DSM TP 2012
Theory and Practice

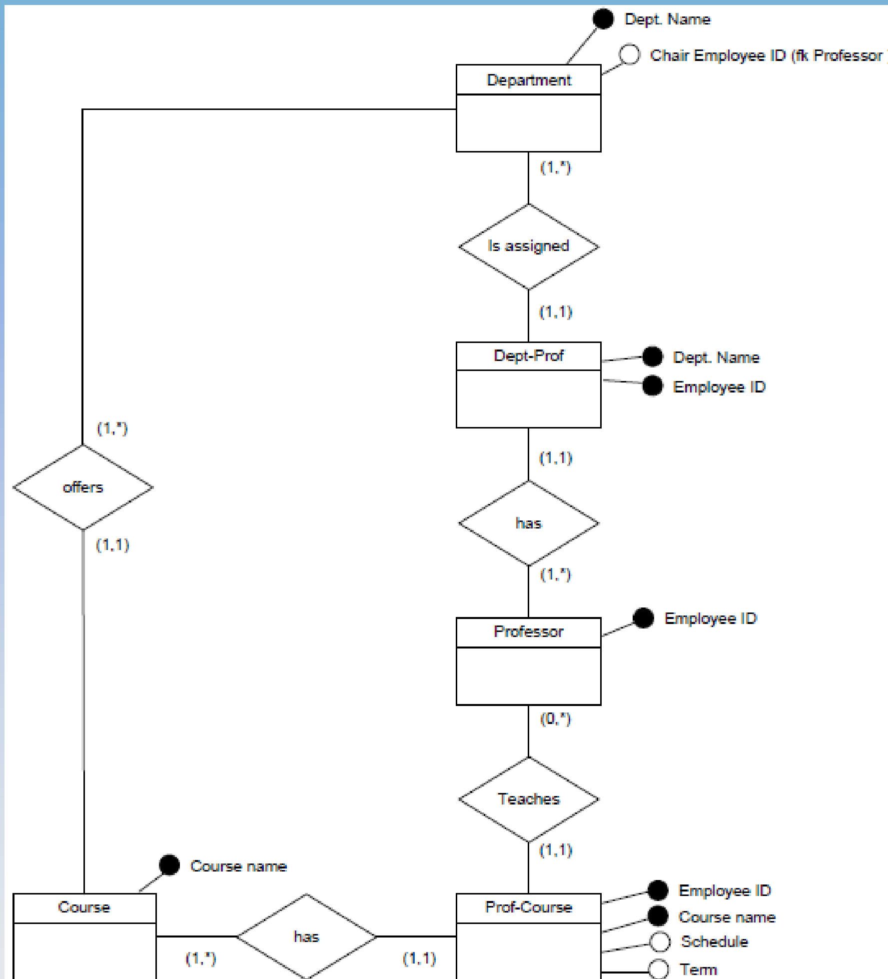
Model Transformation

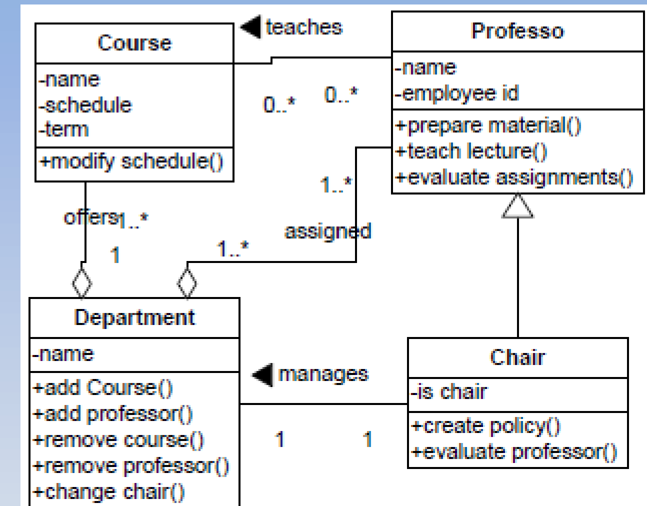
Eugene Syriani

Software Engineering Group
Department of Computer Science
College of Engineering
University of Alabama

MOTIVATION

- Suppose I ask you to provide a software that converts any E-R diagram into a UML class diagram, how would you achieve that?





MOTIVATION

- Suppose I ask you to provide a software that converts any E-R diagram into a UML class diagram, how would you achieve that?
- Assumptions in E-R:
  - Entities & relations can contain attributes
  - Attributes can be of type:
    NUM, CHAR, TIMESTAMP, BIT
  - An entity may have one or more primary attributes
  - Relations relate 1-* or *-* entities
  - IS-A relationship between entities can be used
- Assumptions in UML CD:
  - Classes, associations, attributes, and inheritance can be used
  - Attributes may be of any type
  - OCL constraints may be defined

THE “PROGRAMMING” SOLUTION

- Write a program that takes as input a .ER file and outputs a .UML file (or something similar)
- What are the issues?
  - What if the ER file is a diagram? in XML format? Probably end up limiting input from a specific tool only
  - Similarly in UML, should I output a diagram (in Dia or Visio)? In XMI? In code (Java, C#)?
  - How do I organize my program?
    - Requires knowledge from both domains
    - Need a loader (from input file)
    - Need some kind of visitor to traverse the model, probably graph-like data structure
    - Need to encode a “transformer”
    - Need to develop a UML printer
- Not an easy task after all…

THE “MODELING” WAY

1. Describe a meta-model of ER
   – Define concepts and concrete visual syntax
   – Generate an editor
2. Describe a meta-model of UML (same thing)
3. Define a transformation T: MM_{ER}->MM_{UML}
   – This is done in the form of rules with pre/post-conditions
     ▪ describes “what to transform” instead of “how to transform”

- Code is automatically generated from the trafo model to a trafo instance that produces the result
- Some MT languages give you a bi-directional solution for free!

PROS & CONS

Programing solution

+ Programming techniques are well-proven, it is a reliable solution
— Defined at the level of the code
— Evolution, extension and maintenance more tedious
— More likely to make errors
— Incoherent abstraction mismatch between
  - The in/output artifacts: they represent designs models
  - The transformation between them: which is pure code

PROS & CONS

Modeling solution

+ In/output & trafo models are all defined at the same level of abstraction, in the same domain:
  - No need to add an extra "programmer" resource to the project
+ Much faster solution thanks to rule-based approach & automatic code synthesis
+ Alteration of the transformation process are automatically reflected in the final software product
+ You get a modeling environment for ER & UML for free!
  - No need to read from external non-standard tool anymore

— Young technology, few people understand it & master it, many challenges still need to be solved

PROS & CONS

In practice

- You typically encounter the same problems in the modeling solution as in the programing solution
- The difference is that you can find the problems more easily, fix them very quickly and re-deploy the solution automatically
- Also, it does not require the developer to be a computer scientist or a software engineer. The person who defines the requirements can develop the solution as well
- The bottom line is that you save time, reduce the cost, fulfill the entire scope and deliver a high-quality product

SO WHAT ARE WE DOING HERE?

- It seems that Model-based Design is the “Holy Grail” of software engineering
- Well, the devil is in the details…
- We will explore
  - The techniques that I mentioned
  - Identify some of the remaining hot challenges in MDE
  - Solve some of these challenges

What is Model Transformation?

MODELS ARE EVERYWHERE

- How to **modify** them in a safe, structured way?
- How to establish logical relations, **mapping** between them?
- How to explicitly specify their **semantics**?
- How to **generate code** from them?

- In fact, how can we **manipulate** them?

- Model Transformation is a sub-field of MDE, responsible for bringing your models to life

SOME DEFINITIONS

"The process of converting one model to another model of the same system."

OMG 2003

"The automatic generation of a target model from a source model, according to a transformation definition."

Kleppe 2003

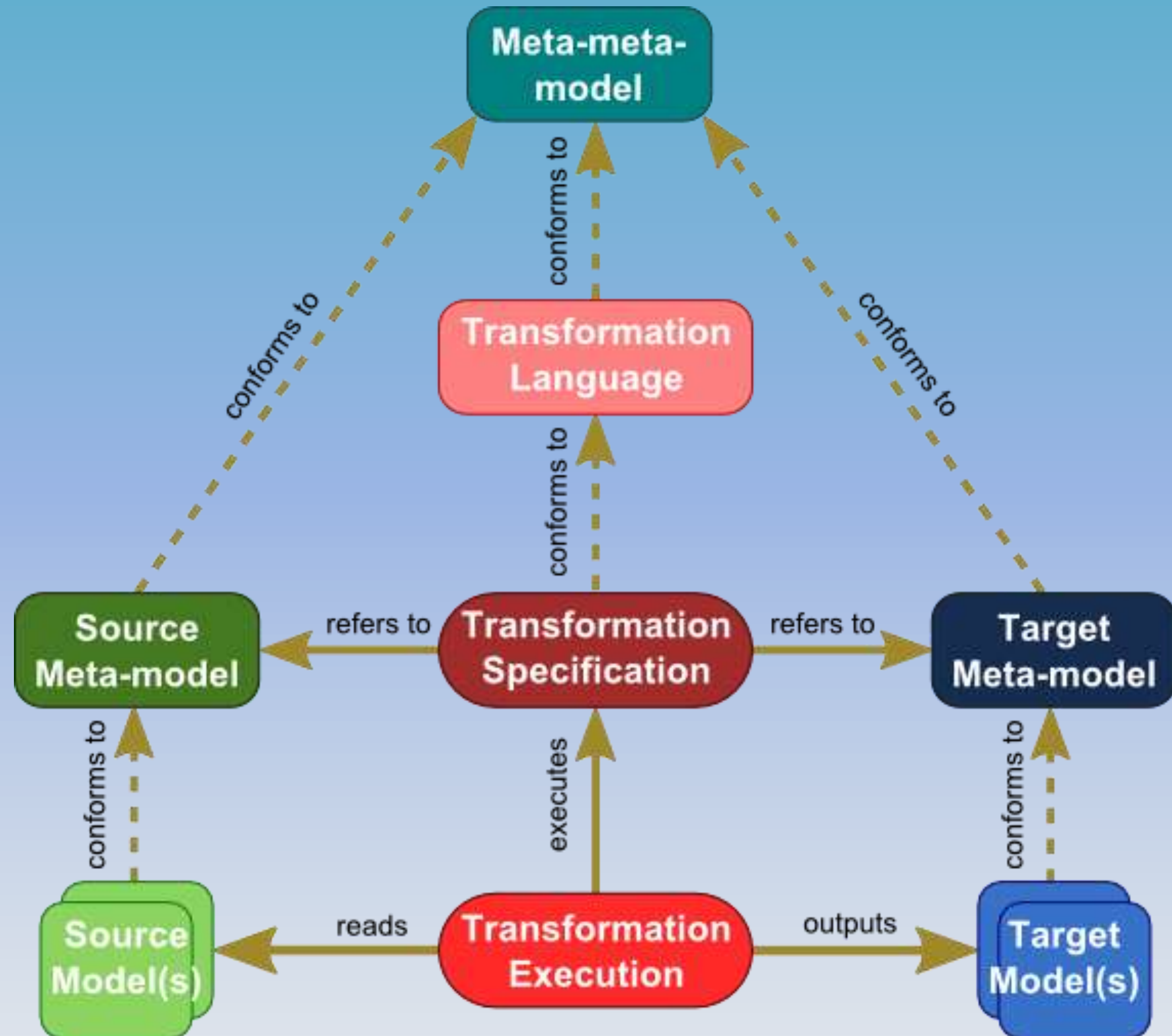
"The automatic manipulation of a model with a specific intention."

Syriani 2011

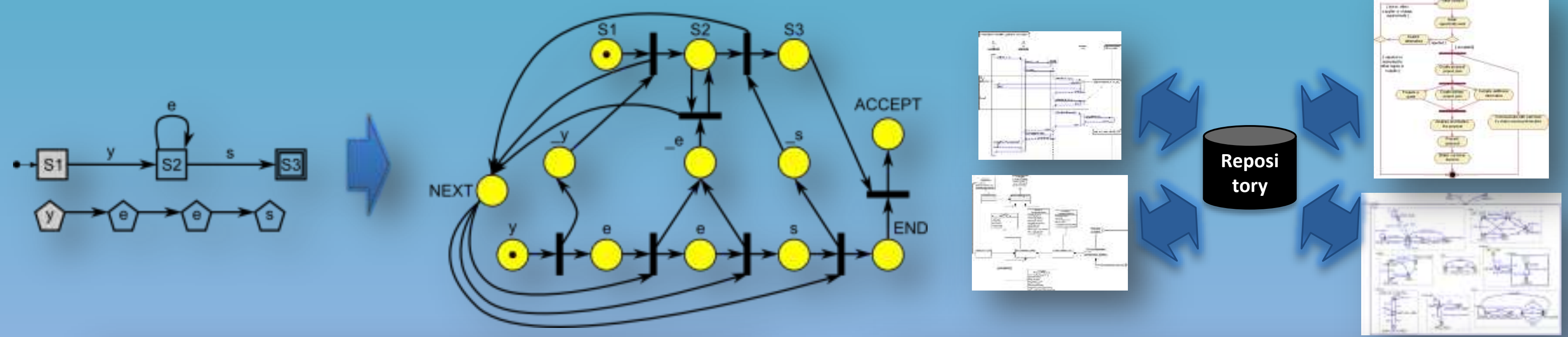
TERMINOLOGY

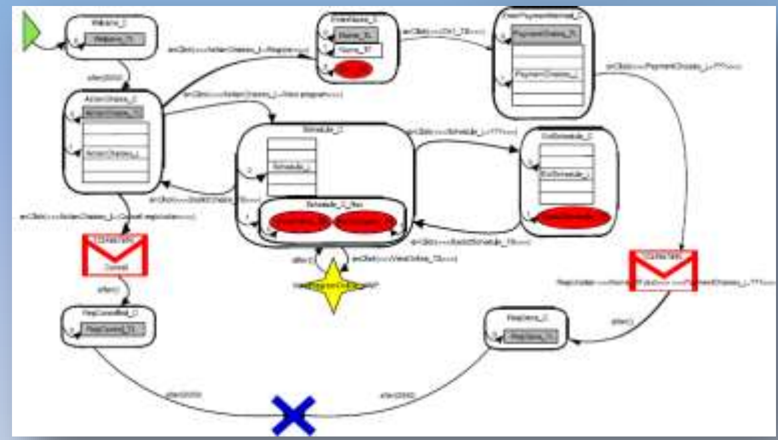
Model-to-model transformation

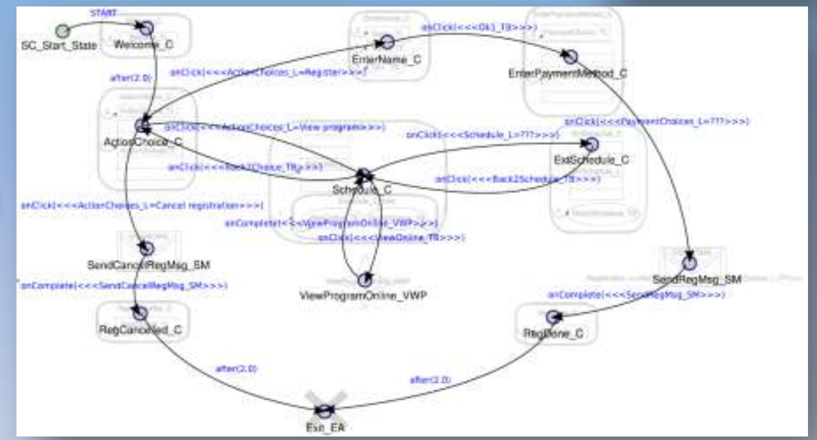
- Transformation defined at the meta-model level
- Execution of transformation is applied on the models to automatically transform them

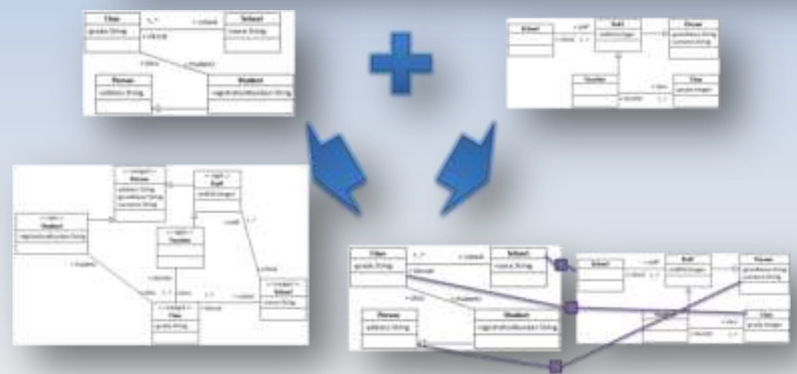


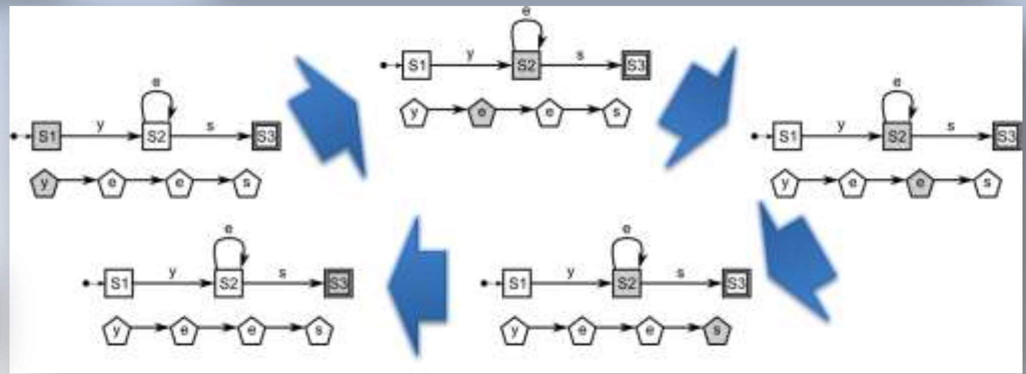
TRANSFORMATIONS





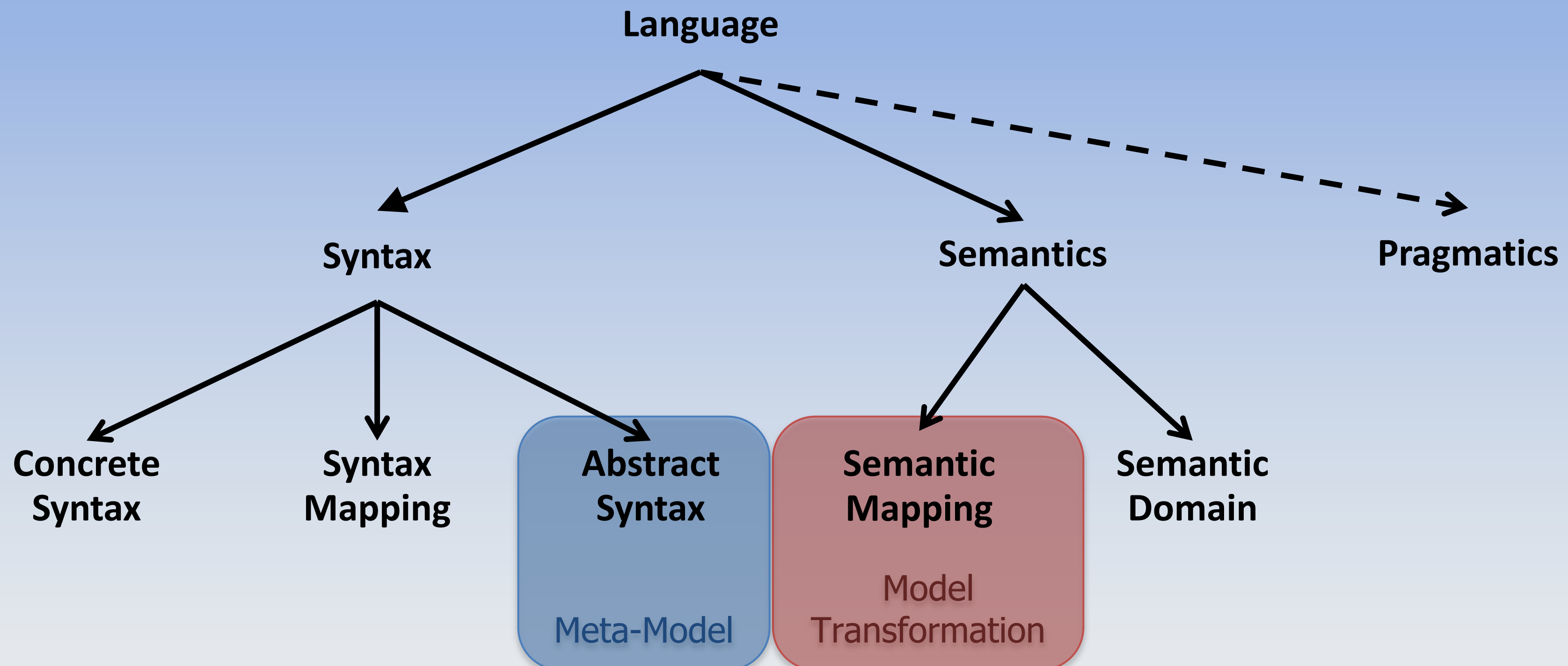






SOFTWARE LANGUAGE ENGINEERING P.O.V.

- The **meta-model** of a language L defines:
  - The abstract syntax of L
  - The static semantics of L
- A transformation defined on L can define the **dynamic semantics** of L: how model instances behave



TYPICAL USES OF MODEL TRANSFORMATION

Manipulation

- A model transformation performs a **manipulation** on a model.
- Simple operations on a model:
  - **Add** an element to the model;
  - **Remove** an element from the model;
  - **Update** an element's properties;
  - **Access** an element or its properties.
- These primitive operations are know as the **CRUD operations** (Create, Read, Update, Delete)

Amrani, M.; Dingel, J.; Lambers, L.; Lúcio, L.; Salay, R.; Selim, G.; Syriani, E. & Wimmer, M. Towards a Model Transformation Intent Catalog. MoDELS Workshop on Analysis of Model Transformation, ACM, 2012

TYPICAL USES OF MODEL TRANSFORMATION

Query

A query is still a transformation

- What is a query?
  - A query is an operation that requests some information about a system.
  - This operation takes as input the model M and outputs a view of M.
  - A view is a projection of (a sub-set of) of M.
- **Restrictive view**: Reveal a proper subset of M (all, none, some)
  - Retrieve all cycles in a causal block diagram
  - Show only classes/associations of a class diagram
- **Aggregated view**: Restriction of M modifying some of its properties
  - Get the average of all costs per catalogue product in a relational database schema
  - In a hierarchical model, show top-level elements only, with an extra attribute denoting the number of sub-elements

QUESTION

Is a query a transformation? Why?

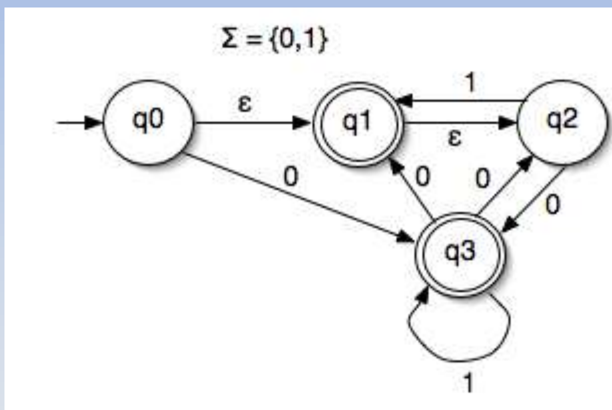
- **It is a projection, obtained by CRUD operations on the properties of M.**

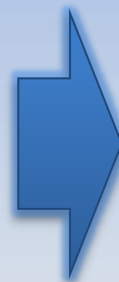
TYPICAL USES OF MODEL TRANSFORMATION

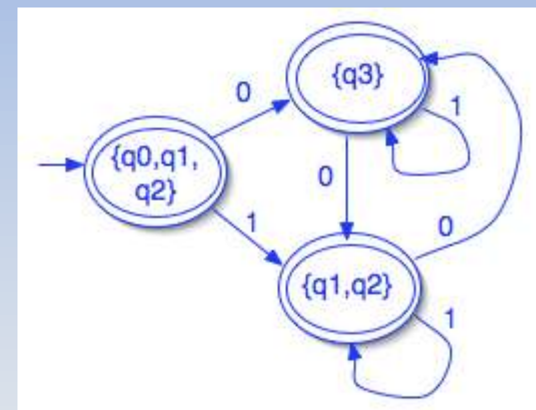
Refinement

- **Transform from a higher level specification to a lower level description**
  - **M1 refines M2 if M1 can answer all questions that M2 can**

NFA to DFA







Non-deterministic state automata (NFA)

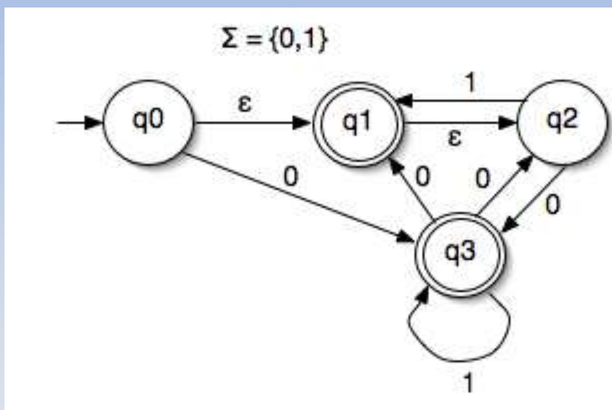
Deterministic state automata (DFA)

TYPICAL USES OF MODEL TRANSFORMATION

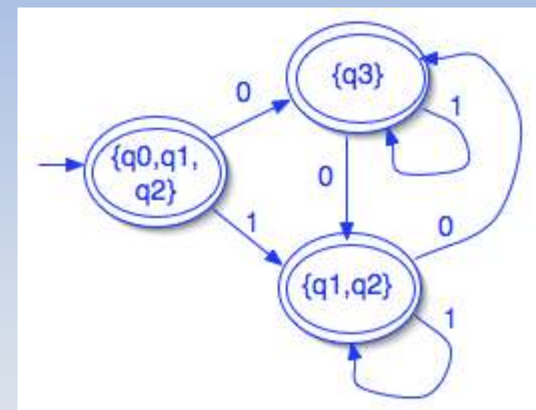
Abstraction

- Inverse of refinement
  - M1 refines M2 then M2 is an abstraction of M1

DFA to NFA



Non-deterministic state automata (NFA)



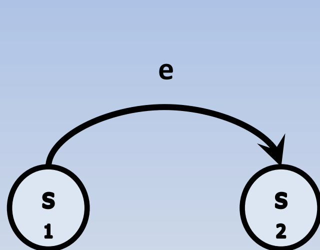
Deterministic state automata (DFA)

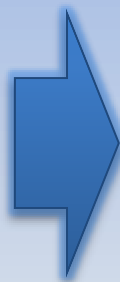
TYPICAL USES OF MODEL TRANSFORMATION

Synthesis

- Model is synthesized into a well-defined language format that can be stored, such as in **serialization**
- **Model-to-code generation**
  - Case where the target language is source code in a programming language

Statecharts to Python Compiler





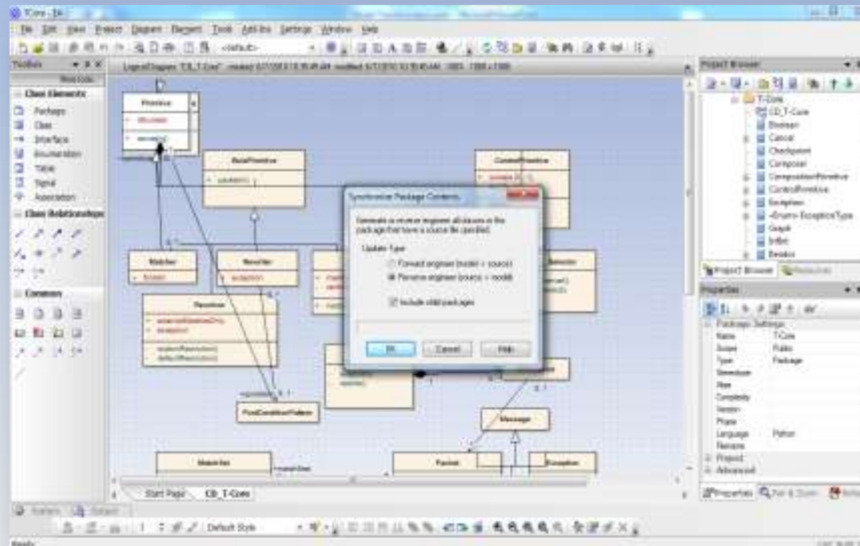
```
if e == 0:    # event "e"
    if table[1] and self.isInState(1) and self.testCondition(3):
    if (scheduler == self or scheduler == None) and table[1]:
        self.runActionCode(4)             # output action(s1)
        self.runExitActionsForStates(-1)
        self.clearEnteredStates()
        self.changeState(1, 0)
        self.runEnterActionsForStates(self.StatesEntered, 1)
        self.applyMask(DigitalWatchStatechart.OrthogonalTable[1], table)
        handled = 1
    if table[0] and self.isInState(0) and self.testCondition(4):
        if (scheduler == self or scheduler == None) and table[0]:
            self.runActionCode(5)         # output action(s2)
            self.runExitActionsForStates(-1)
            self.clearEnteredStates()
            self.changeState(0, 0)
            self.runEnterActionsForStates(self.StatesEntered, 1)
            self.applyMask(DigitalWatchStatechart.OrthogonalTable[0], table)
            handled = 1
```

Statecharts model　　**Generated Python code**

TYPICAL USES OF MODEL TRANSFORMATION

Reverse Engineering

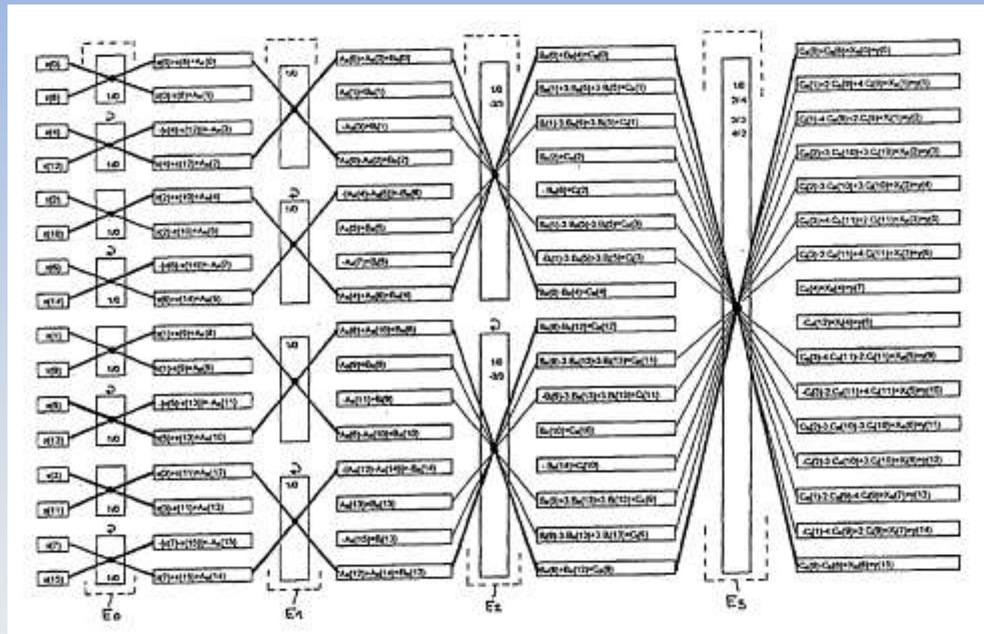
- Inverse of synthesis: extracts higher level specifications from lower level ones.
  - UML class diagrams can be generated from Java with Fujaba
- If the same model transformation T synthesizes M1 into M2 and reverse engineers M2 to M1, then T is said to be a **bi-directional transformation**.



TYPICAL USES OF MODEL TRANSFORMATION

Approximation

- **Refinement with respect to negated properties**
  - **M1 approximates M2 if M1 negates the answer to all questions that M1 negates**
- **In practice, M2 is an idealization of M1 where an approximation is only extremely likely**

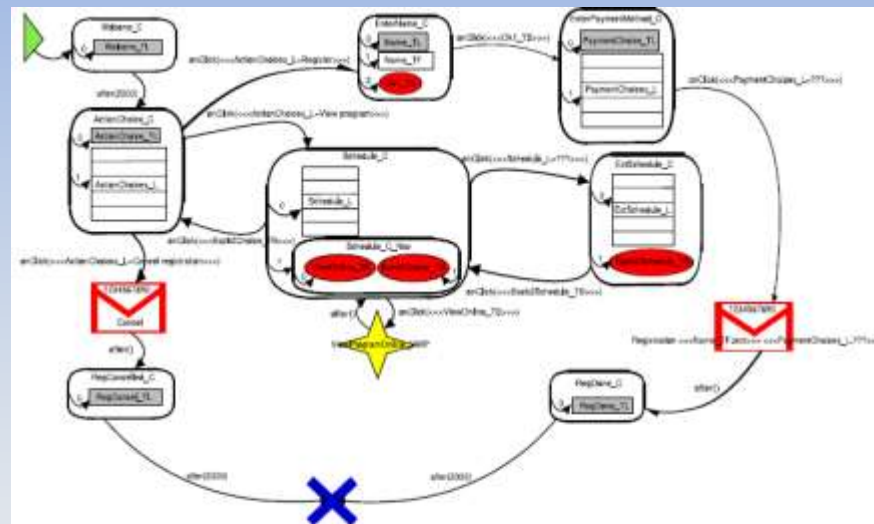


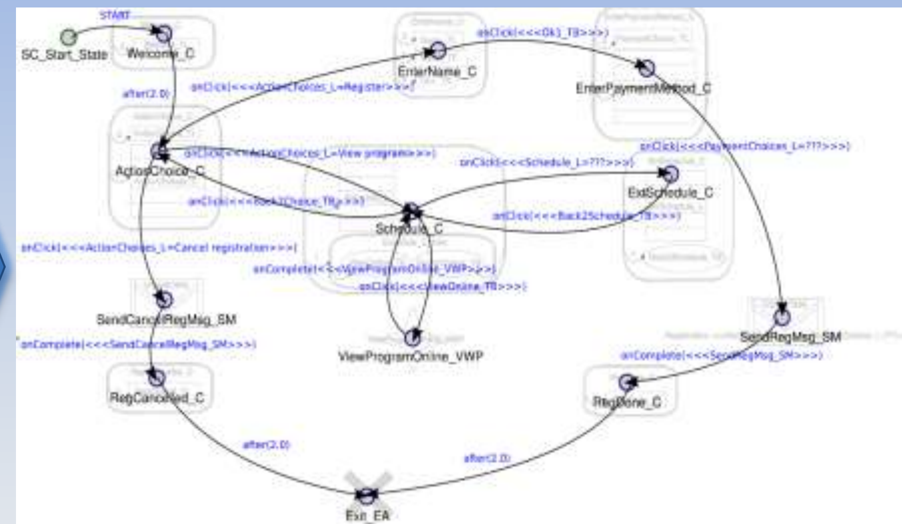
TYPICAL USES OF MODEL TRANSFORMATION

Translational semantics

- The **semantics** of the source formalism is given in terms of the semantics of the target formalism.
- Semantic mapping function of the original language defined by a MT that translates any of its instances to a valid instance of the reference formalism with well-defined semantics.
- **Inter-formalism transformation** (a.k.a. m2m transformation)

PhoneApps To Statecharts





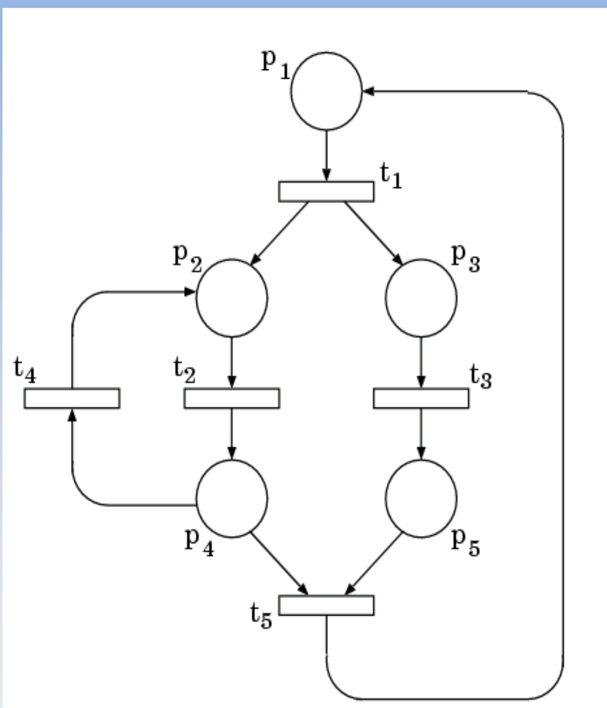
PhoneApps DSM of a conference registration mobile application

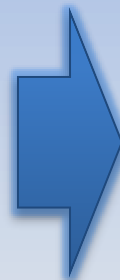
Representation of the model in Statecharts

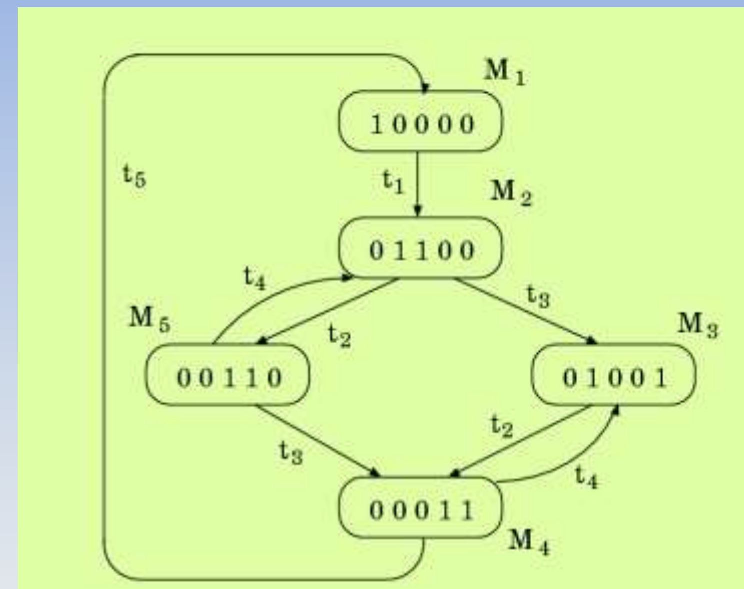
TYPICAL USES OF MODEL TRANSFORMATION

Analysis

- Map a modeling language to a formalism that can be analyzed more appropriately than the original language
  - The target language is typically a formal language with known analysis techniques





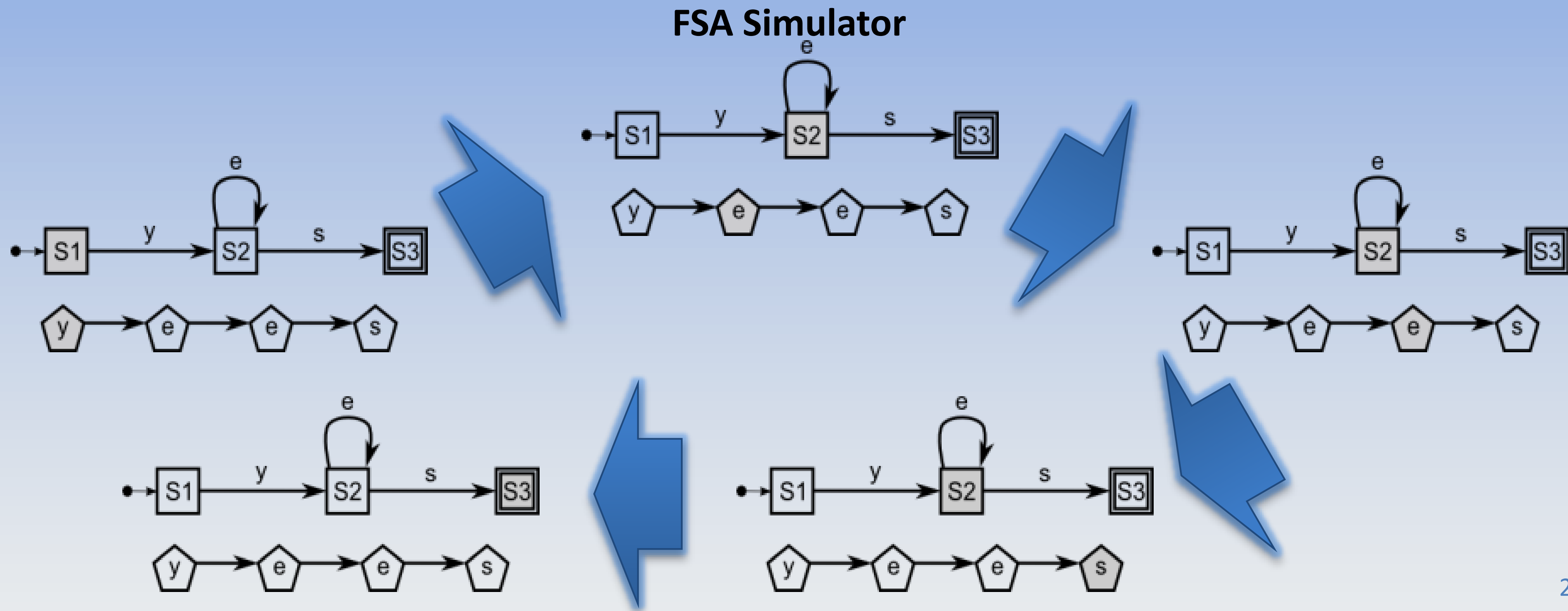


TYPICAL USES OF MODEL TRANSFORMATION

Operational Semantics – Simulation

- **Update the state of the model**
- **In this case, the source and target meta-models are identical.**
- **Moreover, the target model is an "updated" version of the source model: no new model is created**

FSA Simulator



TYPICAL USES OF MODEL TRANSFORMATION

Relation between Abstract and Concrete syntax

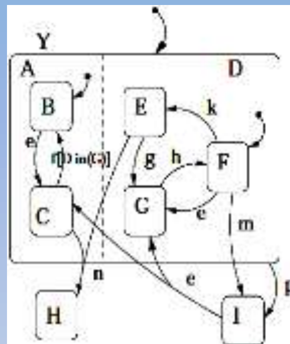
Model transformation can be used to specify mappings within the language too. It can be used only if both the abstract and concrete syntax are themselves modelled.

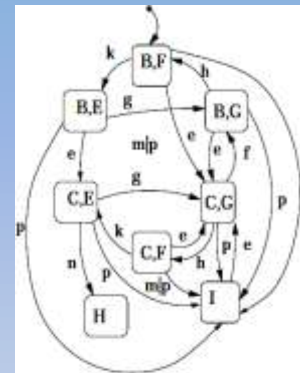
- **Rendering**
  - Mapping from the abstract syntax to possibly several concrete representations (textual, graphical, ...)
  - 1 abstract syntax to many concrete syntaxes
- **Parser**
  - Mapping from the concrete syntax to the corresponding abstract syntax (graph)
  - 1 concrete syntax to 1 abstract syntax

TYPICAL USES OF MODEL TRANSFORMATION

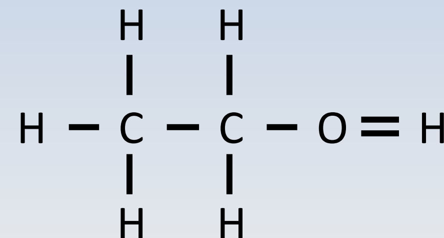
Normalization

- Decrease syntactic complexity
  - Translate complex language constructs into more primitive language constructs





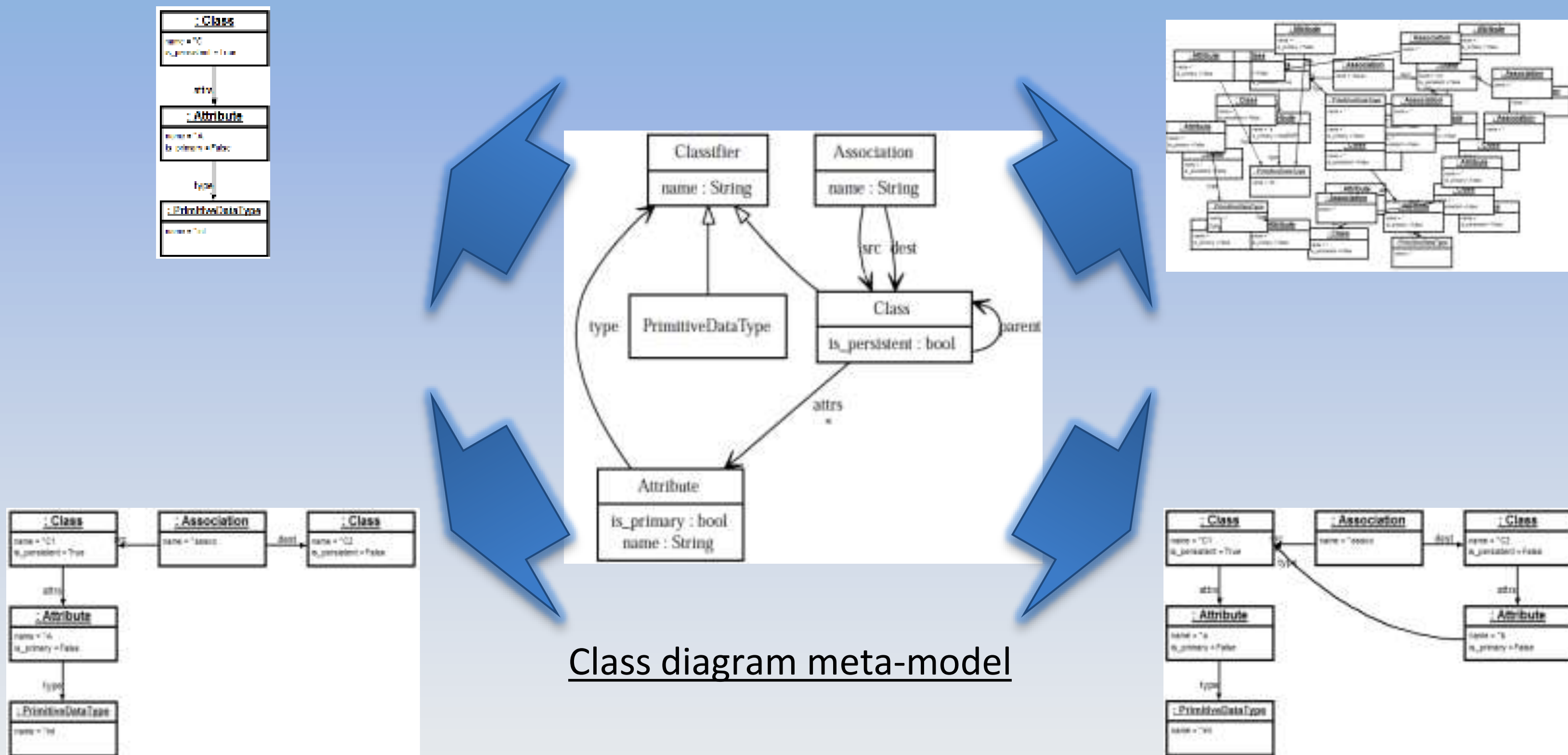
  - Transform all uses of a language construct in a normal or canonical form



TYPICAL USES OF MODEL TRANSFORMATION

Meta-model instance generation

- Automatically generate random models that conform to the language
- This is very useful, especially for model-based testing

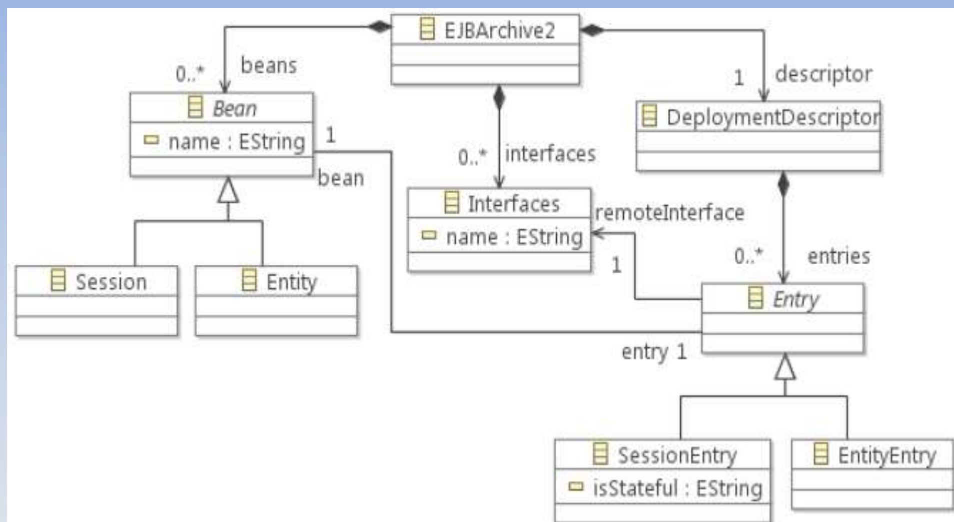


Class diagram meta-model

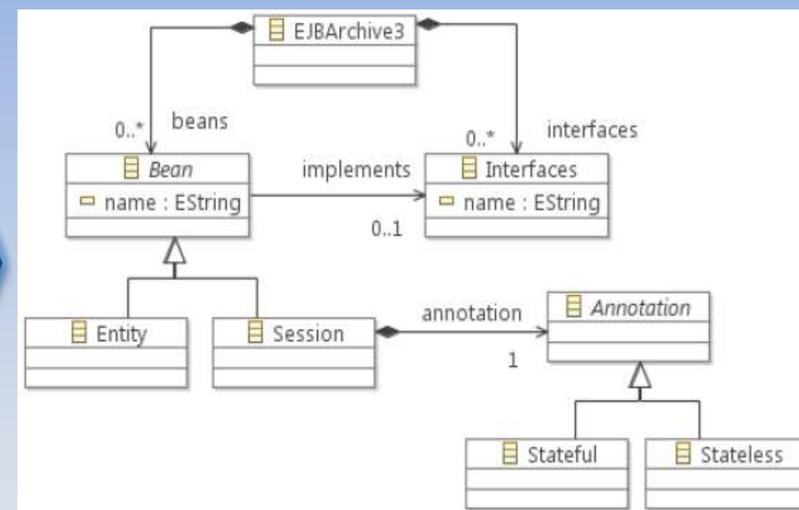
TYPICAL USES OF MODEL TRANSFORMATION

Migration

- **Transform from a software model written in one language or framework into another, but keeping the same level of abstraction**
- **Evolution to new version**



Enterprise Java Beans 2.0



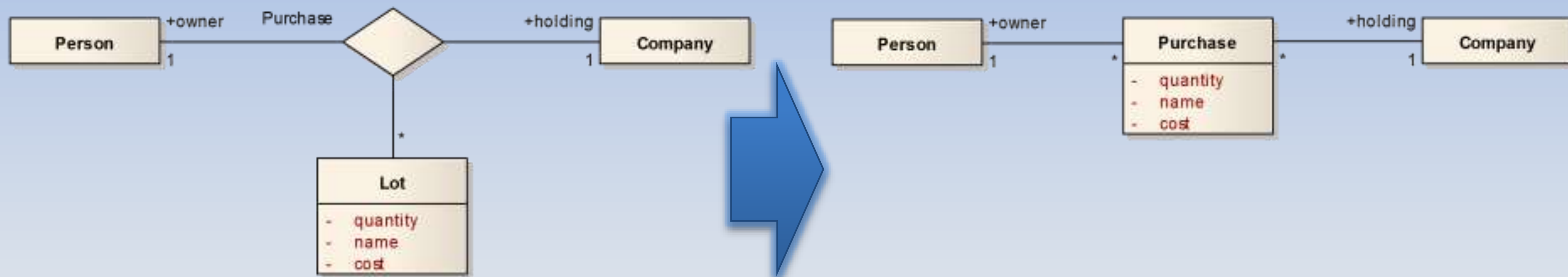
Enterprise Java Beans 3.0

TYPICAL USES OF MODEL TRANSFORMATION

Optimization

- Improve certain operational qualities of the model while preserving its semantics
- Typically used on architecture or design models

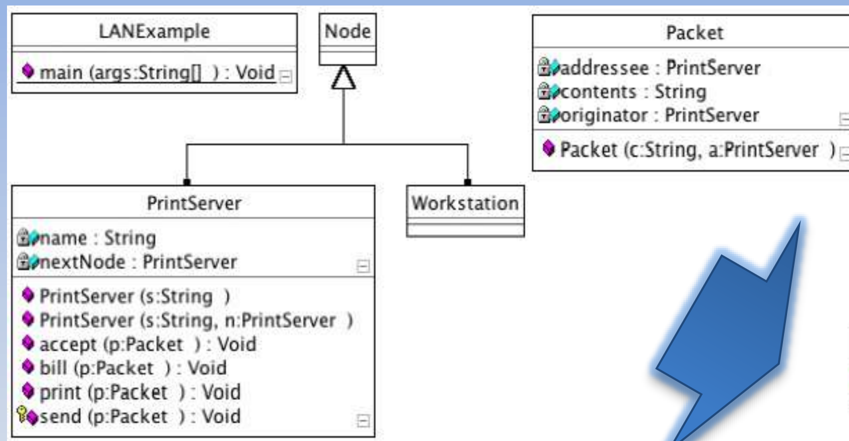
N-ary to binary association



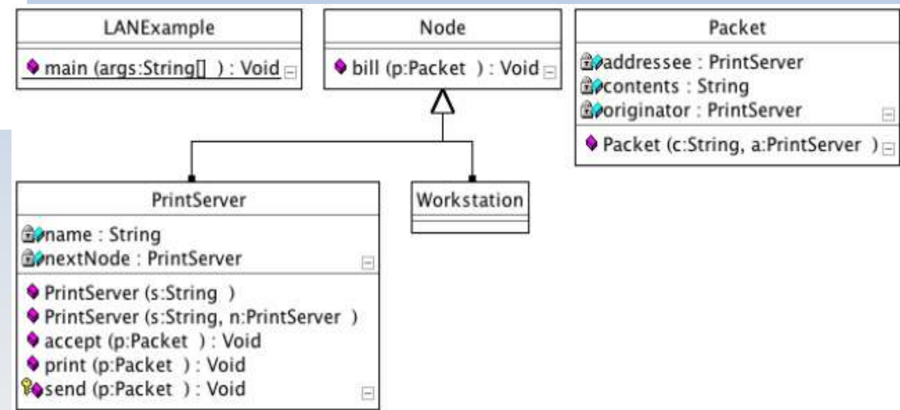
TYPICAL USES OF MODEL TRANSFORMATION

Refactoring

- Change the internal structure of the model to improve certain quality characteristics without changing its observable behaviour
  - Understandability, modifiability, reusability, modularity, adaptability



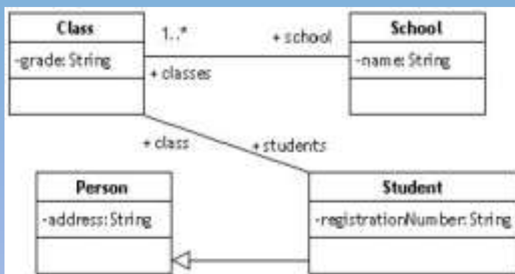
Pull up method refactoring

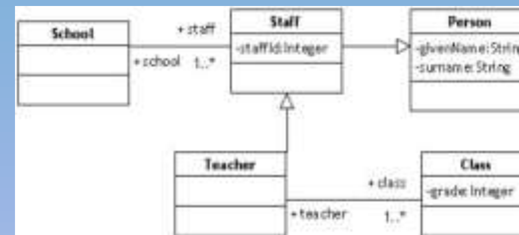


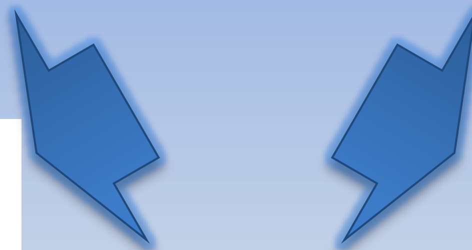
TYPICAL USES OF MODEL TRANSFORMATION

Composition

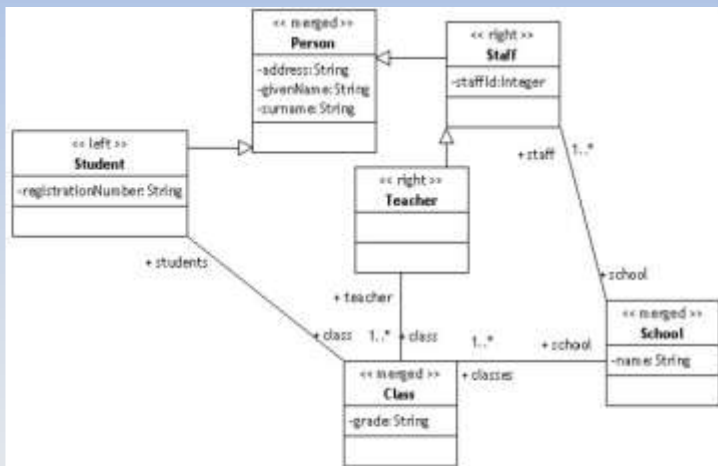
- Integrate models that have been produced in isolation into a compound model



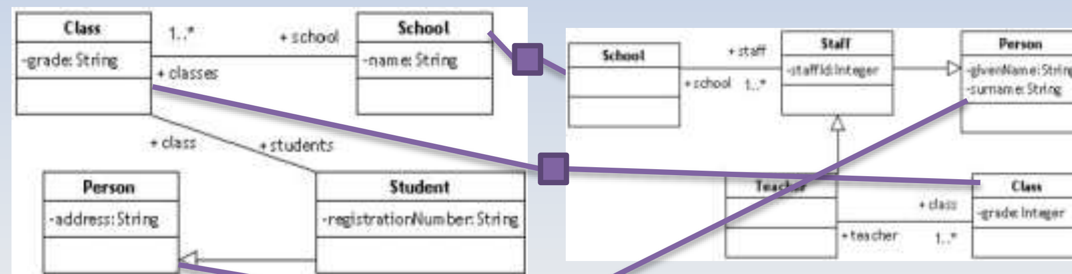




Model merging



Model weaving

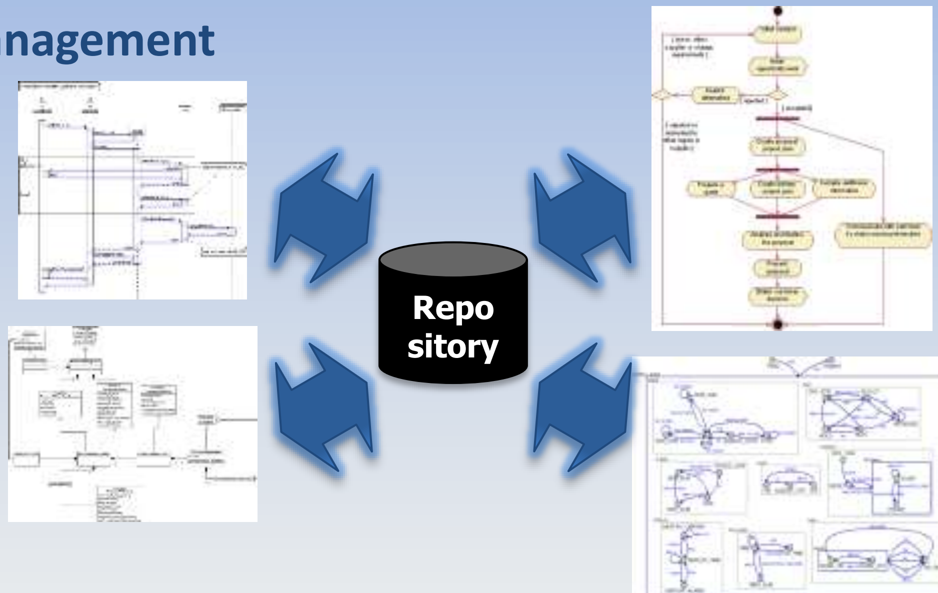


TYPICAL USES OF MODEL TRANSFORMATION

Synchronization

Integrate models that have evolved in isolation but that are subject to global consistency constraints

- In contrast with composition, synchronization requires that changes are propagated to the models that are being integrated
- Source model changes are propagated to corresponding target model changes: **Incremental / Change-driven** transformation.
- Synchronization must be ensured in both directions: **multi-directional** transformation.
- **Inconsistency** management



VOCABULARY

- Relationship between source & target meta-models
  - **Endogenous**: Source meta-model = Target meta-model
  - **Exogenous**: Source meta-model ≠ Target meta-model
- Relationship between source & target models
  - **In-place**: Transformation executed within the same model
  - **Out-place**: Transformation produces a different model

	Endogenous	Exogenous	Either
In-place	Manipulation, Simulation	X	X
Out-place	Restrictive query	Aggregate query, Synthesis, Reverse engineering, Migration	Normalization, Composition, Synchronization
Either	Optimization, Refactoring	X	X

VOCABULARY

Abstraction level

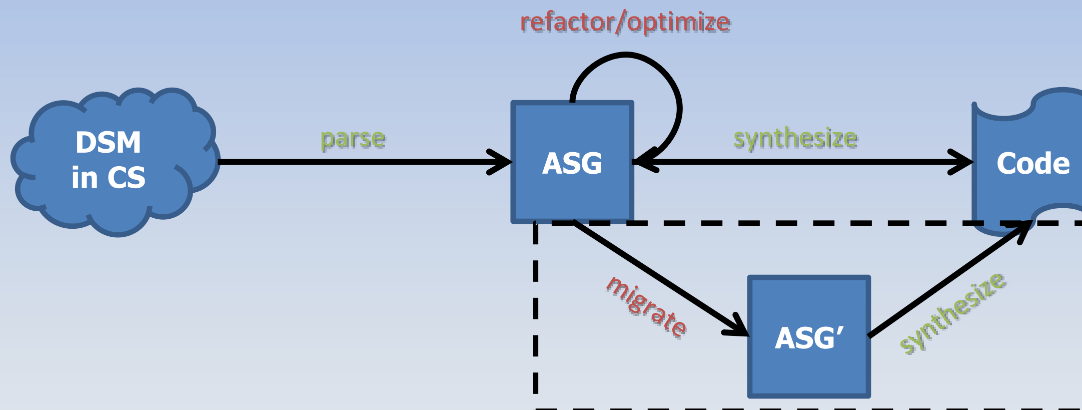
- **Horizontal**: source and target models reside at the same abstraction level
- **Vertical**: source and target models reside at different abstraction levels

	Endogenous	Exogenous	Either
Horizontal	Manipulation, Simulation, API migration	Language migration	Composition
Vertical	Refinement, Refactoring, Restrictive query, Optimization, Normalization	Aggregate query, Synthesis, Reverse engineering, Desugaring	X

VOCABULARY

- Syntactical vs. Semantical Transformations
  - A **syntactical** transformation solely modifies the representation of the model
  - In a **semantical** transformation, the output model has a different meaning than the input model, although the representation of the latter may or may not have been modified.

Model transformation chain to compile a DSM into executable Java code



QUESTION

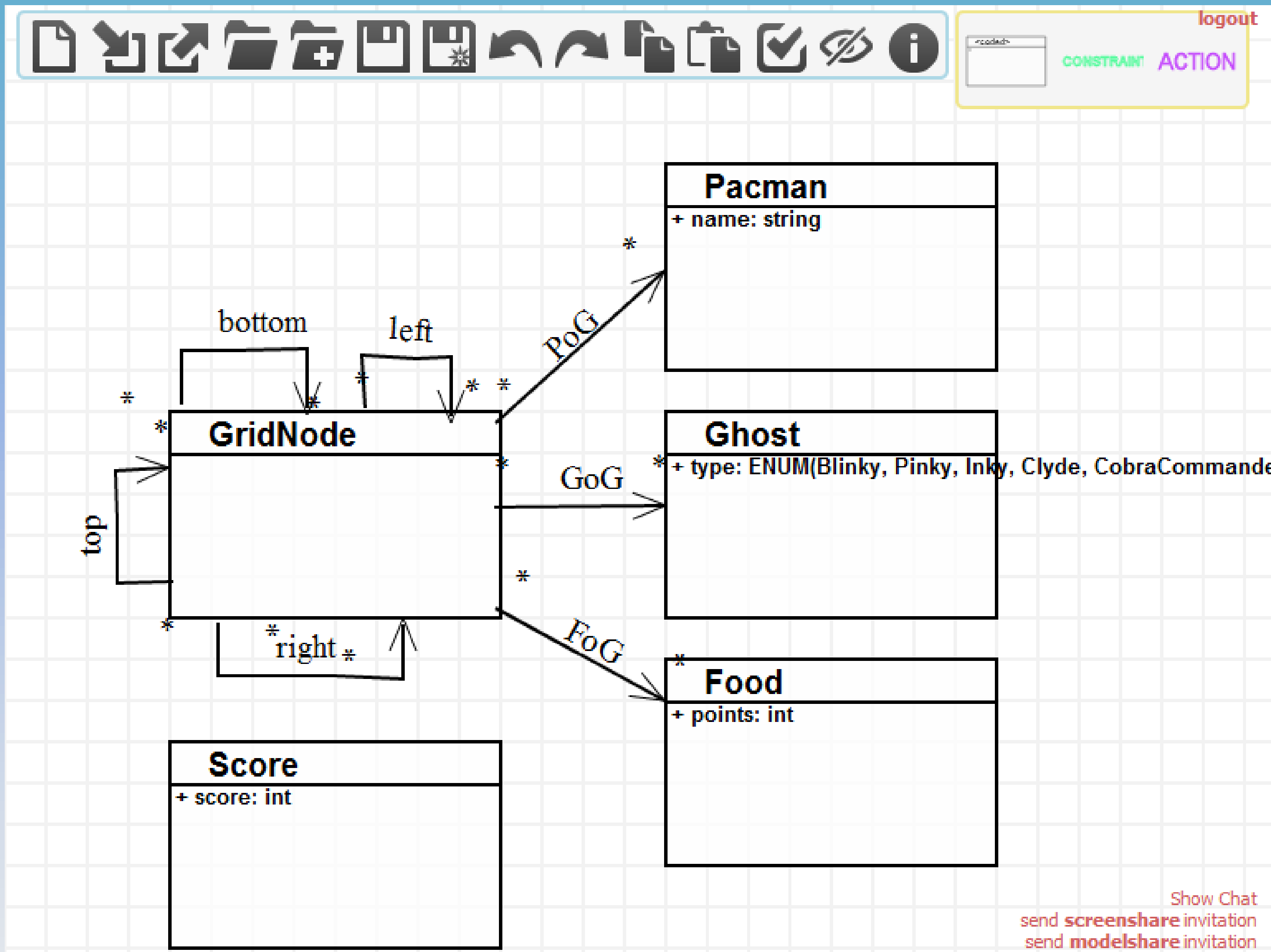
Which transformation intent is syntactical and which is semantical?

Manipulation, Query, Synthesis, Abstraction, Refinement, Approximation, Translation, Analysis, Simulation, Rendering, Parsing, Normalization, Model Generation, Migration, Optimization, Refactoring, Composition, Synchronization

- **Syntactical: Query, Synthesis, Rendering, Parsing, Normalization, Model Generation**
- **Semantical: Manipulation, Abstraction, Refinement, Approximation, Translation, Analysis, Simulation, Migration, Optimization, Refactoring, Composition, Synchronization**

Model Transformation Example

STATIC SEMANTICS (META-MODEL)



de Lara, J. & Vangheluwe, H. AToM3: A Tool for Multi-formalism and Meta-Modelling. FASE'02, LNCS: 2306, pp. 174-188, Springer-Verlag 2002.

CONCRETE SYNTAX

logout

Pacmanicon

Ghosticon

GridNodeIcon

Foodicon

Score

Scoreboardicon

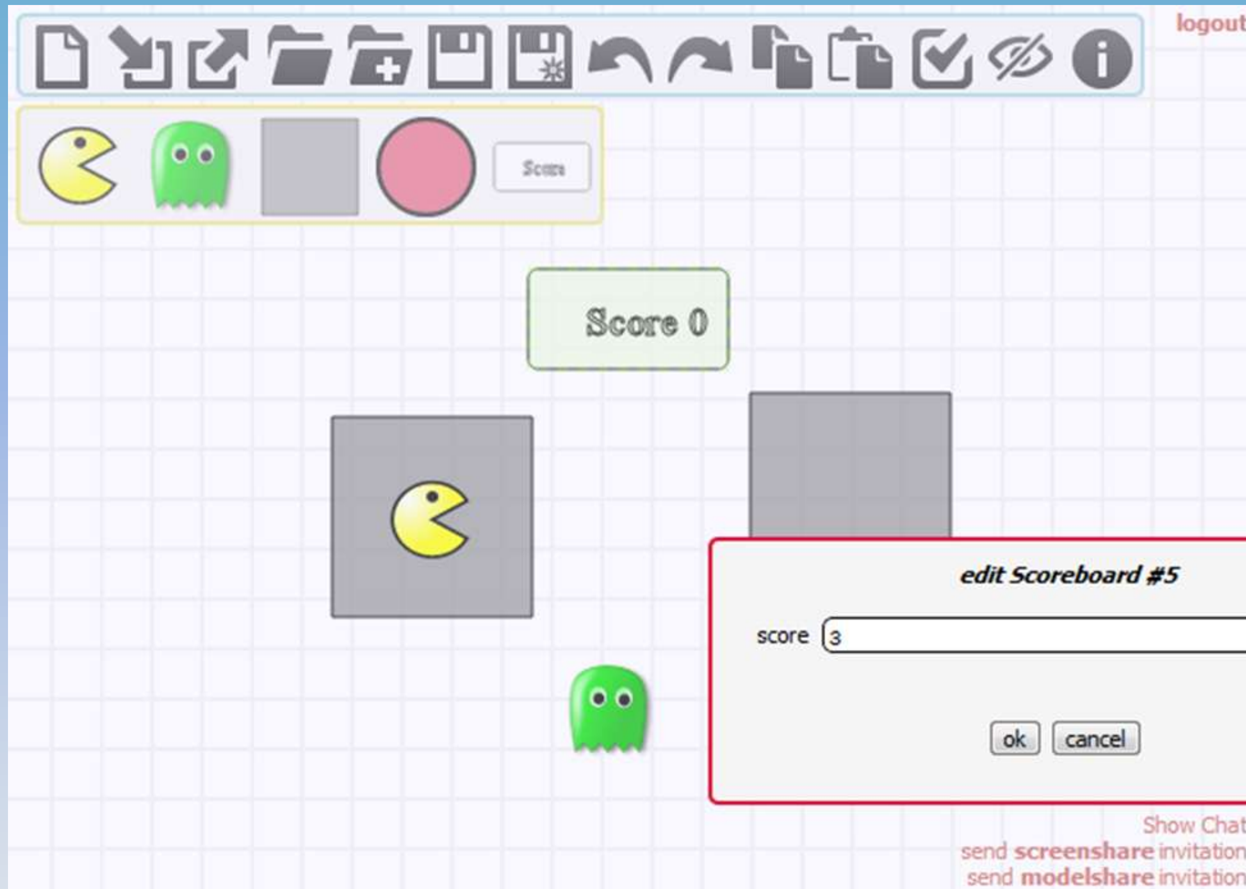
PoGLink

FoGLink

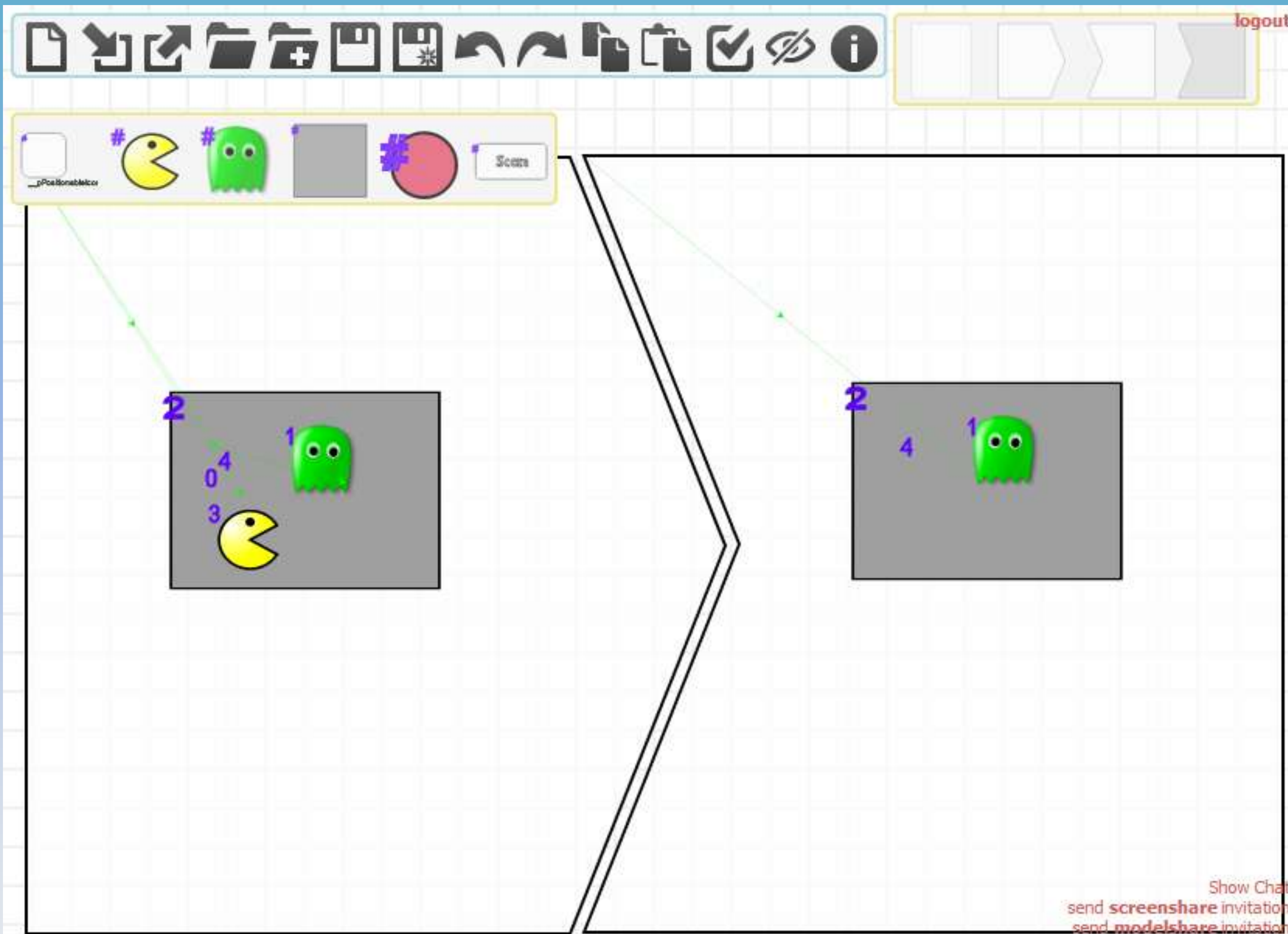
GoGLink

Show Chat
send screenshare invitation
send modelshare invitation

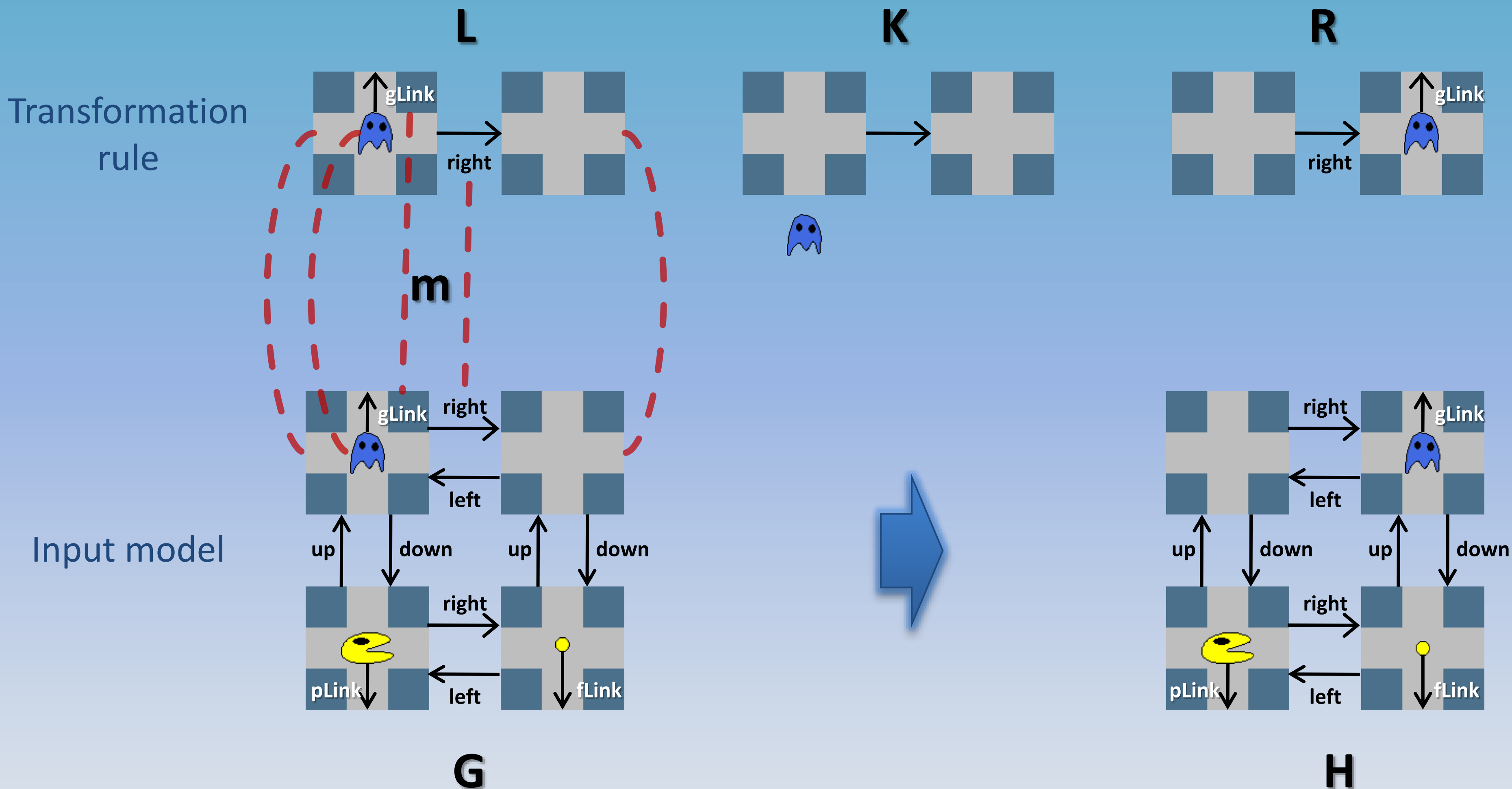
GENERATE MODELING ENVIRONMENT



GRAPH TRANSFORMATION TO SPECIFY SEMANTICS OF LANGUAGE

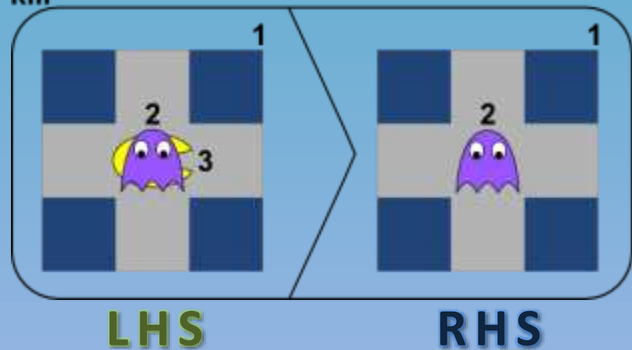


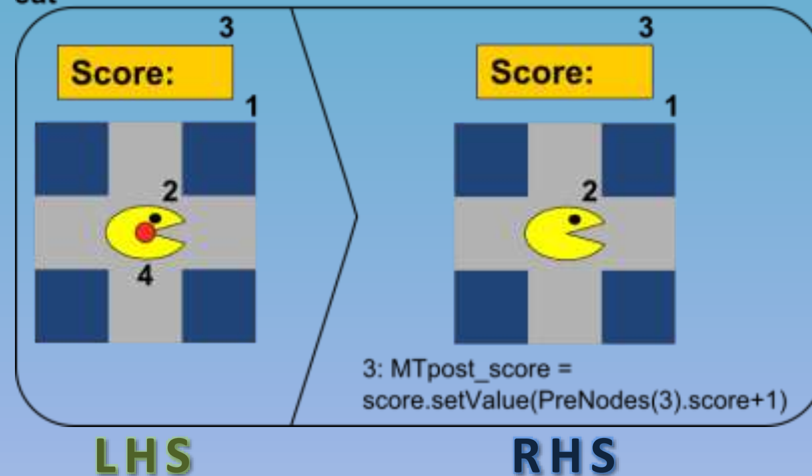
RULE-BASED MODEL TRANSFORMATION

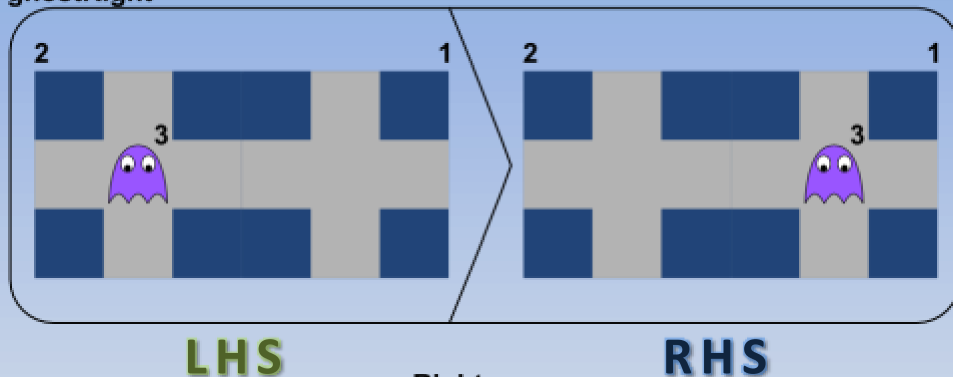


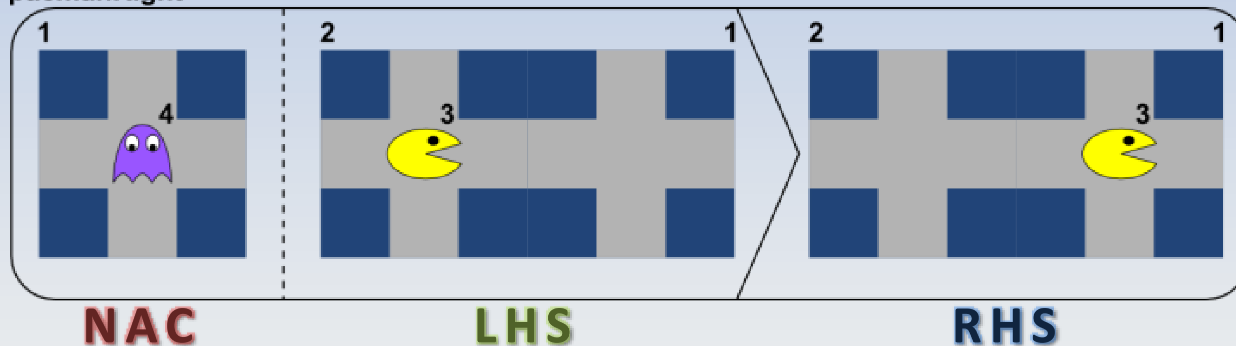
If there exists an occurrence of L in G then replace it with R

OPERATIONAL SEMANTICS



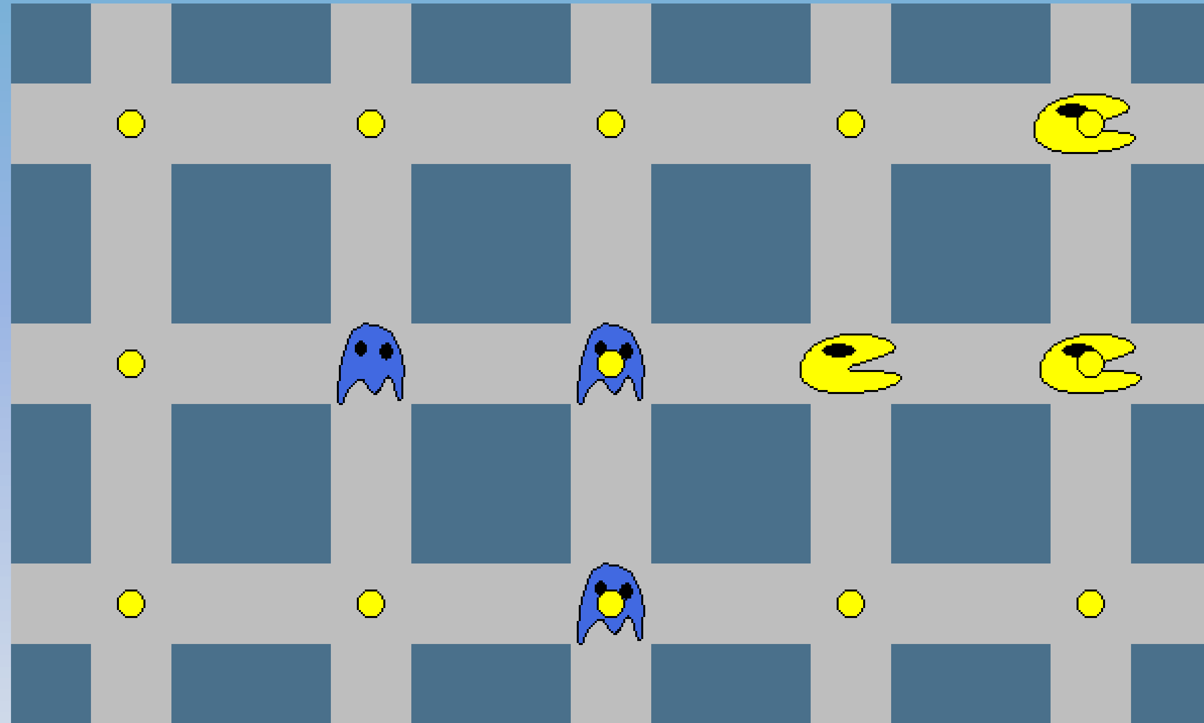






SIMULATION OF A MODEL

P1	1. pacmanDie
P2	2. pacmanEat
P3	3. ghostMoveLeft
	4. ghostMoveRight
	5. ghostMoveUp
	6. ghostMoveDown
	7. pacmanMoveLeft
	8. pacmanMoveRight
	9. pacmanMoveUp
	10. pacmanMoveDown



MODEL TRANSFORMATION DEVELOPMENT

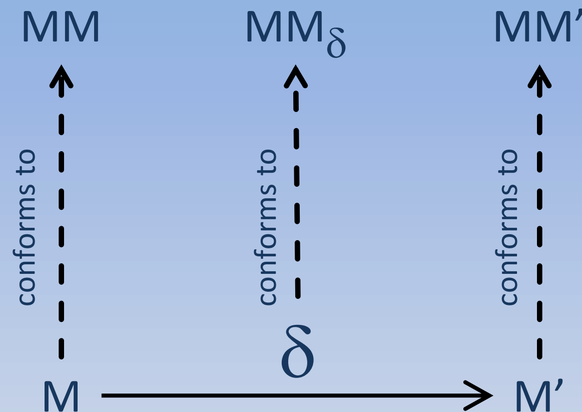
Execution

- **Given input model**
- **Run transformation**
  - Rules
  - Unordered, Priority, Layer, Control Flow
- **Output**
  - New model
  - Modified model

Main Concepts of Model Transformation

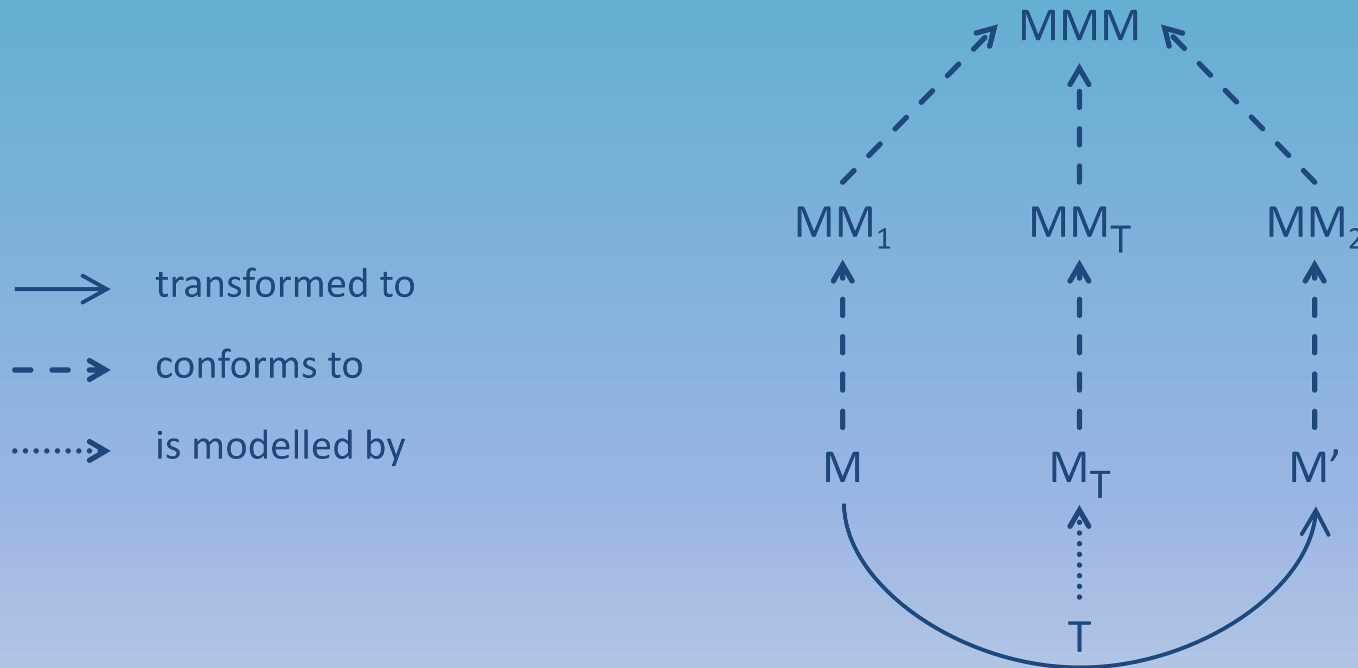
IN A MDE FRAMEWORK

- Everything is **modelled**
  Therefore a change will always be on a model.
- We explicitly **model** everything
  A change or modification must itself be **modeled** ⇒ models of transformations.



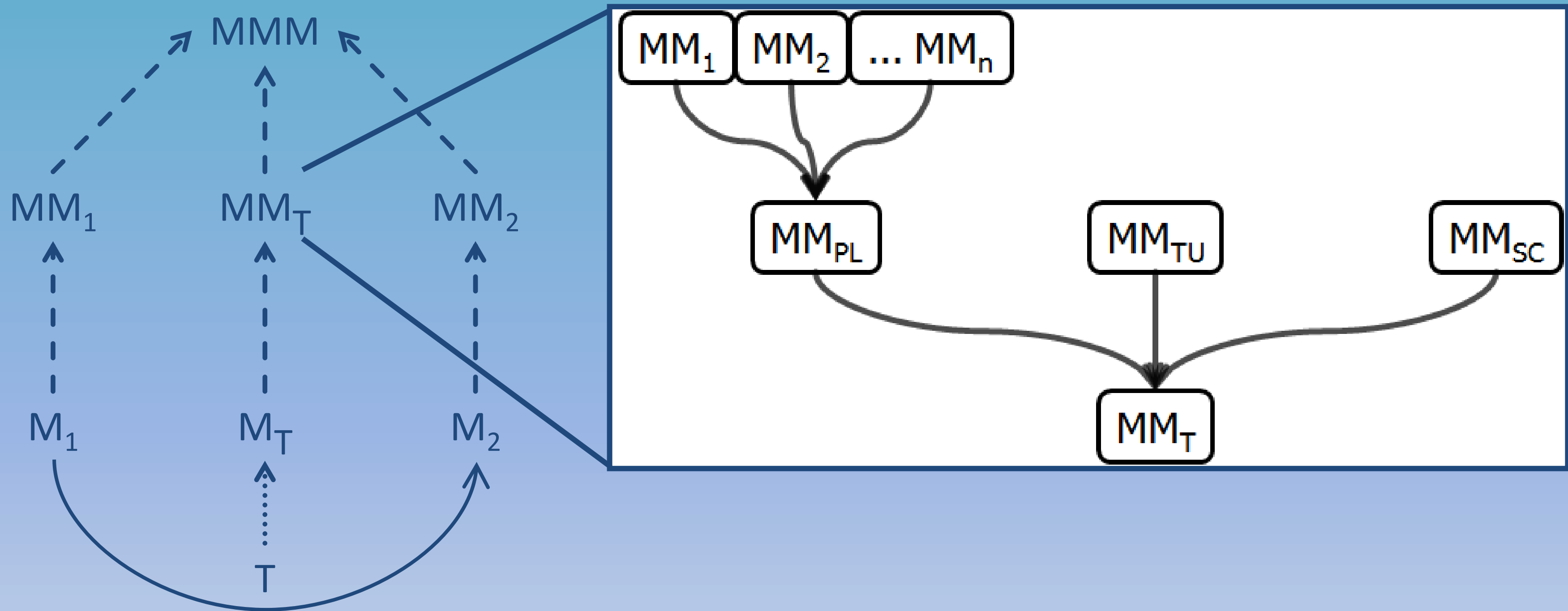
- δ represents an intentional change (or alteration) of M, which yields M'
- MM_δ defines all possible changes for the same intention from an instance of MM to an instance of MM'

MODELS, META-MODELS & TRANSFORMATIONS



- **T**: operation that transforms the model M_1 into M_2.
- **M_T**: model of a transformation that transforms any model from MM_1 into a model from MM_2.
- **MM_T**: meta-model of all transformations that transform models from any meta-model.
- **MMM**: meta-model of the language used to describe meta-models.

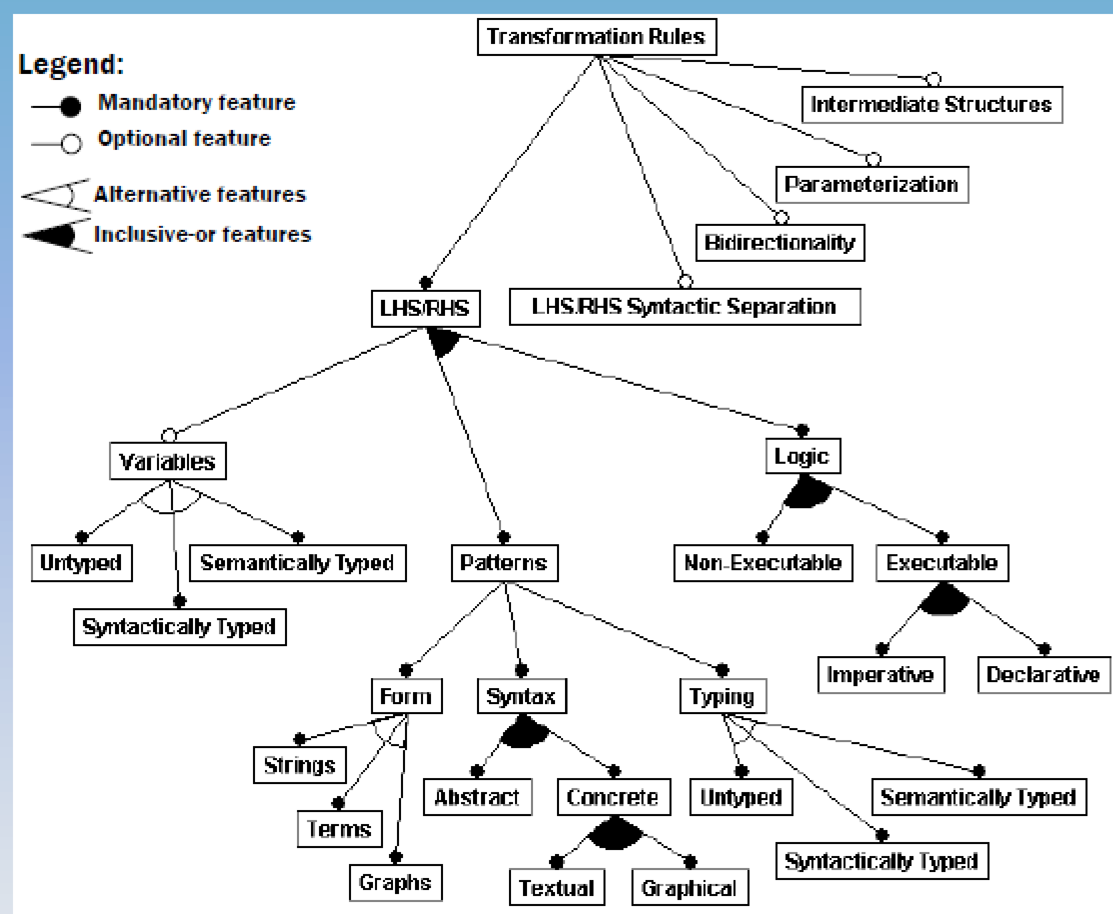
MODEL OF MODEL TRANSFORMATION



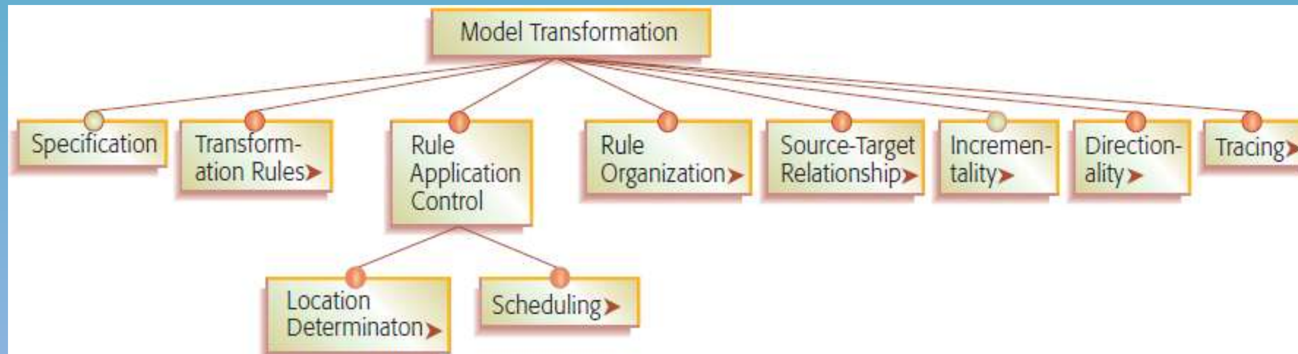
- **MM_{TU}**: meta-model of the transformation units
  - rules, queries, primitive operators, helper functions, modules/packages
- **MM_{SC}**: meta-model of the scheduling language
  - programing lang, workflow lang, modeling lang, DSL for scheduling trafos
- **MM_{PL}**: meta-model of the pattern language
  - model fragments specified in the pre- and post-conditions of transformation rules

FEATURES OF MTL

Using feature diagrams



FEATURES OF MODEL TRANSFORMATION LANGUAGES [Czarnecki06]



- **Specification**

Pre/post condition on the transformation:
  - Function between source & target models
  - Relation may be executable or not

- **Transformation Rules**

Smallest transformation unit, used to specify a transformation
  - Rule-based transformations: pre-condition & post-condition for rewriting
  - Transformation units defined as functions
  - Templates

- **Rule Application Control**
  - Where is a rule applied on the model
  - In what order are the rules executed

- **Source-Target Relationship**
  - In-place
  - Out-place

- **Rule Organization**

General structuring issues of rules
  - Modularization
  - Composition
  - Re-use

- **Incrementality**

Ability to update existing target models based on changes in the source models

- **Directionality**

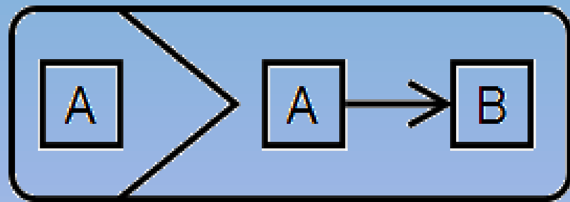
Transformation executed in one direction or in multiple directions (uni-/multi-directional transformation)

- **Tracing**

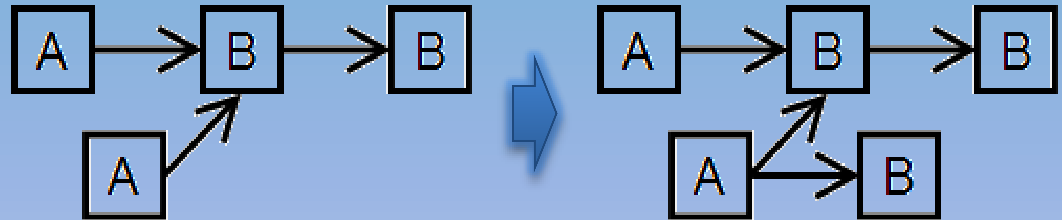
Mechanisms for recording different aspects of transformation execution:
  - Create & maintain trace links between source & target model elements

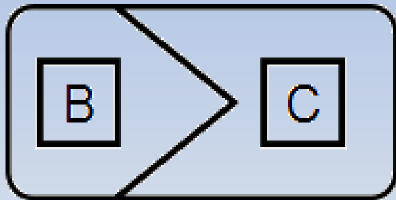
TRANSFORMATION RULES

- Smallest transformation units
- A model transformation is mainly specified in terms of rules

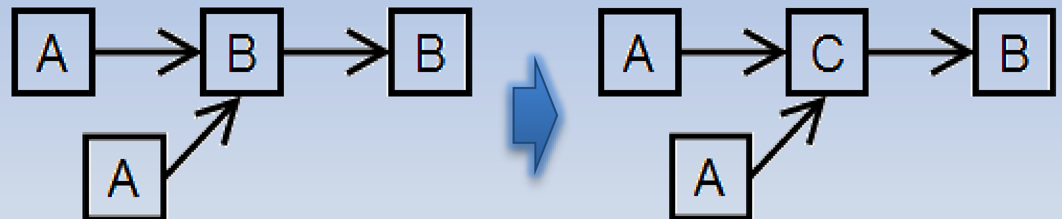


:





:



DOMAIN OF A TRANSFORMATION

- Defines how a rule can access elements of the models
- 1..* domains: examples of 1-way transformation? 2-way? n-way?
- Domain language
  - The language in which models are defined. Typically MOF
- Domain Modes
  - Read-only: source domain of synthesis
  - Write-only: target domain of synthesis
  - Read-write: domain of simulation

QVT-Relations rule

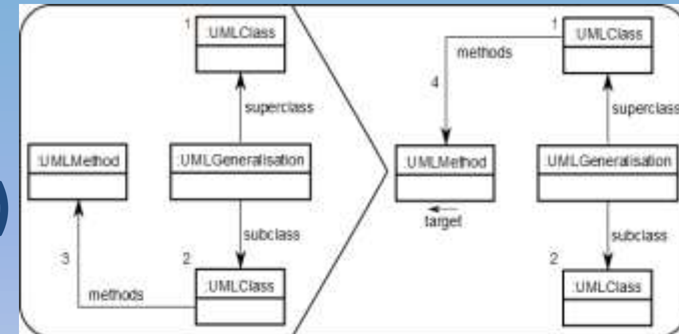
```
top relation PackageToSchema {
domain uml p:Package {name = pn}
domain rdbms s:Schema {name = pn}
}
```

BODY OF A RULE

Patterns

- Model fragment internally represented as:
  - **Strings**: Template-based transformation
  - **Terms**: tree representation of models
  - **Graph**: Model-to-model transformation
- Using a specific syntax (textual, graphical)
  - **Abstract syntax**
  - **Concrete syntax**
- Syntactic separation

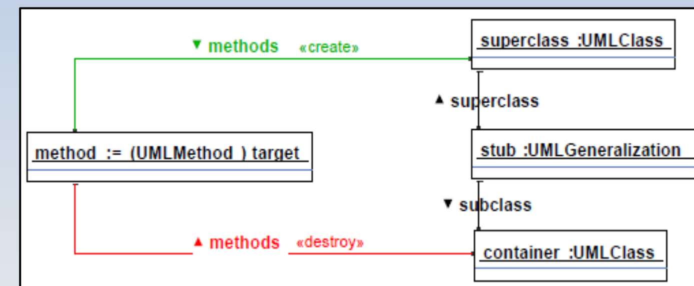
MoTif rule



ATL rule

```
module Person2Contact;
create OUT: MMb from IN: MMa {

rule Start {
   form p: MMa!Person(
      p.function = 'Boss'
   )
   to c: MMb!Contact(
      name <- p.first_name +
p.last_name)
}
```

FUJABA: compact notation



RULE LOGIC

How computations & constraints are specified on model elements

	Imperative	Declarative
Executable	Java API for MOF models	OCL query
Non-executable	X	Graph trafo rule

QVT-Relations rule

```
top relation PackageToSchema {
   domain uml p:Package{name=pn}
   domain rdbms s:Schema{name=pn}
}
```

Kermeta operation [Falleri06]

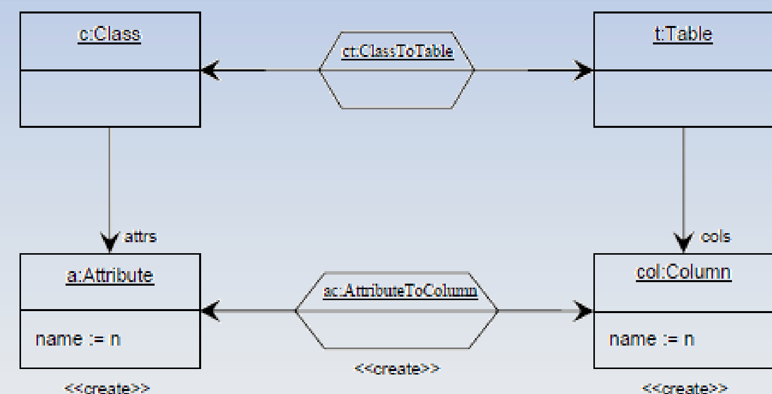
```
operation transform(source:PackageHierarchy): DataBase is
do
    result := DataBase.new
    trace.initStep("uml2db")
    source.hierarchy.each{ pkg |
       var scm: Schema init Schema.new
       scm.name := String.clone(pkg.name)
       result.schema.add(scm)
       trace.addlink("uml2db", "package2schema", pkg, scm)
    }
end
```

DIRECTIONALITY

Ability to execute the transformation in different directions

- **Unidirectional** : create (or update) the target model only
- **Multi-directional**: can be executed in any direction
  - Multi-directional rules
- **Operational** rules have a functional character: given an input model, produce a target model.
  - **Causality** from source to target model
- **Declarative** rules: gives a relation between both models that must be satisfied
  - **Acausal** relationship between the models

TGG rule [Schurr94]



QVT-Relations rule

```
top relation PackageToSchema {
domain uml p:Package {name = n}
domain rdbms s:Schema {name = n}
}
```

INCREMENTALITY

- An **incremental transformation** is defined as a set of relations between a source and target meta-models. These relations define constraints on models to be synchronized.
- The first time it is run, it creates a target model. Trace links are often automatically created.
- Then, if a change is detected in one of the models, it **propagates** this change to the other model, by adding, removing, or updating an element so that the relations are still satisfied.
- There are 4 standard scenarios in model synchronization:
  - Create a target model from the source model
  - Propagate changes in the source model to the target model
  - Propagate changes in the target model to the source model
  - Verify consistency between the two models

TRACING

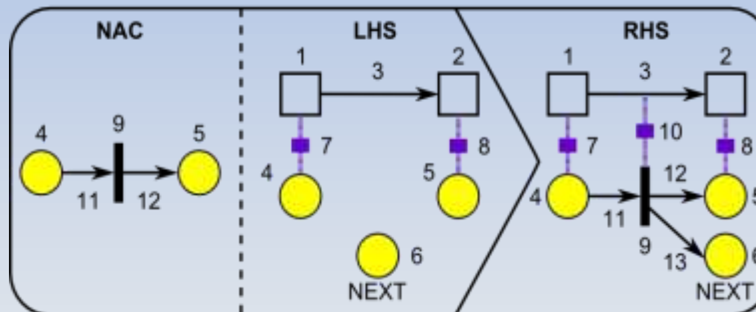
Runtime footprint of a transformation execution

- **Traceability links** connect source & target elements
  They are instances of the mappings between the domains
  - Impact analysis
  - Direction of the synchronization
  - Debugging transformations
- Automatic creation of trace links: QVT, ATL
- Traces can be considered as any other model, but has to be manually created:
  e.g., AGG , $ATOM^3$, VIATRA

INTERMEDIATE STRUCTURES

Creation of additional elements which are not part of the domain

- **AToM^3: generic links, simplifies the transformation rules**
- **ATL: automatic creation of traceability links.**
  Each newly created element is linked back to element(s) of the source model.
- **AGG and VIATRA: make use of traceability to prevent a rule from being applied on the same element.**



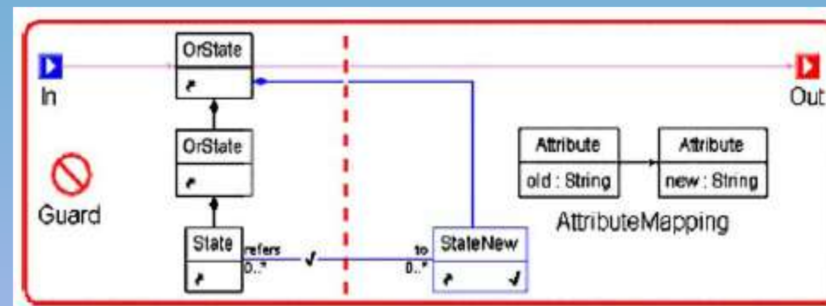
MoTif rule

PARAMETERIZATION

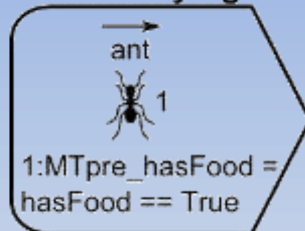
Control Parameters

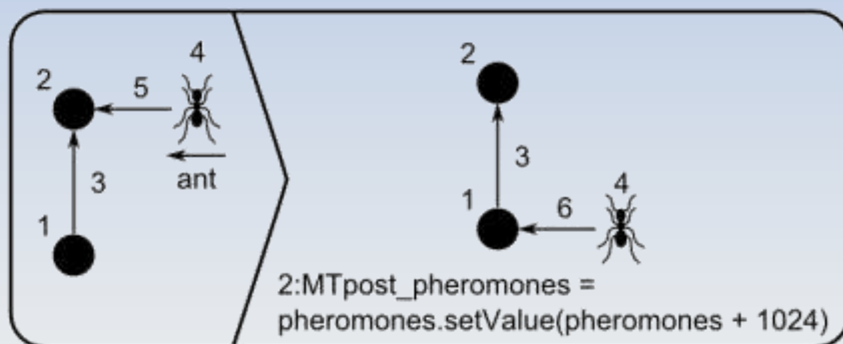
- **Pre-defined binding of some model elements**
- **ProGreS: variable parameter passing**
- **GReAT, MoTif: pivot nodes**

GReAT pivot passing

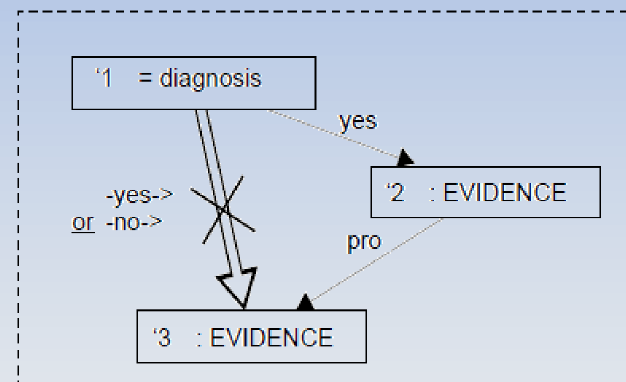


MoTif pivots





ProGreS in/out parameters



PARAMETERIZATION

Generics

- **Pass element types to rules**
- **In this case, the types of the elements in the patterns are variable**

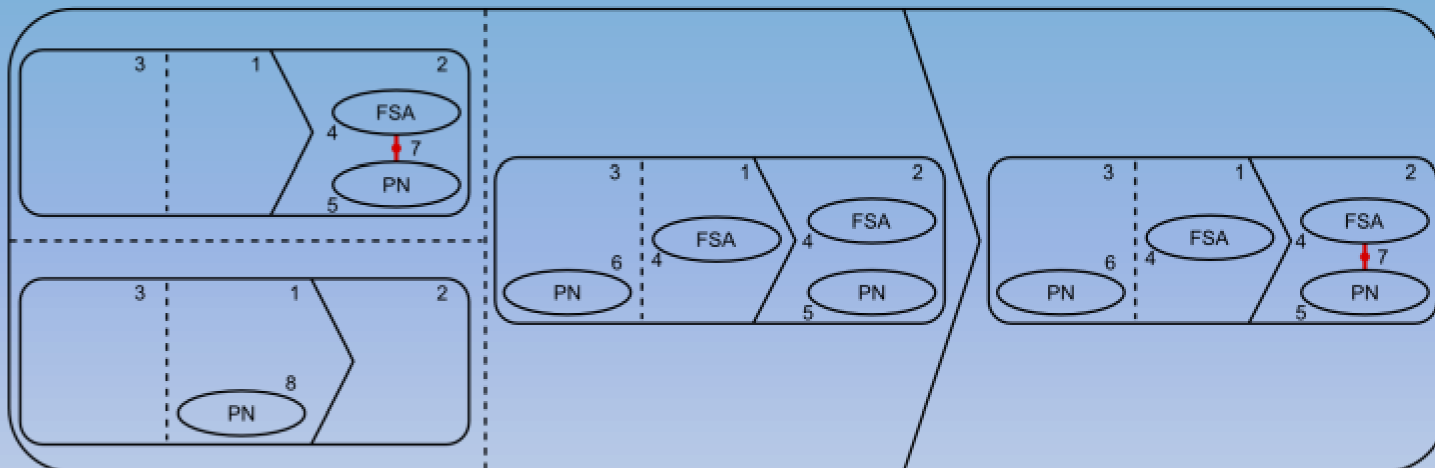
VIATRA generic rule

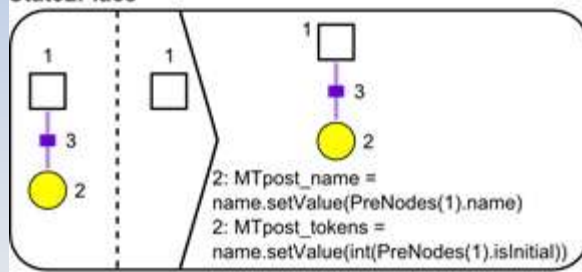
```
condition pattern transClose(CP,CS,A,
          ClsE, AttE, ParR, AttR) =
{
   // Pattern on the meta-level
   entity(ClsE);
   entity(AttE);
   relation(ParR,ClsE,ClsE);
   relation(AttR,ClsE,AttE);
   // Pattern on the model-level
   entity(CP);
   // Dynamic type checking
   instanceOf(CP,ClsE);
   entity(CS);
   instanceOf(CS,ClsE);
   entity(A);
   instanceOf(A,AttE);
   relation(Par,CS,CP);
   instanceOf(Par,ParR);
   del relation(Attr,CS,A);
   del instanceOf(Attr,AttR);
   new relation(Attr2,CP,A);
   new instanceOf(Attr2,AttR);

}
```

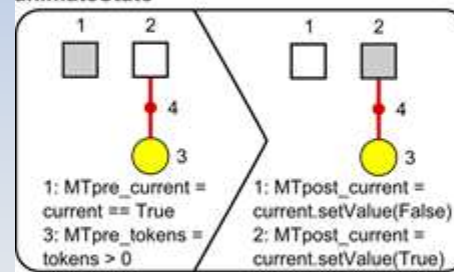
PARAMETERIZATION

Higher-Order Transformation (HOT)

- Takes a rule as input and outputs another rule





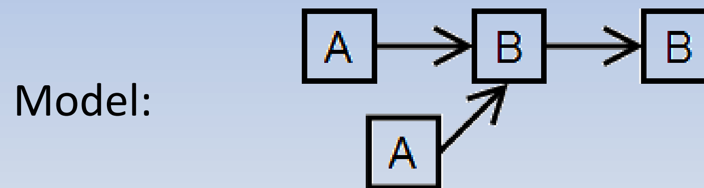


LOCATION DETERMINATION

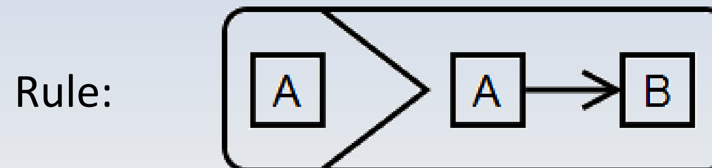
Strategy for determining the application locations of a rule

- **Deterministic**: same choices will be made every time
- **Non-deterministic**
  - **One-point**: once choice is made, at random (repeated?)
  - **Concurrent**: all occurrences
    - **Critical pair analysis** to ensure there are no overlapping matches
- **Interactive**: choice resolved by user/external intervention

Model:



Rule:

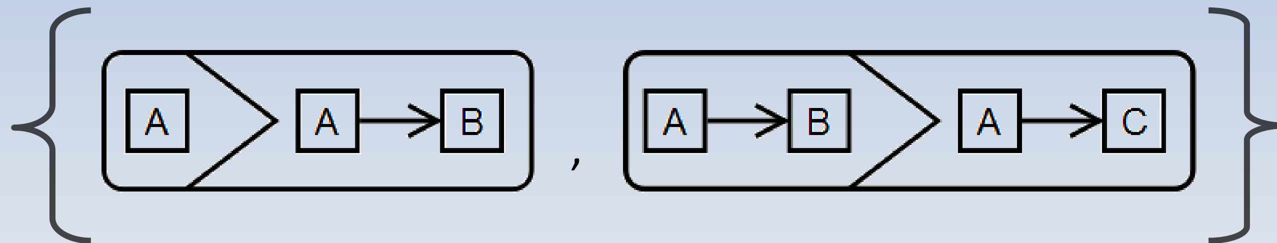


RULE SCHEDULING

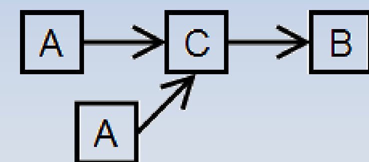
Strategy for determining the order in which the rules are applied

- **Implicit**: completely determined by the design logic of the rules
  - **Unordered**: One rule that is applicable is selected to be applied non-deterministically at each iteration
  - **Grammar**: unordered with start model and terminal states (generation or recognition)
- E.g., Groove, MOMENT2

Rules:



Model:



RULE SCHEDULING

Strategy for determining the order in which the rules are applied

- **Explicit internal**: a rule may invoke other rules.
  - In ATL, a matched rule (implicitly scheduled) may invoke a called rule in its imperative part. Lazy rules
  - In QVT, the when/where clauses of a rule may have a reference to other rules.
    - When: the former will be applied after the latter
    - Where: the latter will be applied after the former

```
top relation ClassToTable {
    domain uml c:Class {
      package = p:Package{},
      isPersistent = true,
      name = cn
    }
    domain rdbms t:Table {
        schema = s:Schema{},
        name = cn,
        cols = cl:Column {
            name = cn + '_tid',
            type = 'NUMBER'},
        pkey = cl
    }
    when {
        PackageToSchema (p, s);
    }
    where {
        AttributeToColumn (c, t);
    }
}
```

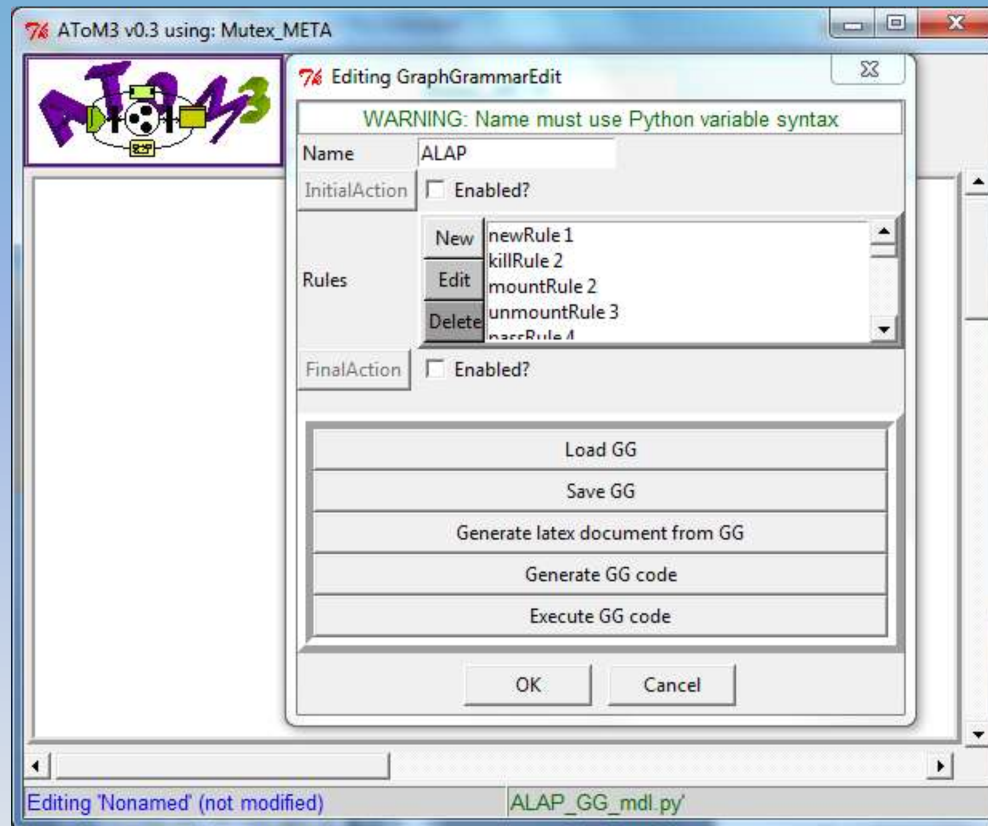
RULE SCHEDULING

Strategy for determining the order in which the rules are applied

- **Explicit external: clear separation between the rules and the scheduling logic.**
  Use a control structure to define rule scheduling
- **Ordered: priority, layer/phased, explicit workflow structure, …**
- **Event-driven: rule execution is triggered by external events**

RULE SCHEDULING

Priority-based: $AToM^3$



RULE SCHEDULING

Layered/Phased: AGG

AGG V1.2.0b
File Edit Mode Transform Parser Analyzer Preferences Help

GraGras
- Shopping
  - [Em] Type Graph
  - Graph
  - takeCart
  - selectGood
  - payBill
  - createBill
  - billGood
  - settleBill
  - UniqueGoodOwnership
    - belongsToShop
    - belongsToCustomer
  - RackOwnership
    - belongsToShop
  - Ownerships

Node Type: Good
Edge Type: owns

UniqueGoodOwnership with belongsToShop

1:Customer cash=x; 4:; 2:Bill total=y; 5:; 3:Shop; 6:Good value=z

1:Customer cash=x; 4:; 2:Bill total=y; 5:; 6:Good value=z; owns; 3:Shop

Type Graph of Shopping

Customer int cash; Bill int total; Shop; Cart; Good int value; Rack; CashBox int amount; owns; owns; 1; 0..1; 0..1; *; 1; 0..1; 1; 1; 0..1; 0..1; *; *; 1; 0..1; 0..1; *; *; *; 0..1

Edit Constraint (Formula)

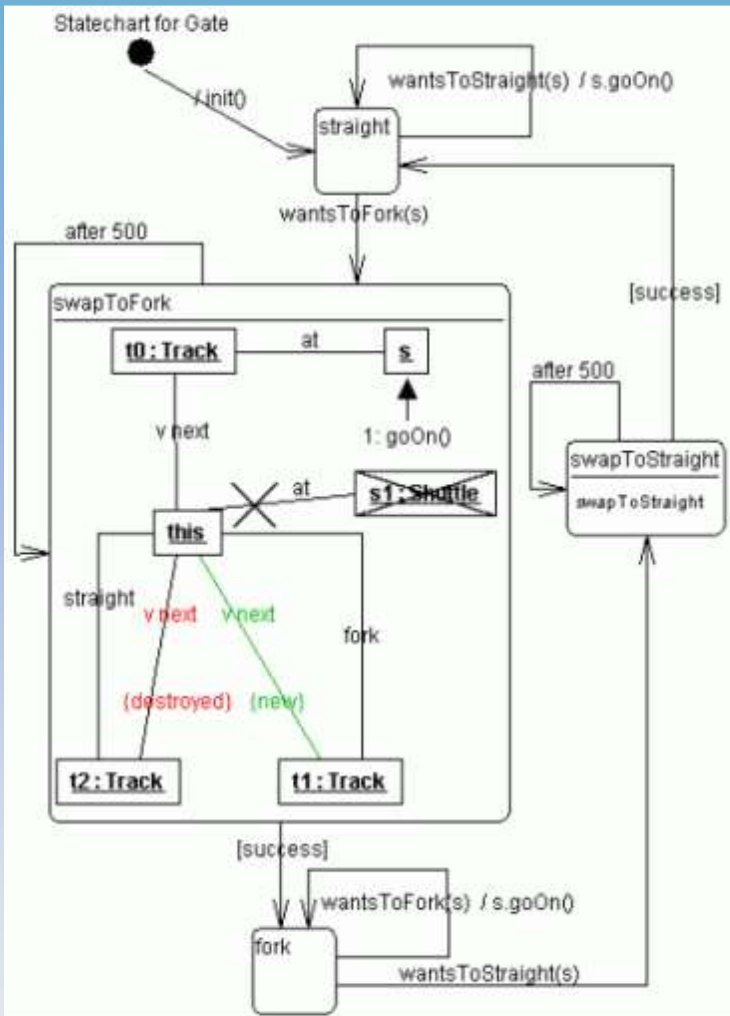
1 UniqueGoodOwnership
2 RackOwnership

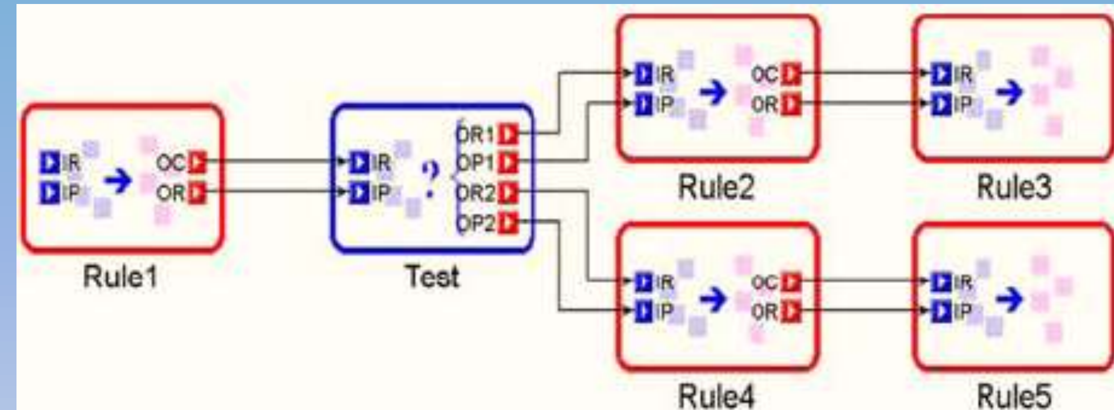
(1 && 2)

Ok Cancel

RULE SCHEDULING

Priority-based: AToM^3





GReAT data flow

FUJABA story diagram

EXPLICIT SCHEDULING FEATURES

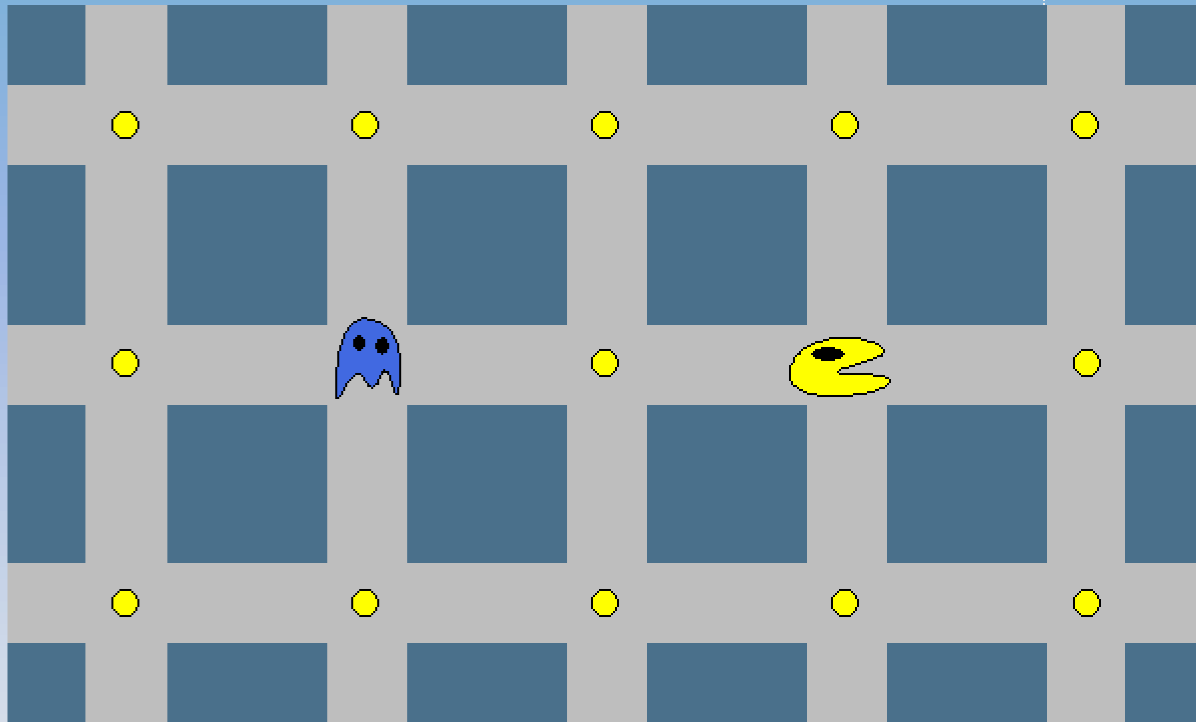
	PRoGReS	FUJABA	VIATRA	AToM^3	GReAT	VMTS	MoTif
Control structure	Imperative language	Story Diagram	Abstract state machine	Priority ordering	Data flow	Activity diagram	DEVS
Atomicity	transaction, rule	Rule	gtrule	Rule	Expression	Step	ARule
Sequencing	&	Yes	seq	Implicit	Yes	Yes	Yes
Branching	choose...else	Branch activity	if-then-else	No	Test / Case	Decision step, OCL	Query
Looping	loop	For-all pattern	iterate, forall	Implicit	Yes	Self loop	FRule, SRule, LRule
Non-determinism	and, or	No	random	Within layer	$1-n$ *connection*	No	Selector pattern
Recursion	Yes	No	Yes	No	Yes	Yes	Yes
Parallelism	No	Optional	No	Optional	No	Fork, Join	Synchronizer pattern
Back-tracking	Implicit	No	Choose (implicit)	No	No	No	XRule
Hierarchy	Modularisation	Nested state	Yes	No	Block, ForBlock	High-level step	CRule

DIFFERENT MODEL TRANSFORMATION APPROACHES

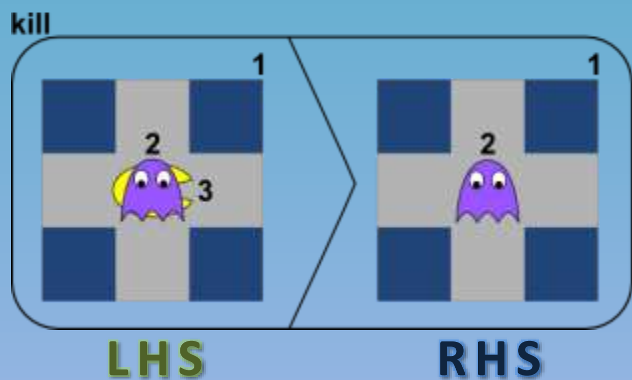
- **Model-to-text** (concrete syntax)
  - **Visitor-based:** traverse the model in an object-oriented framework
  - **Template-based:** target syntax with meta-code to access source model
- **Model-to-Model**
  - **Direct manipulation:** access to the API of M3 and modify the models directly
  - **Operational:** similar to direct manipulation but at the model-level (OCL)
  - **Rule-based**
    - **Graph transformation:** implements directly the theory of graph transformation, where models are represented as typed, attributed, labelled, graphs in category theory. It is a declarative way of describing operations on models.
    - **Relational:** declarative describing mathematical relations. It define constraints relating source and target elements that need to be solved. They are naturally multi-directional, but in-place transformation is harder to achieve

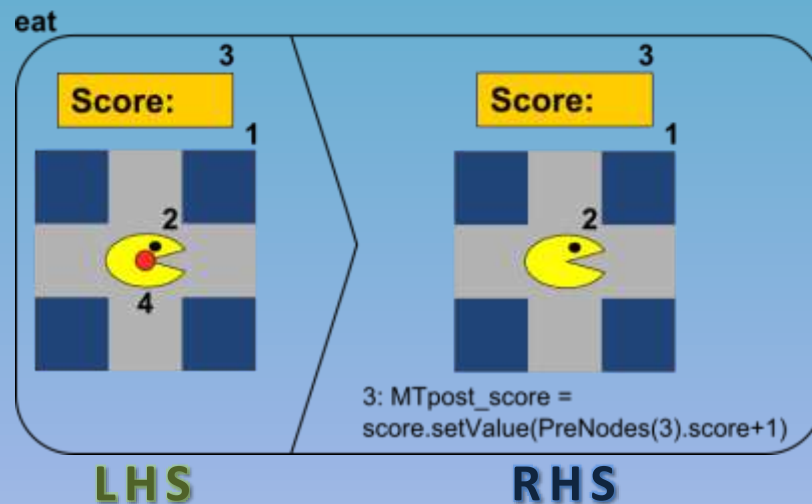
Rule-based Transformation

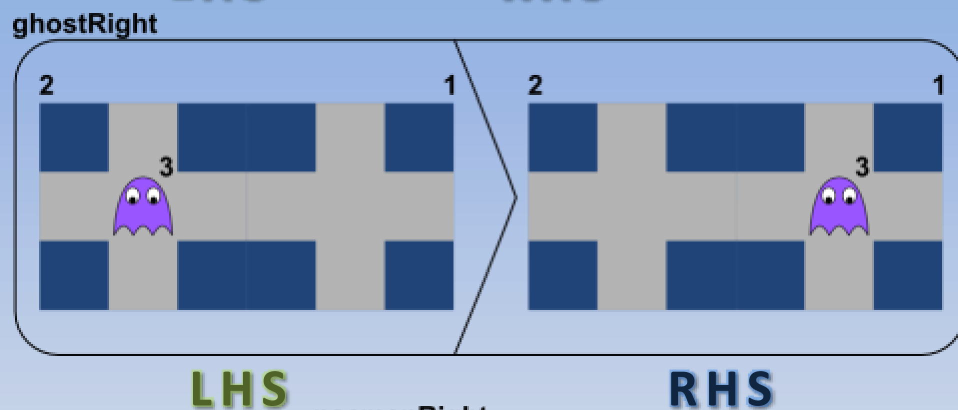
BY EXAMPLE

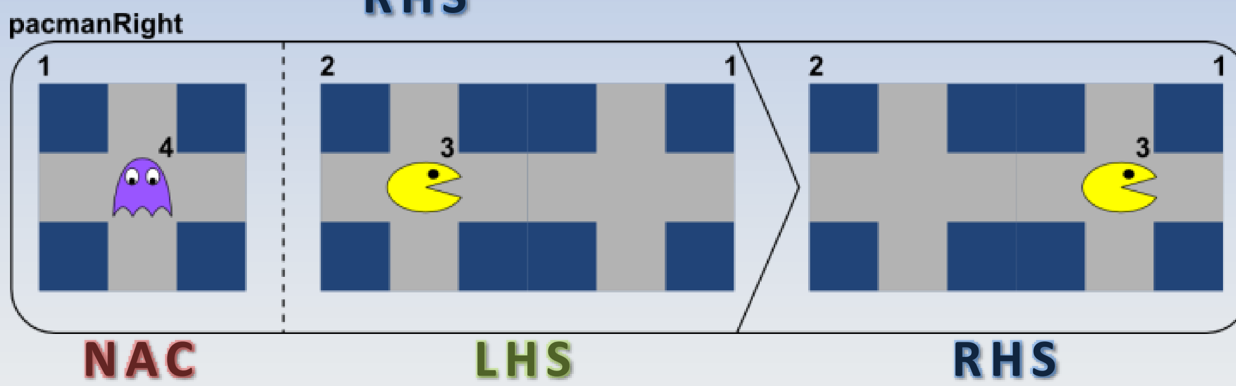


UNORDERED RULES

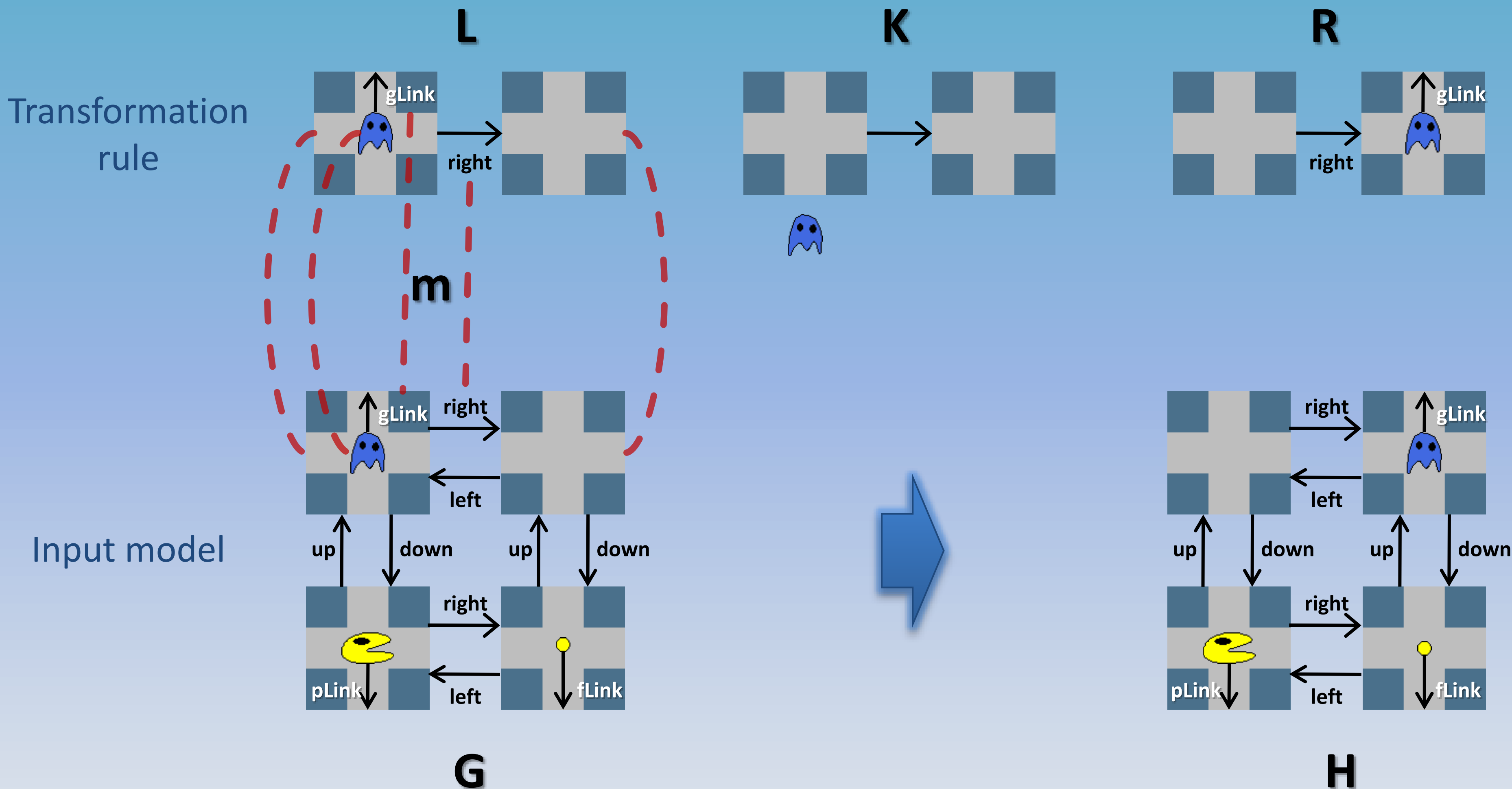








RULE-BASED MODEL TRANSFORMATION



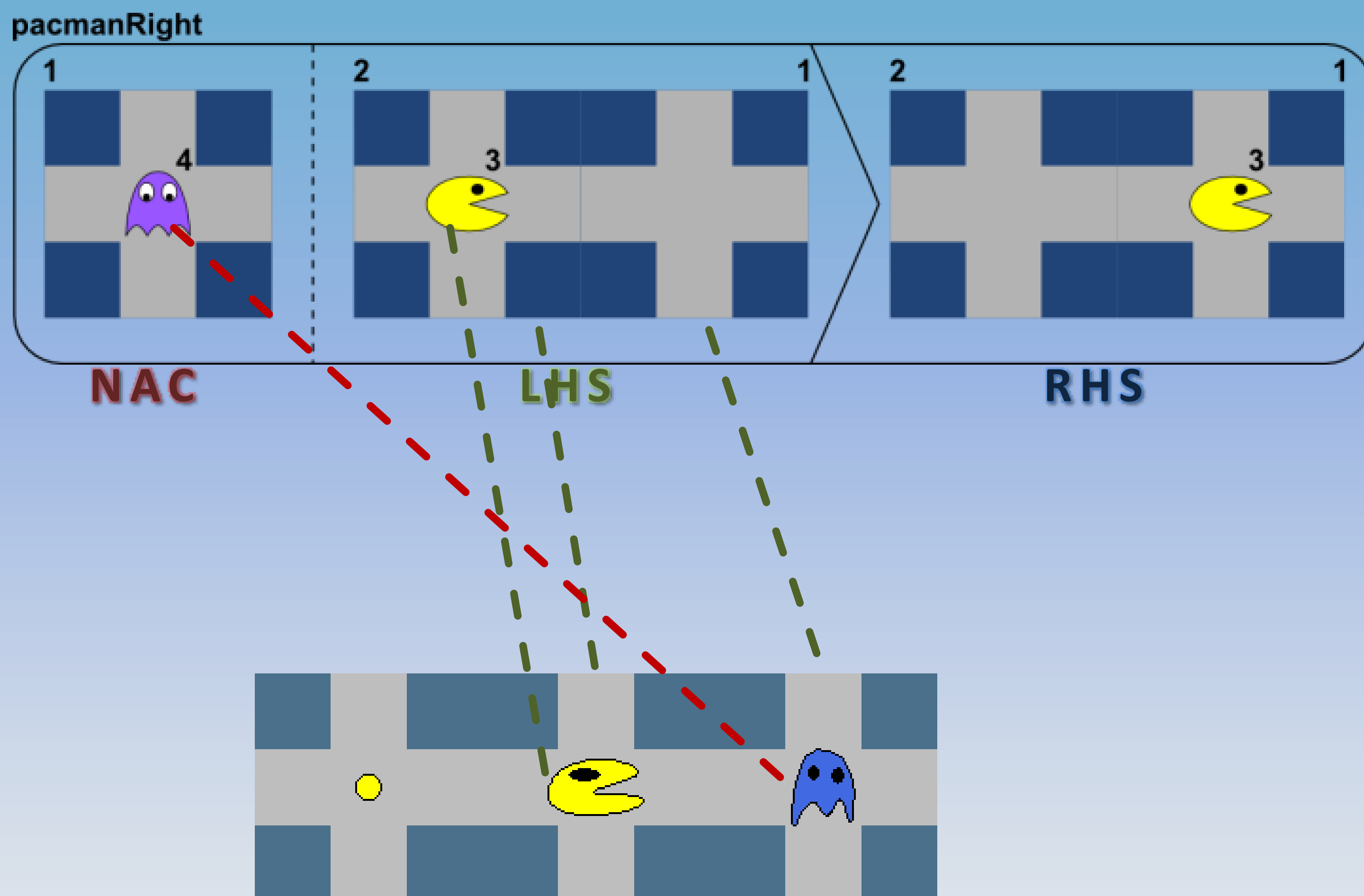
If there exists an occurrence of L in G then replace it with R

MECHANICS OF RULE APPLICATION

- Matching Phase
  - Find an embedding m of the LHS pattern L in the host graph G
  - An occurrence of L is called a **match**: $m(L)$
  - Thus, $m(L)$ is a sub-graph of G
- Rewriting Phase
  Transform G so that it satisfies the RHS pattern:
  - **Remove** all elements from $m(L - K)$ from G
  - **Create** the new elements of $R - K$ in G
  - **Update** the properties of the elements in $m(L \cap K)$
- When a match of the LHS can be found in G, the rule is **applicable**
- When the rewriting phase has been performed, the rule was **successfully applied**

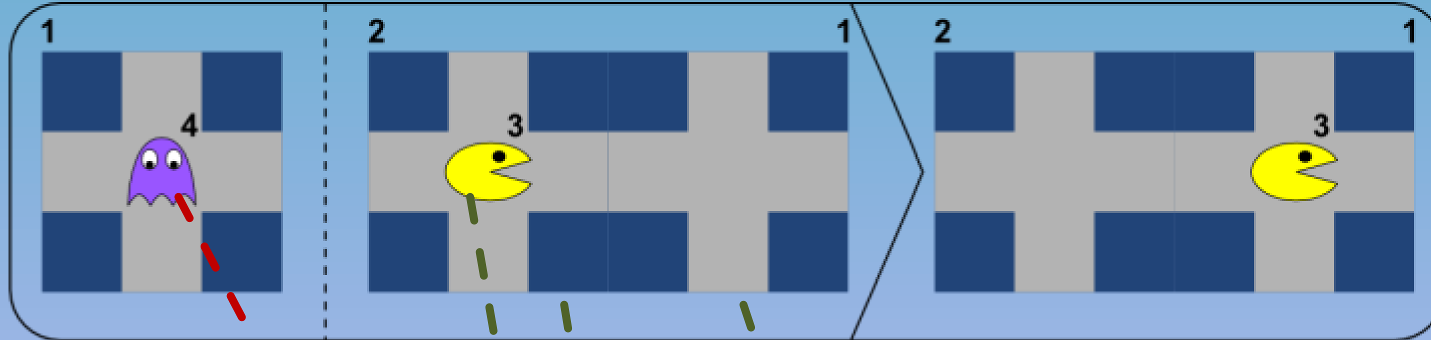
NEGATIVE APPLICATION CONDITIONS

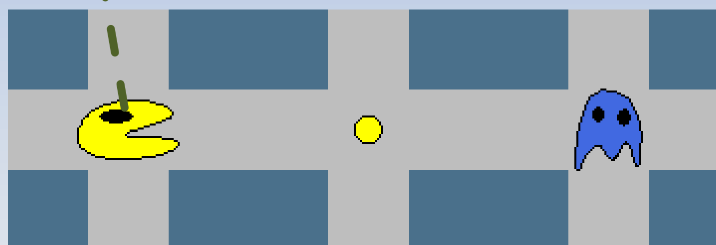
Non-applicable rule



NEGATIVE APPLICATION CONDITIONS

Applicable rule





HOW TO FIND A MATCH?

- The matching phase is NP-Complete, the rewriting phase is linear.
- There are various exponential-time worst case solutions for pattern matching, for which the average-time complexity can be reduced with the help of heuristics
- **Search Plan Approach**
  - Define the traversal order for the nodes of the model to check whether the LHS can be matched.
  - Compute the cost tree of the different search paths and choosing the less costly one.
  - Complex model-specific optimization steps can be carried out for generating efficient adaptive search plans.
- **Constraint Satisfaction Solving Approach (CSP)**
  - Consider the LHS elements as variables, the elements of model form the domain and typing, and the links and attribute values form the set of constraints.
  - Based on back-tracking algorithms

QUESTION

What is the worst upper-bound of the complexity for applying a rule?

- $O\left(|L|^{|G|} + |R|\right) = O\left(|L|^{|G|}\right)$ **CRUD operations**

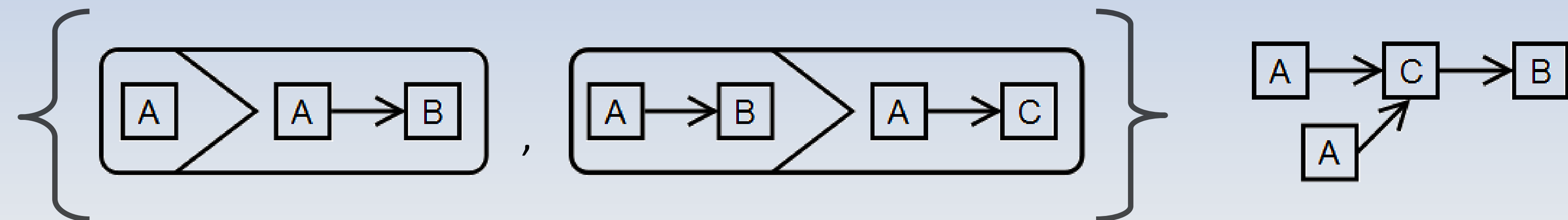
Graph Transformation

REWRITING SYSTEMS

From Chomsky Grammars to Graph Grammars

- **Start symbol:** S
- **Terminals:** $\{a, b\}$
- **Non-terminals:** $\{S, A\}$
- **Production rules:**

$$S \rightarrow ASb$$
$$A \rightarrow a$$
$$S \rightarrow \varepsilon$$

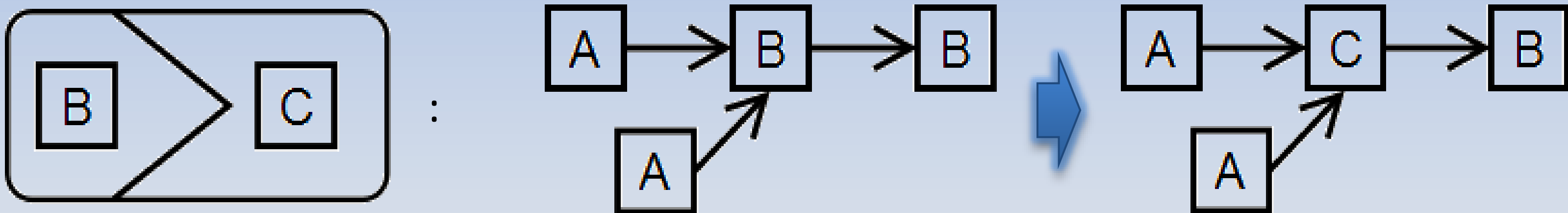


REWRITING SYSTEMS

From Term Rewriting to Graph Rewriting

- **Signature:** $\{0, s, add\}$
- **Rewrite rules:**

$$add(0, y) \rightarrow y$$
$$add(s(x), y) \rightarrow s(add(x, y))$$



ALGEBRAIC GRAPH TRANSFORMATION

- **Based on category theory**
- **Category:** Graphs
  - **Objects: typed, attributed, labeled, directed graphs**
$$G = (V, E, s, t)$$
$$s, t: E \rightarrow V$$
  - **Morphisms: total graph morphisms in the form**
$$f: G \rightarrow H = (f_V: V_G \rightarrow V_H, f_E: E_G \rightarrow E_H)$$
  - **Composition:**
$$f(B, C) \circ g(A, B) = h(A, C)$$
  - **Identity:**
$$f \circ id = f$$

PUSHOUT

A pushout over morphisms $m: L \to G$ and $r: L \to R$ is defined by

- a pushout object H
- morphisms $n: R \to H$ and $r': G \to H$

such that the following diagram commutes

$$\begin{array}{ccc} L & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle n} \\ G & \xrightarrow{r'} & H \end{array}$$

GRAPH TRANSFORMATION RULE

A production $p: \left(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R\right)$ is composed of a pair of injective morphisms $l: K \rightarrow L$ and $r: K \rightarrow R$ where:

- **L is the LHS**
- **R is the RHS**
- **K is the interface**

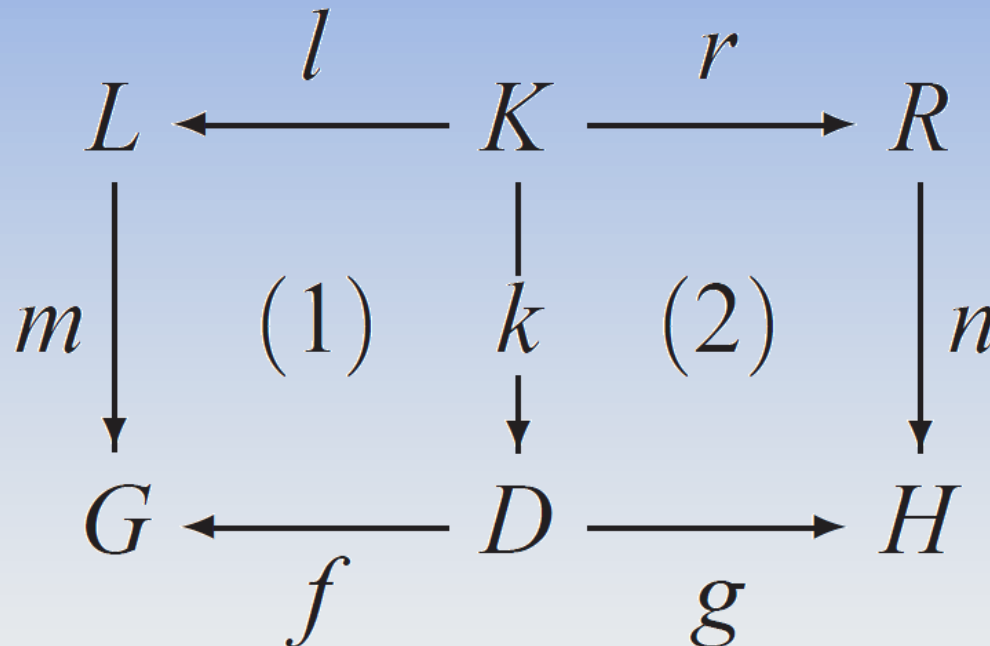
GRAPH TRANSFORMATION

- **Let** $p: \left(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R\right)$ **be a graph production**
- **Let** D **be a context graph**
- **Let** $m: K \rightarrow G$ **be a total graph morphism (match)**
- **A Double Pushout (DPO) graph transformation** $G \overset{p,m}{\Longrightarrow} H$ **is given by the DPO diagrams**

$$\begin{array}{ccccc} L & \xleftarrow{\;l\;} & K & \xrightarrow{\;r\;} & R \\ m \big\downarrow & (1) & \big\downarrow k & (2) & \big\downarrow n \\ G & \xleftarrow[\;f\;]{} & D & \xrightarrow[\;g\;]{} & H \end{array}$$

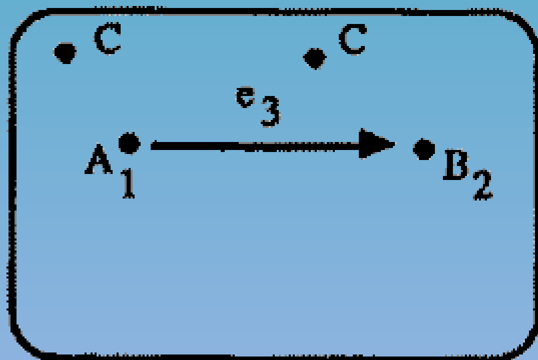
GRAPH TRAFO APPLICATION

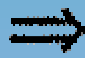
1. Find a **match** $\mathrm{M} = m(L)$ in G
2. **Remove** $L - K$ from M such that the gluing condition $(G - M) \cup k(K) = D$ still holds
3. **Glue** $R - K$ to D in order to obtain H

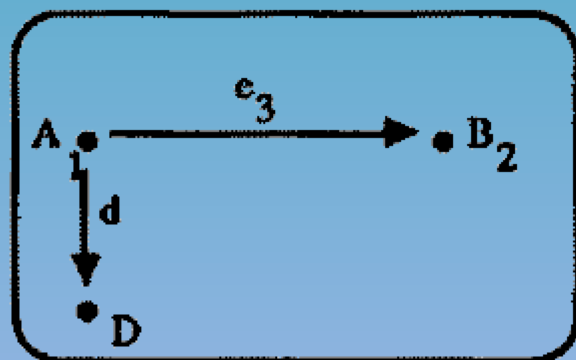


DPO EXAMPLE

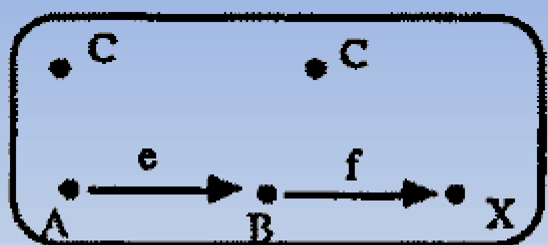
p_0:



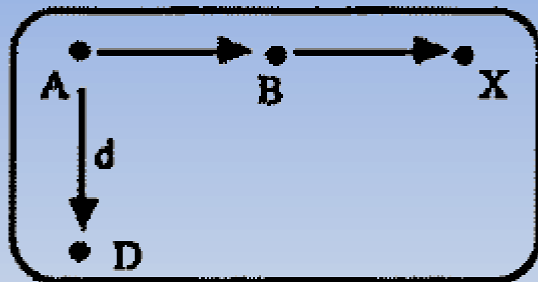




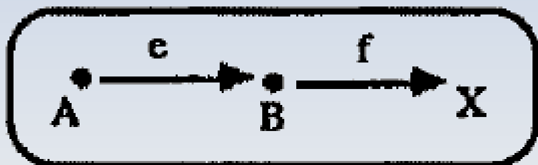
G:



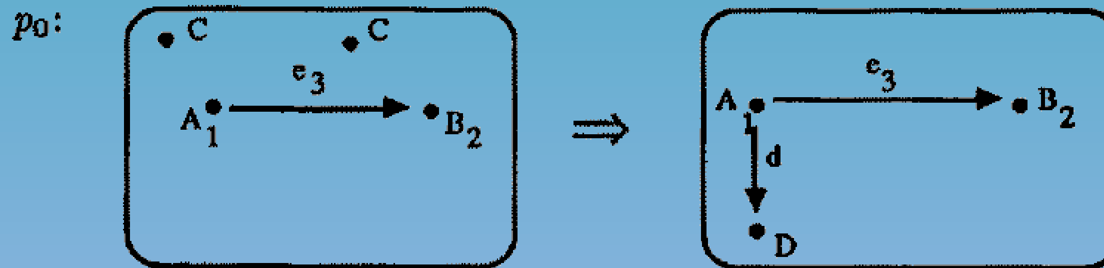
H:



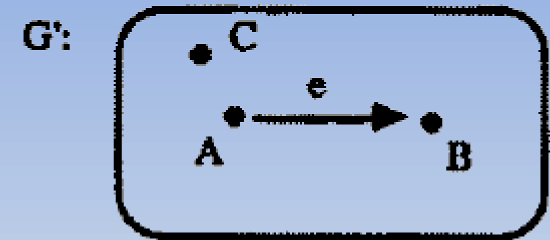
D:



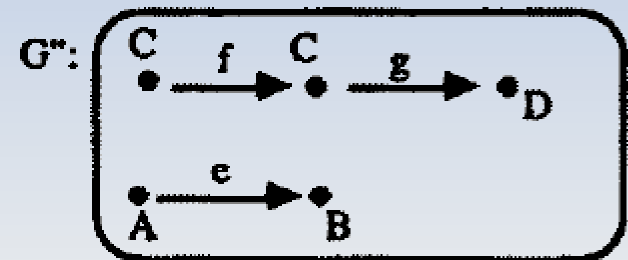
DPO GLUING CONDITIONS



- **Identification condition**
  - No two vertices in the LHS shall be mapped to the same element such that they must be deleted
  - p_0 cannot be applied on G'

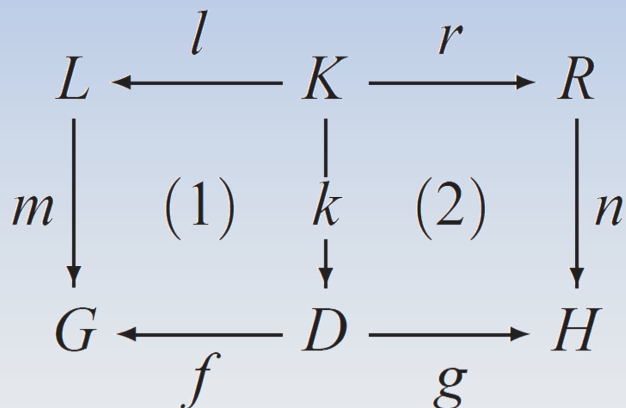


- **Dangling condition**
  - The LHS specifying the deletion of a vertex shall include all its incident edges
  - p_0 cannot be applied on G''

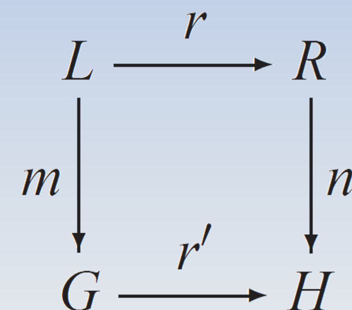


DPO VS. SPO

- Interface graph: l, r are total morphisms
- Restrictions on deletion of nodes & edges
- Safe by construction



- r is a partial morphism
- Dangling problem resolution
  - Implicitly delete the edges adjacent to the to-be-deleted vertex
- Identification problem
  - In practice, rule becomes inapplicable. But still allowed in theory
- Unsafe, care should be taken



FAMILY OF TRANSFORMATION LANGUAGES

Model Transformation has many applications:

- **Generate** PSMs form PIMs and **reverse engineering**
- **Map** and **synchronize** among models at the same or different level of abstraction
- Create **views** of a system
- Model **evolution** tasks

- Since the applications are very different in nature, it is not optimal to have a *single* model transformation language that supports all of the above.
- Instead, it is more appropriate to have dedicated transformation languages tailored to specific transformation problems.