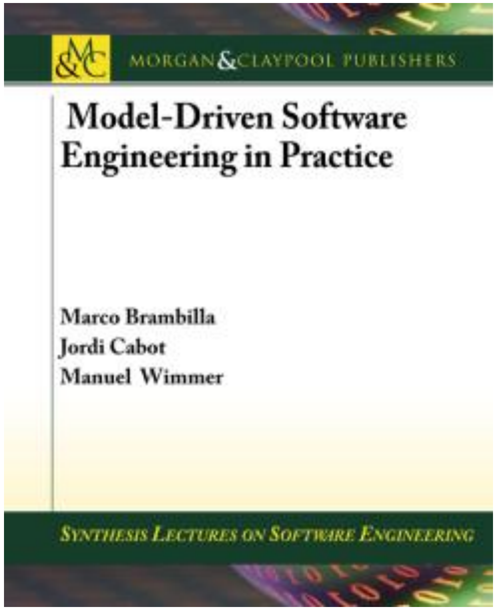
MORGAN & CLAYPOOL PUBLISHERS

Chapter 8

MODEL-TO-MODEL TRANSFORMATIONS

Teaching material for the book
Model-Driven Software Engineering in Practice
by Marco Brambilla, Jordi Cabot, Manuel Wimmer.
Morgan & Claypool, USA, 2012.



Content

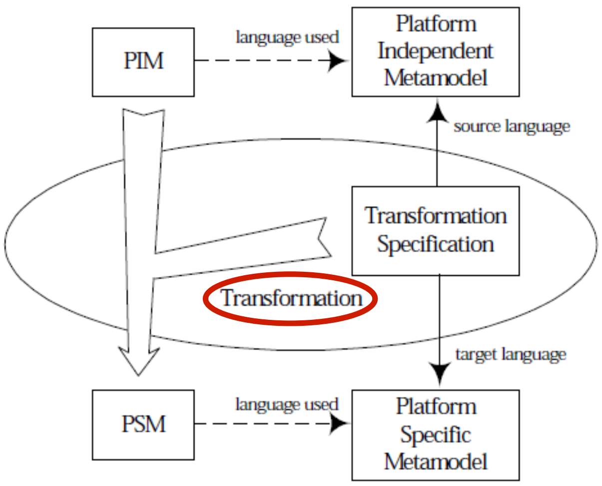
- Introduction
- Out-place Transformations: ATL
- In-place Transformations: Graph Transformations
- Mastering Model Transformations

INTRODUCTION

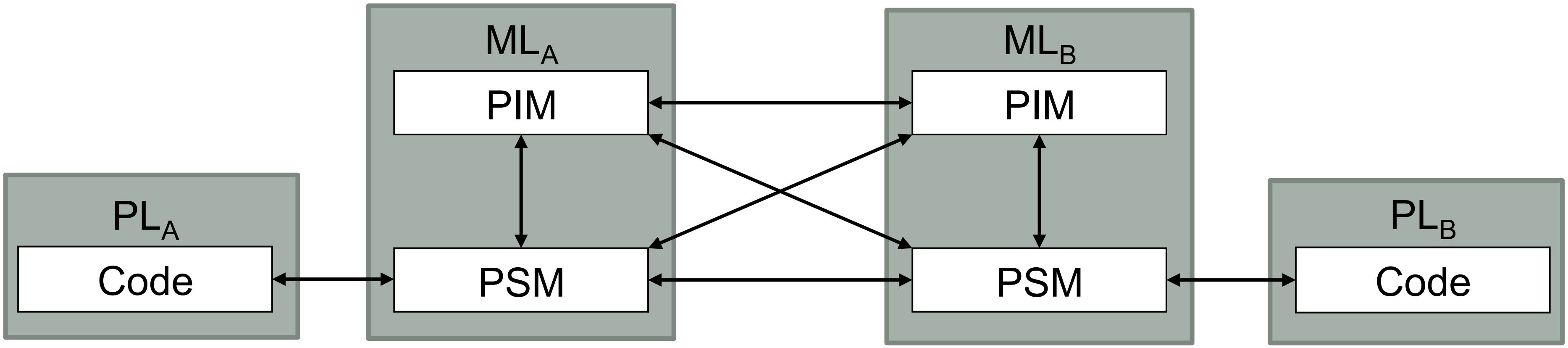
Motivation

Transformations are everywhere!

- **Before** MDE
  - Program compilation, refactoring, migration, optimization, ..
- **With** MDE
  - Transformations are **key technologies**!
  - Every **systematic manipulation** of a model is a model transformation!
- **Dimensions**
  - Horizontal vs. vertical
  - Endogenous vs. exogenous
  - Model-to-text vs. text-to-model vs. model-to-model



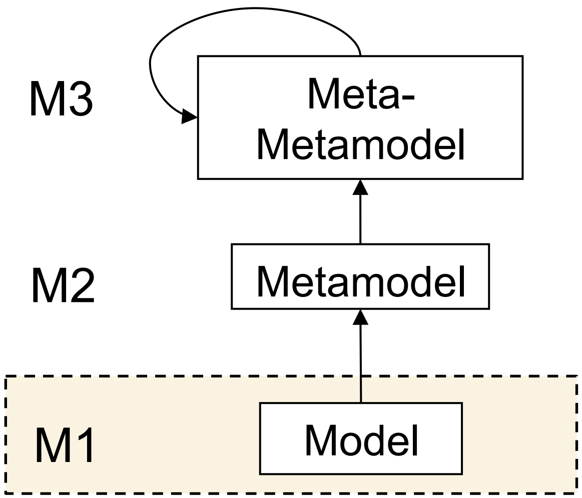
[Excerpt of MDA Guide from the OMG]



Definitions

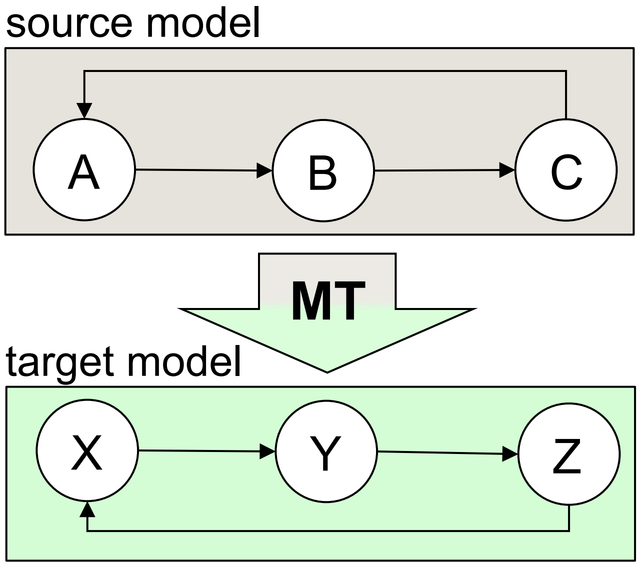
- A **model-to-model (M2M) transformation** is the automatic creation of target models from source models.
  - 1-to-1 transformations
  - 1-to-N, N-to-1, N-to-M transformations
  - target model* = T(source model*)

Metamodeling Stack



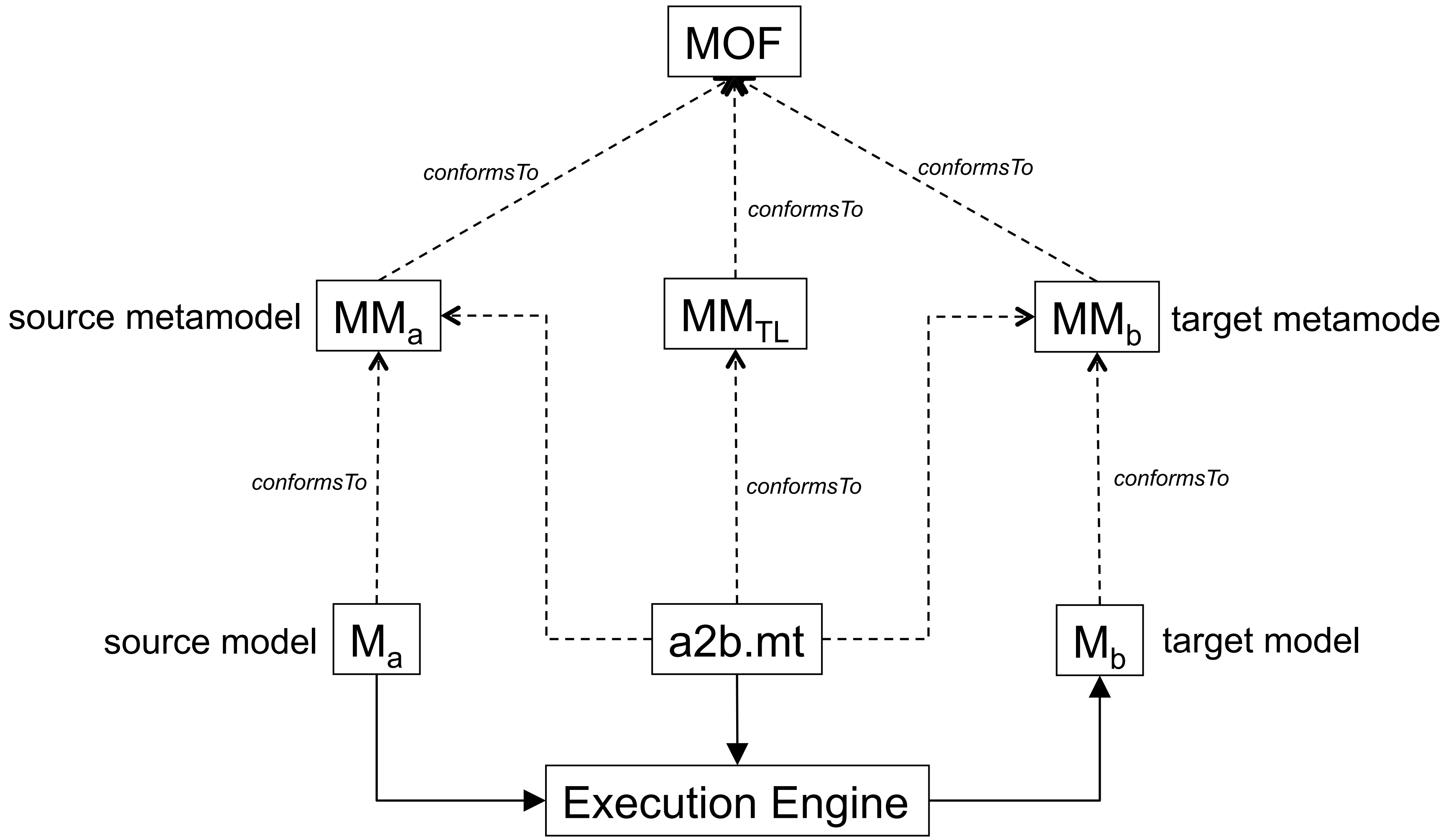
Focus of model transformations, but **not limited** to M1 **(remember: everything is a model!)**

Example



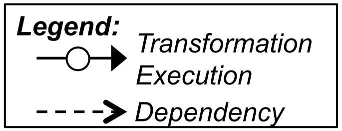
Architecture

Model-to-Model Transformation Pattern

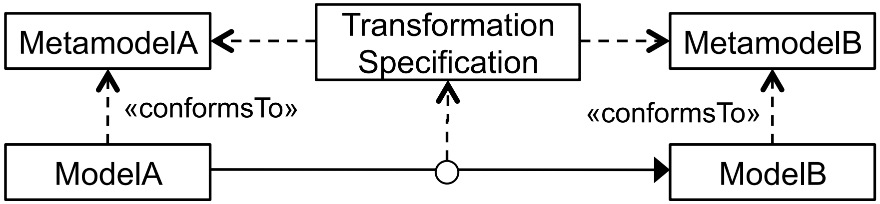


Two Transformation Strategies

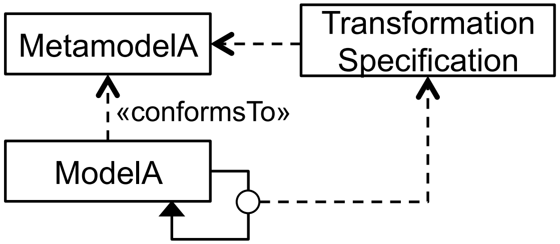
Out-place vs. in-place transformations



Out-place Transformations build a new model from scratch



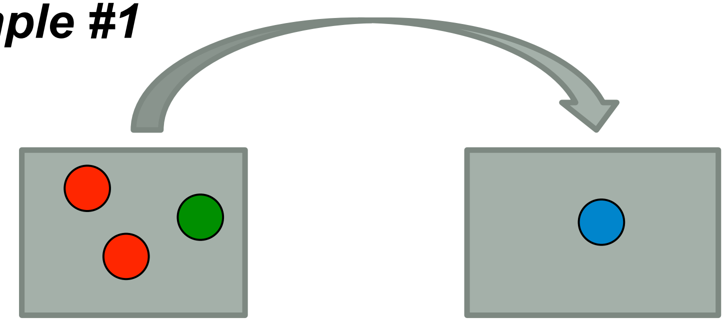
In-place Transformations change some parts in the model



Two Transformation Strategies

Out-place vs. in-place transformations

Example #1



Out-place Transformation

For each green element, create a blue element.

In-place Transformation

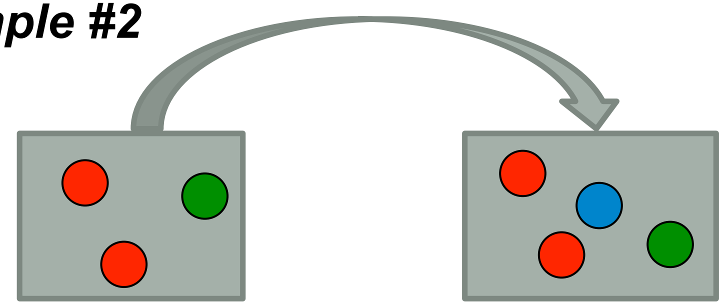
For each green element, create a blue element.

Delete all elemens except blue ones.

Two Transformation Strategies

Out-place vs. in-place transformations

Example #2



Out-place Transformation

For each green element, create a blue element.

For each green element, create a green element.

For each red element, create a red element.

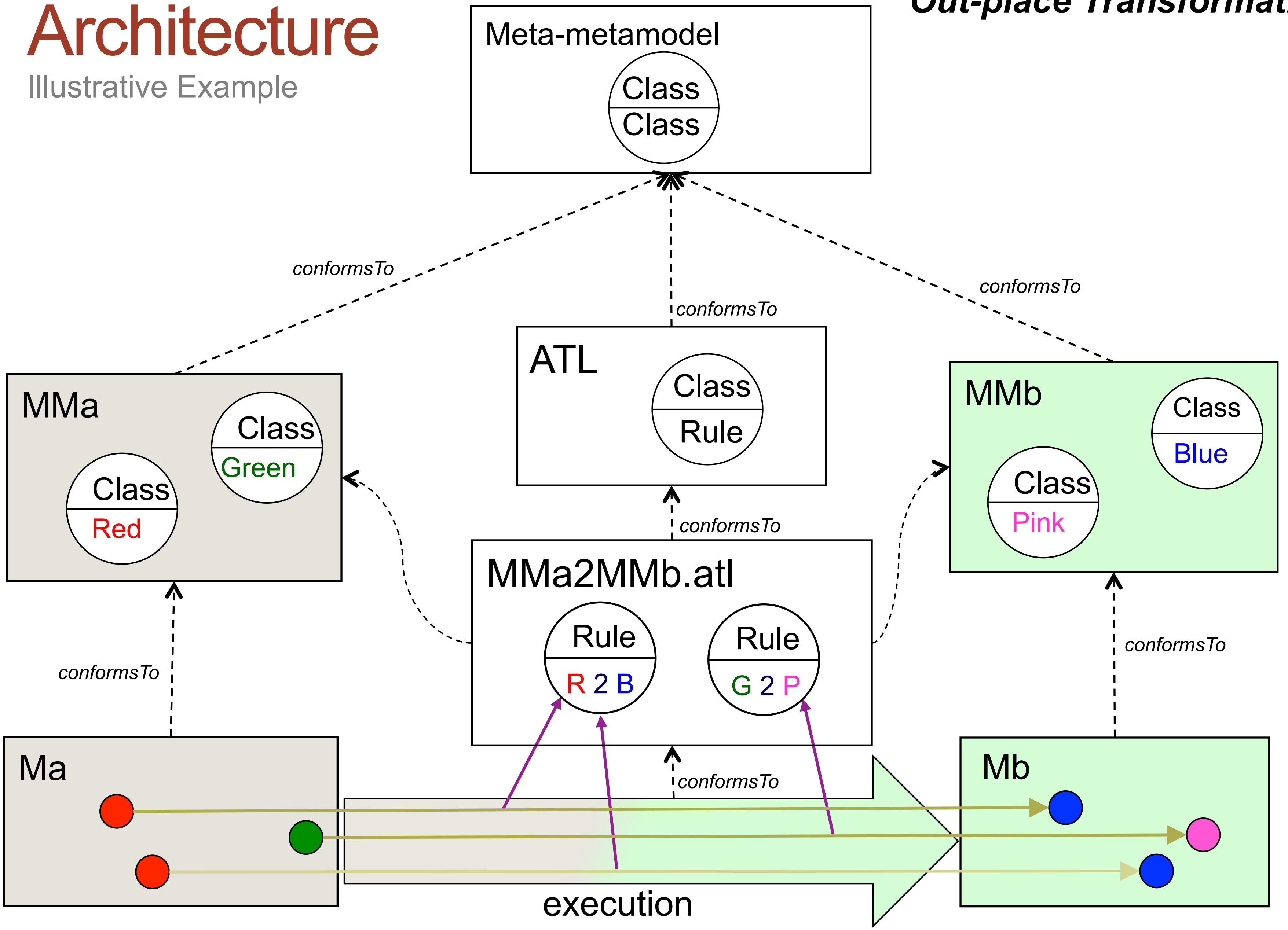
In-place Transformation

For each green element, create a blue element.

Architecture

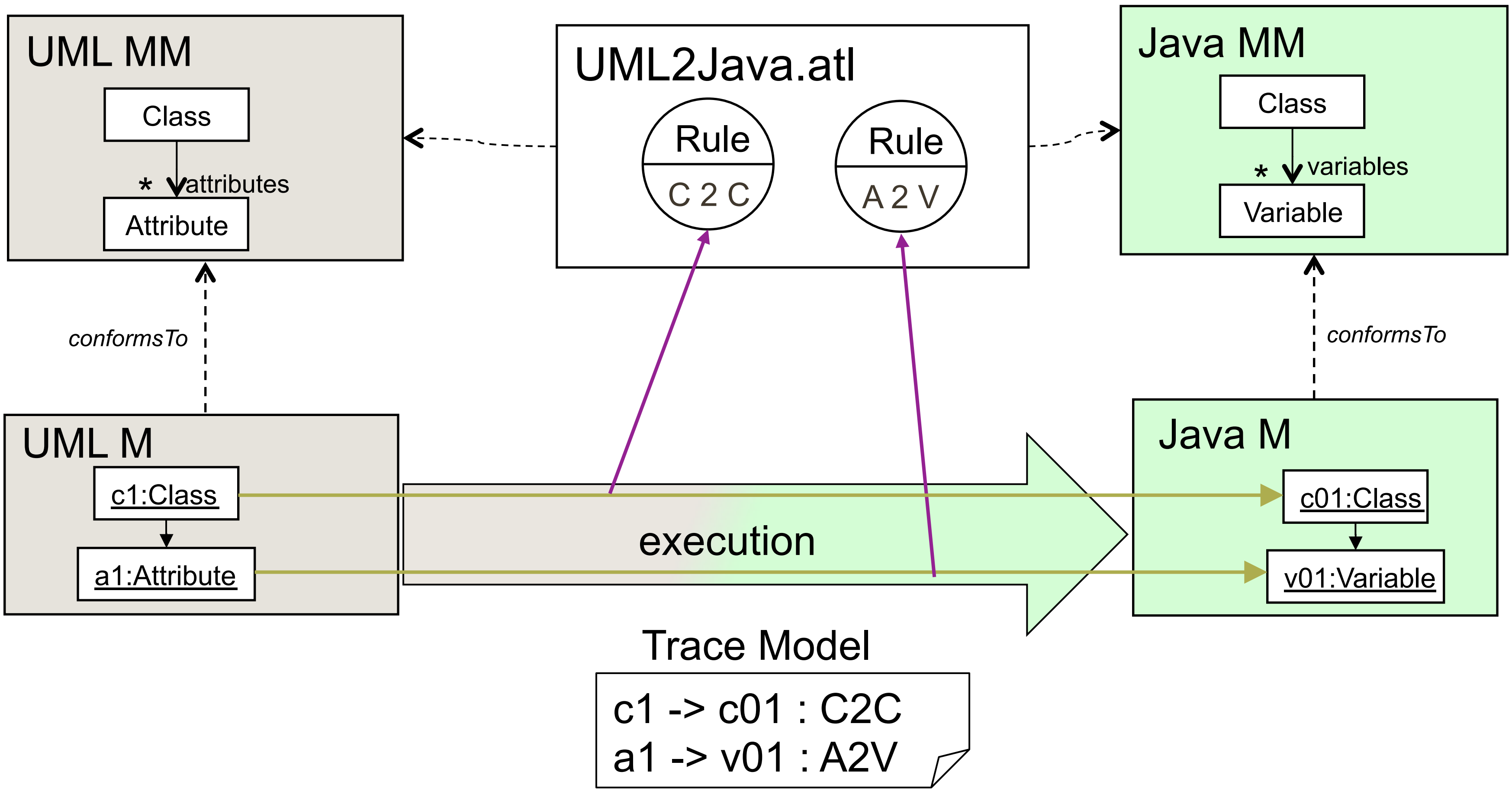
Illustrative Example

Out-place Transformations



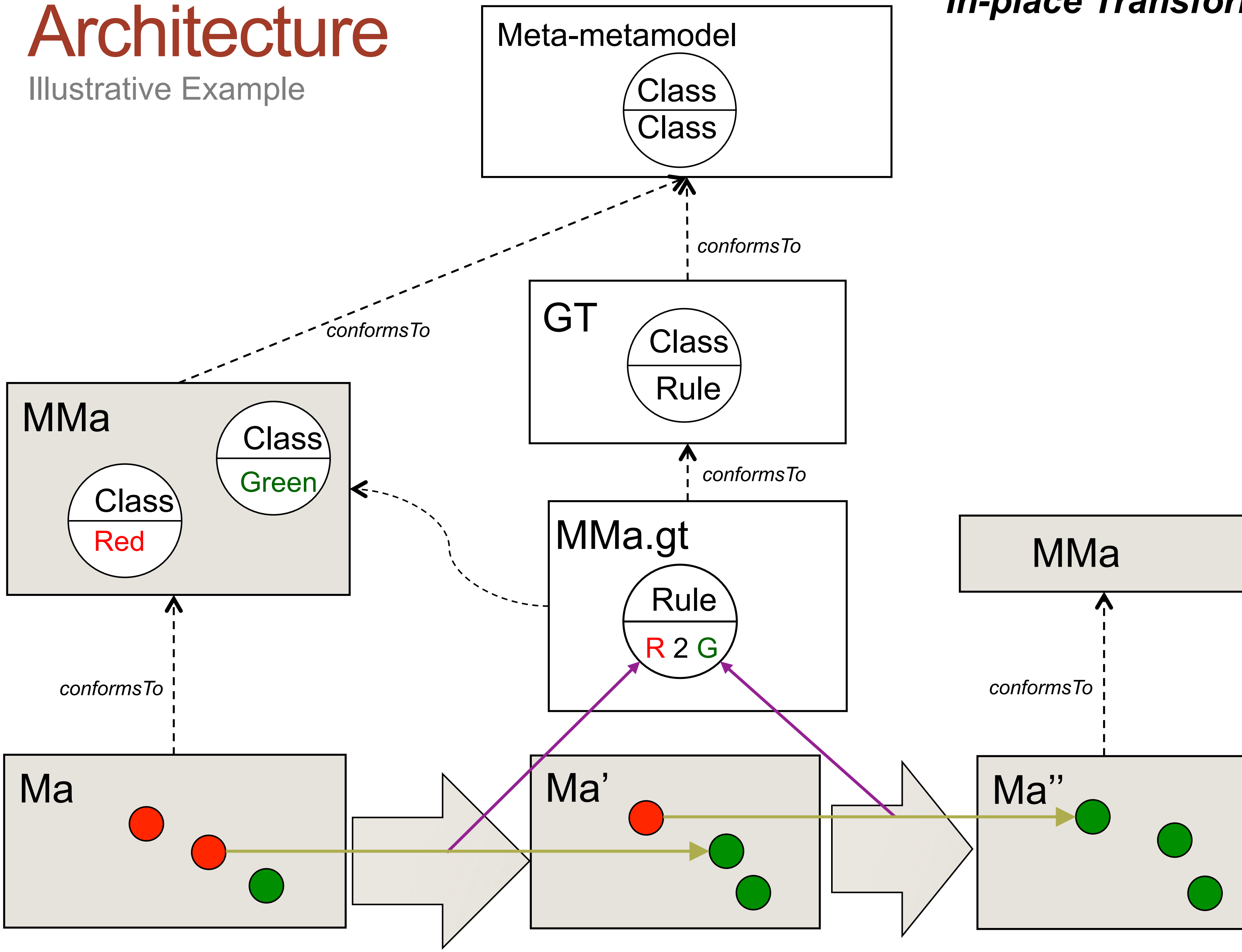
Architecture

Concrete Example



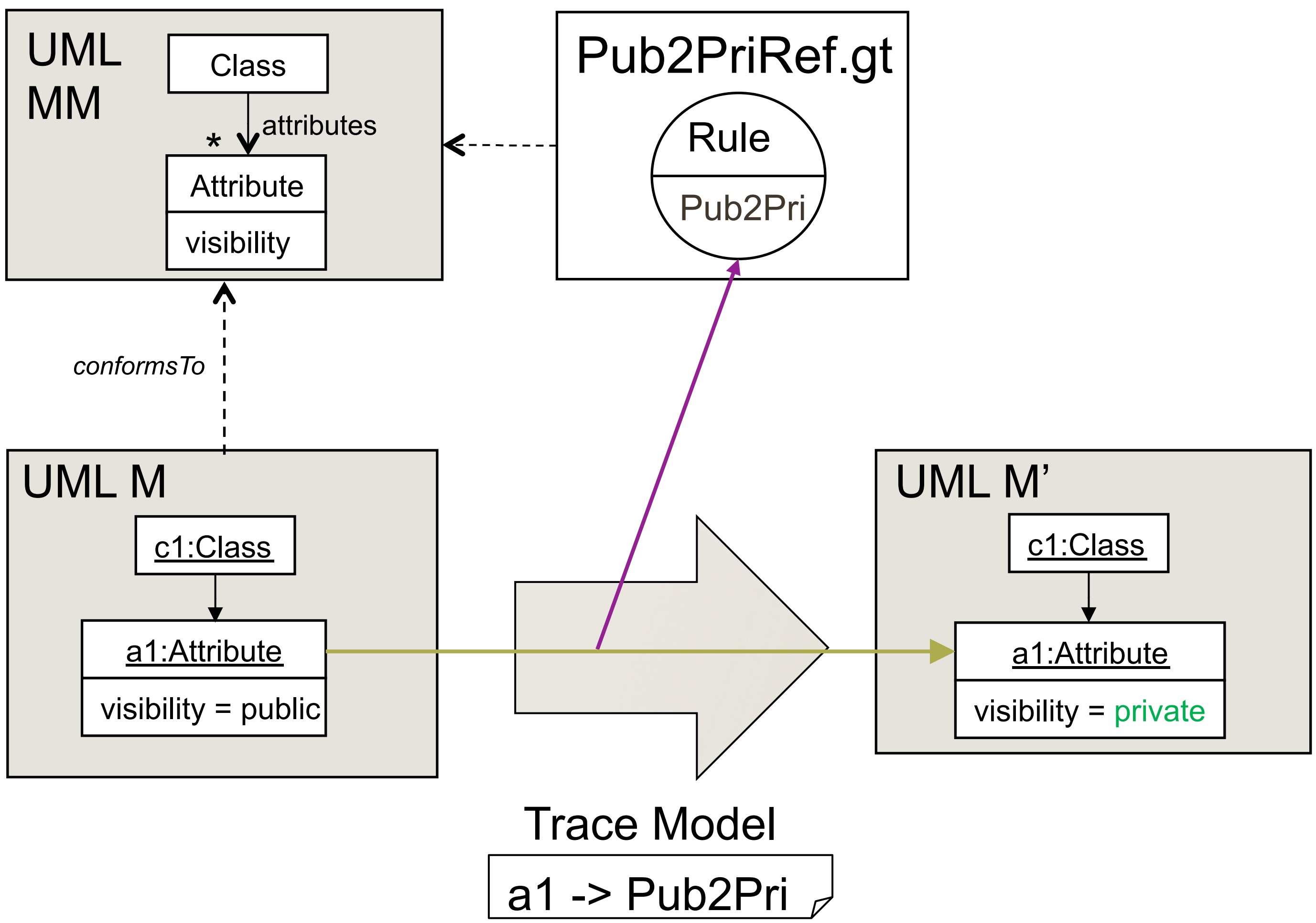
Architecture

Illustrative Example



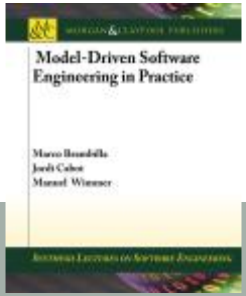
Architecture

Concrete Example



OUT-PLACE TRANSFORMATIONS:

ATLAS TRANSFORMATION LANGUAGE



ATL overview

- Source models and target models are distinct
  - **Source** models are **read-only**
  - **Target** models are **write-only**

- The language is a declarative-imperative hybrid
  - Declarative part
    - **Matched** rules with automatic traceability support
    - Side-effect free query language: **OCL**
  - Imperative part
    - **Called/Lazy** rules
    - **Action blocks**
    - **Global variables via Helpers**

- Recommended programming style: **declarative**

ATL overview (continued)

- A **declarative** rule specifies
  - A **source pattern** to be **matched** in the **source models**
  - A **target pattern** to be **created** in the **target models** for each match during rule **application**
  - An optional action block (i.e., a sequence of imperative statements)

- An **imperative** rule is basically a procedure
  - It is called by its name
  - It may take arguments
  - It contains
    - A declarative target pattern
    - An optional action block

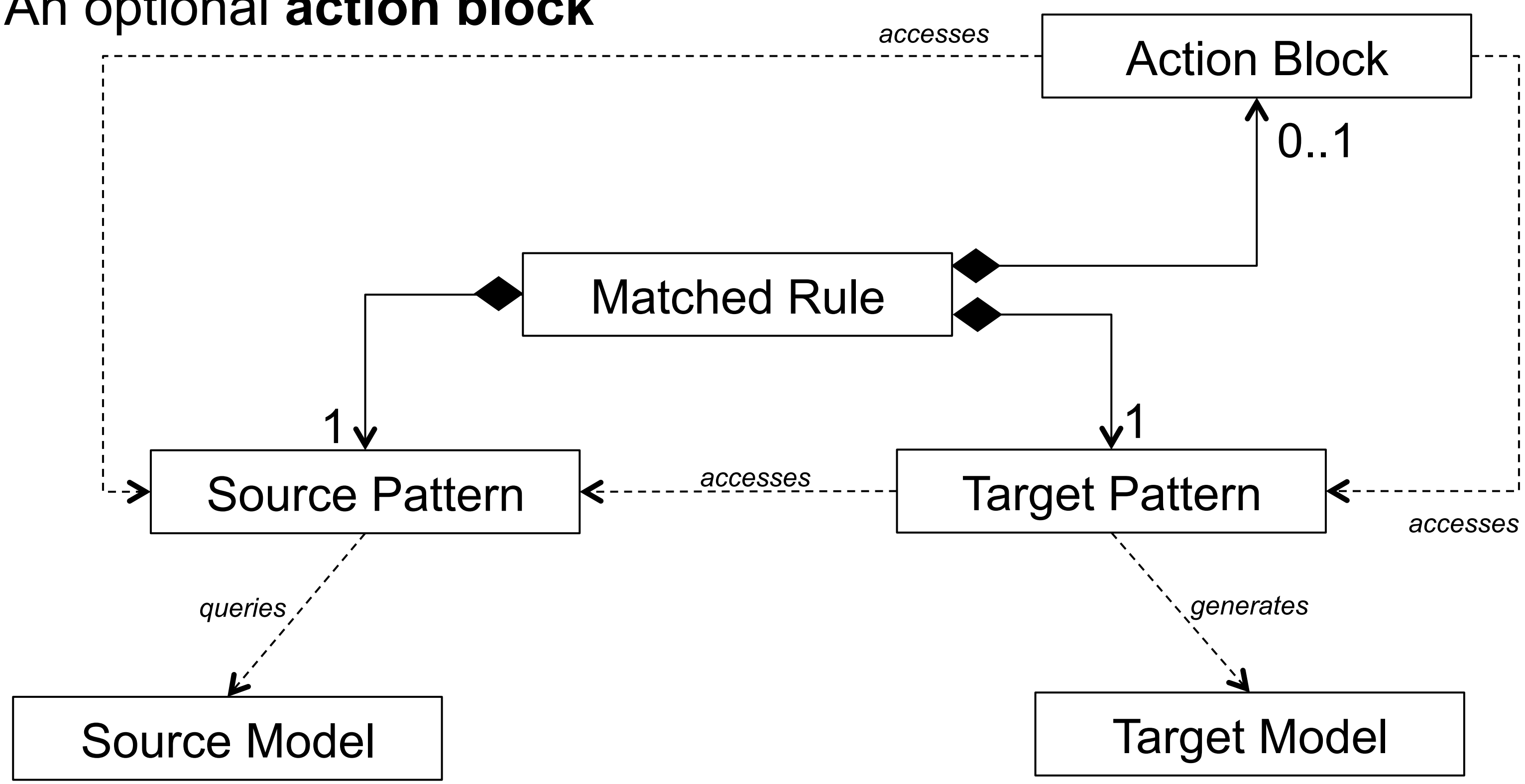
ATL overview (continued)

- **Applying** a rule means
  1. Creating the specified target elements
  2. Initializing the properties of the newly created elements

- There are **two** types of rules concerning their application
  - **Matched** rules are applied **once** for each match by the execution engine
    - A given set of elements may only be matched by one matched rule
  - **Called/Lazy** rules are applied **as many times** as they are called from other rules

Matched rules: Overview

- A **Matched rule** is composed of
  - A **source pattern**
  - A **target pattern**
  - An optional **action block**



Matched rules: source pattern

- The **source pattern** is composed of
  - A set of labeled **source pattern elements**
  - A source pattern element refers to a type contained in the source metamodels
  - A **guard** (Boolean OCL expression) used to filter matches
- A **match** corresponds to a **set of elements** coming from the source models that
  - Fulfill the types specified by the source pattern elements
  - Satisfy the guard

```
rule Rule1{
   from
          v1  : SourceMM!Type1(cond1)
   to
          v2  : TargetMM!Type1 (
              prop <- v1.prop
            )
}
```

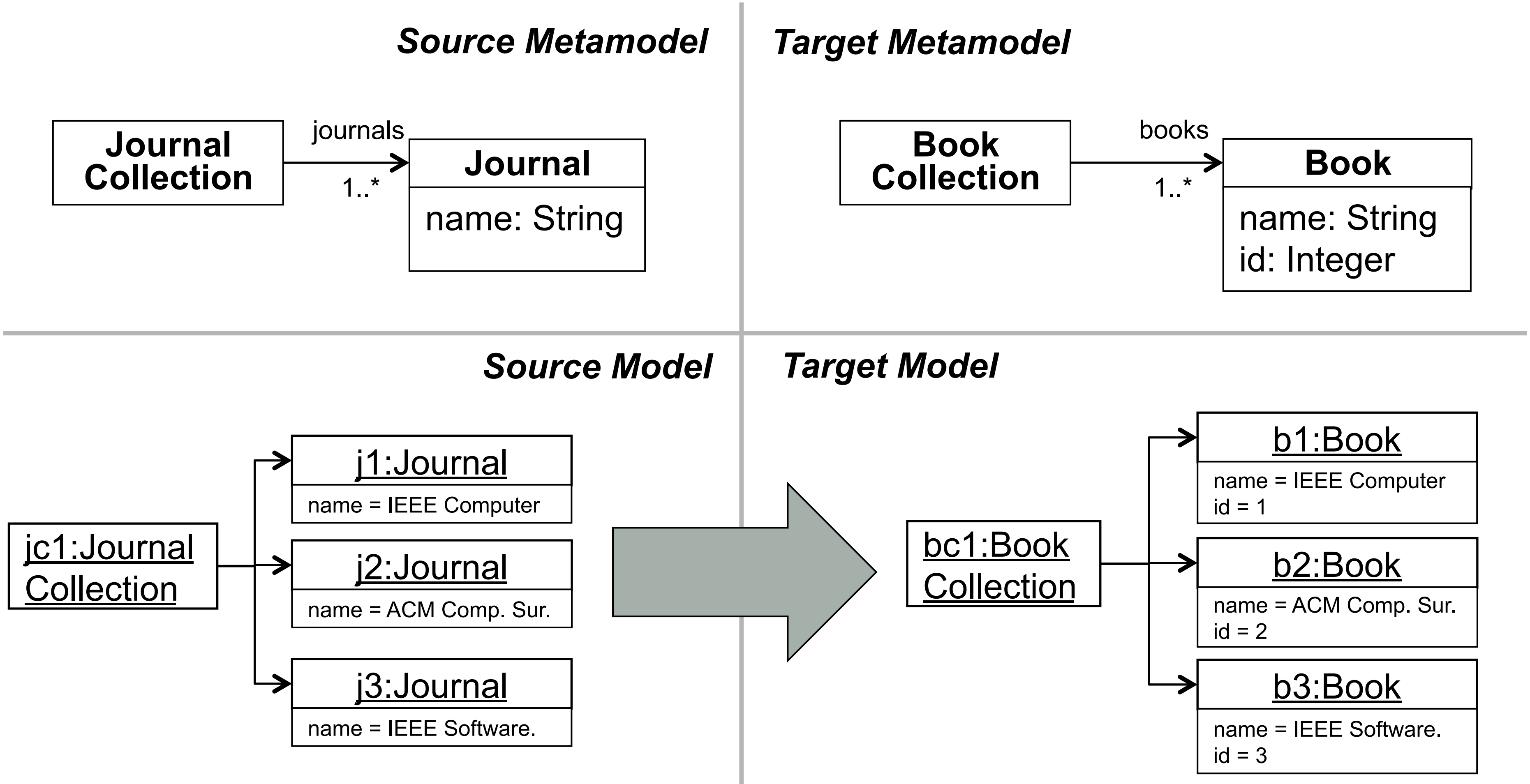
Matched rules: target pattern

- The target pattern is composed of
  - A set of labeled **target pattern elements**
  - Each target pattern element
    - refers to a type in the target metamodels
    - contains a set of bindings
  - A **binding** initializes a property of a target element using an OCL expression
- For each match, the target pattern is applied
  - Elements are created in the target models
  - Target elements are initialized by executing the bindings

```
rule Rule1{
   from
         v1 : SourceMM!Type1(cond1)
   to
         v2 : TargetMM!Type1 (
            prop <- v1.prop
          )
}
```

Example #1 – Publication 2 Book

Concepts
1) **Matched Rule**
2) **Helper**



Example #1

Configuration of Source/Target Metamodels

- **Header**

```
module Publication2Book;
create OUT : Book from IN : Publication;
```

- **For code completion**
  - `--@path MM_name =Path_to_metamodel_definition`
  - Activate code completion: CTRL + space

Example #1

Matched Rules

- **Example Publication 2 Book**

Header

```
module Publication2Book;
create OUT : Book from IN : Publication;

rule Collection2Collection {
  from
        jc : Publication!JournalCollection
  to
        bc : Book!BookCollection(
              books <- jc.journals
        )
}

rule Journal2Book {
  from
        j : Publication!Journal
  to
        b : Book!Book (
           name <- j.name
        )
}
```

Source Pattern

Target Pattern

Matched Rule

Binding

Example #1

Helpers 1/2

- **Syntax**

  `helper context` Type **`def`** **:** Name(Par1 : Type, …) : Type = *EXP;*

- **Global Variable**

  ```
  helper def: id : Integer = 0;
  ```

- **Global Operation**

  ```
  helper context Integer def : inc() : Integer = self + 1;
  ```

- **Calling a Helper**
  - `thisModule.HelperName(…)` – for global variables/operations without context
  - `value.HelperName(…)` – for global variables/operations with context

Example #1

Helpers 2/2

- **Example Publication 2 Book**

```
module Publication2Book;
create OUT : Book from IN : Publication;

helper def : id : Integer = 0;
helper context Integer def : inc() : Integer = self + 1;

rule Journal2Book {
  from
          j : Publication!Journal
  to
          b : Book!Book (
             name <- j.name
          )
  do {
           thisModule.id <- thisModule.id.inc();
           b.id <- thisModule.id;
  }
}
```

Global Variable

Global Operation

Action Block

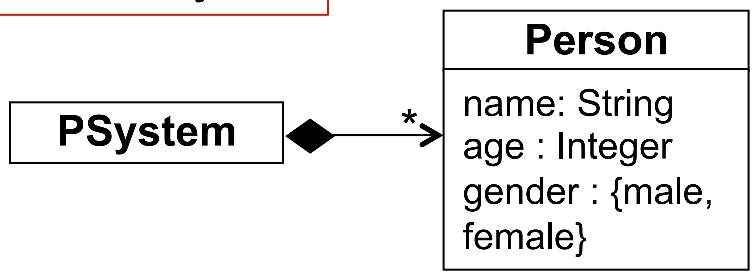
Global Operation call

Module Instance for accessing global variables

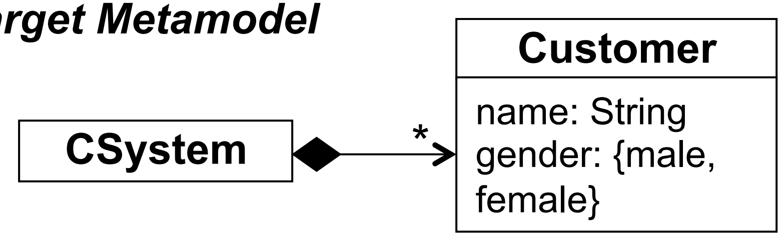
Example #2 – Person 2 Customer

Concepts:
1) Guards
2) Trace Model
3) Called/Lazy Rule

Source Metamodel



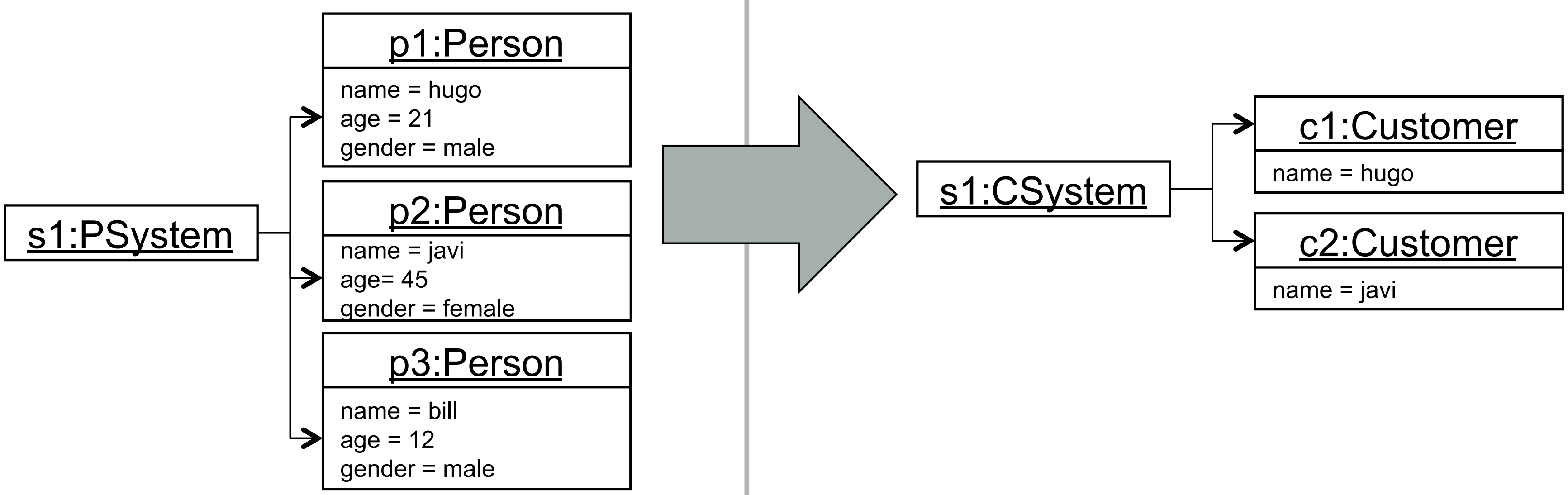
Target Metamodel



Constraint: Customer must be older than 18 years

Source Model

Target Model



Example #2

Describing more complex correspondences and bindings

- **Declarative Statements**
  - If/Else, OCL Operations, Global Operations, …
  - Application: Guard Condition and Feature Binding

- **Example: IF/ELSE**

```
if condition then
  exp1
else
  exp2
endif
```

Example #2

Guards Conditions in Source Patterns

- **Example Person2Customer**

Guard Condition

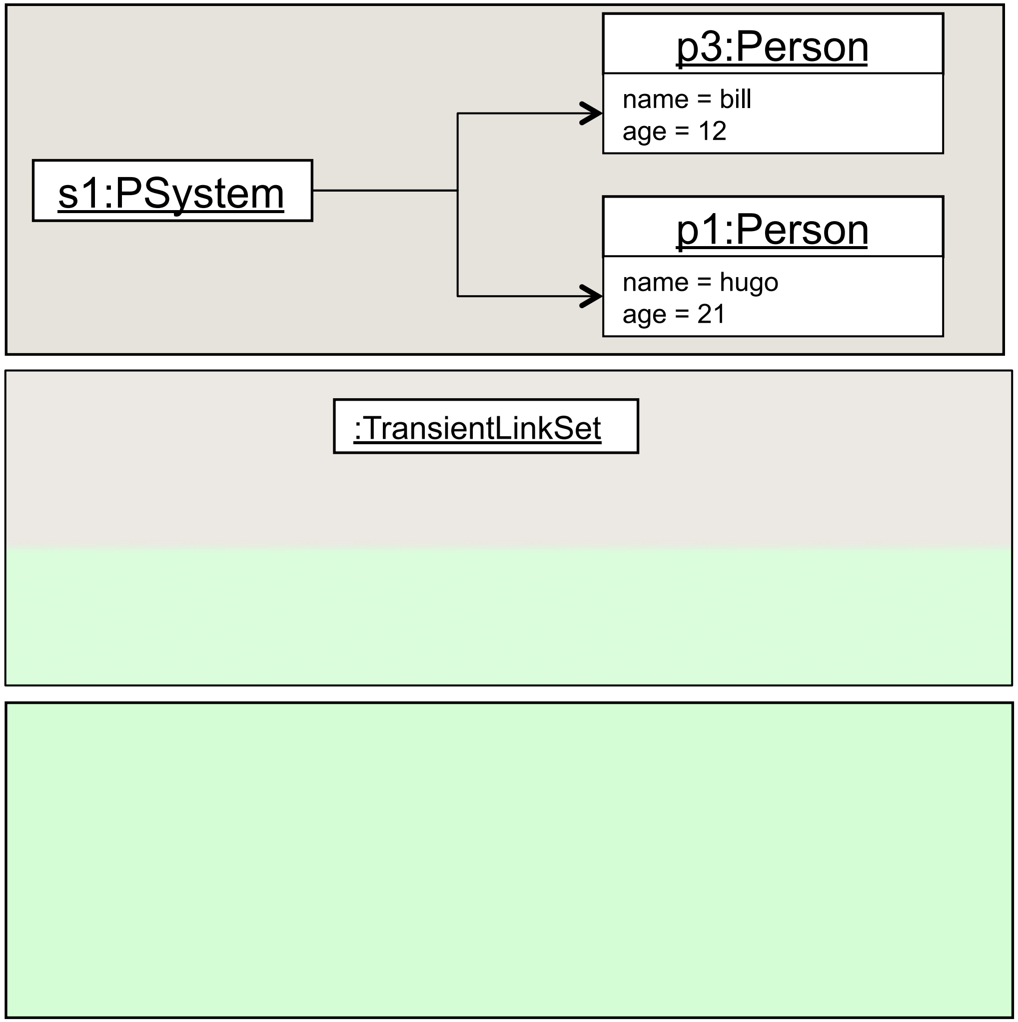
```
rule Person2Customer {
  from
        p : Person!Person (p.age > 18)
  to
        c : Customer!Customer (
            name <- p.name
        )
}

rule PSystem2CSystem {
  from
        ps : Person!PSystem
  to
        cs : Customer!CSystem (
            customer <- ps.person -> select(p | p.age > 18)
        )
}
```

Compute Subset for Feature Binding

Example #2

Implicit Trace Model – Phase 1: Module Initialization Phase



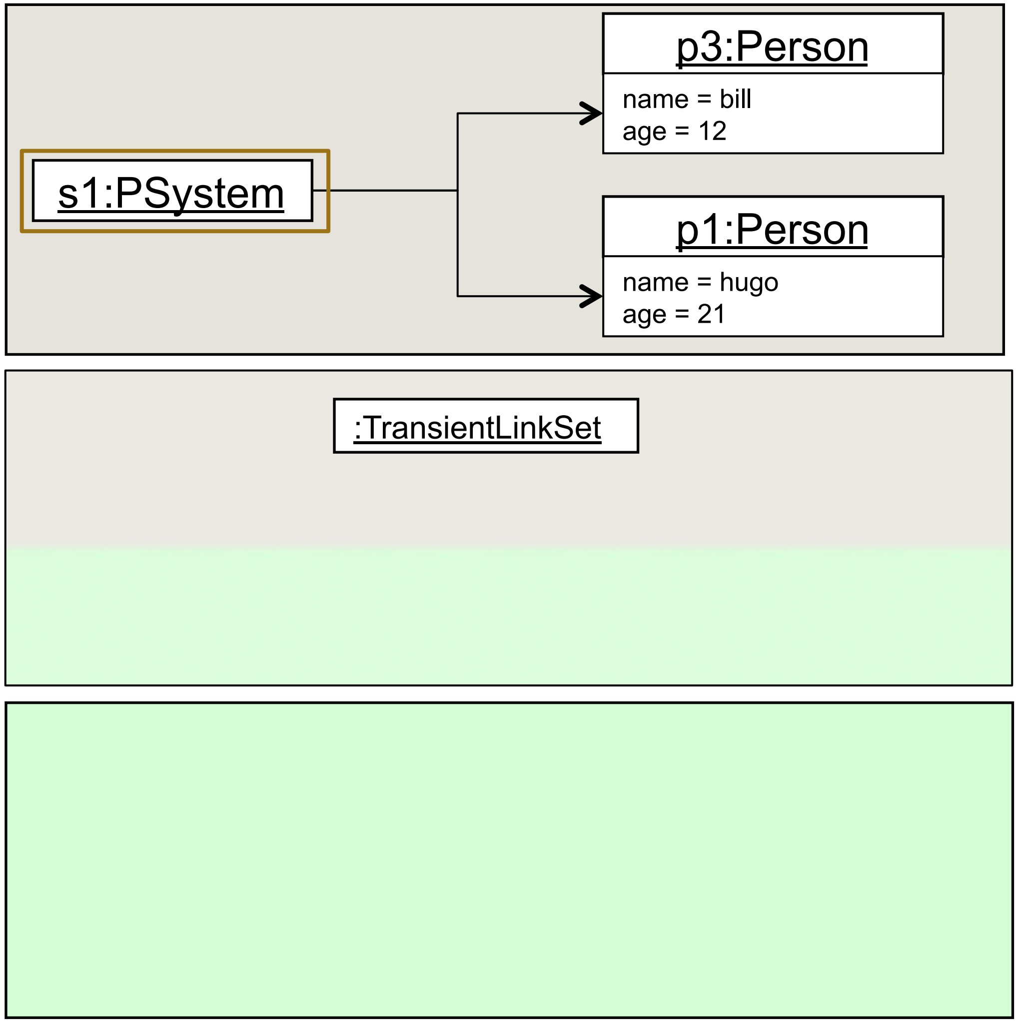

```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
          )
}


rule Person2Customer {
  from
          p : Person!Person(p.age > 18)
  to
          c : Customer!Customer (
             name <- p.name
          )
}
```

Trace Model is a set (cf. TransientLinkSet) of **links** (cf. TransientLink) that relate **source** elements with their corresponding **target** elements

Example #2

Implicit Trace Model – Phase 2: Matching Phase

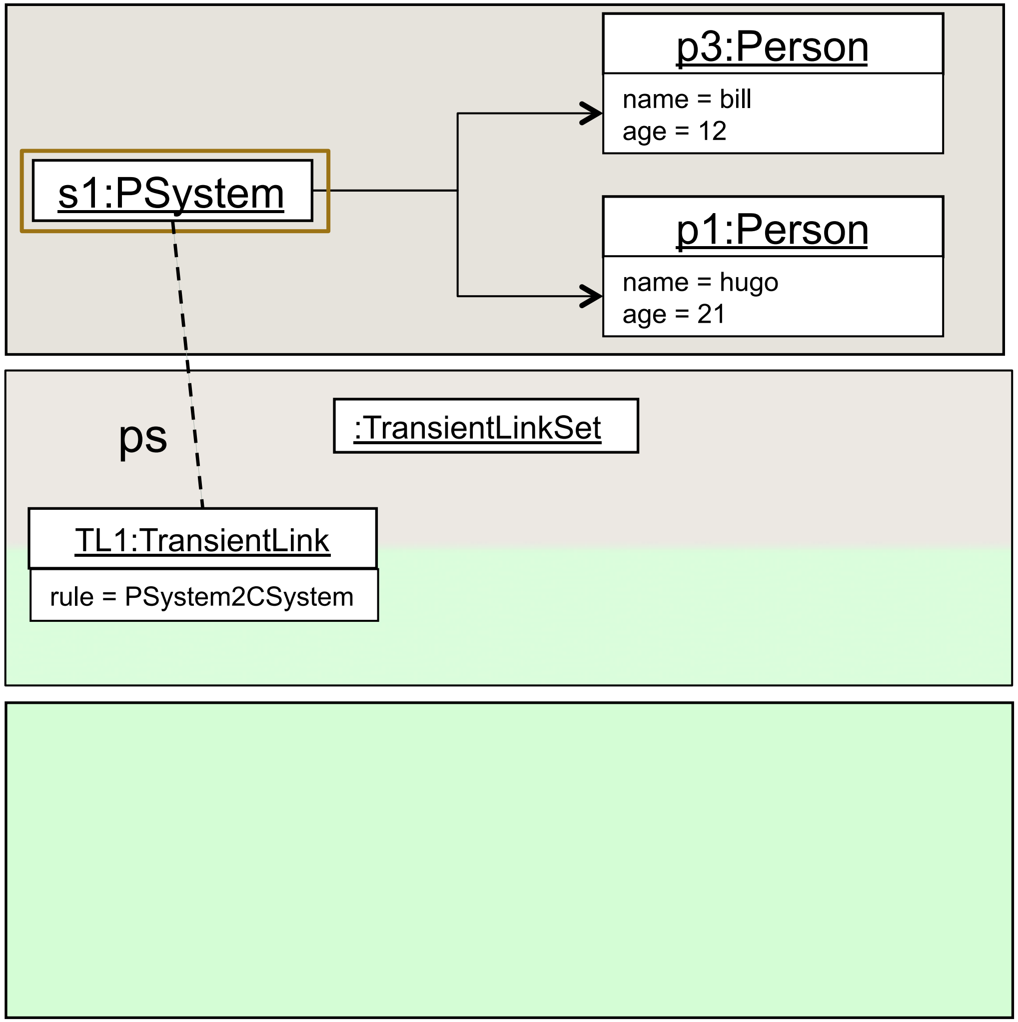



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
          )
}


rule Person2Customer {
  from
          p : Person!Person(p.age > 18)
  to
          c : Customer!Customer (
             name <- p.name
          )
}
```

Example #2

Implicit Trace Model – Phase 2: Matching Phase

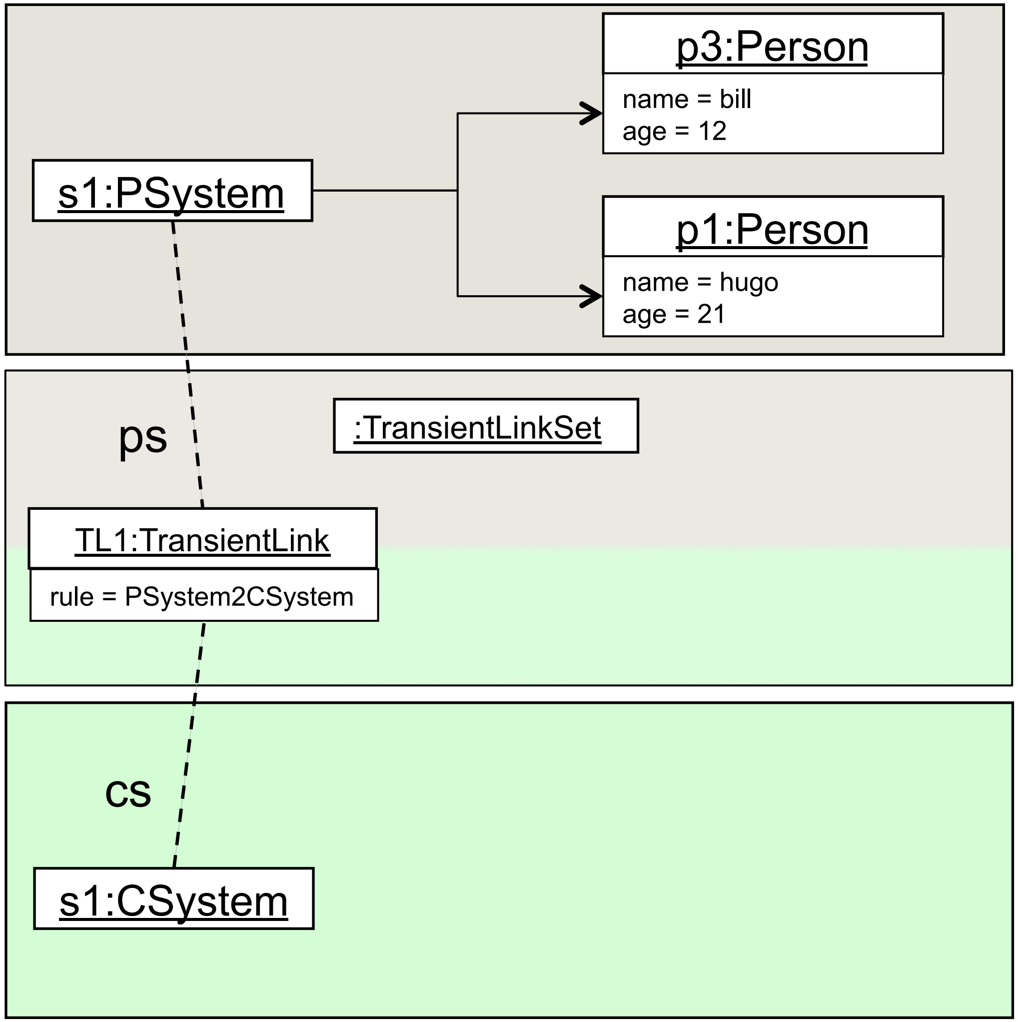



```
rule PSystem2CSystem {
  from
           ps : Person!PSystem
  to
           cs : Customer!CSystem (
              customer <- ps.person
           )
}


rule Person2Customer {
  from
           p : Person!Person(p.age > 18)
  to
           c : Customer!Customer (
              name <- p.name
           )
}
```

Example #2

Implicit Trace Model – Phase 2: Matching Phase

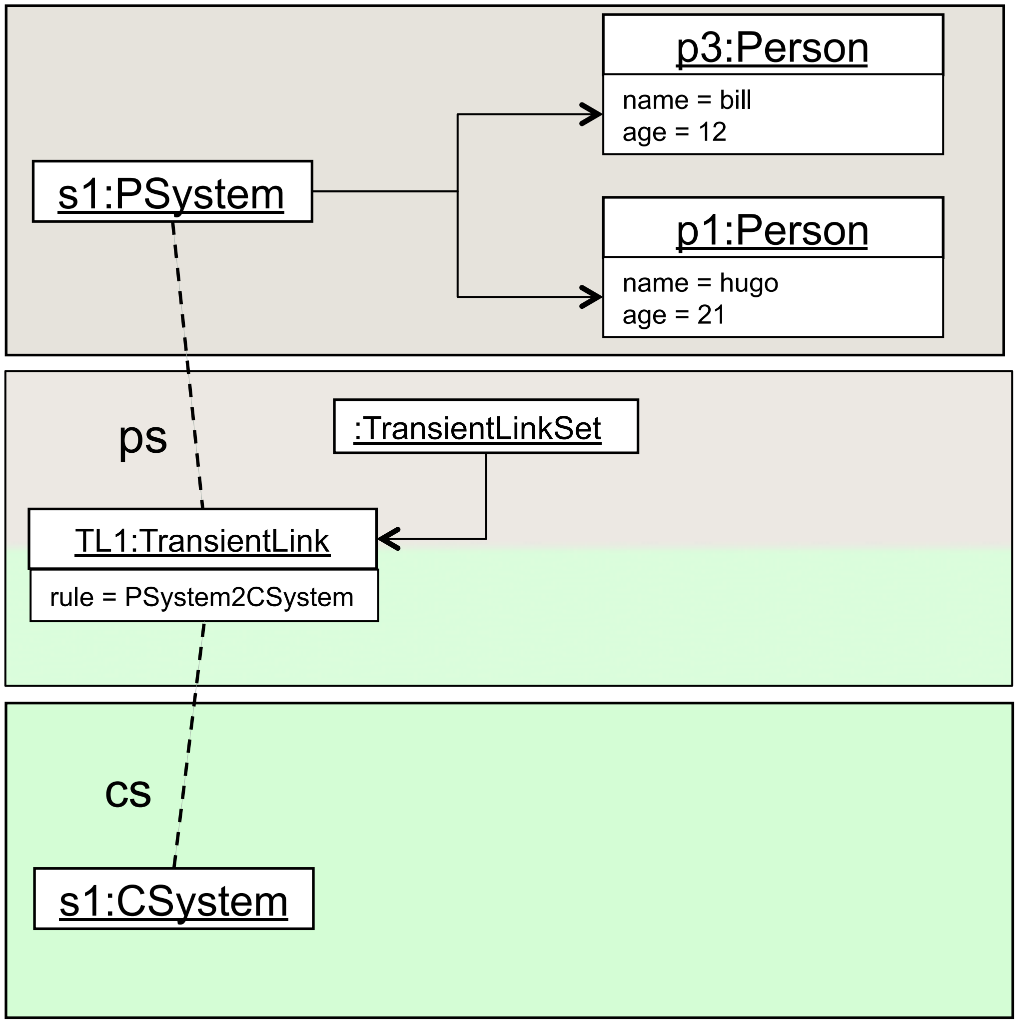



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
          )
}


rule Person2Customer {
  from
          p : Person!Person(p.age > 18)
  to
          c : Customer!Customer (
             name <- p.name
          )
}
```

Example #2

Implicit Trace Model – Phase 2: Matching Phase

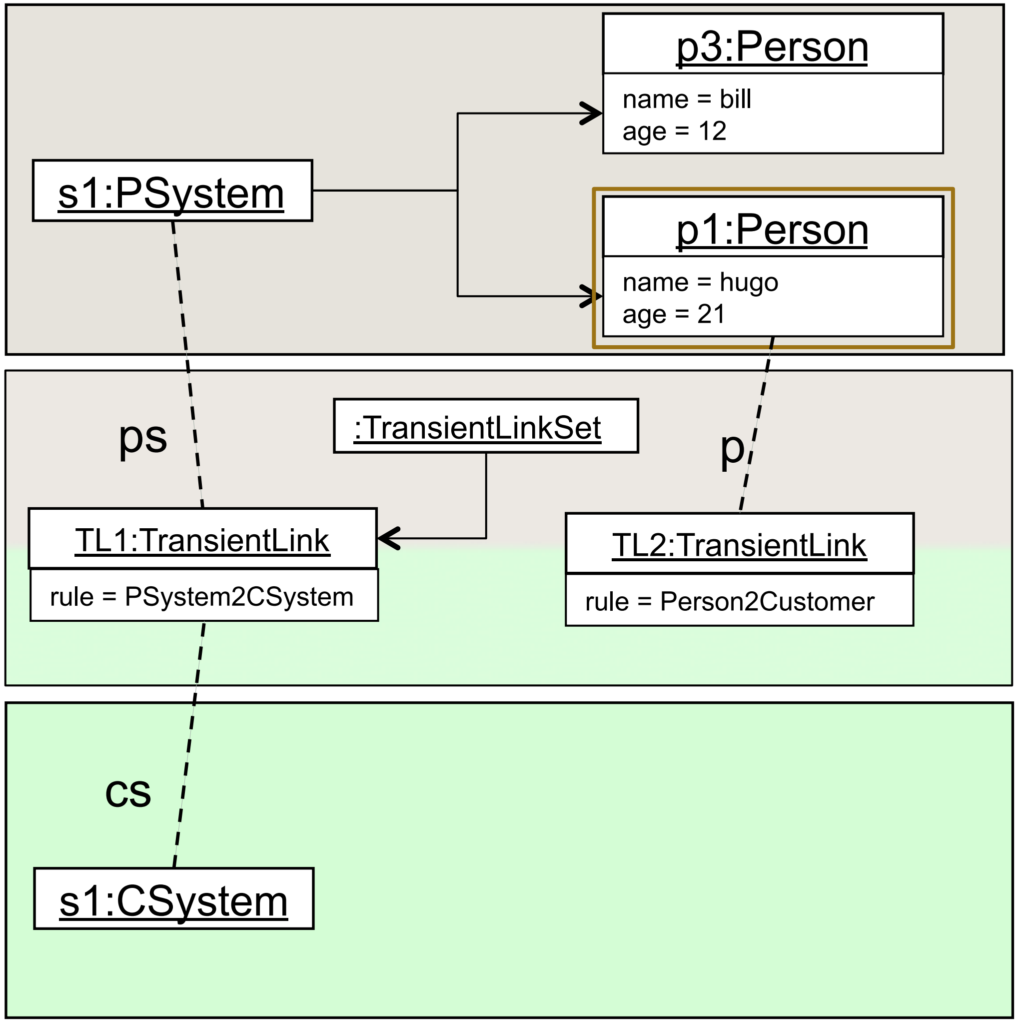



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
          )
}


rule Person2Customer {
  from
          p : Person!Person(p.age > 18)
  to
          c : Customer!Customer (
             name <- p.name
          )
}
```

Example #2

Implicit Trace Model – Phase 2: Matching Phase

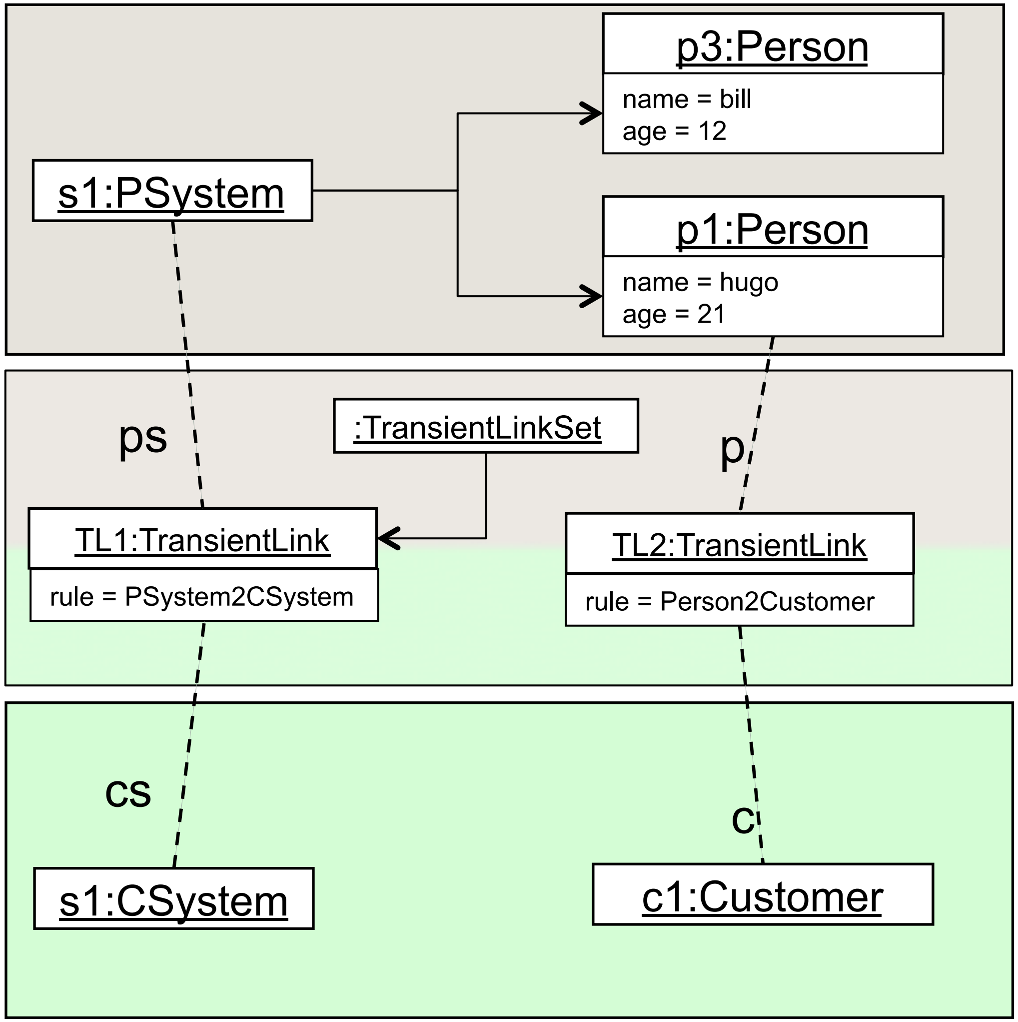



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
          )
}


rule Person2Customer {
  from
          p : Person!Person(p.age > 18)
  to
          c : Customer!Customer (
             name <- p.name
          )
}
```

Example #2

Implicit Trace Model – Phase 2: Matching Phase

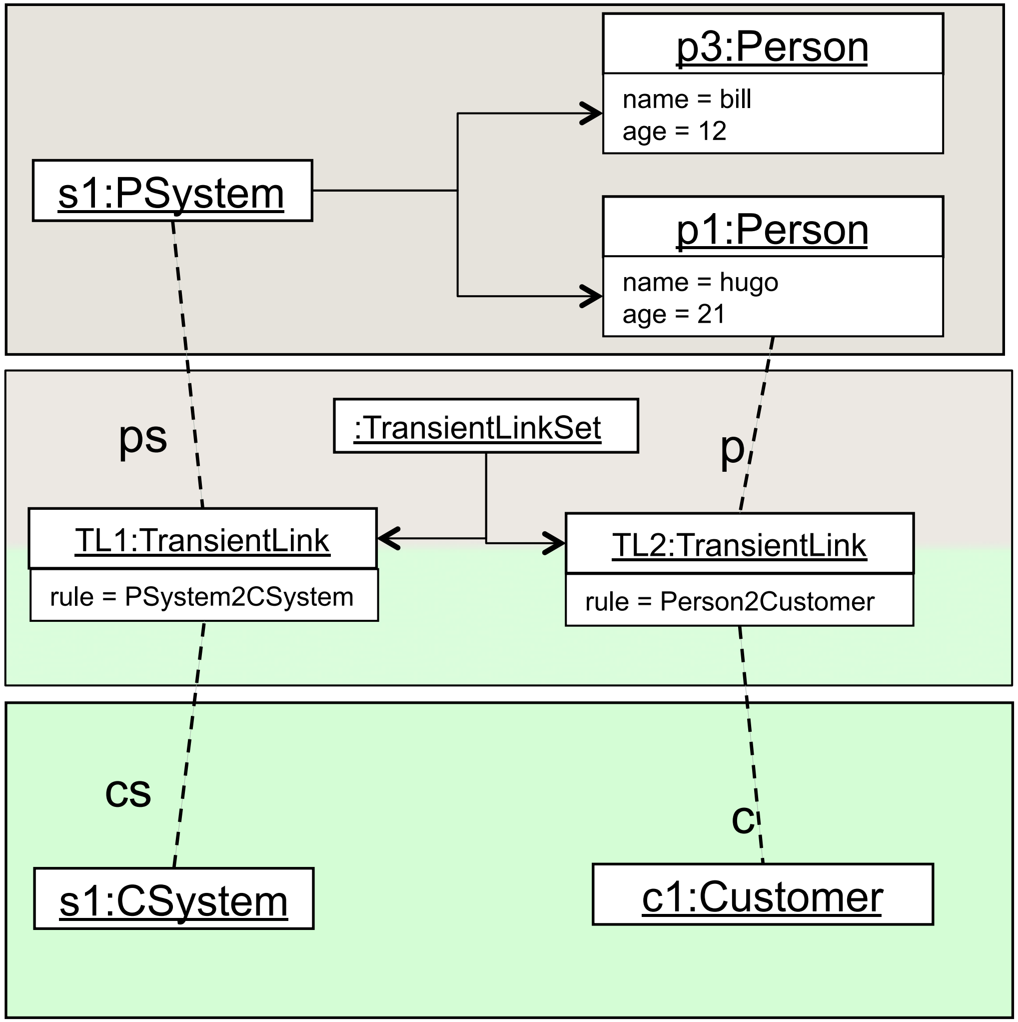



```
rule PSystem2CSystem {
  from
        ps : Person!PSystem
  to
        cs : Customer!CSystem (
           customer <- ps.person
        )
}


rule Person2Customer {
  from
        p : Person!Person(p.age > 18)
  to
        c : Customer!Customer (
           name <- p.name
        )
}
```

Example #2

Implicit Trace Model – Phase 2: Matching Phase

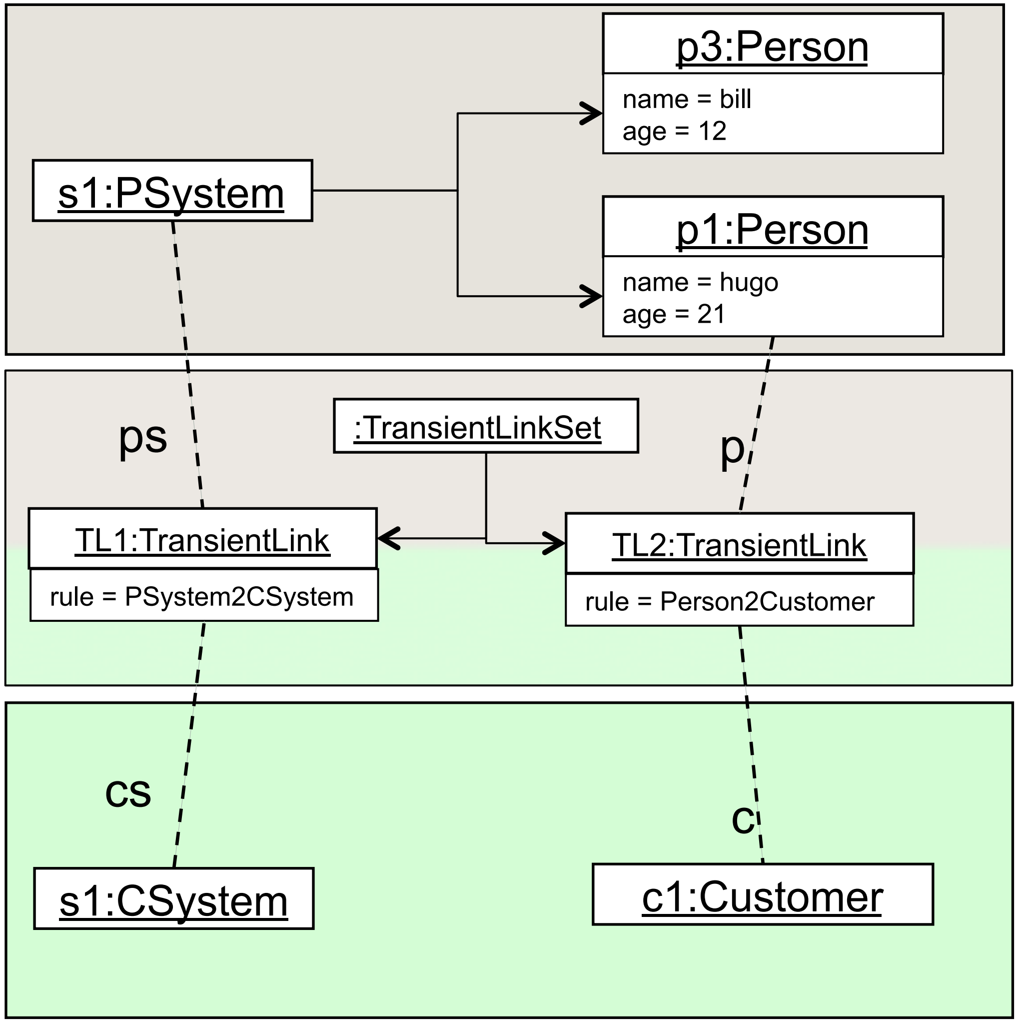



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
          )
}


rule Person2Customer {
  from
          p : Person!Person(p.age > 18)
  to
          c : Customer!Customer (
             name <- p.name
          )
}
```

Example #2

Implicit Trace Model – Phase 3: Target Initialization Phase

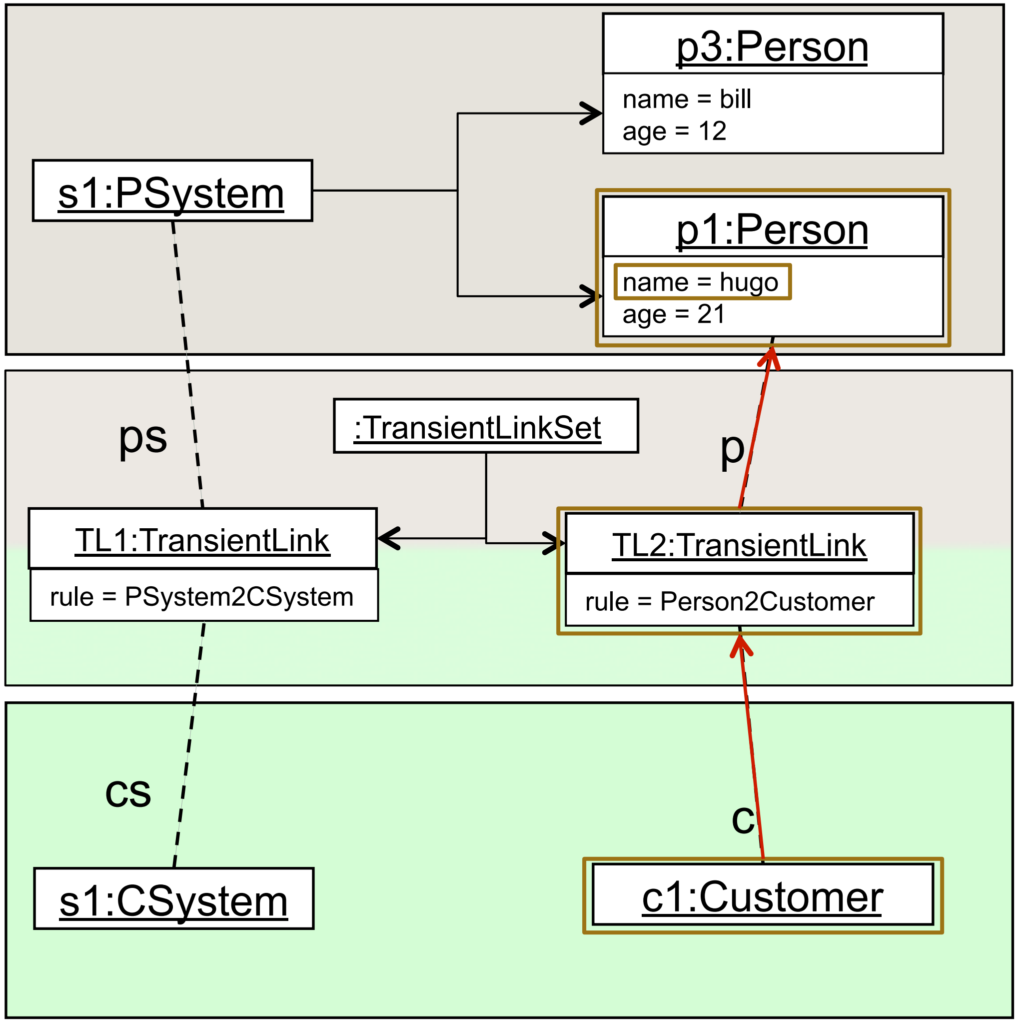



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
          )
}


rule Person2Customer {
  from
          p : Person!Person(p.age > 18)
  to
          c : Customer!Customer (
             name <- p.name
          )
}
```

Example #2

Implicit Trace Model – Phase 3: Target Initialization Phase

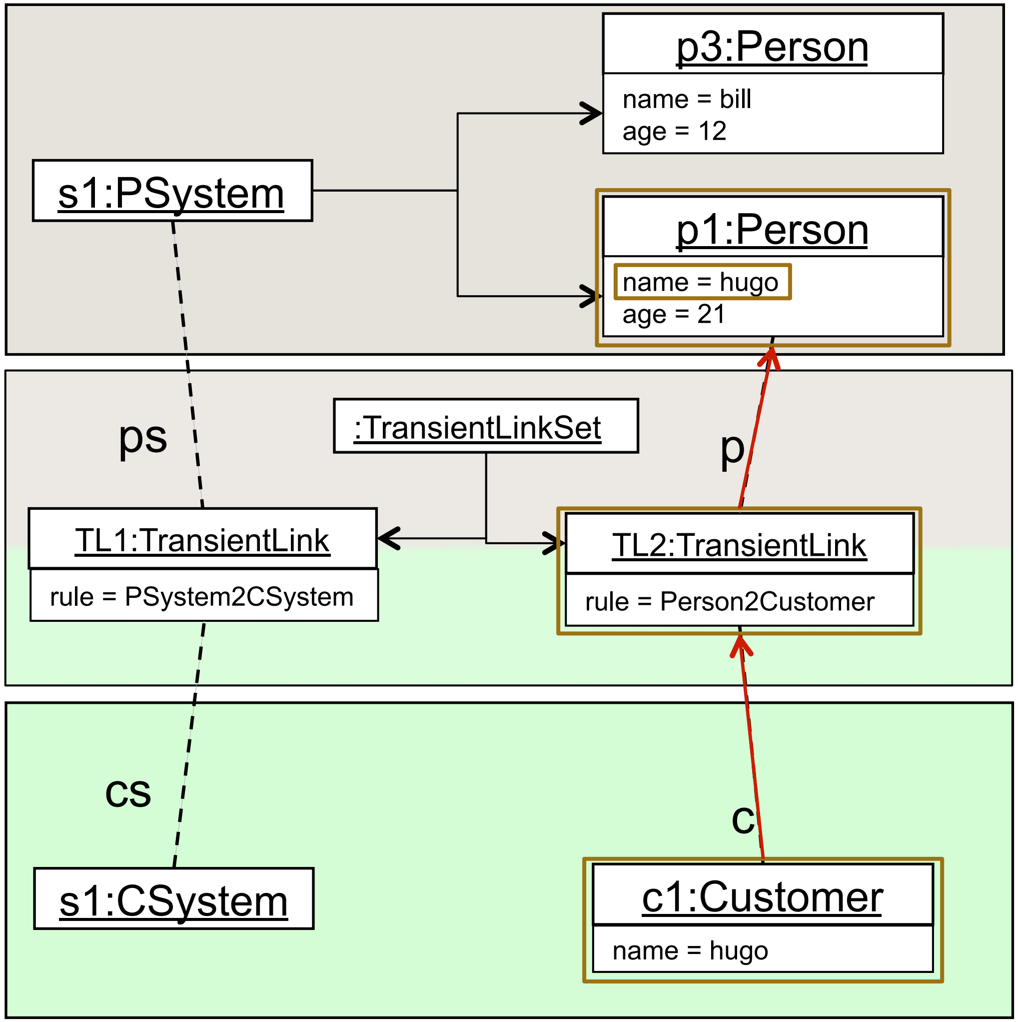



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
          )
}


rule Person2Customer {
  from
          p : Person!Person(p.age > 18)
  to
          c : Customer!Customer (
             name <- p.name
          )
}
```

Example #2

Implicit Trace Model – Phase 3: Target Initialization Phase

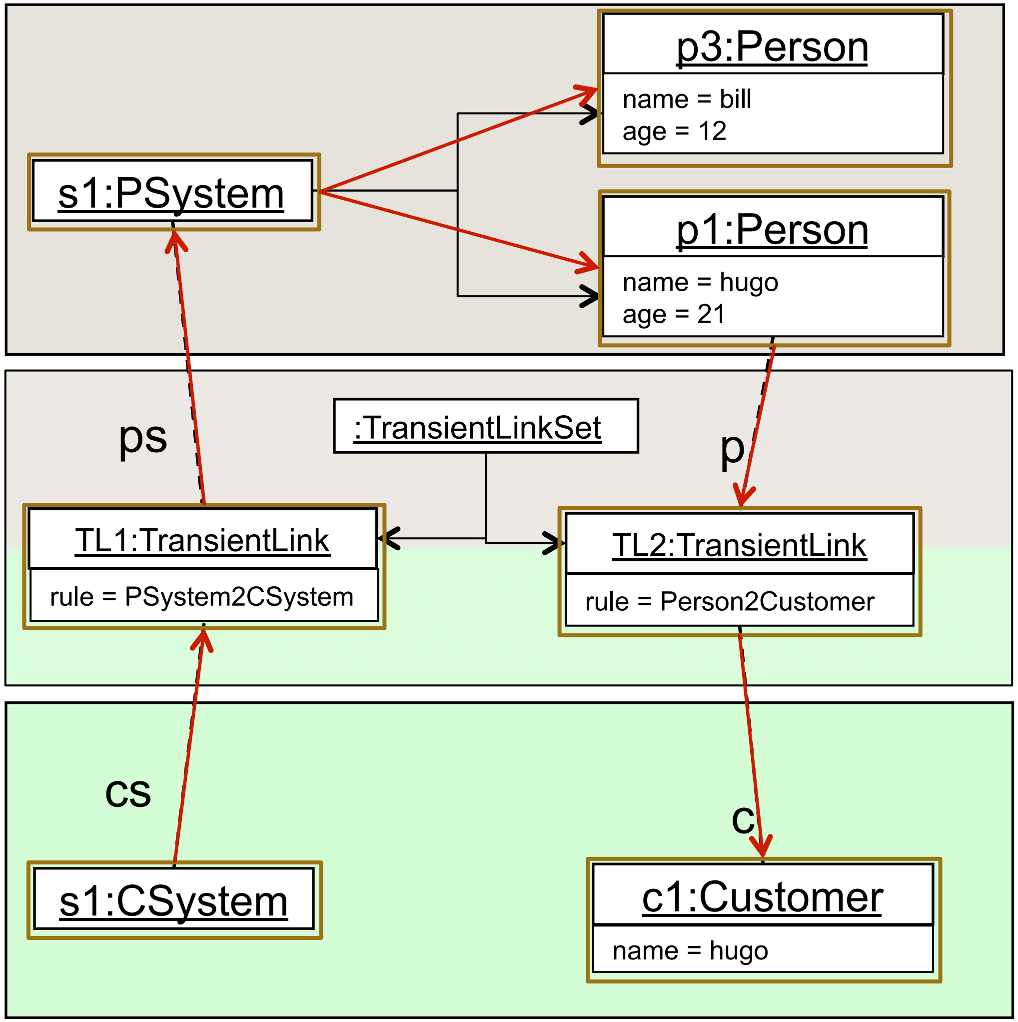



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
          )
}


rule Person2Customer {
  from
          p : Person!Person(p.age > 18)
  to
          c : Customer!Customer (
             name <- p.name
          )
}
```

Example #2

Implicit Trace Model – Phase 3: Target Initialization Phase

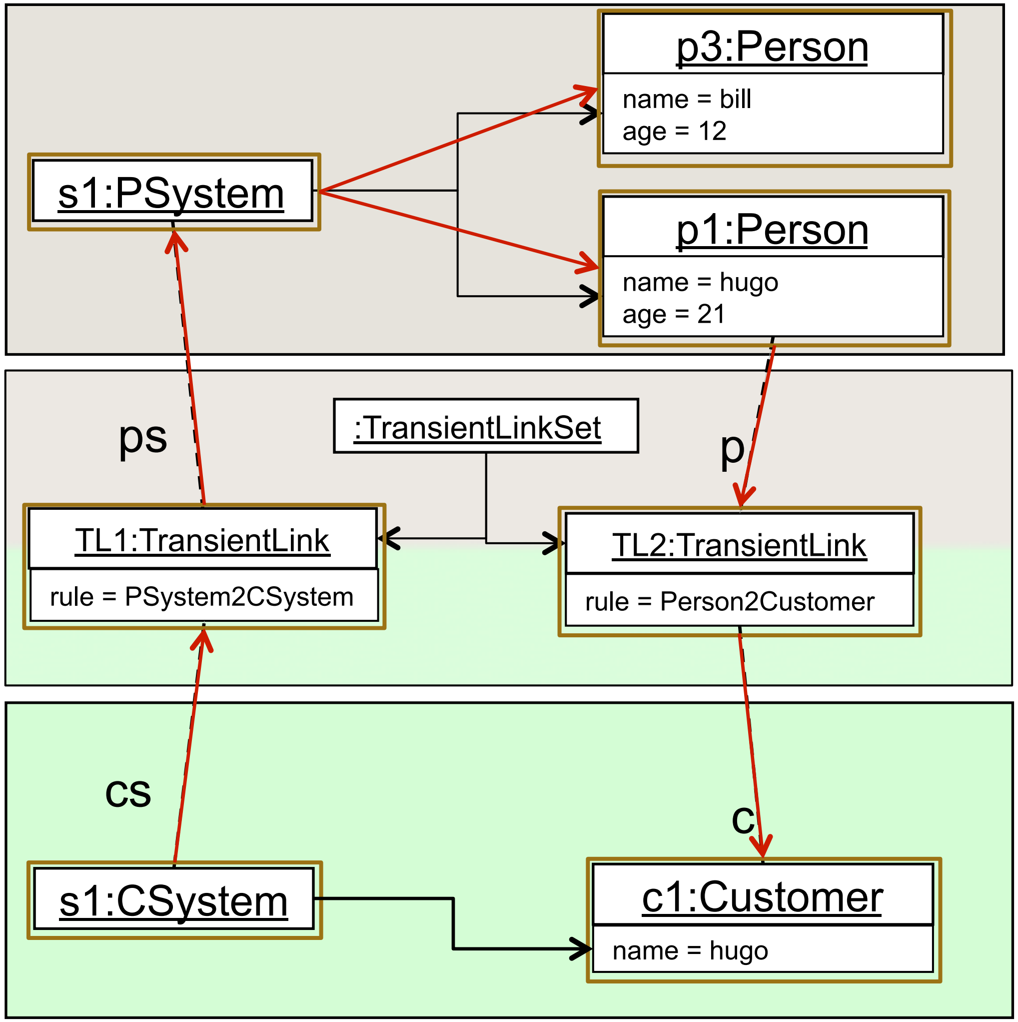



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
          )
}


rule Person2Customer {
  from
          p : Person!Person(p.age > 18)
  to
          c : Customer!Customer (
             name <- p.name
          )
}
```

Example #2

Implicit Trace Model – Phase 3: Target Initialization Phase




```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
          )
}

rule Person2Customer {
  from
          p : Person!Person(p.age > 18)
  to
          c : Customer!Customer (
             name <- p.name
          )
}
```

Transformation Execution Phases

1. **Module Initialization** Phase
   - Module variables (attribute helpers) and trace model are initialized
   - If an **entry point** *called rule* is defined, it is executed in this step
2. **Matching** Phase
   - Using the source patterns (**from**) of *matched rules,* elements are selected in the source model (each match has to be **unique**)
   - Via the *target patterns* (**to**) corresponding elements are created in the target model (for each match there are as much target elements created as target patterns are used)
   - Traceability information is stored
3. **Target Initialization** Phase
   - The elements in the target model are initialized based on the *bindings* (**<-**)
   - The resolveTemp function is evaluated, based on the traceability information
   - Imperative code (**do**) is executed, including possible calls to *called rules*

Example #2

Alternative Solution with Called Rule and Action Block (1/3)

Imperative Statements in Action Blocks (do)

IF/[ELSE]

```
if( aPerson.gender = #male )
  thisModule.men->including(aPerson);
else
  thisModule.women->including(aPerson);
```

FOR

```
for( p in Person!Person.allInstances() ) {

     if(p.gender = #male)
               thisModule.men->including(p);
     else
               thisModule.women->including(p);
}
```

Example #2

Alternative Solution with Called Rule and Action Block (2/3)

```
rule PSystem2CSystem {
  from
        ps : Person!PSystem
  to
        cs : Customer!CSystem ()

  do{
           for( p in Person!Person.allInstances() ) {
             if(p.age > 18)
                cs.customer <- thisModule.NewCustomer(p.name,
                                                 p.gender);
           }
  }
}
```

Matched Rule

Explicit Query on Source Model

Action Block

Binding

Called Rule Call

Example #2

Alternative Solution with Called Rule and Action Block (3/3)

Called Rule

```
rule NewCustomer (name: String, gender: Person::Gender){
  to
      c : Customer!Customer (
        c.name <- name
  )
  do {
      c.gender <- gender;
      c;
  }
}
```

Target Pattern

Action Block

Result of Called Rules is the last statement executed in the Action Block

Example #2

Alternative Solution with Lazy Rule and Action Block (1/2)

```
rule PSystem2CSystem {
  from
        ps : Person!PSystem
  to
        cs : Customer!CSystem (
   )
   do{
          for( p in Person!Person.allInstances() ) {
            if(p.age > 18)
               cs.customer <- thisModule.NewCustomer(p);
          }
   }
}
```

Matched Rule

Explicit Query on Source Model

Action Block

Binding

Lazy Rule Call

Example #2

Alternative Solution with Lazy Rule and Action Block (2/2)

Lazy Rule

```
lazy rule NewCustomer{
  from
      p : Person!Person
  to
      c : Customer!Customer (
        name <- p.name,
        gender <- p.gender
      )
}
```

Source Pattern

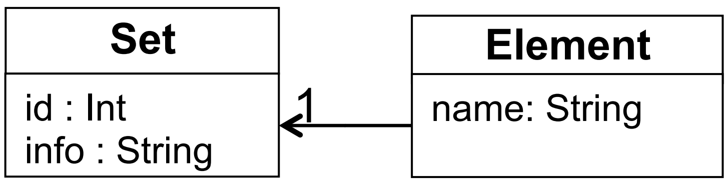
Target Pattern

Result of Lazy Rules is the first specified target pattern element

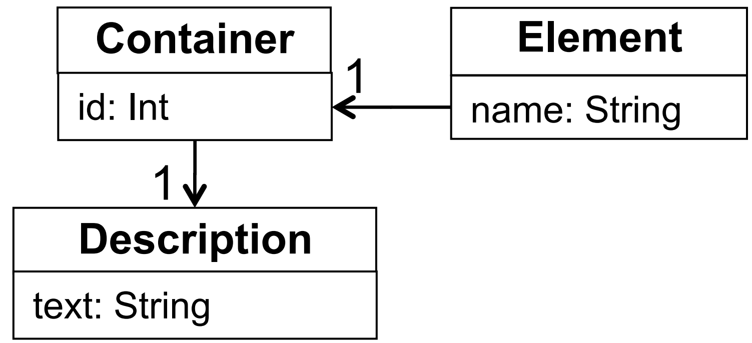
Example #3

Resolve Temp – Explicitly Querying Trace Models (1/4)

Source Metamodel



Target Metamodel



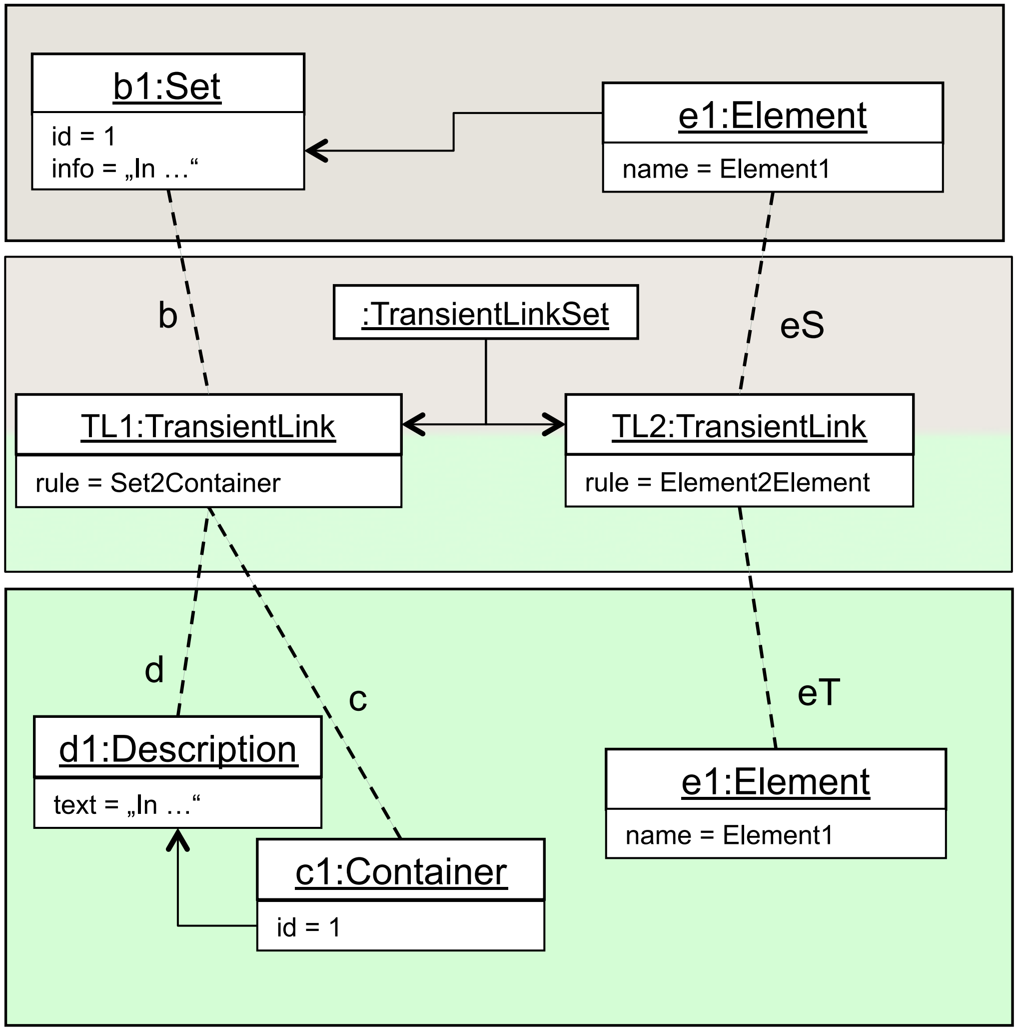
Transformation

```
rule Set2Container {
   from
      b : source!Set
   to
      d : target!Description(
         text <- b.info
      ),
      c : target!Container(
         id <- b.id,
         description <- d
      )
}
```

```
rule Element2Element{
   from
      eS : source!Element
   to
      eT : target!Element (
         name <- eS.name,
         container <- thisModule.resolveTemp(
                         eS.set, 'c')
      )
}
```

Example #3

Resolve Temp – Explicitly Querying Trace Models (2/4)



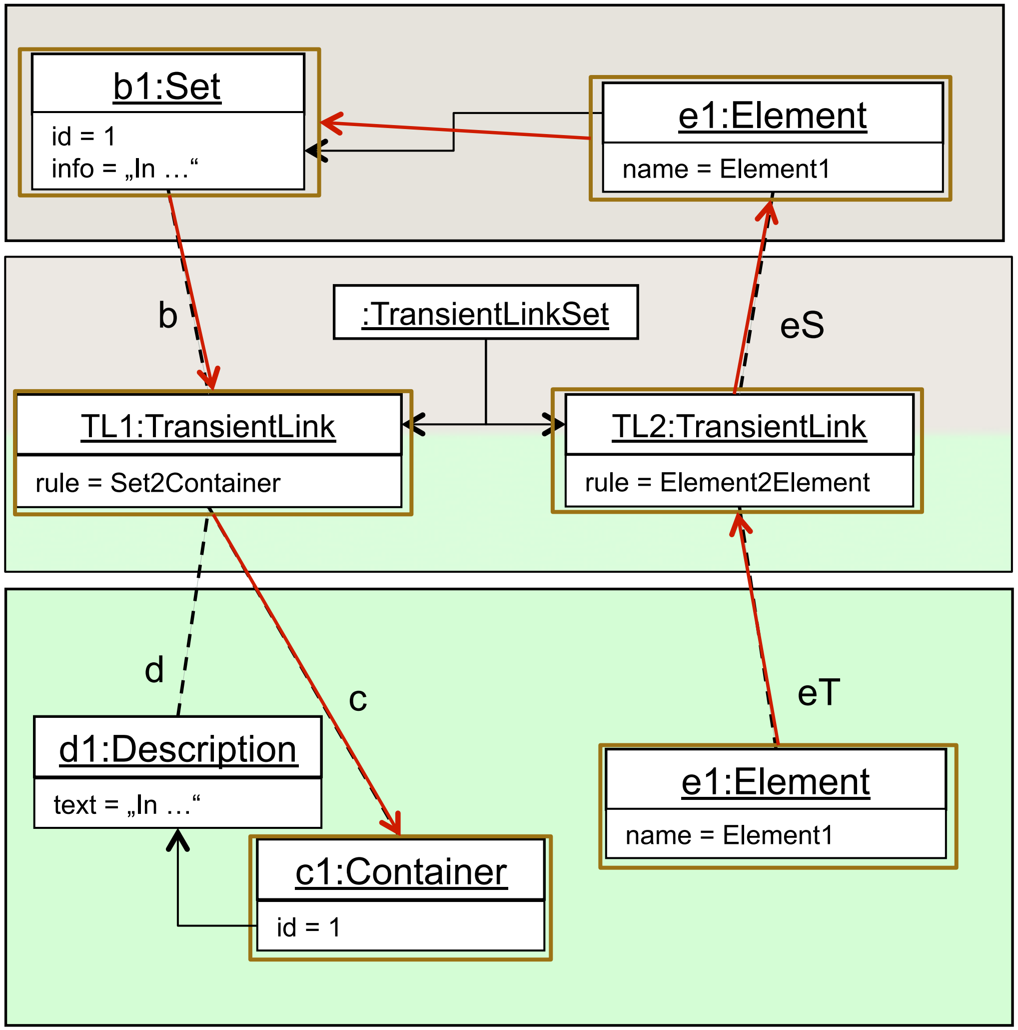

```
rule Set2Container {
    from
       b : source!Set
    to
       d : target!Description(
          text <- b.info
       )
       c : target!Container(
          id <- b.id,
          description <- d

       )
}
```

```
rule Element2Element{
    from
       eS : source!Element
    to
       eT : target!Element (
        name <- eS.name,
        container <-thisModule.resolveTemp(
                      eS.set, 'c')
       )
}
```

Example #3

Resolve Temp – Explicitly Querying Trace Models (3/4)



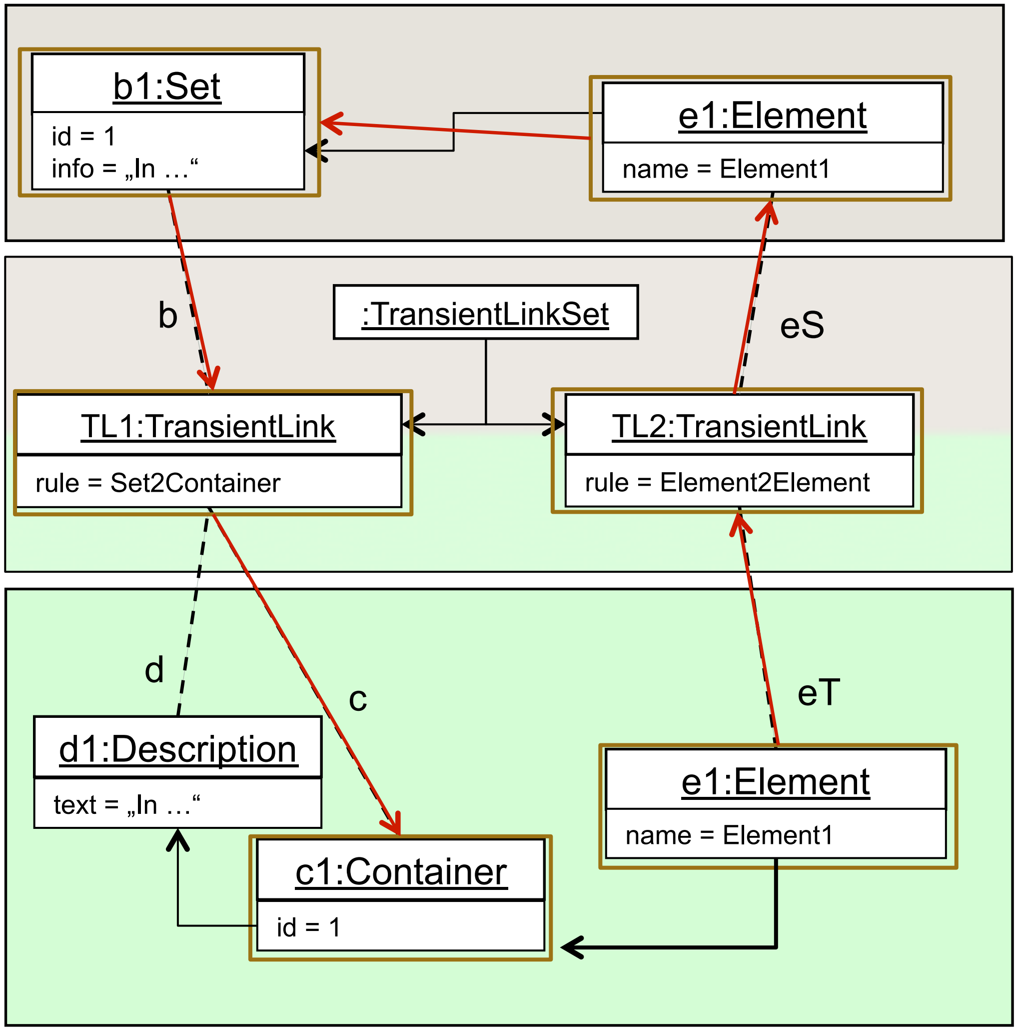

```
rule Set2Container {
    from
        b : source!Set
    to
        d : target!Description(
            text <- b.info
        )
        c : target!Container(
            id <- b.id,
            description <- d

        )
}
```

```
rule Element2Element{
    from
        eS : source!Element
    to
        eT : target!Element (
         name <- eS.name,
         container <-thisModule.resolveTemp(
                        eS.set, 'c')
        )
}
```


Example #3

Resolve Temp – Explicitly Querying Trace Models (4/4)



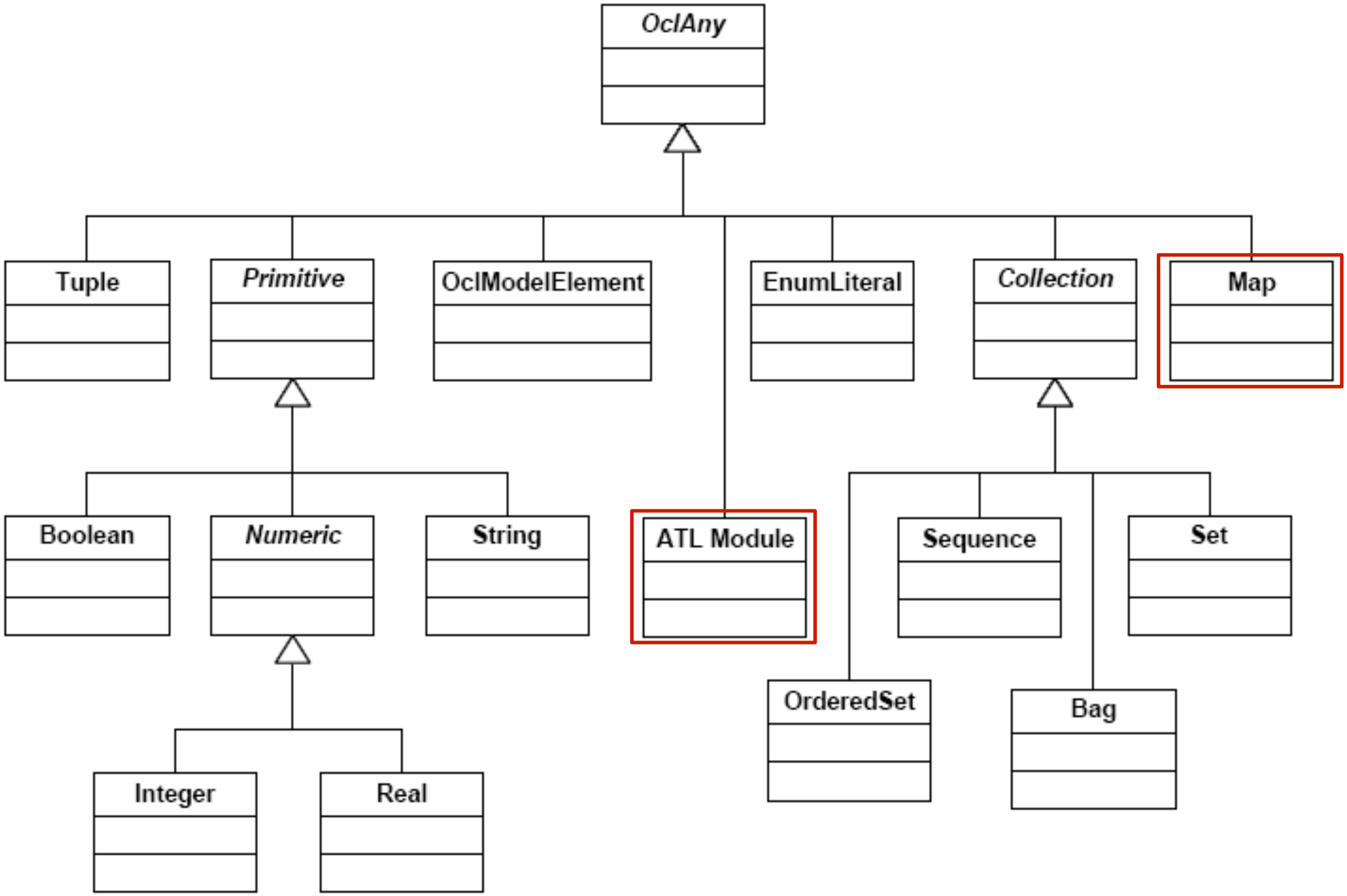

```
rule Set2Container {
    from
        b : source!Set
    to
        d : target!Description(
            text <- b.info
        )
        c : target!Container(
            id <- b.id,
            description <- d

        )
}
```

```
rule Element2Element{
    from
        eS : source!Element
    to
        eT : target!Element (
         name <- eS.name,
         container <-thisModule.resolveTemp(
                       eS.set, 'c')
        )
}
```

ATL Data Types

OCL Operations for each Type



Rule inheritance

- Rule inheritance allows for **reusing** transformation rules

- A child rule may match a **subset** of what its parent rule matches
  - All the **bindings** and **filters** of the parent still make sense for the child
- A child rule may **specialize** target elements of its parent rule
  - Initialization of existing elements may be specialized
- A child rule may **extend** target elements of its parent rule
  - New elements may be created
- A parent rule may be declared as **abstract**
  - Then the rule is not executed, but only reused by its children

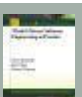
- **Syntax**

```
abstract rule R1 {
   ...
}
rule R2 extends R1 {
   ...
}
```

Rule inheritance

Example #1


```
abstract rule Person2Entity {
   from
      p : source!Person
   to
      e : target!Entity(
         name <- p.name
   )
}
rule Customer2Client extends Person2Entity{
   from
      p : source!Customer
   to
      e : target!Client (
         id <- p.id
   )
}
```

Rule inheritance

Example #2


```
rule Person2Entity {
   from
      p : source!Person
   to
      e : target!Entity(
         name <- p.name
   )
}
rule Customer2Client extends Person2Entity{
   from
      p : source!Customer
   to
      e : target!Client (
         id <- p.id
   )
}
```


Debugging Hints

- Quick and Dirty: Make use of .debug()
- Proceed in tiny increments
- Immediately test changes
- Read Exception Trace
  - Look top-down the stack trace for „ERROR“ to find meaningful message:

```
****** BEGIN Stack Trace
    message: ERROR: could not find operation ChXXXapter2TitleValue on Module having supertypes: [OclAny]
```

  - Check line number:

Line number

```
A.__applyBook2Line(1 : NTransientLink;) : ??#32 14:25-14:76
```

- do-blocks can be used for temporary debug output

ATL in Use

- ATL tools and documentation are available at http://www.eclipse.org/atl
  - Execution engine
    - Virtual machine
    - ATL to byte code compiler
  - Integrated Development Environment (IDE) for
    - Editor with syntax highlighting and outline
    - Execution support with launch configurations
    - Source-level debugger
  - Documentation
    - Starter's guide
    - User manual
    - User guide
    - Basic examples

Summary

- ATL is specialized in out-place model transformations
  - Simple problems are generally solved easily
- ATL supports advanced features
  - Complex OCL navigation, called rules, refining mode, rule inheritance, etc
  - Many complex problems can be handled declaratively
- ATL has declarative and imperative features
  - Any out-place transformation problem can be handled
- Further information
  - Documentation: http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language
  - Examples: http://www.eclipse.org/m2m/atl/basicExamples_Patterns

Alternatives to ATL

- **QVT**: Query-View-Transformation standard of the OMG
  - Declarative QVT Relational language
  - Imperative QVT Operational language
  - Low-level QVT Core language (VM level)
- **TGG**: Triple Graph Grammars
  - Correspondence graphs between metamodels
  - Transform models in both directions, integrate and synchronize models
- **JTL**: Janus Transformation Language
  - Strong focus on model synchronization by change propagating
- **ETL**: Epsilon Transformation Language
  - Designated language in the Epsilon framework for out-place transformations
- **RubyTL**: Ruby Transformation Language
  - Extension of the Ruby programming language
  - Core concepts for out-place model transformations (extendable)
- **Many** more languages such as VIATRA, Tefkat, Kermeta, or SiTra, …

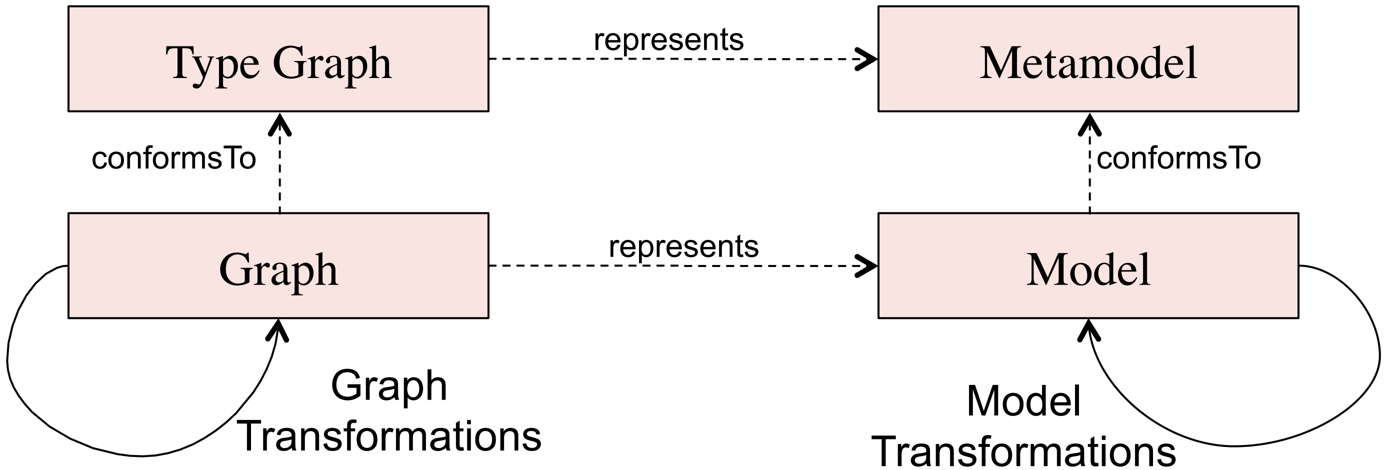
IN-PLACE TRANSFORMATIONS:

GRAPH TRANSFORMATIONS

Why graph transformations?

- **Models** are **graphs**
  - Class diagram, Petri net, State machine, …
- **Type Graph**:
  - Generalization of graph elements
- **Graph transformations**
  - Generalization of the graphs' **evolutions**

⇨ Graph transformations are applicable for models!

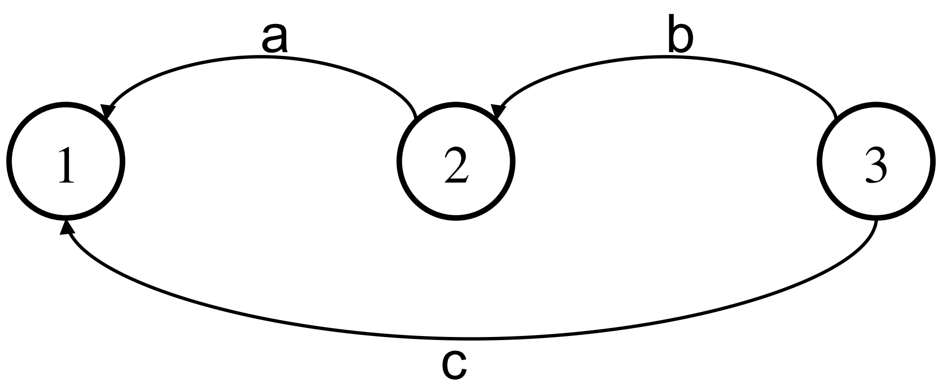


Basics: Directed graph

- A **directed graph** *G* consists of two disjoint sets
  - Vertices *V (Vertex)*
  - Edges *E (Edge)*
- Each edge has a source vertex *s* and a target vertex *t*

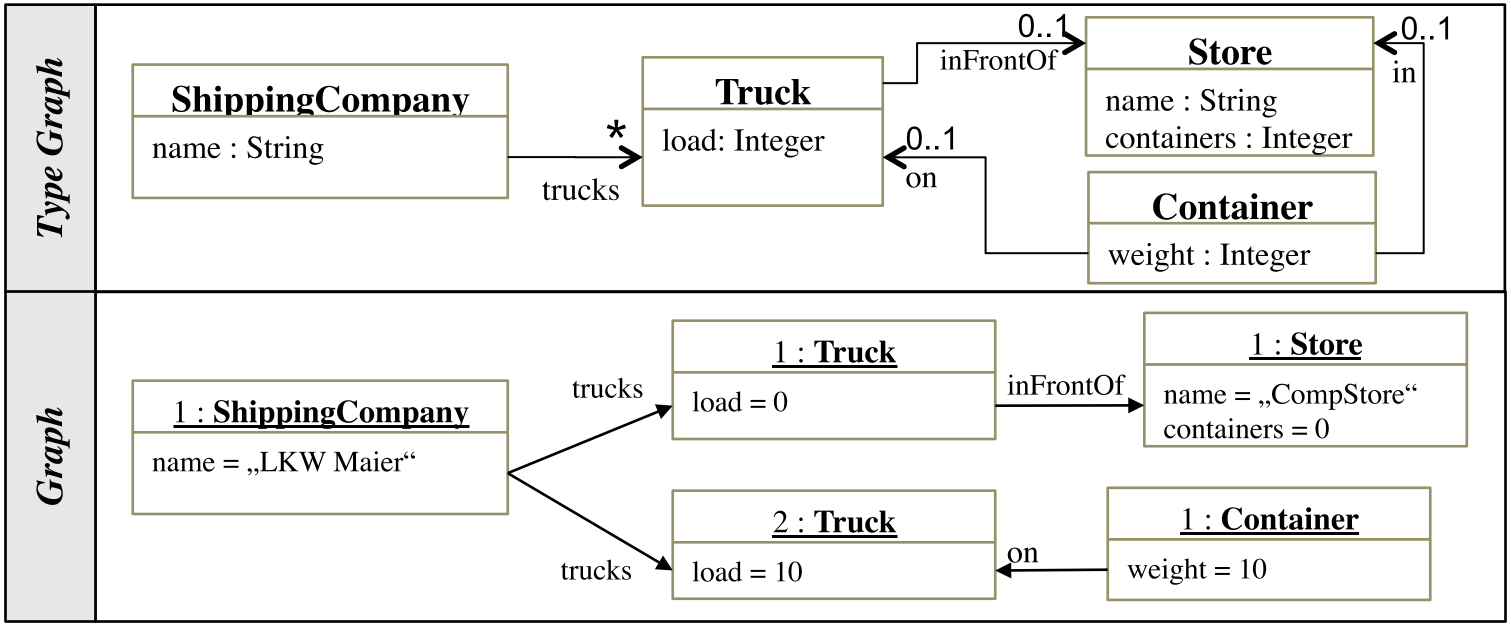
- Summarized: $G = (V, E, s: E \rightarrow V, t: E \rightarrow V)$

- **Example graph**
  - V = {1, 2, 3}
  - E = {a, b, c}
  - s(a) = 2
  - t(a) = 1
  - s(b) = 3
  - t(b) = 2
  - …



Typed attributed graph (1/3)

- To represent models further information is needed
  - **Typing**: Each vertex and each edge has a **type**
  - **Attribution**: Each vertex/edge has an **arbitrary number of name/value pairs**
- **Notation** for *directed*, *typed* and *attributed* graphs
  - **Graph** is represented as an **object diagram**
  - **Type graph** is represented as a **class diagram**
- **Example**

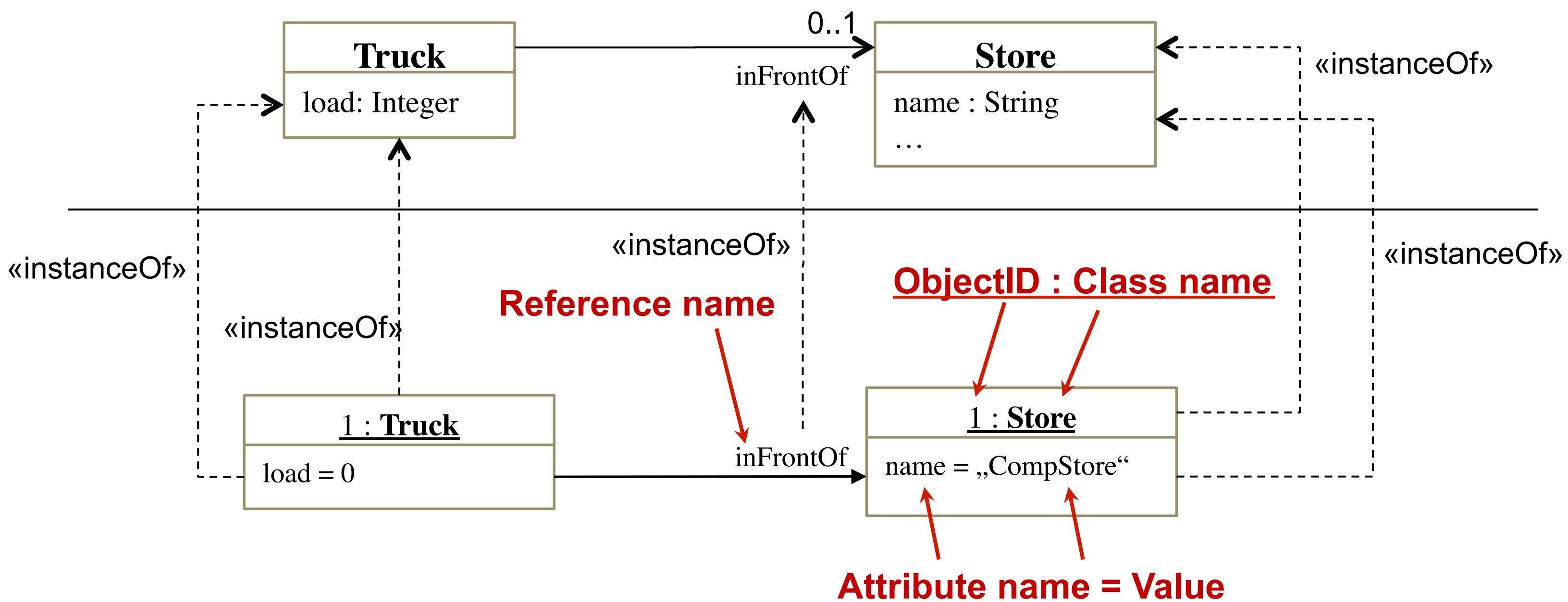


Example taken from: C. Ermel, M. Rudolf, and G. Taentzer, **The AGG approach: Language and environment**, *Handbook of Graph Grammars and Computing by Graph Transformation, Application, Languages and Tools,* volume 2, World Scientific, 1999.

Typed attributed graph (2/3)

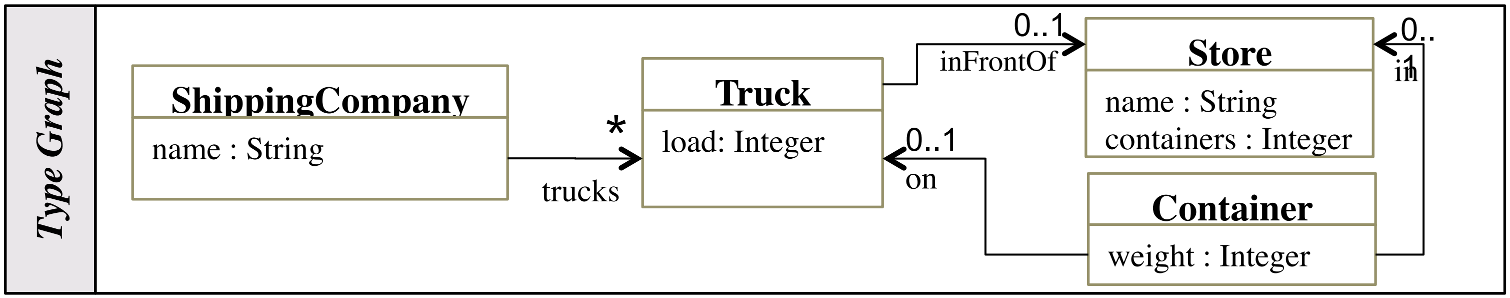
- **Object diagram**
  - Instance of class diagram

- **Basic concepts**
  - **Object** : Instance of class
  - **Value**: Instance of attribute
  - **Link**: Instance of reference



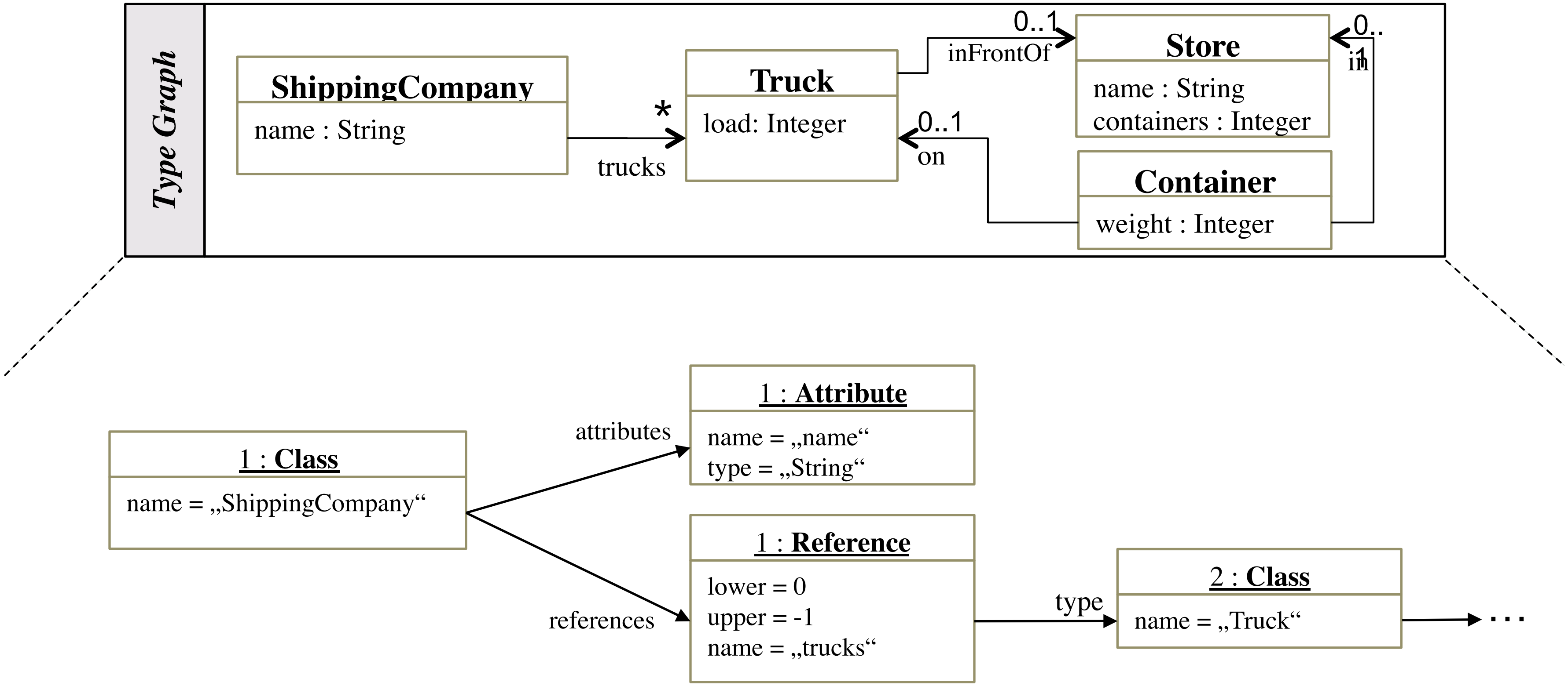
Typed attributed graph (3/3)

- Question: How does the type graph look in pure graph shape (object diagram)?



Typed attributed graph (3/3)

- Question: How does the type graph look in pure graph shape (object diagram)?



Until now…

- …we considered models as static entities
  - A modeler creates a model using an editor – Done!

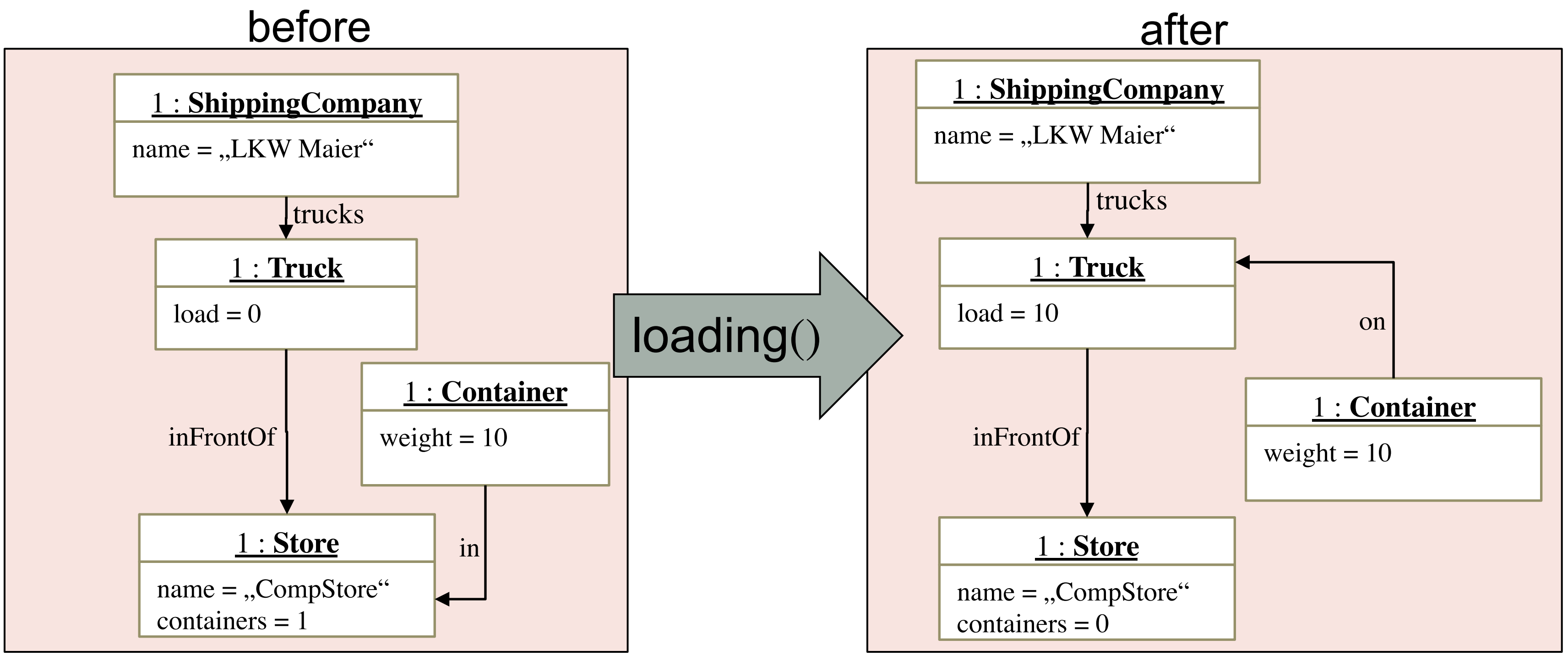
- *But what is about **dynamic** model modifications?*

- They are **needed** for
  - Simulation
  - Execution
  - Animation
  - Transformation
  - Extension
  - Improvement
  - …

- *How can graphs be modified?*
  - Imperative: Java Program + Model API
  - Declarative: Graph transformations by means of graph transformation rules

Example

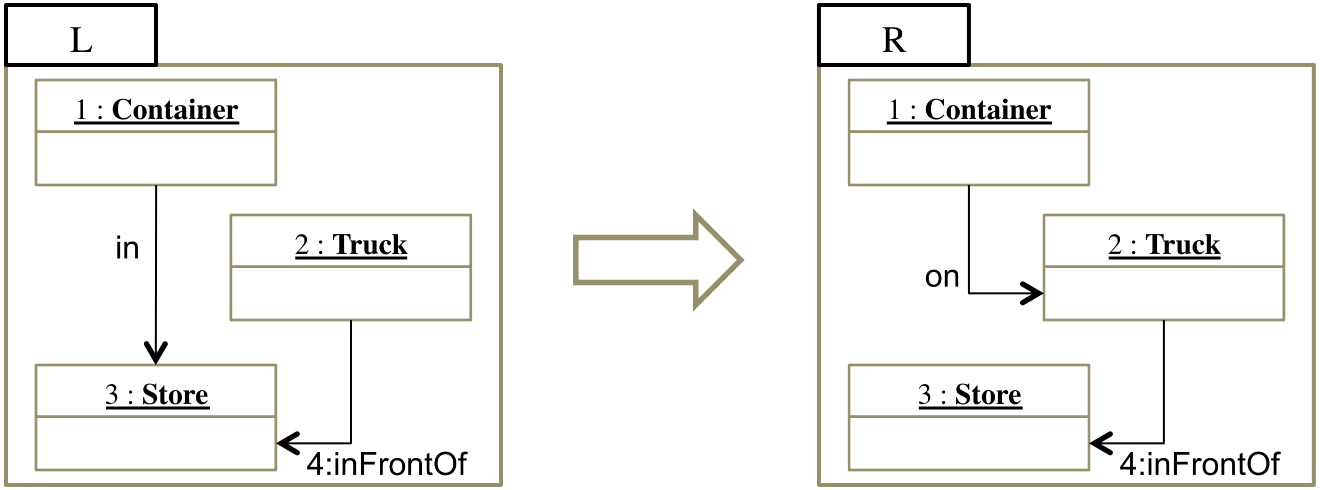
- Operation: Loading of Container onto a Truck



- How can this behaviour be described in a generic way?

Graph transformation rule

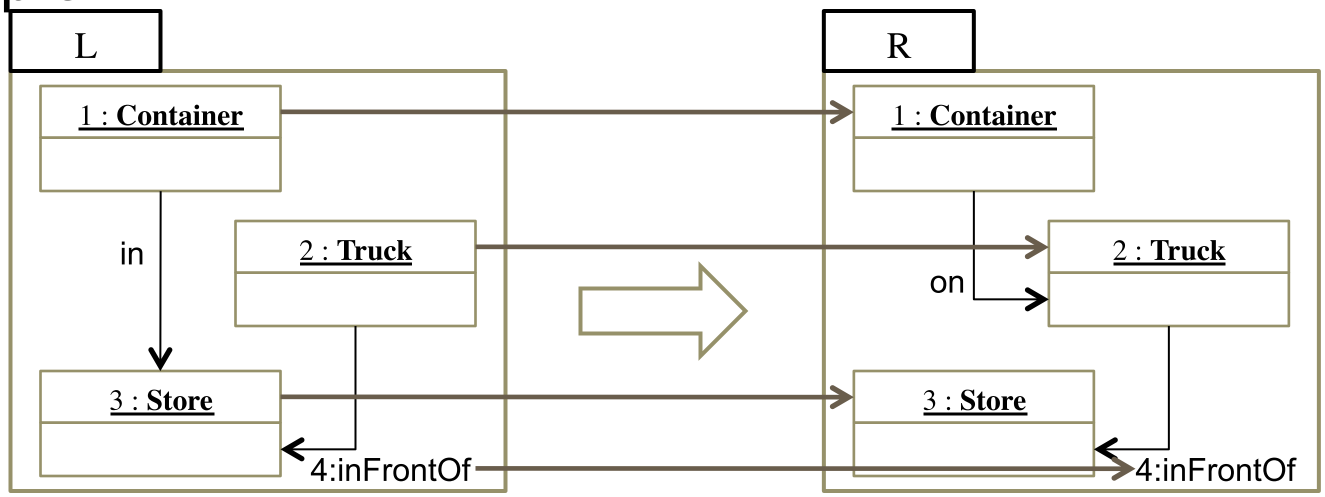
- A graph transformation rule *p: L → R is* a structure preserving, partial mapping between two graphs
  - L and R are two directed, typed and attributed graphs themselves
  - Structure preserving, because vertices, edges and values may be preserved
  - Partial, because vertices and edges may be added/deleted
- Example: Loading of Container onto a Truck



Pre-condition   Post-condition

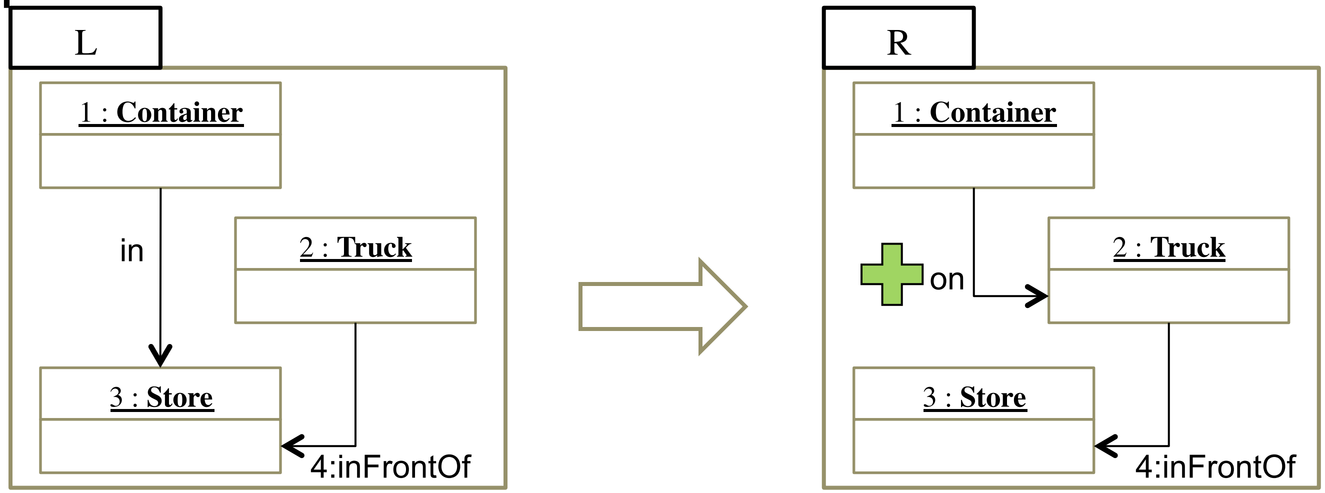
Graph transformation rule

- Structure preserving
  - All vertices and edges which are contained in the set L ∩ R
- Example



Graph transformation rule

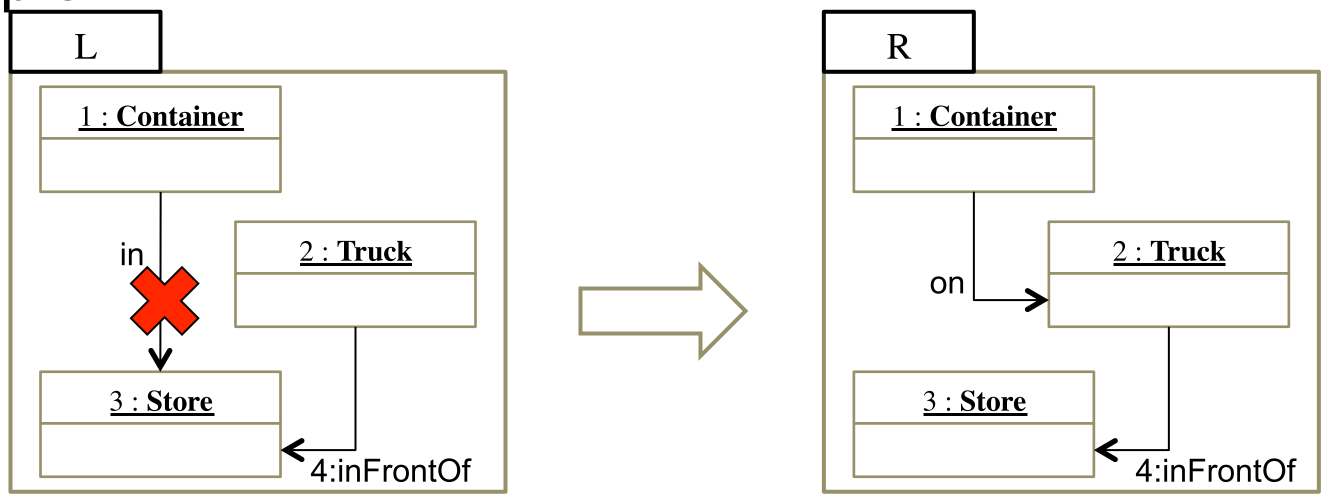
- Adding
  - All vertices and edges which are contained in the set R \ L
- Example



Graph transformation rule

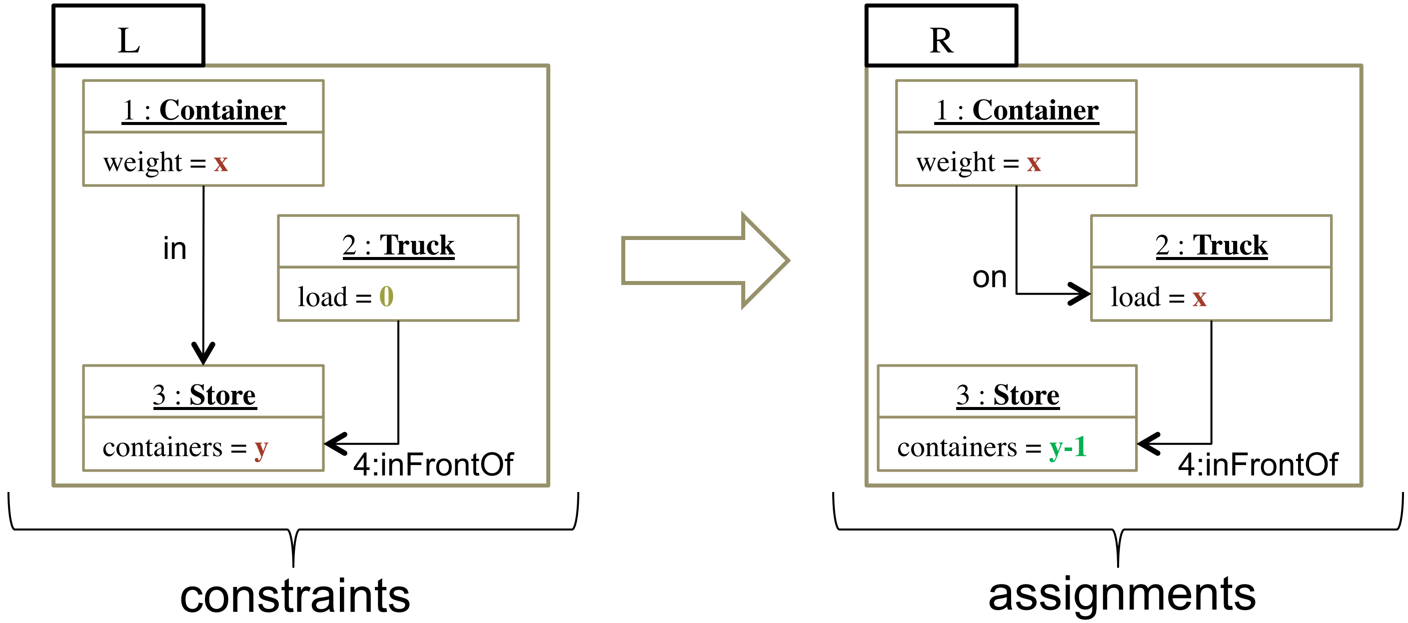
- Deleting
  - All vertices and edges which are contained in the set L \ R

- Example



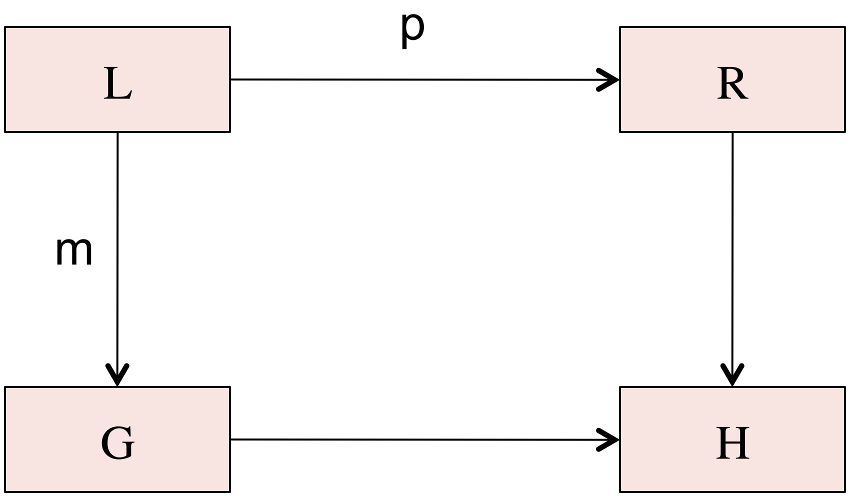
Graph transformation rule

- Calculating – Make use of attributes
  - Constants
  - Variables
  - Expressions (OCL & Co)
- Example



Graph transformation

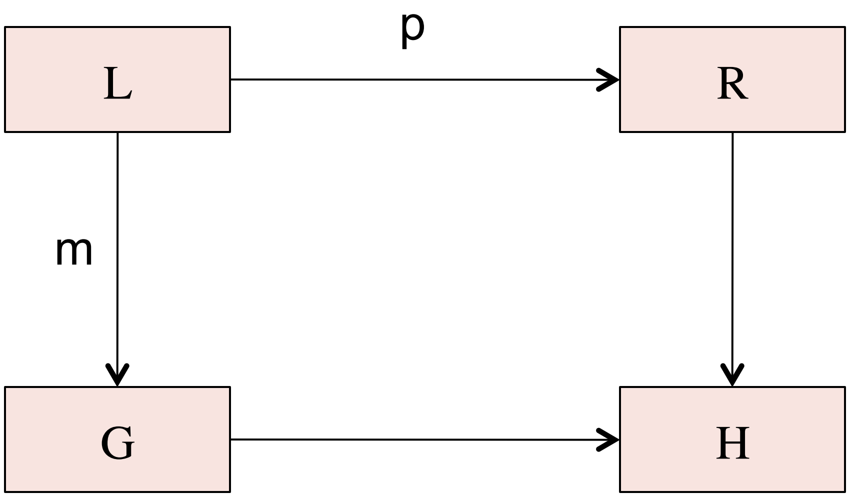
- A graph transformation $t: G \rightarrow H$ is the result of the execution of a graph transformation rule $p: L \rightarrow R$ in the context of G
  - t = (p,m) where $m: L \rightarrow G$ is an injective graph morphism (**match**)



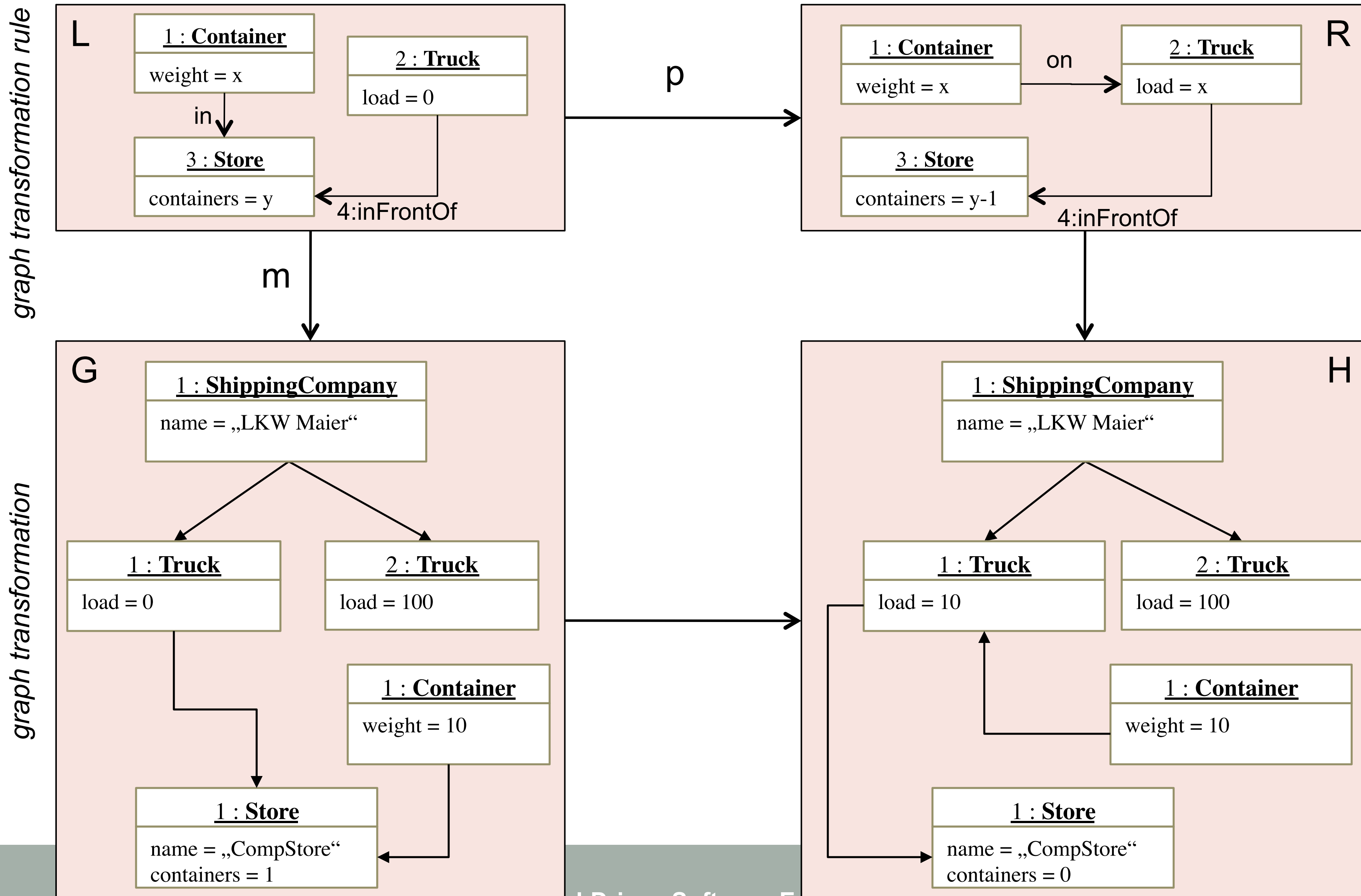
Graph transformation

Operational specification

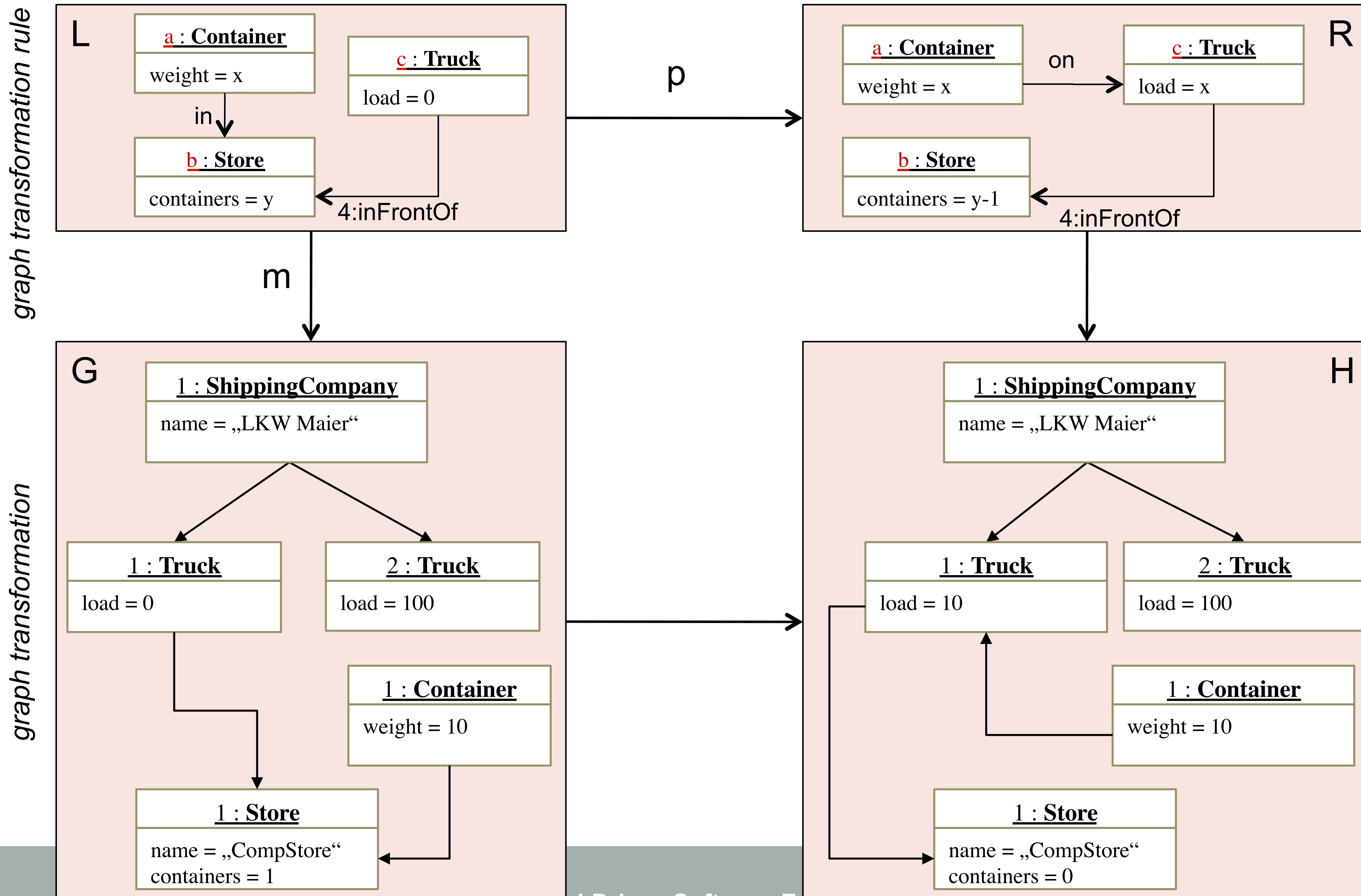
- Prepare the transformation
  - Select rule p: L -> R
  - Select match m: L -> G
- Generate new graph H by
  - Deletion of L \ R
  - Addition of R \ L



Graph transformation example (1/2)

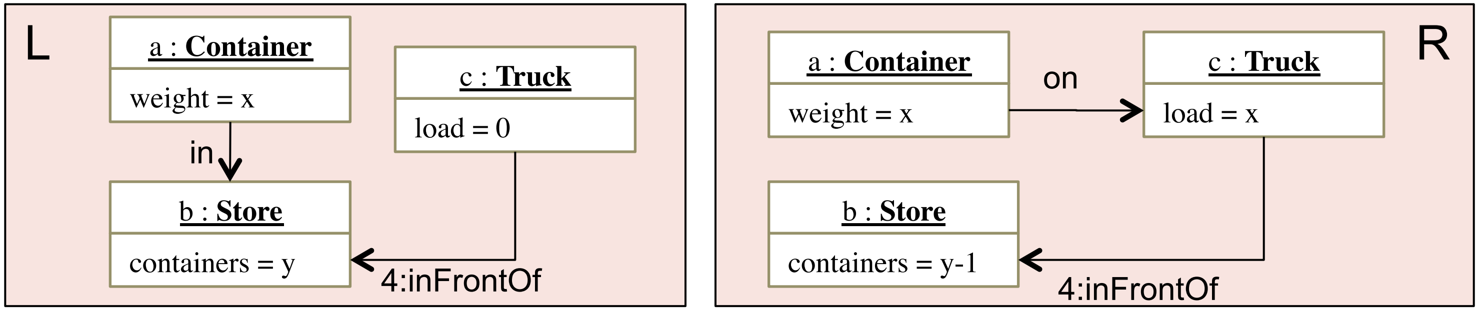


Graph transformation example (2/2)



Graph transformations in ATL

- **Graph transformation**



- **ATL (Refining Mode)**

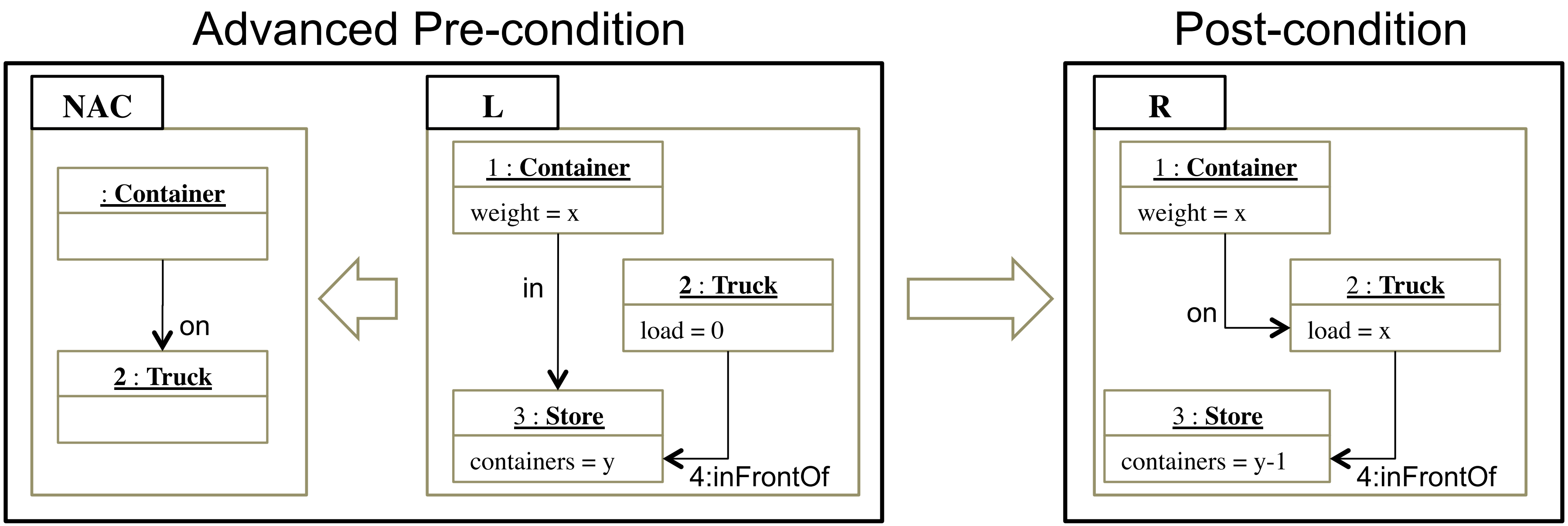
```
rule Loading {
  from
       a : Container (a.in = b)
       b : Store
       c : Truck (c.inFrontOf = b AND c.load = 0)
  to
       _a : Container(weight <- a.weight, on <- _c)
       _b : Store(containers <- b.containers - 1)
       _c : Truck(load <- a.weight, inFrontOf <- _b)
}
```


Negative Application Condition (NAC)

- Left side of a rule specifies what must be **existing** to execute the rule
  - *Application Condition*
- Often needed to describe what **must not be existing**
  - *Negative Application Condition (NAC)*
- NAC is a graph which describes a **forbidden** sub graph structure
  - Absence of specific vertices and edges must be granted
- Graph transformation rule is executed when NAC is **not** fulfilled

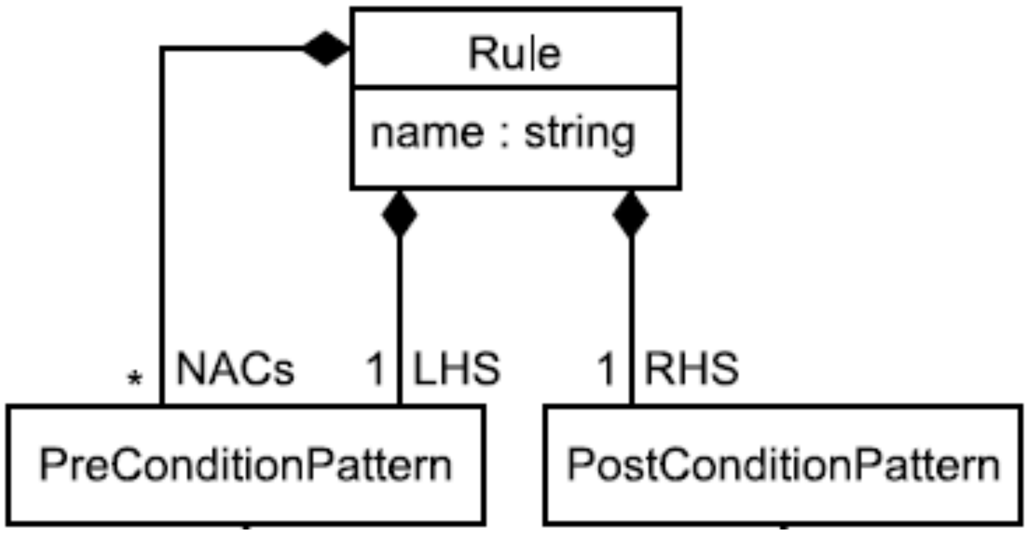
Negative Application Condition (NAC)

- Example: A truck should only be loaded if there is no container on the truck



Negative Application Condition (NAC)

- **Multiple** NACs for one rule possible
- **But**: LHS and RHS must only be specified once

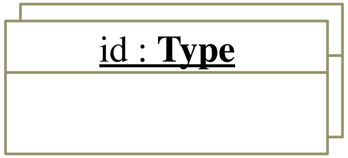


Multi-Object

- The **number** of matching objects is not always known **in advance**

- Maximal set of objects which fulfill a certain condition
  - Selection of all objects x from a set y which fulfill condition z
  - In OCL: *y -> select(x | z(x))*

- **Notation: Multi-Object from UML 1.4**
  - A multi-object symbol maps to a collection of instances in which each instance conforms to a given type and conditions
  - Example: *select(x | x.oclIsTypeOf(Type))*



Execution order of graph transformation rules

- A graph transformation system consists of a set of different graph transformation rules
- In which **order** are they being executed?
- Multiple procedures
  - nondeterministic
  - deterministic
    - Priorities
    - Programs (aka „*programmable graph transformations*“)
- Example – Specialized UML activity diagrams
  - [failure]
  - [success]

Graph transformation Tools

- VIATRA
- PROGRES
- GrGen.NET
- VMTS
- MOMENT2
- EMT
- Fujaba
- AGG
- MoTif
- GROOVE
- MoTMoT
- ATL Refining Mode
- eMotions
- …

Summary

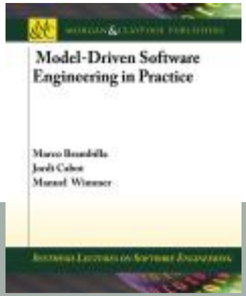
- Model transformations are graph transformations
  - Type Graph = Metamodel
  - Graph = Model

- Programmable graph transformations allow for the development of complex graph evolution scenarios

- Exogenous, out-place transformations can also be specified as graph transformations
  - However more complex as with ATL

- Graph transformations are becoming more and more relevant in practice
  - Found their way into Eclipse!

MASTERING MODEL TRANSFORMATIONS

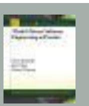


HOTs

- **Transformations** can be regarded as **models** themselves
  - They are instances of a **transformation metamodel**!
  - This **uniformity** allows reusing tools and methods defined for models

- Thus, a transformation model can itself be created or manipulated by transformations, by so-called ***Higher Order Transformations*** (HOTs)
  - Transformations that take as input a model transformation and/or generate a model transformation as output

- **Examples**
  - Refactoring support for transformations to improve their internal structure
  - Adding a logging aspect to transformations
  - …



Bi-directional Transformations

- **Bi-directional** model transformation languages
  - do **not impose** a transformation **direction** when specifying a transformation
  - **allow** for **different execution modes** such as *transformation*, *integration*, and *synchronization*

- **Transformation mode** is further divided into **forward** and **backward** transformation (source -> target -> source)
- **Integration mode** assumes to have source and target models given and checks if expected elements (that would be produced in the transformation mode) exist
- **Synchronization mode** updates the models in case the integration mode has reported unsatisfied correspondences

- Languages allowing for bi-directional transformations:
  - JTL, TGG, QVT Relational, …

Lazy & Incremental Transformations

- **Standard execution strategy** for out-place transformations
  1) Read the **complete input** model
  2) Produce the **output model from scratch** by applying **all** matching transformation **rules**.
- Such executions are often referred as ***batch*** transformations.

- Two scenarios may benefit from alternative execution strategies
  1) An **output model already exists** from a previous transformation run for a given input model → ***incremental*** transformations
  2) Only a **part of the output model** is needed by a consumer → ***lazy*** transformations

- Experimental implementations for lazy and incremental transformations are available for ATL

Transformation Chains

- **Transformations** may be **complex processes**
- **Divide and Conquer**
  - Use different transformation steps to avoid having one monolithic transformation!
- **Transformation chains** are the technique of choice for modeling the **orchestration** of different model transformations
- **Transformation chains** are defined with **orchestration languages**
  - Simplest form: sequential steps of transformations executions
  - More complex forms: conditional branches, loops, and further control constructs
  - Even HOTs may produce dynamically transformations used by the chain
- **Smaller transformations** focusing on certain aspects allow for **higher reusability**

MORGAN & CLAYPOOL PUBLISHERS

MODEL-DRIVEN SOFTWARE ENGINEERING IN PRACTICE

Marco Brambilla,
Jordi Cabot,
Manuel Wimmer.
Morgan & Claypool, USA, 2012.

www.mdse-book.com
www.morganclaypool.com