transformation verification techniques to check compatibility

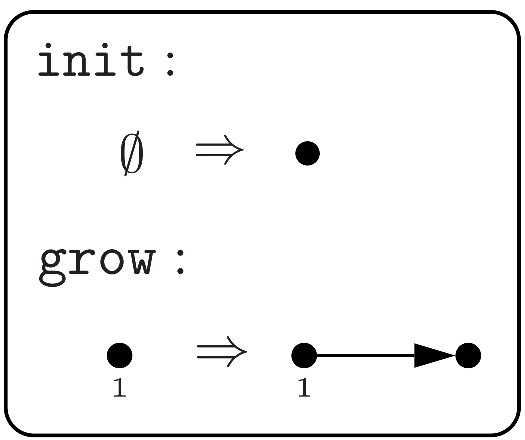
- graph transformation (GT) is a computation abstraction
  => state is represented as a graph
  => computational steps represented as GT rule applications

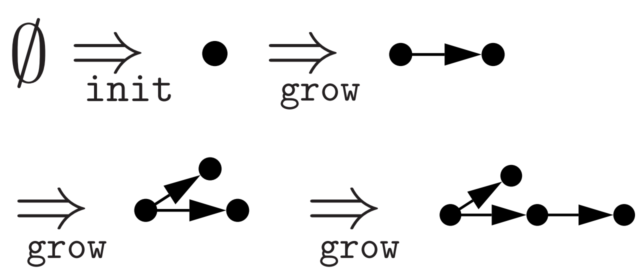


Our proposal

Exploit graph transformation verification techniques to check compatibility

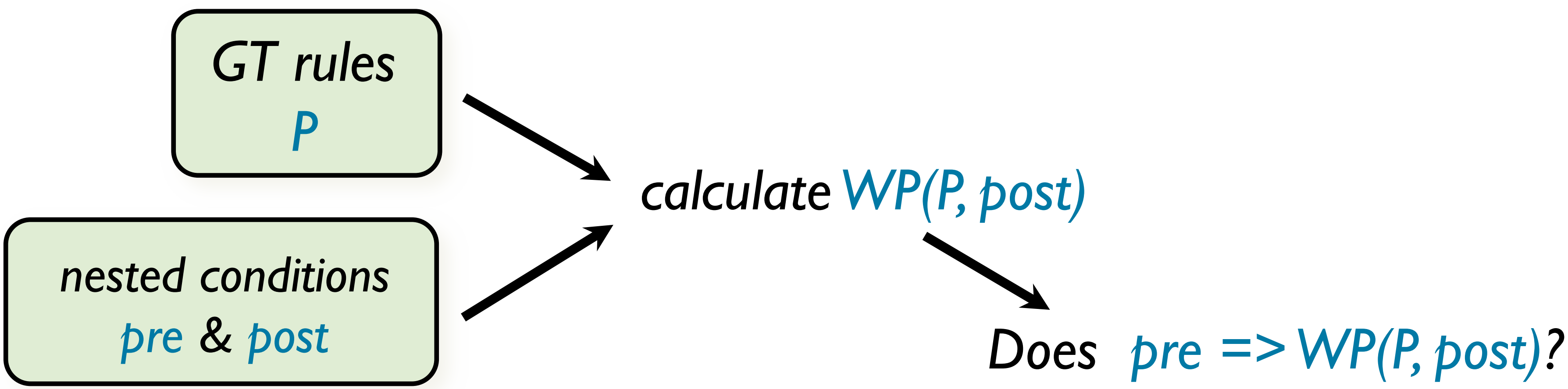
- graph transformation (GT) is a computation abstraction
  => state is represented as a graph
  => computational steps represented as GT rule applications





GT Verification

- functional correctness of GT rules can be verified in a weakest precondition style
- pre- and postconditions are expressed in the graph-based logic of nested conditions, equiv. to FO logic
- roughly, to verify {pre} P {post}:



Rigorous "faking"

- translate the unidirectional transformations to GT rules

  => denoted P_S and P_T

- translate the inter-model constraints to nested conditions

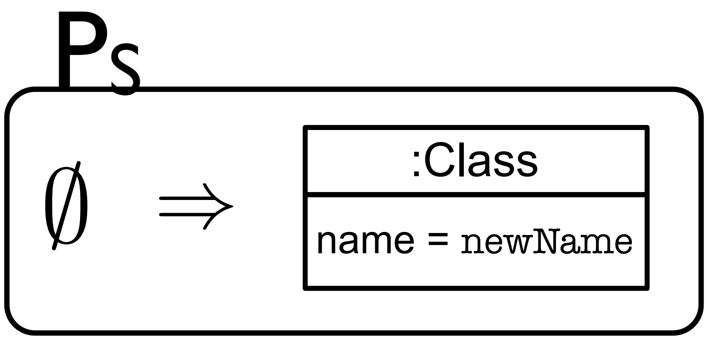
  => denoted evl

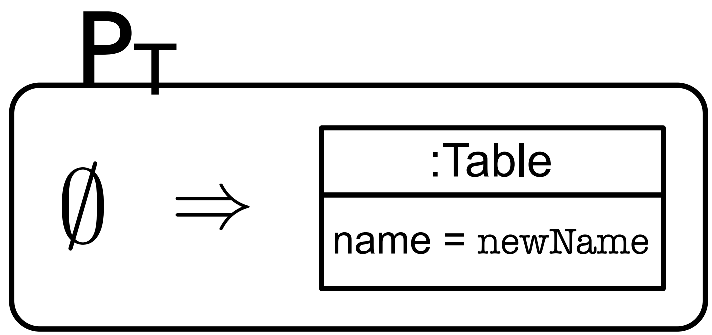
- automatically discharge the following specifications using the weakest precondition calculi

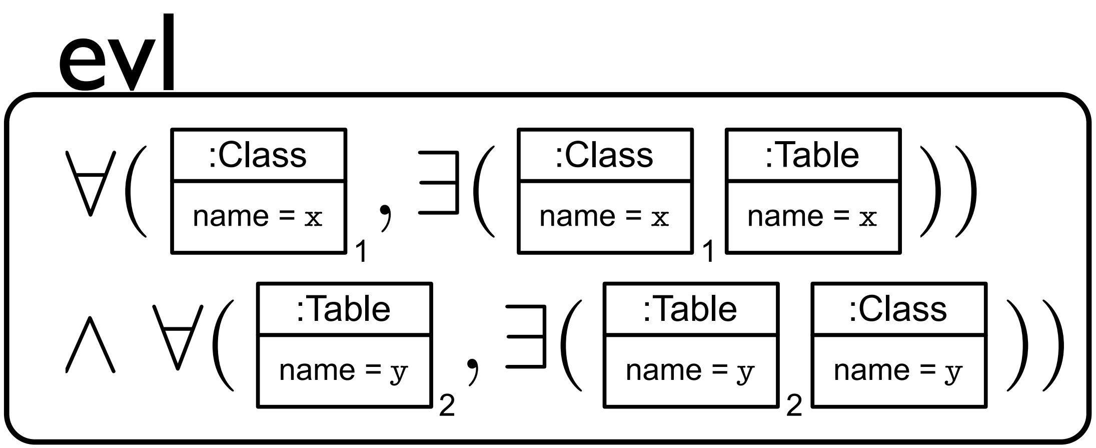
$$\{evl\}\ P_S; P_T\ \{evl\}$$

$$\{evl\}\ P_T; P_S\ \{evl\}$$

Proving consistency of our CD/DB bx

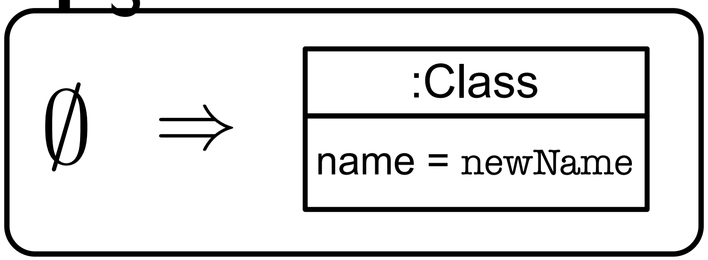




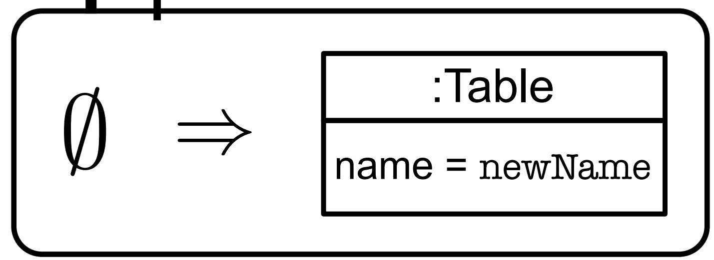


Proving consistency of our CD/DB bx

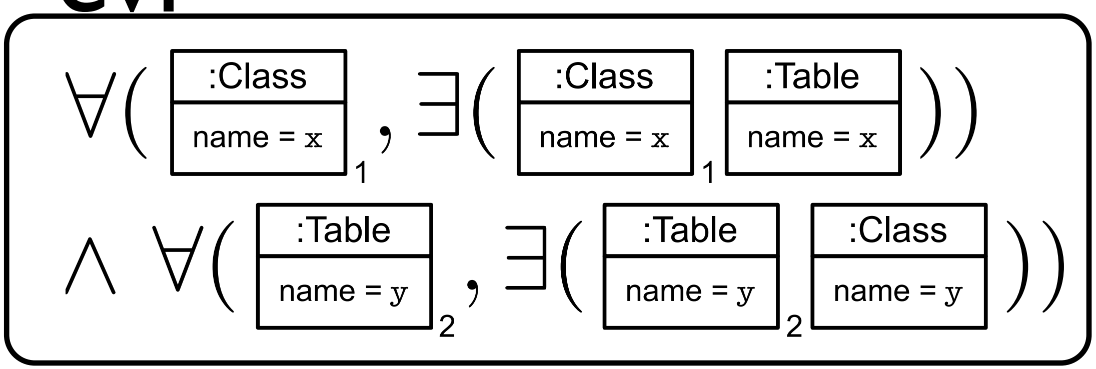
P_S



P_T



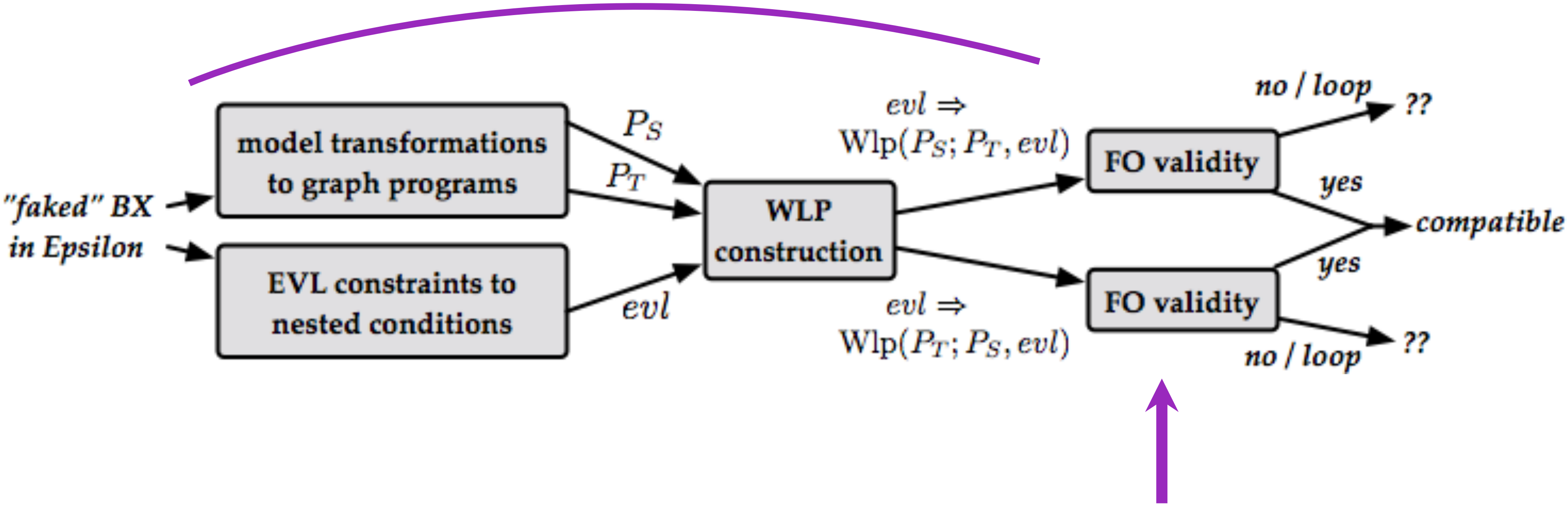
evl



✓ *compatible:* $WP(P_S;P_T,evl) \equiv WP(P_T;P_S,evl) \equiv evl$

Putting it all together

we need to do this bit



exploit existing theorem provers here

Our next steps

- identify a selection of BX case studies ✓
- fake them in Epsilon, manually translate them into GT rules and nested conditions, and verify compatibility ✓
- implement the translations for an expressive subset of the Epsilon languages; implement the WP calculation
- challenges and open questions:
  => finding counterexamples (e.g. using GROOVE)
  => theoretical / practical limitations (e.g. is FO expressive enough?)

Wrap-up

- State of the art in MDE for BX.
- Requirements engineering for BX.
- Architecture and design for BX.
- (A little) Verification of BX.
- What are the future challenges from a SE/MDE perspective?
  - QVT-R: the bugbear.
  - Value proposition of BX versus two unidirectional transformations (Empirical studies! Empirical studies!)
  - When does the requirement for a BX “emerge” in the engineering process? (Work bottom up, top down…)