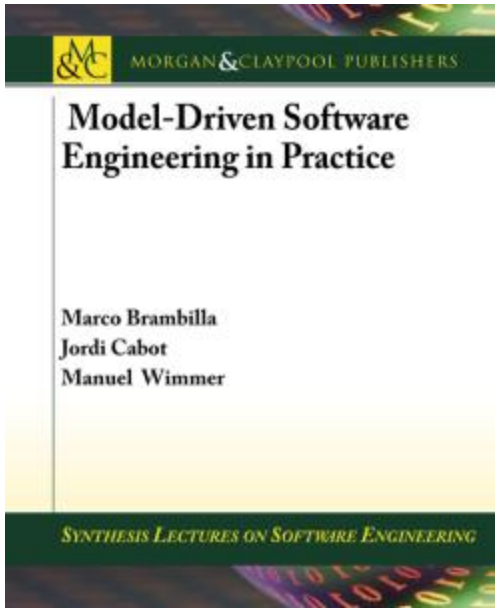
MORGAN & CLAYPOOL PUBLISHERS

Capítulo 8

TRANSFORMAÇÕES MODELO-PARA-MODELO

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



Conteúdo

- Introdução
- Transformações Out-place: ATL
- Transformações In-place: Transformações de grafos
- Dominando Transformações de Modelo

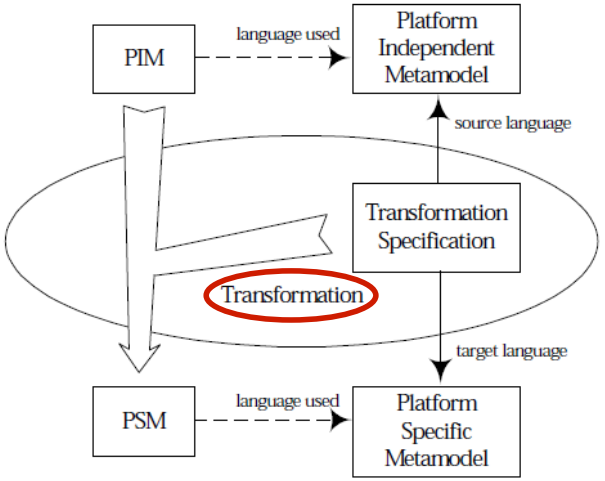
INTRODUÇÃO


Model-Driven Software
Engineering in Practice
Marco Brambilla
Jordi Cabot
Manuel Wimmer

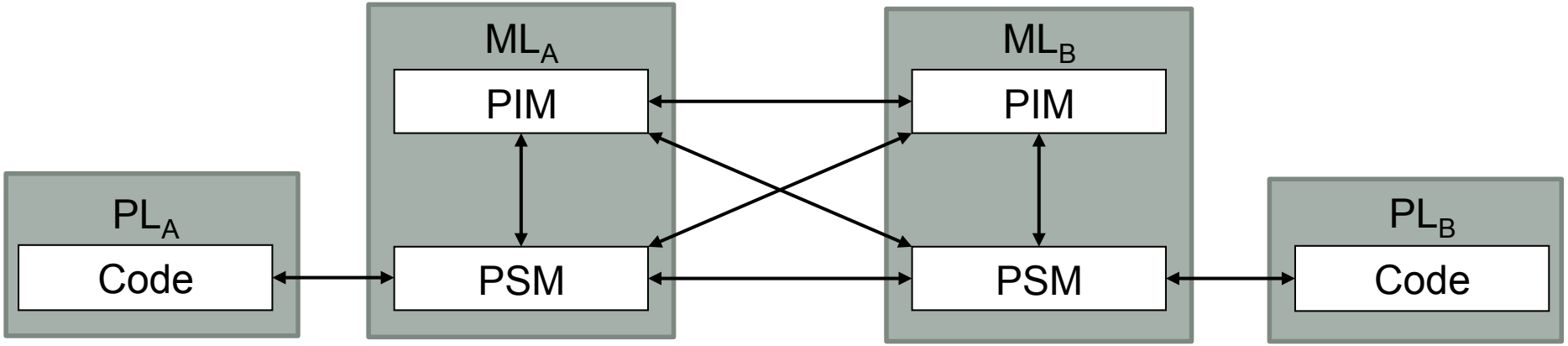
Motivação

Transformações estão em toda parte!

- **Antes** do MDE
  - Compilação do programa, refatoração, migração, otimização ...
- **Com** MDE
  - Transformações são **tecnologias chave**!
  - Toda **manipulação sistemática** de um modelo é uma transformação de modelo!
- **Dimensões**
  - Horizontal vs. vertical
  - Endógeno vs. exógeno
  - Modelo-para-texto vs. texto-para-modelo vs. modelo-para-modelo



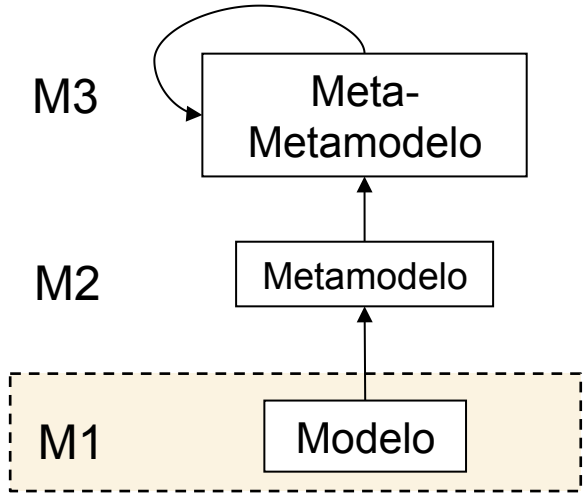
[Trecho do *MDA Guide* da OMG]



Definições

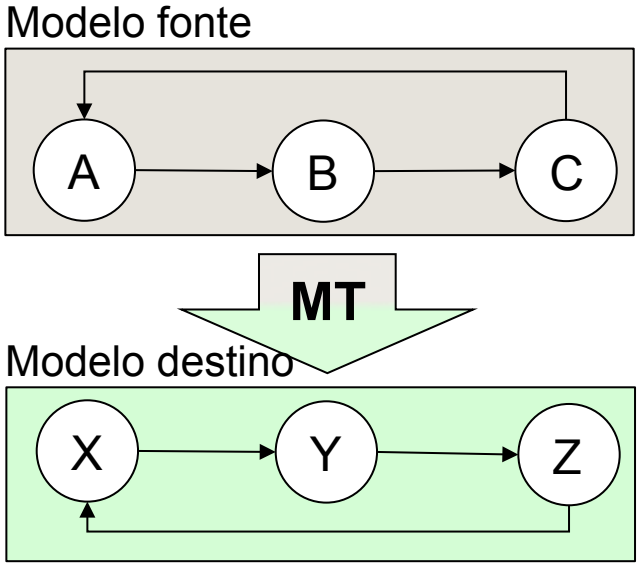
- Uma transformação **modelo-para-modelo (M2M)** é a criação automática de modelos-destino a partir de modelos-origem.
  - transformações 1-para-1
  - transformações 1-para-N, N-para-1, N-para-M
  - modelo origem* = T(modelo destino*)

Pilha de Metamodelagem



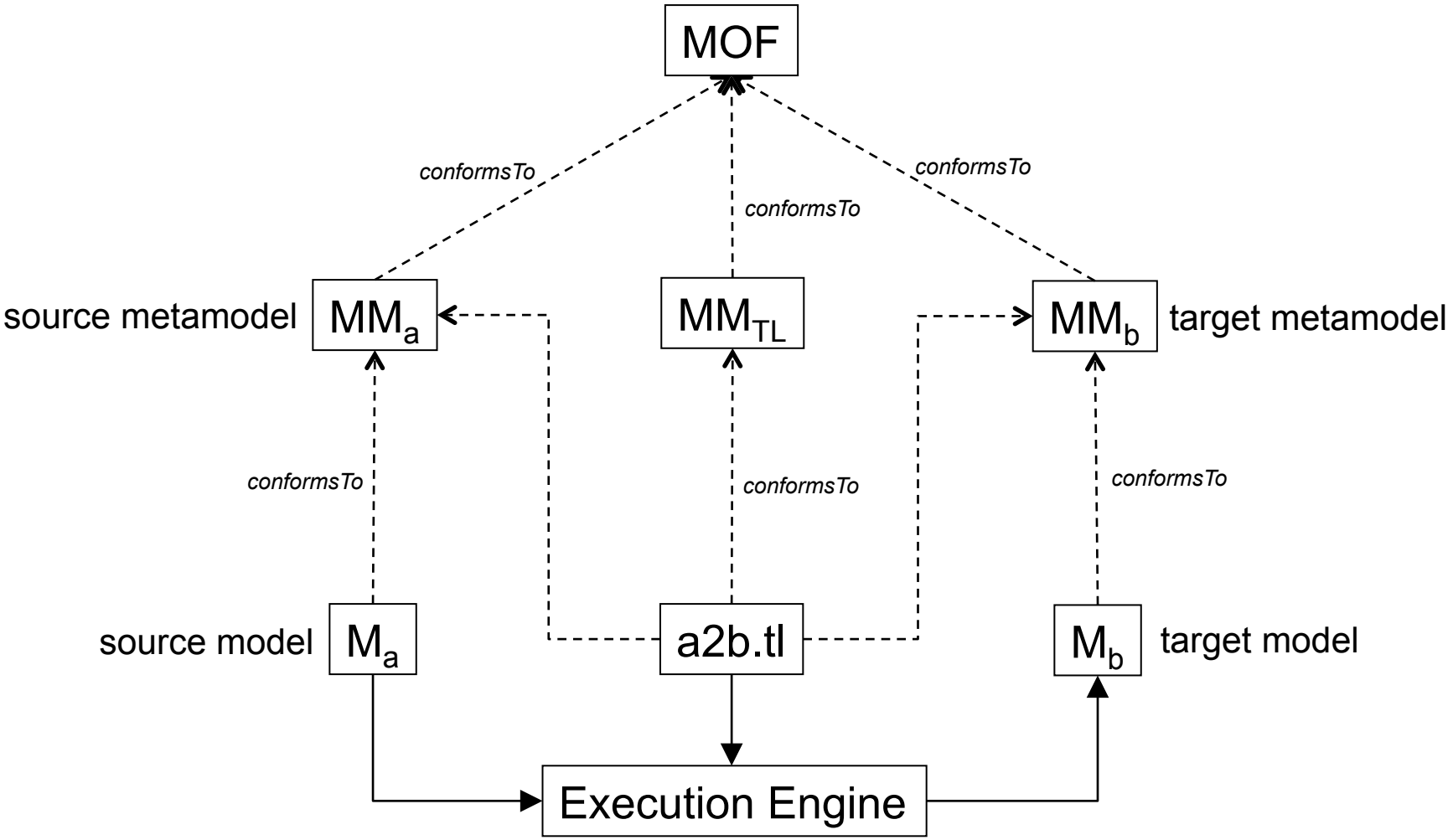
Foca nas transformações do modelo, mas **não é limitado** ao M1
(lembre-se: tudo é um modelo!)

Exemplo



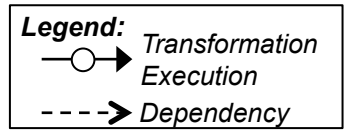
Arquitetura

Padrão de Transformação Modelo-para-Modelo

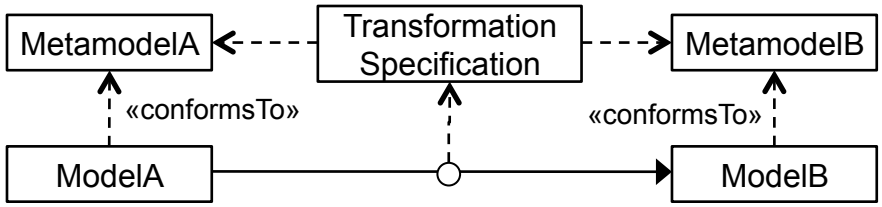


Duas Estratégias de Transformação

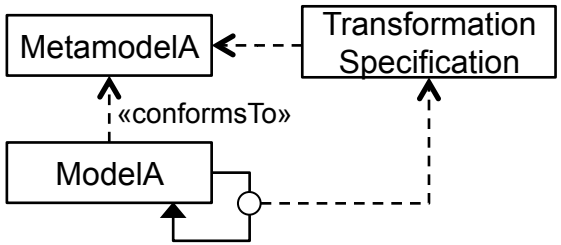
Transformações Out-place vs. In-place



Transformações Out-place controem um novo modelo a partir do zero



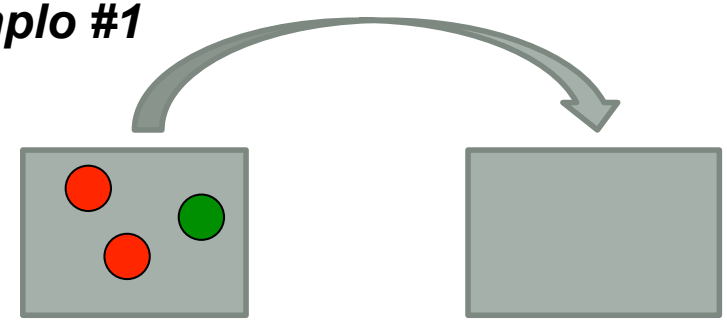
Transformações In-place alteram algumas partes no modelo



Duas Estratégias de Transformação

Transformações Out-place vs. In-place

Exemplo #1



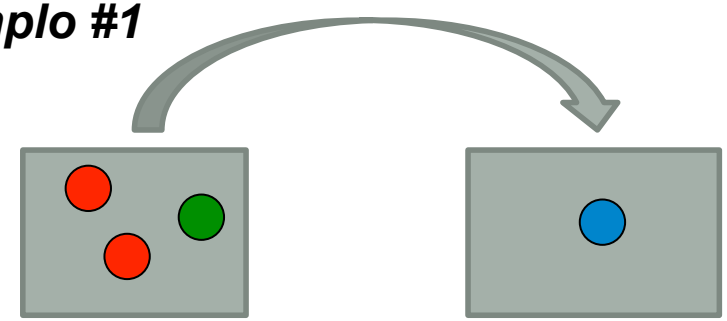
Transformação Out-place

Transformação In-place

Duas Estratégias de Transformação

Transformações Out-place vs. In-place

Exemplo #1



Transformação Out-place

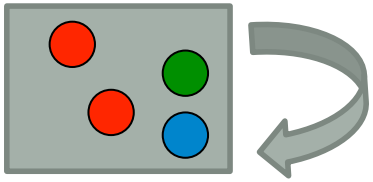
Para cada elemento verde, crie um elemento azul

Transformação In-place

Duas Estratégias de Transformação

Transformações Out-place vs. In-place

Exemplo #1



Transformação Out-place

Para cada elemento verde, crie um elemento azul

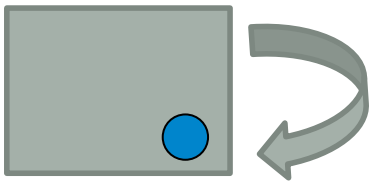
Transformação In-place

Para cada elemento verde, crie um elemento azul.

Duas Estratégias de Transformação

Transformações Out-place vs. In-place

Exemplo #1



Transformação Out-place

Para cada elemento verde, crie um elemento azul

Transformação In-place

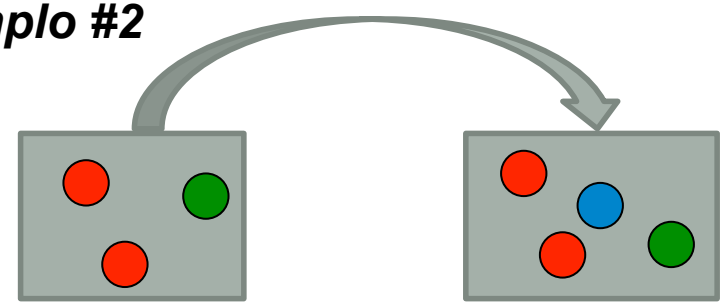
Para cada elemento verde, crie um elemento azul.

Delete todos os elementos exceto os azuis.

Duas Estratégias de Transformação

Transformações Out-place vs. In-place

Exemplo #2



Transformação Out-place

Para cada elemento verde, crie um elemento azul.

Para cada elemento verde, crie um elemento verde.

Para cada elemento vermelho, crie um elemento vermelho.

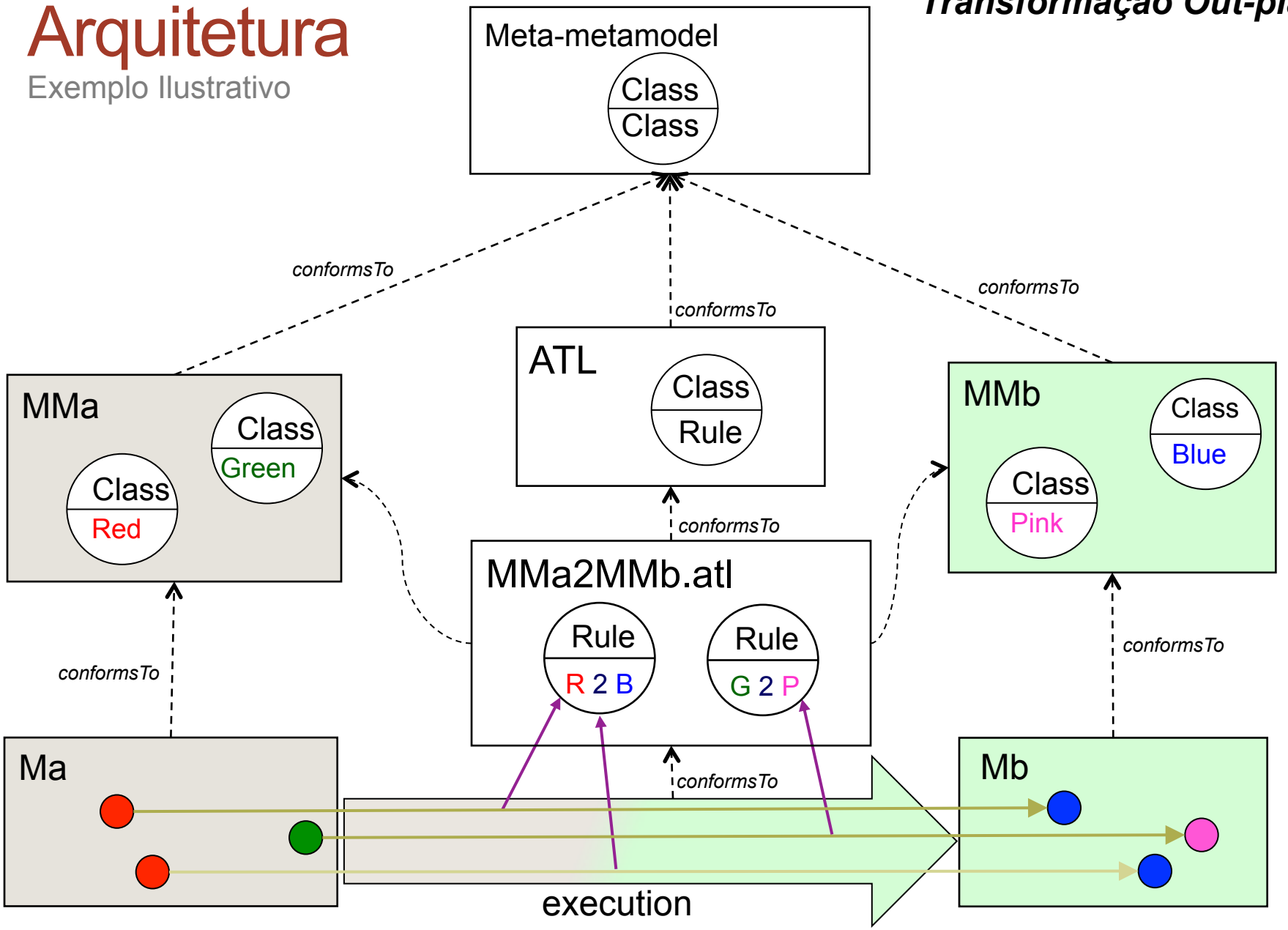
Transformação In-place

Para cada elemento verde, crie um elemento azul.

Arquitetura

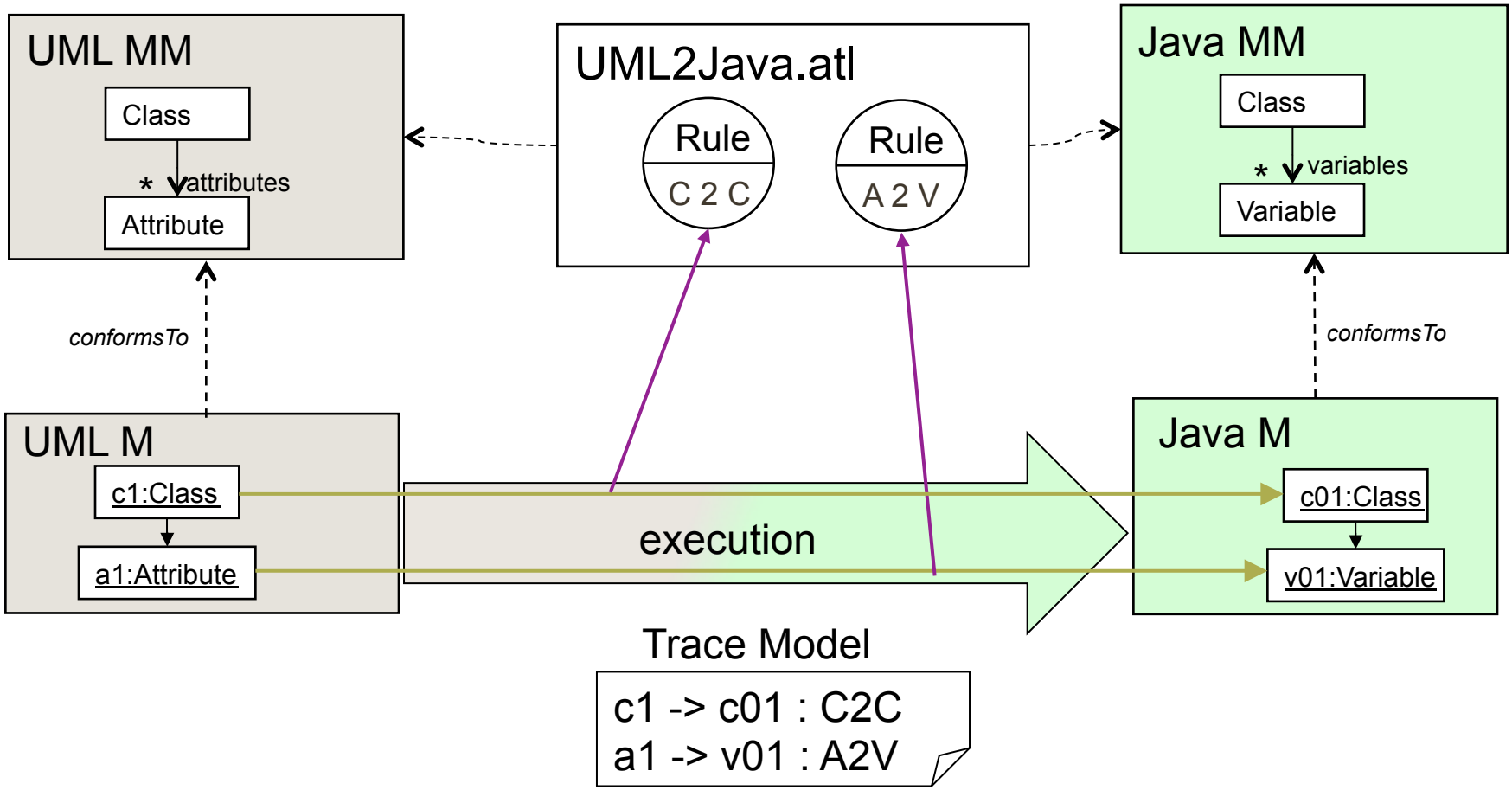
Exemplo Ilustrativo

Transformação Out-place



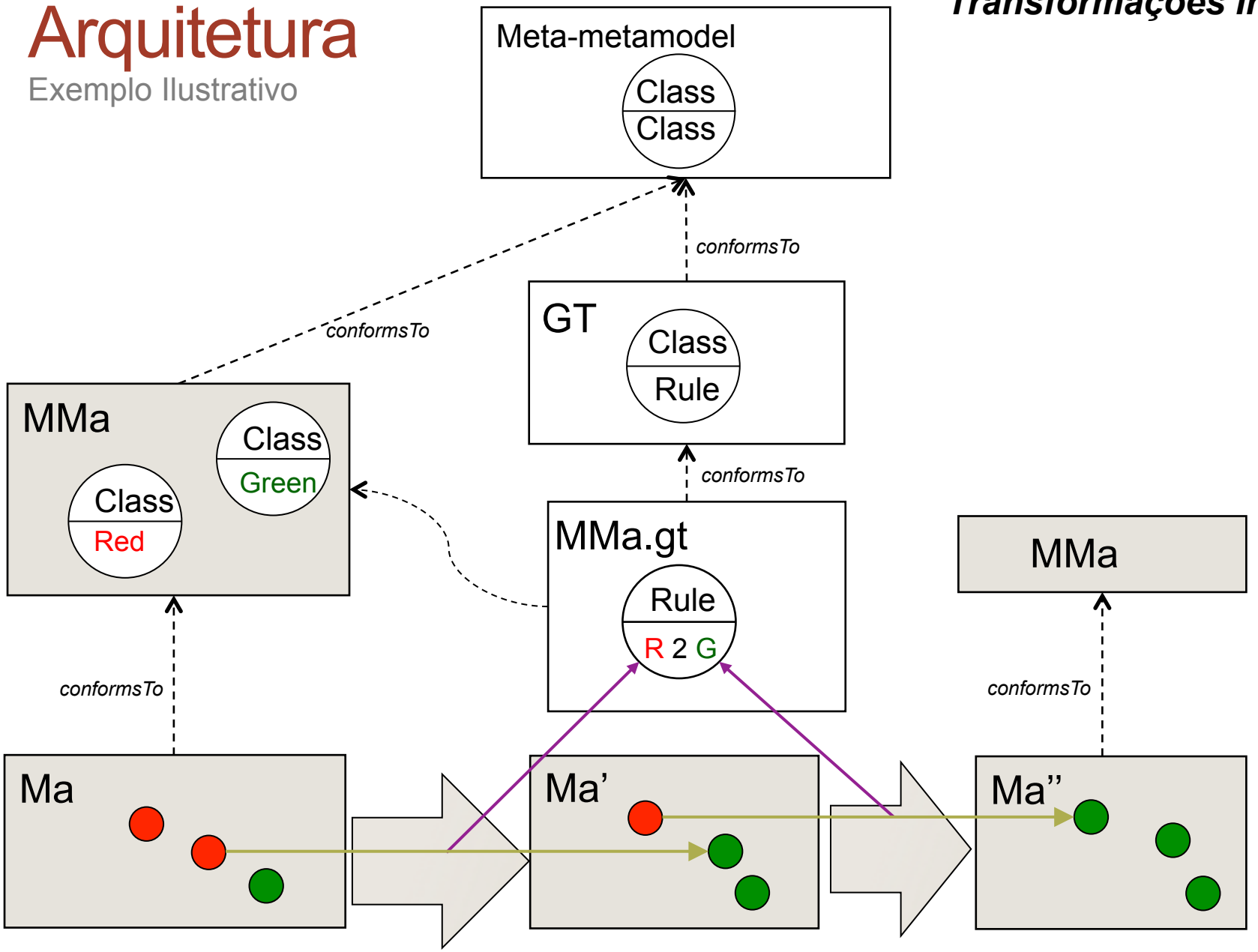
Arquitetura

Exemplo concreto



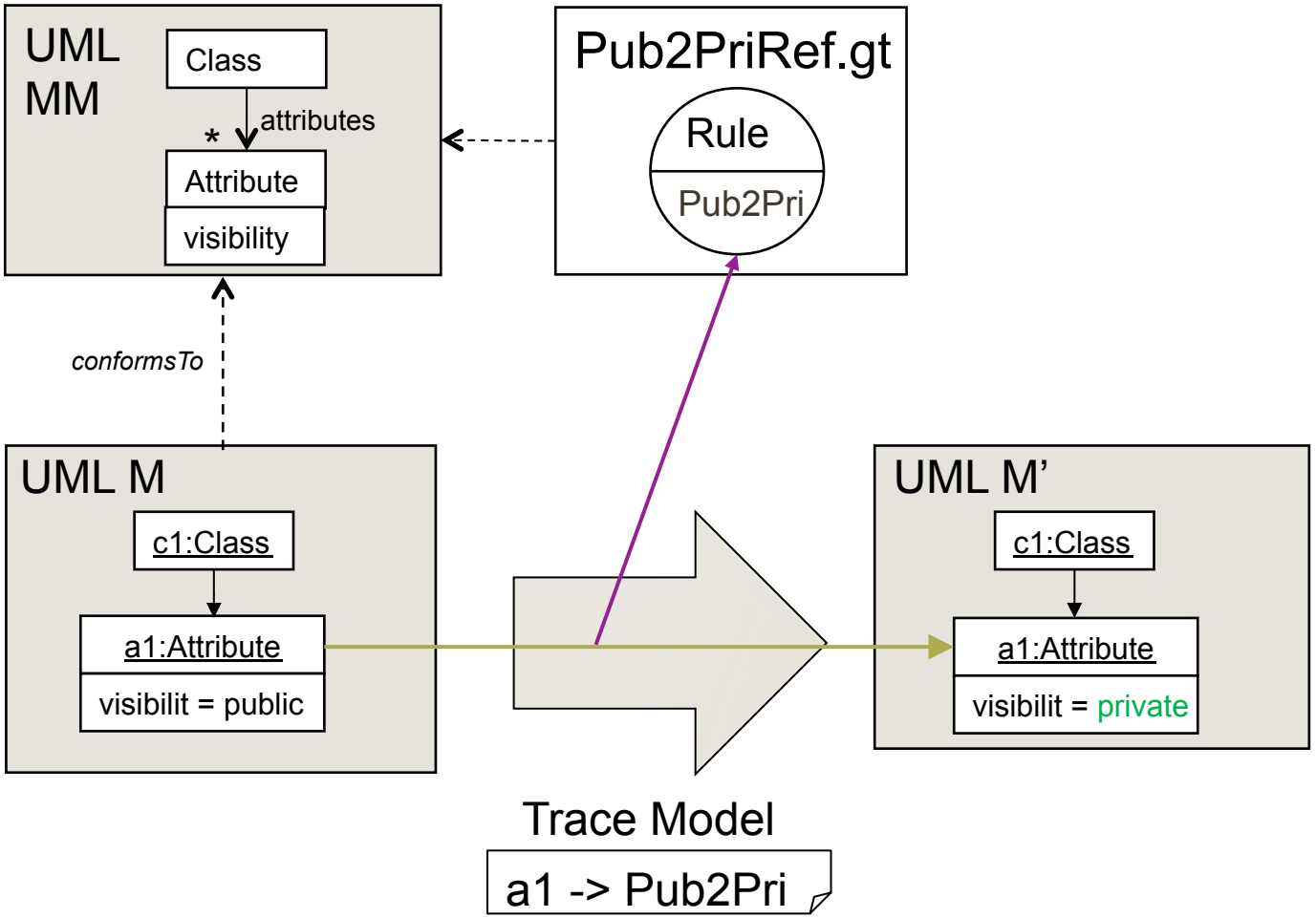
Arquitetura

Exemplo Ilustrativo



Arquitetura

Exemplo Concreto



TRANSFORMAÇÕES OUT-PLACE:

LINGUAGEM DE TRANSFORMAÇÃO ATLAS

ATL visão geral

- Modelos de origem e modelos de destino são distintos
  - Modelos de **Origem** são apenas-leitura (Source model)
  - Modelos de **Destino** são apenas-escrita (Target model)
- A linguagem é híbrida declarativa-imperativa
  - Parte declarativa
    - Regras **combinadas** com apoido da rastreabilidade automática
  - Parte imperativa
    - Regras de **Called/Lazy**
    - **Blocos de ação**
    - **Variáveis globais por meio de Helpers**
  - Estilo de programação recomendado: **declarativo**

ATL visão geral (continuação)

- Uma regra **declarativa** específica
  - Um padrão de origem a ser **combinado** nos modelos de origem
  - Um **padrão de destino** a ser **criado** em um **modelo de destino** para cada combinação durante a **aplicação** das regras
  - Um bloco opcional de ação (ex: uma sequência de instruções imperativas)

- Uma regra **imperativa** é basicamente um procedimento
  - É chamado pelo seu nome
  - Pode conter argumentos
  - Contem
    - Um padrão de destino declarativo
    - Um bloco de ação opcional

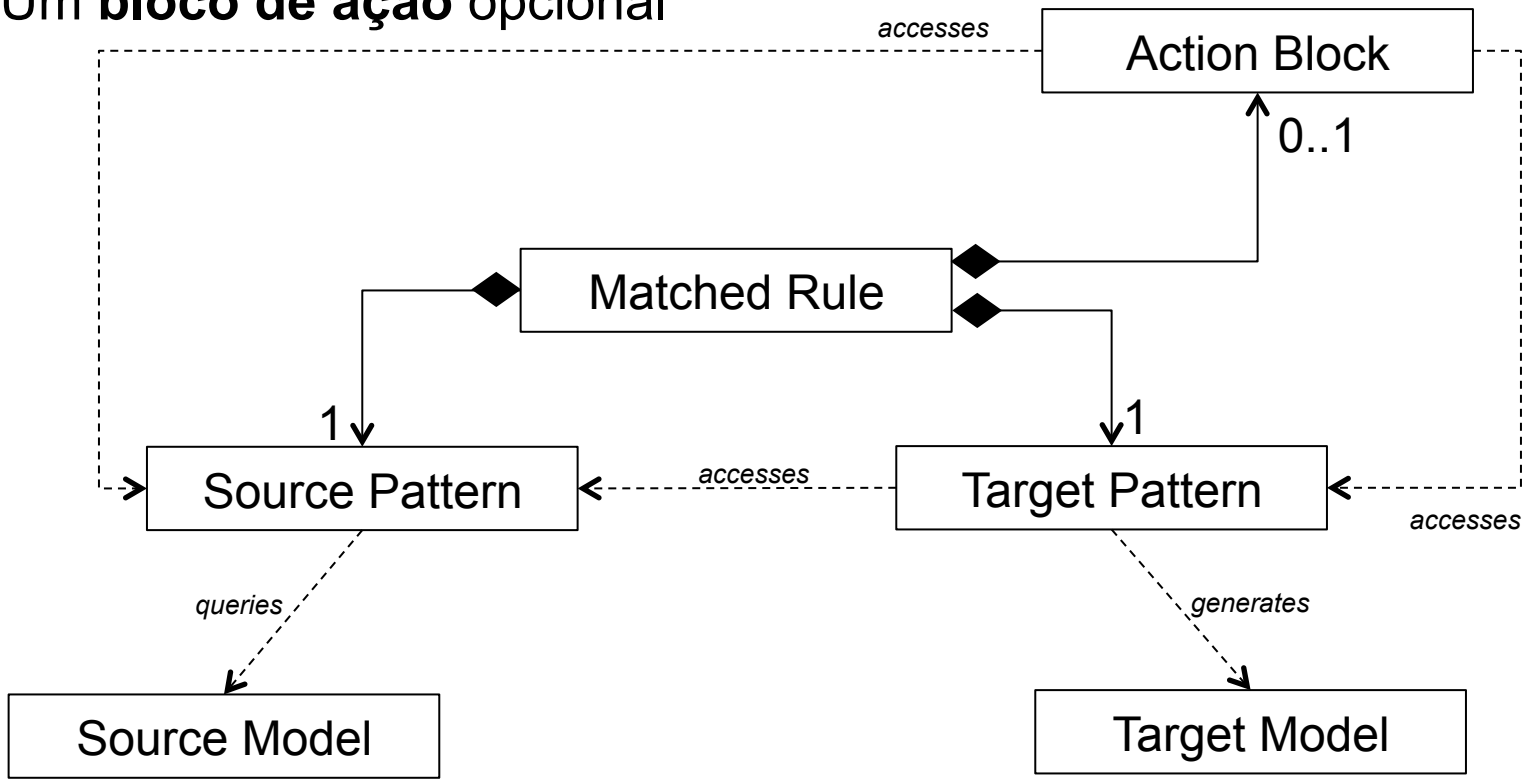
ATL visão geral (continuação)

- **Aplicando** a regra significa:
  - Criar os elementos de destino especificados
  - Inicializar as propriedades dos elementos recém-criados

- Existem dois tipos de regras relacionadas a sua aplicação
  - Regras de **combinação** são aplicadas **uma vez** para cada correspondência pelo mecanismo de execução
    - Um determinado conjunto de elementos só pode ser combinado por uma regra de combinação
  - Regras **Called/Lazy** são aplicadas **quantas vezes** elas forem chamadas por outras regras

Regras de combinação: visão geral

- Uma **regra de combinação** é composta de
  - Uma **origem padrão**
  - Um **destino padrão**
  - Um **bloco de ação** opcional



Regras de combinação: origem padrão

- A **origem padrão** é composta de
  - Um conjunto de **elementos origem padrão** rotulados
  - Um elemento origem padrão refere-se a um tipo contido nos metamodelos origem
  - Um **guard** (Expressão Boleana OCL) usada para filtrar combinações

- Uma combinação corresponde a um **conjunto de elementos** provenientes dos modelos de origem que:
  - Cumpra os tipos especificados pelos elementos de origem padrão
  - Satisfaça o guard

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

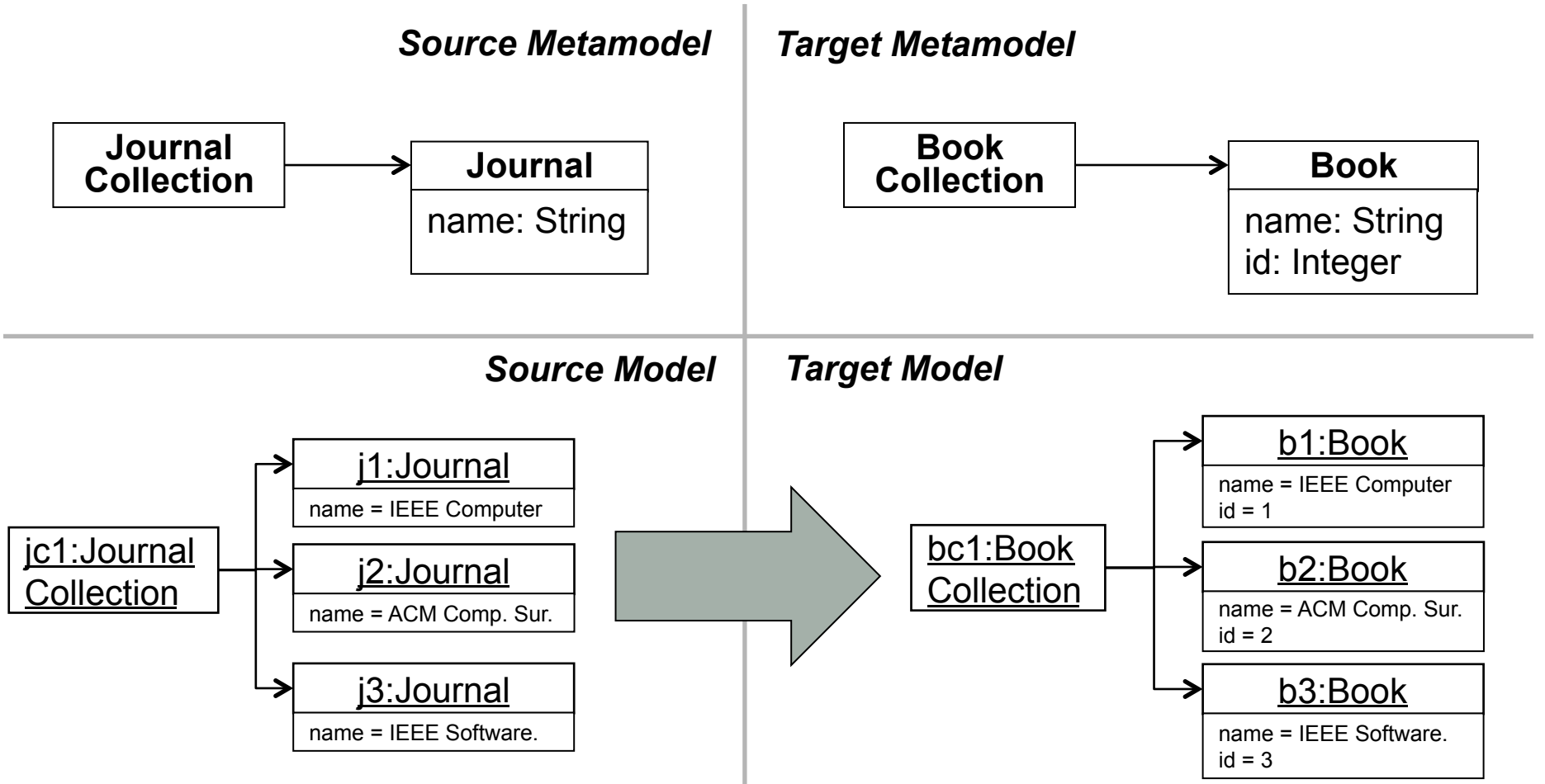

Regras de combinação: destino padrão

- O destino padrão é composto de
  - Um conjunto **elementos destino padrão** rotulados
  - Cada elemento destino padrão
    - refere-se a um tipo de metamodelo destino
    - contém um conjunto de ligações
  - Uma **ligação** inicia uma propriedade de um elemento destino usando uma expressão OCL
- Para cada combinação, o destino padrão é aplicado
  - Elementos são criados nos modelos de destino
  - Elementos de destino são inicializados pela execução das ligações

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


Exemplo #1 – Publication 2 Book

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



Exemplo #1

Configuração de Metamodelos Origem/Destino

- **Header**

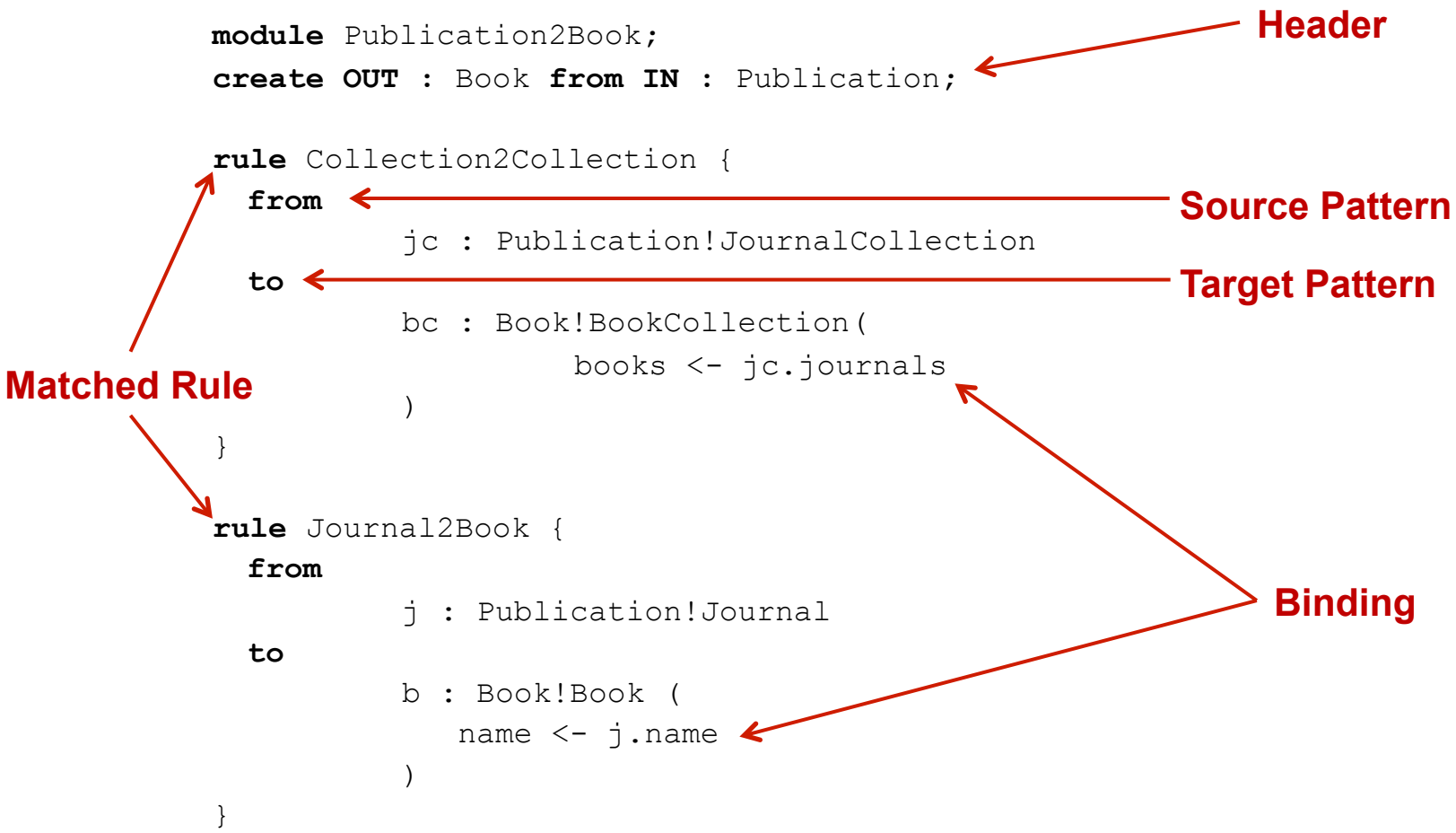
```
module Publication2Book;
create OUT : Book from IN : Publication;
```

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

Exemplo #1

Regras de combinação

- **Exemplo Publication 2 Book**



Exemplo #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

Exemplo #1

Helpers 2/2

- **Exemplo 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

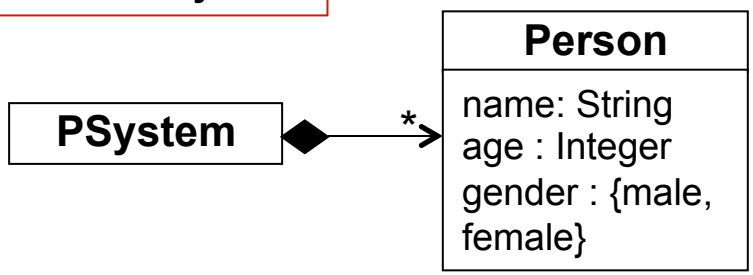
Global Operation call

Module Instance for accessing global variables

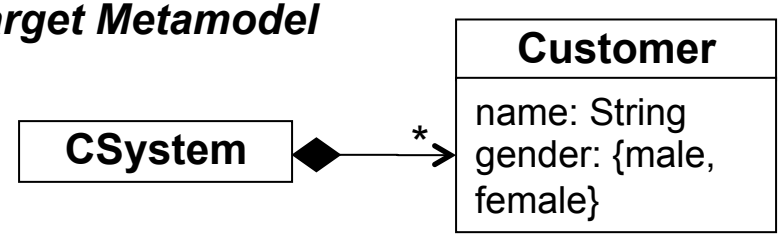
Exemplo #2 – Person 2 Customer

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

Source Metamodel

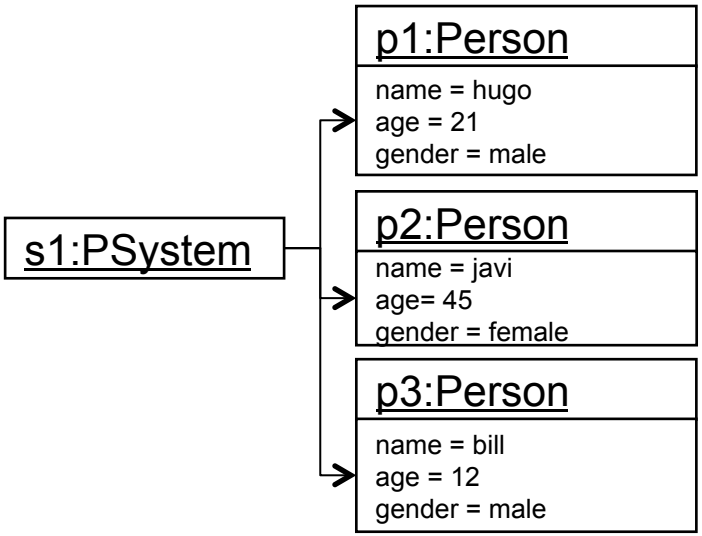


Target Metamodel

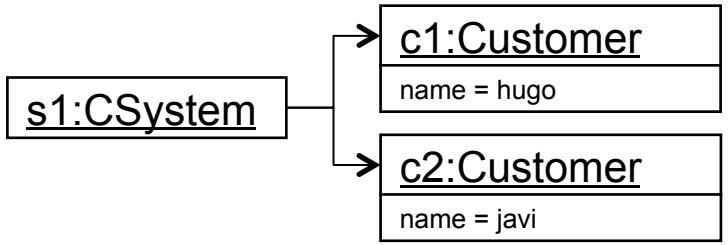


Constraint: Customer must be older than 18 years

Source Model



Target Model



Exemplo #2

Descrevendo correspondências e ligações mais complexas

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

- **Exemplo: IF/ELSE**

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


Exemplo #2

Guards Conditions in Source Patterns (1/2)

- **Exemplo 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

Exemplo #2

Condições de Guards em Padrões de Origem (2/2)

- **Exemplo Person2Customer**

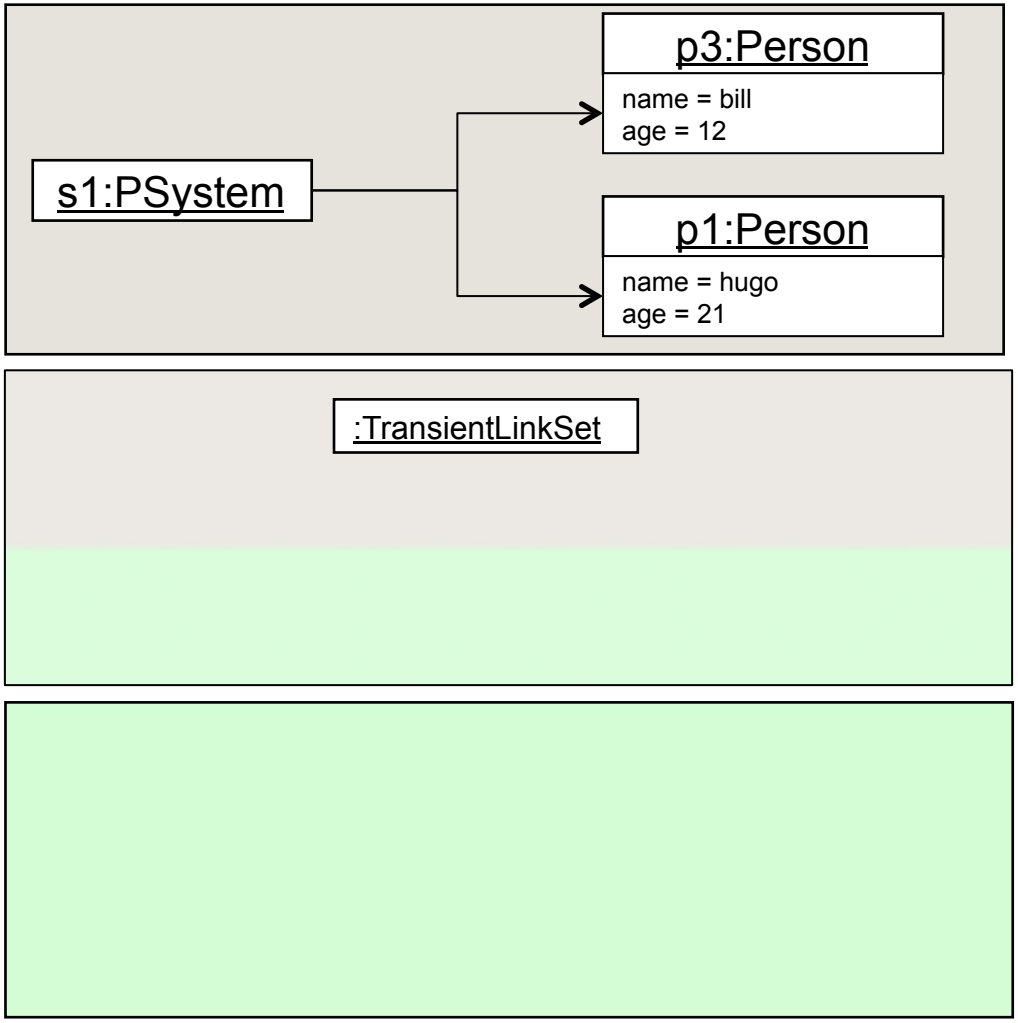
```
rule Person2Customer {
  from
       p : Person!Person (p.age > 18)
  to
       c : Customer!Customer (
          name <- p.name
   )
}

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

Subset for Binding is computed by ATL Engine ☺

Exemplo #2

Modelo de Trace Implicito – Fase 1: Fase de Inicialização do módulo



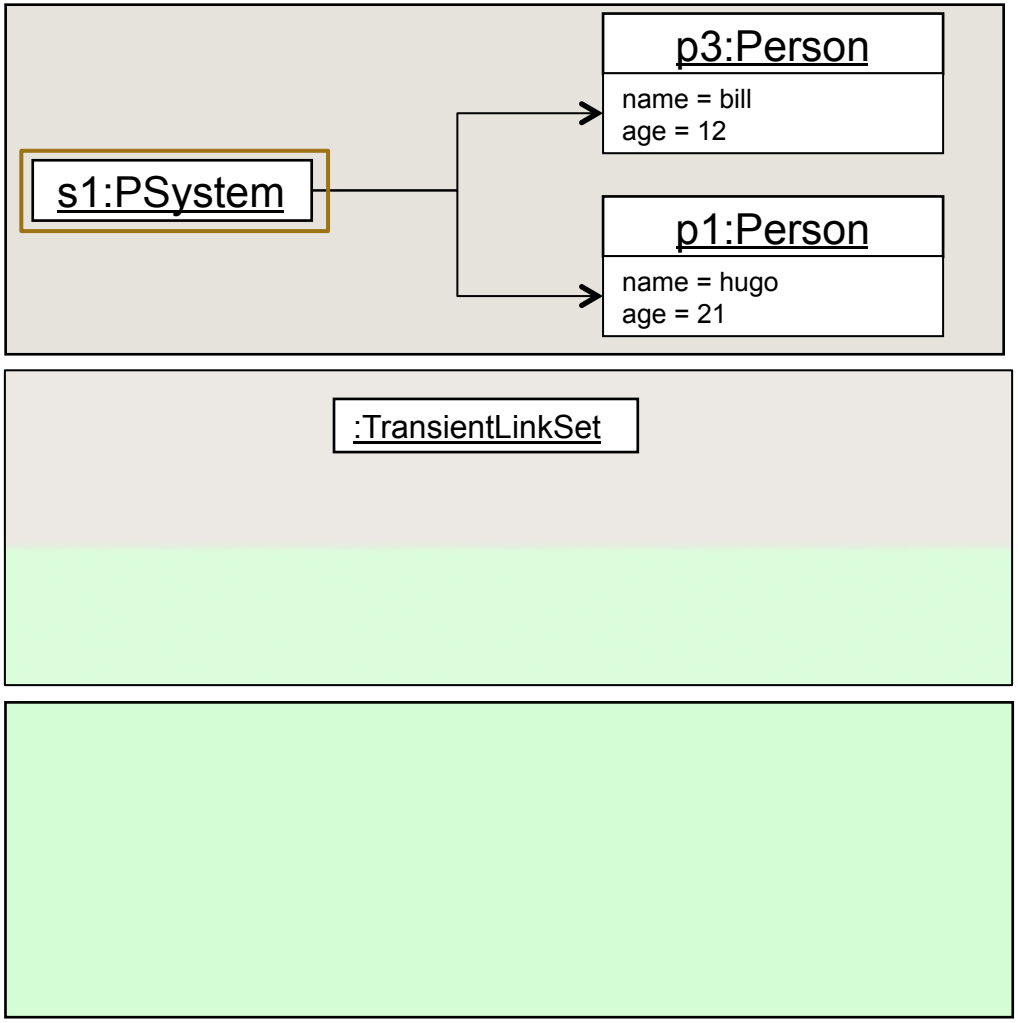

```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
   )
}


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

Modelo de trace é um conjunto (cf. TransientLinkSet) de **links** (cf. TransientLink) que relaciona elementos **de origem** com seus elementos **de destino** correspondentes

Exemplo #2

Modelo de Trace Implícito – Fase 2: Fase de Combinação

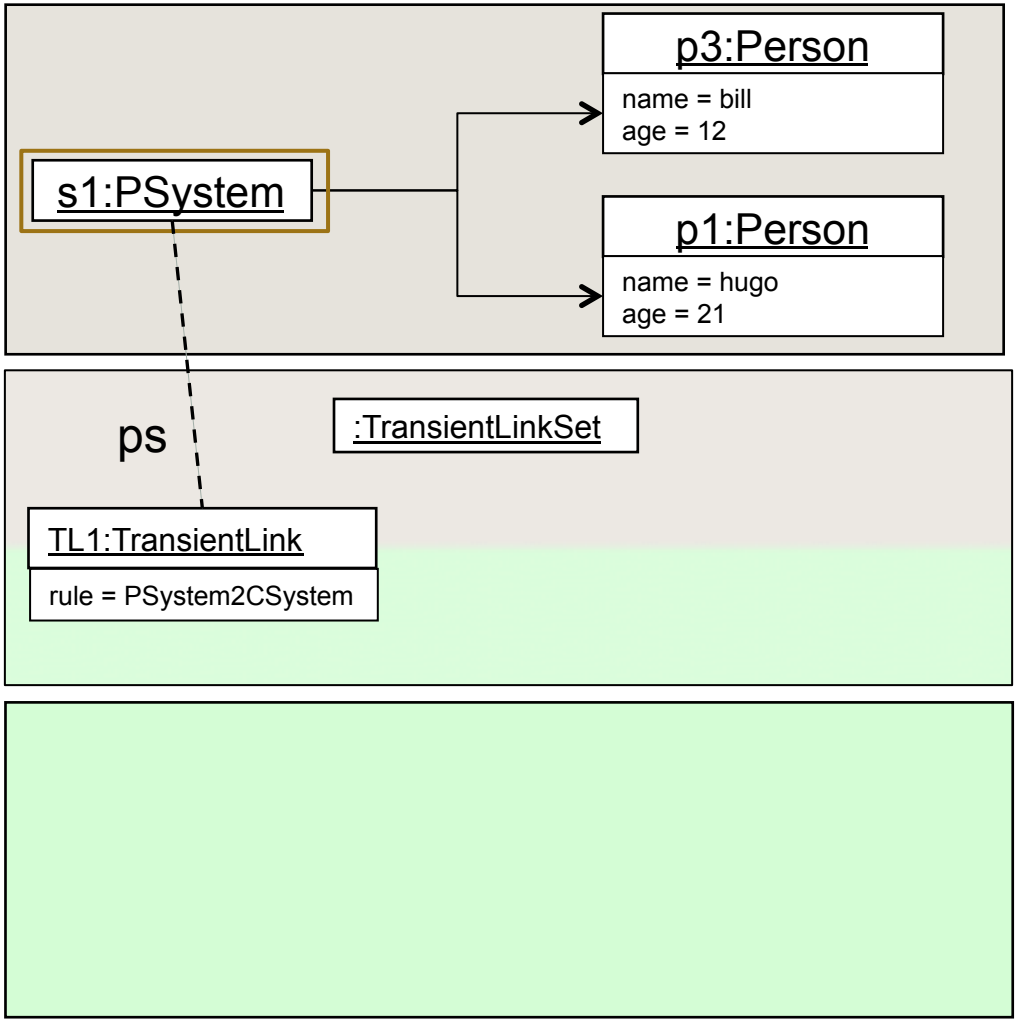



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
   )
}


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


Exemplo #2

Modelo de Trace Implícito – Fase 2: Fase de combinação

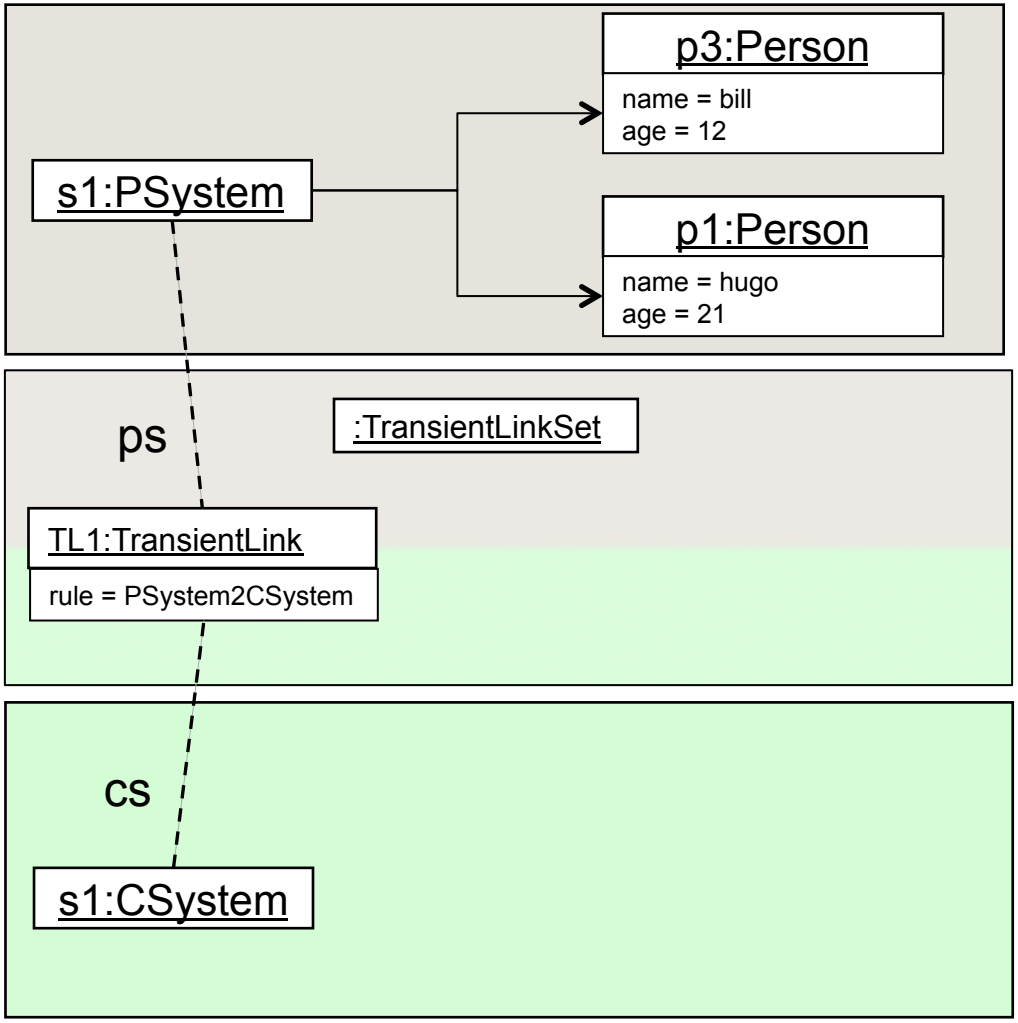



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
   )
}


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


Exemplo #2

Modelo de Trace Implícito – Fase 2: Fase de combinação

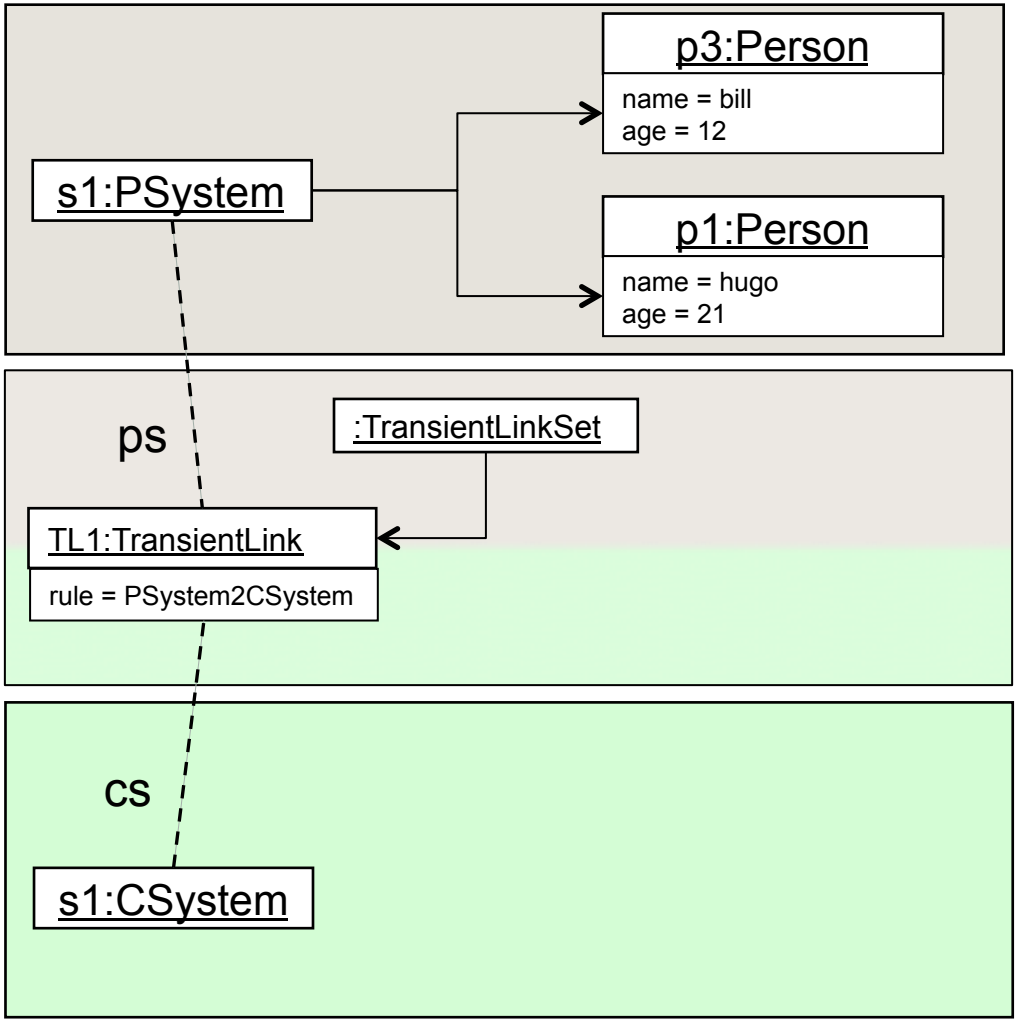



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
   )
}


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


Exemplo #2

Modelo de Trace Implícito – Fase 2: Fase de combinação

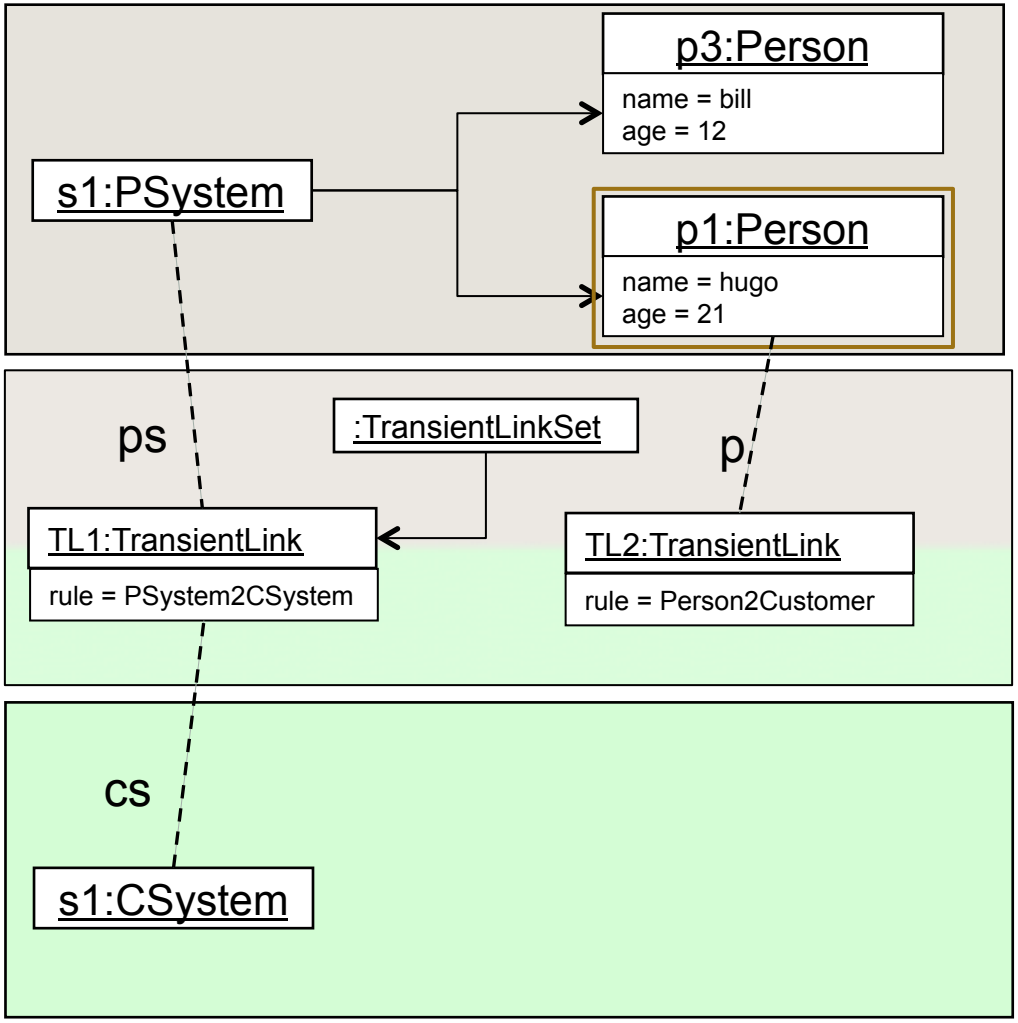



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
   )
}


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


Exemplo #2

Modelo de Trace Implícito – Fase 2: Fase de combinação

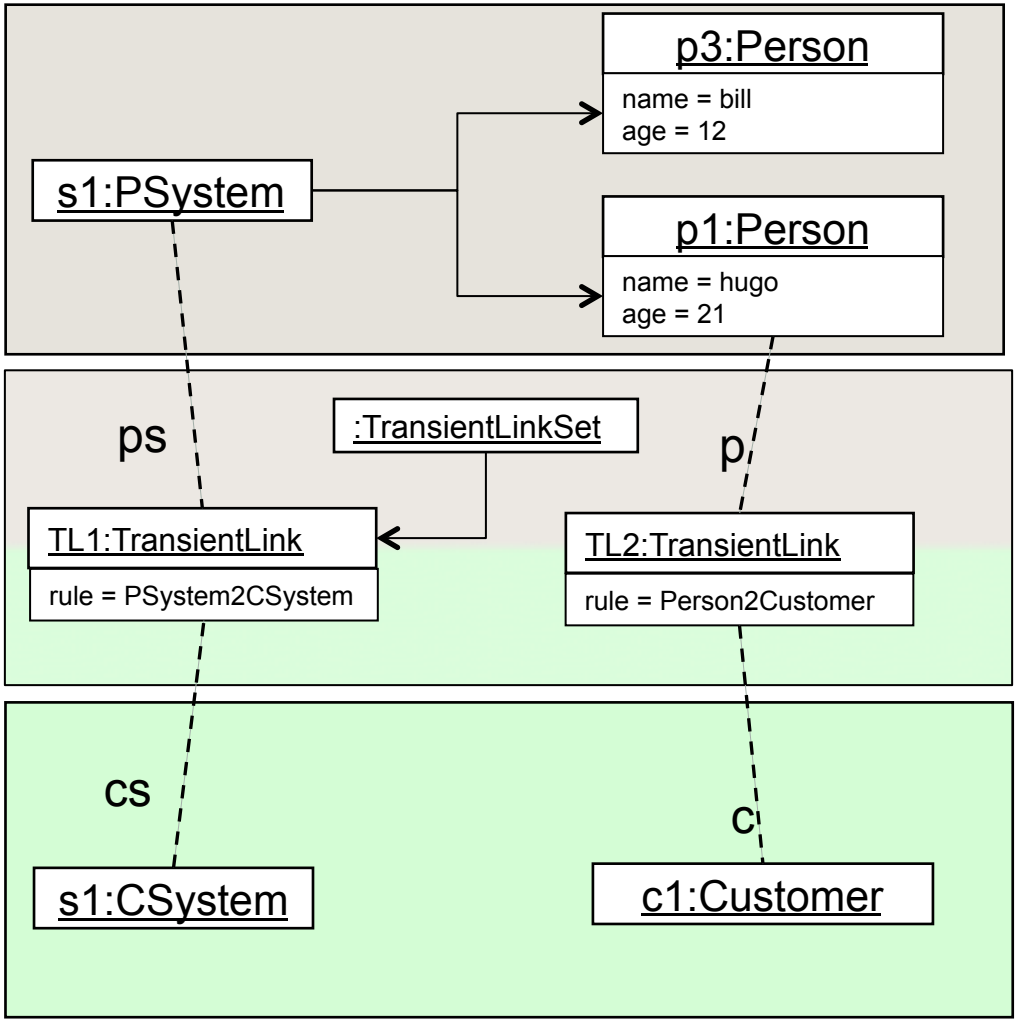



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
   )
}


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


Exemplo #2

Modelo de Trace Implícito – Fase 2: Fase de combinação

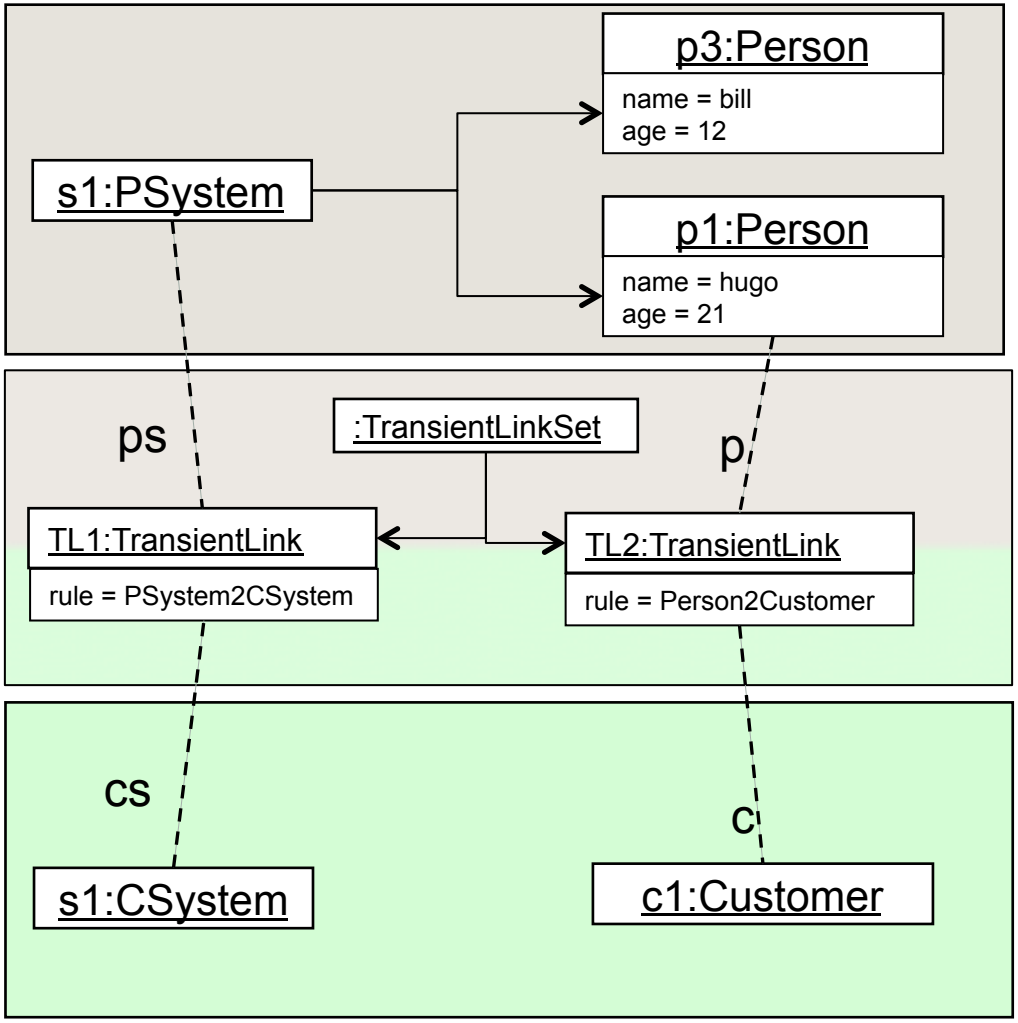



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
   )
}


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


Exemplo #2

Modelo de Trace Implícito – Fase 2: Fase de combinação

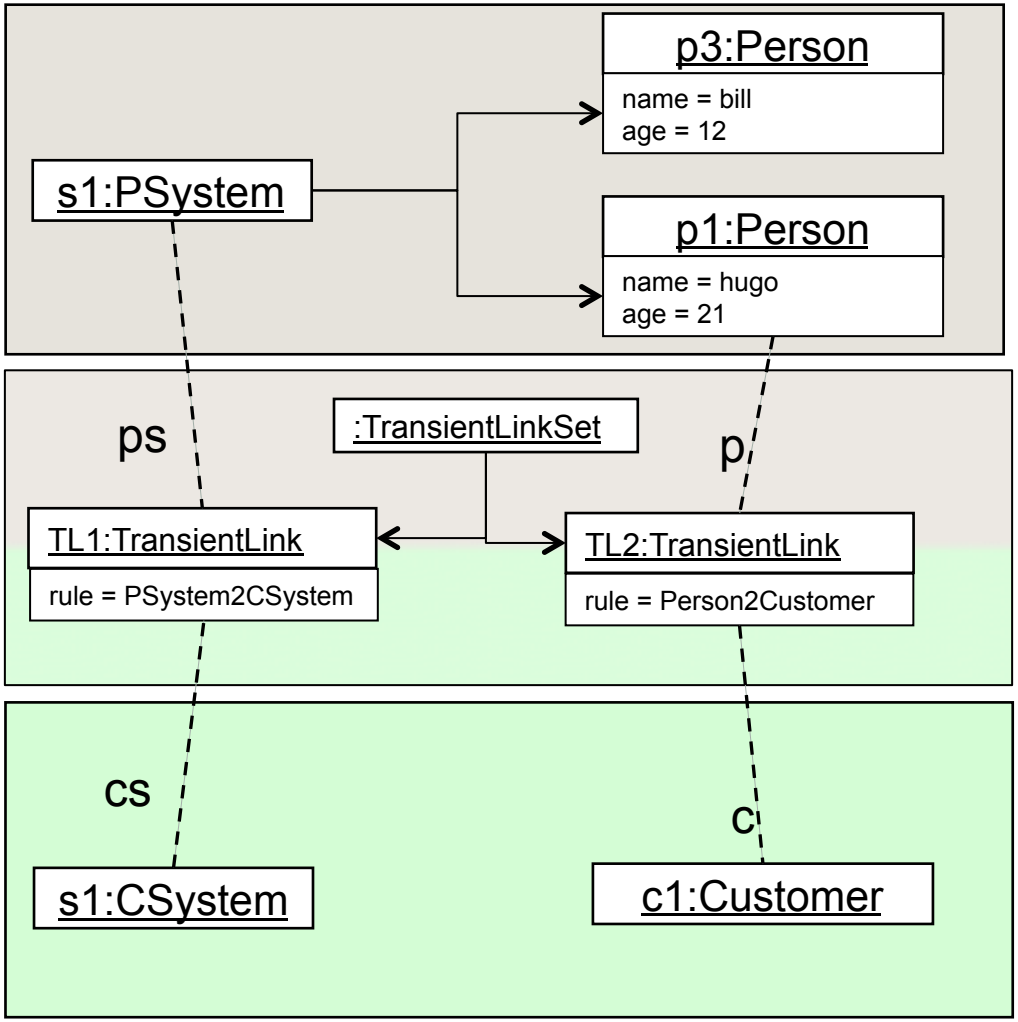



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
   )
}


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


Exemplo #2

Modelo de Trace Implícito – Fase 3: Fase de inicialização do destino

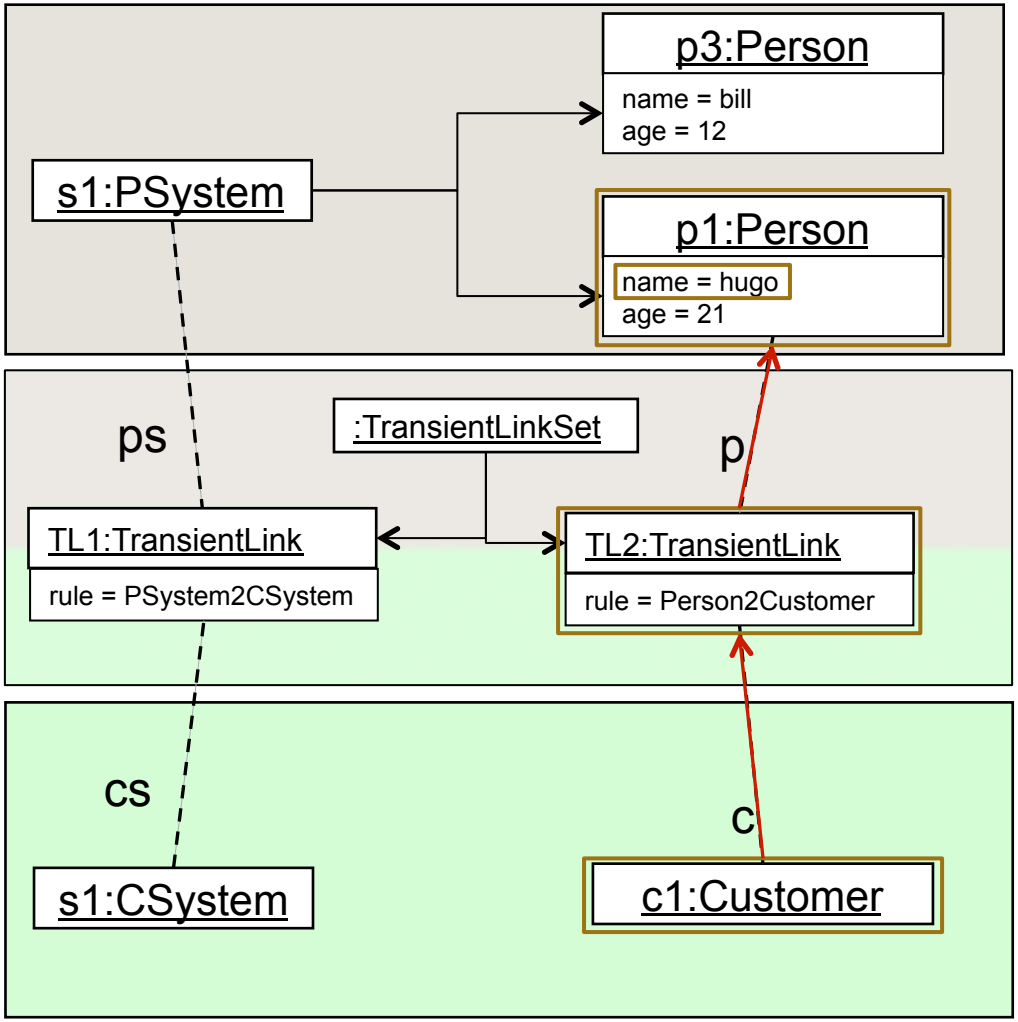



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
   )
}


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


Exemplo #2

Modelo de Trace Implícito – Fase 3: Fase de inicialização do destino

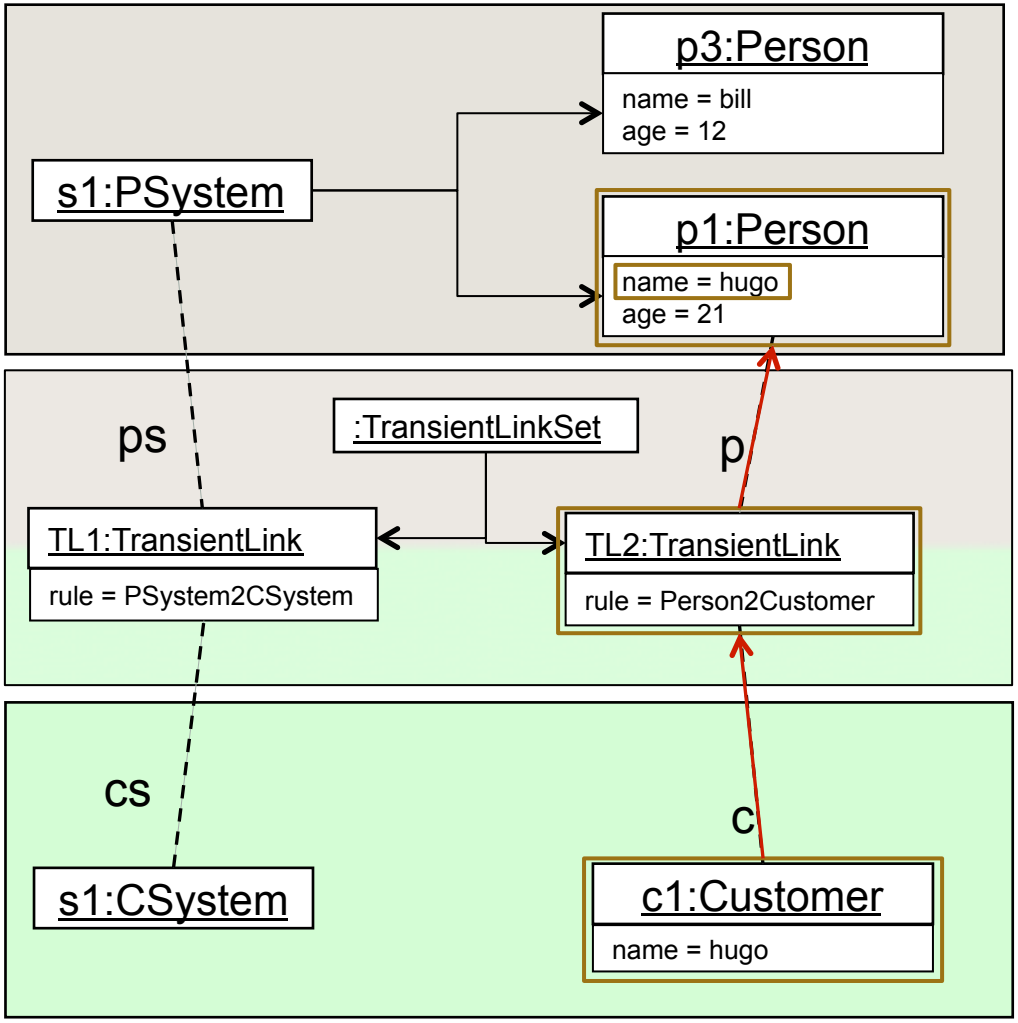



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
   )
}


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


Exemplo #2

Modelo de Trace Implícito – Fase 3: Fase de inicialização do destino

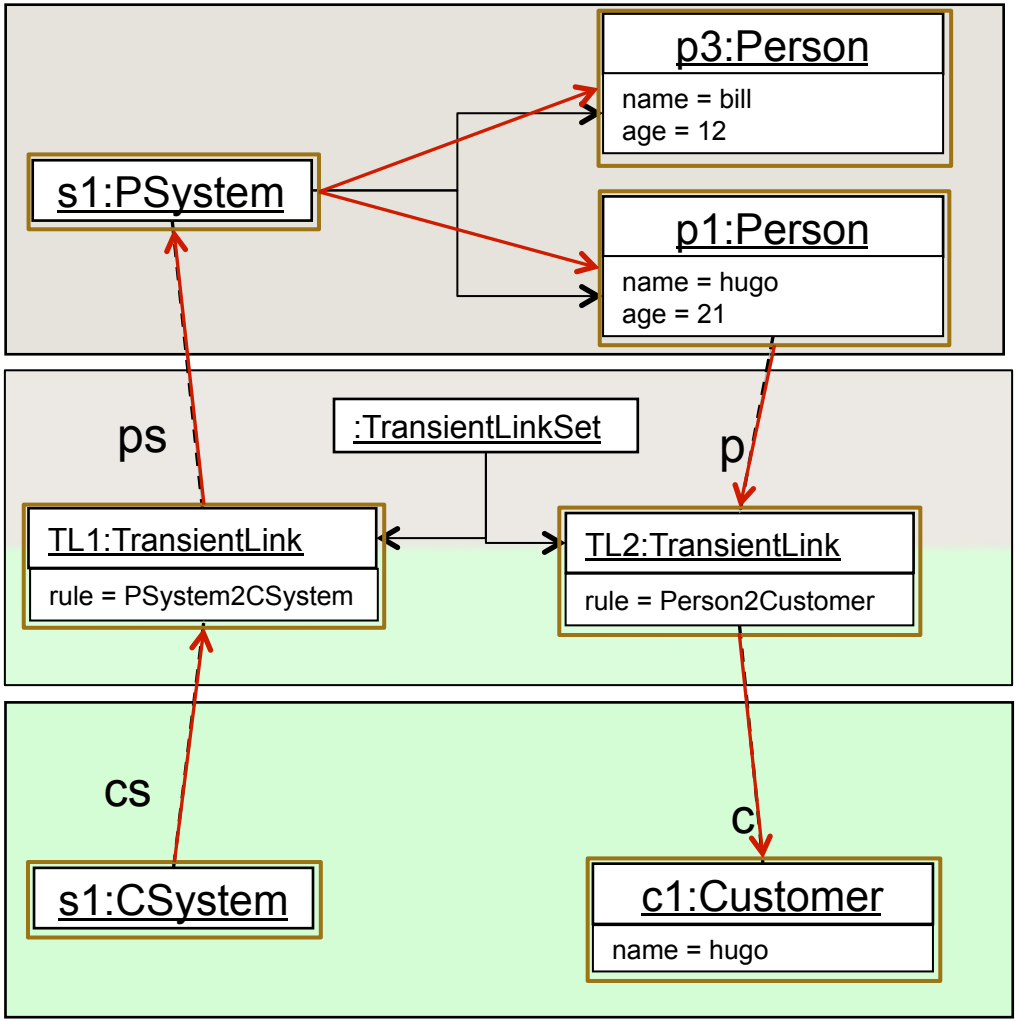



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
   )
}


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


Exemplo #2

Modelo de Trace Implícito – Fase 3: Fase de inicialização do destino

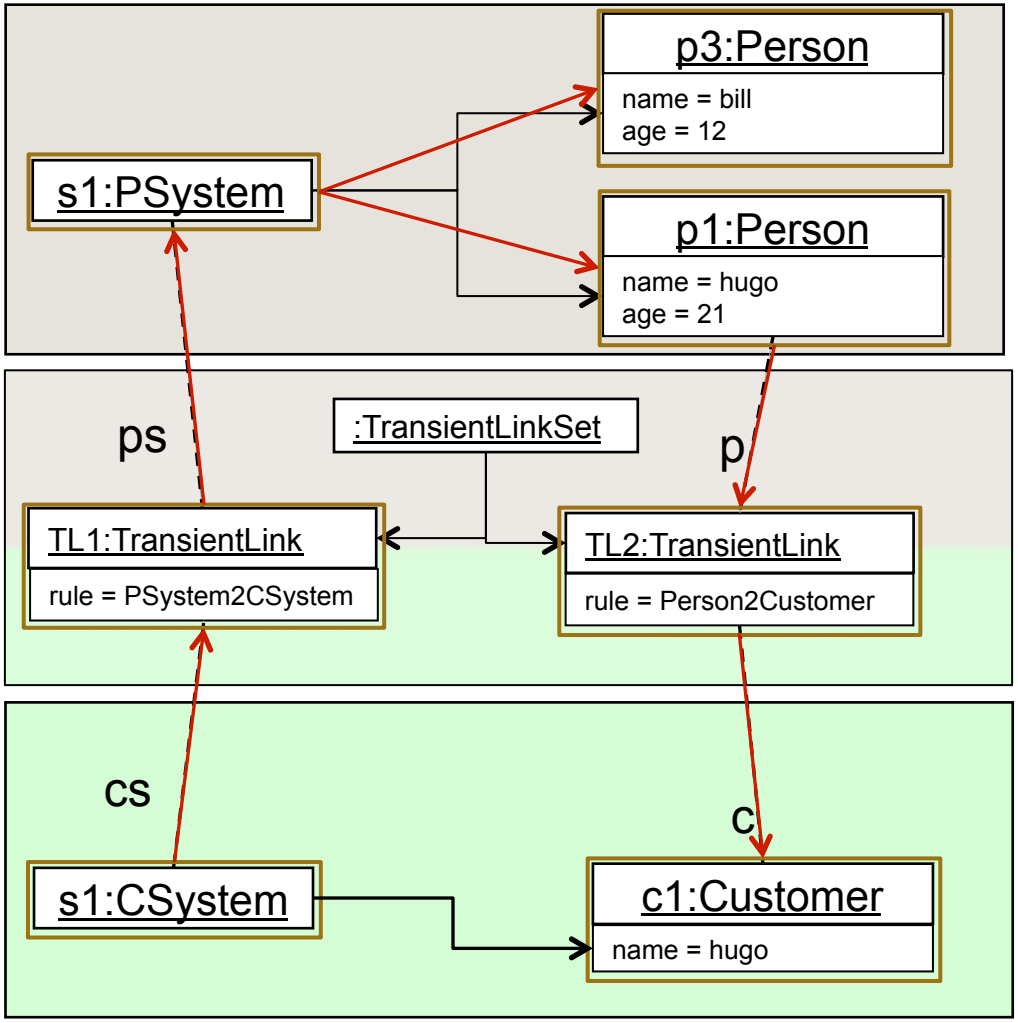



```
rule PSystem2CSystem {
  from
          ps : Person!PSystem
  to
          cs : Customer!CSystem (
             customer <- ps.person
   )
}


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


Exemplo #2

Modelo de Trace Implícito – Fase 3: Fase de inicialização do destino




```
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
   )
}
```


Fases da Execução das Transformações

1. Fase de **Inicialização do Módulo**
   - As variáveis dos módulos (atributos de helpers) e modelos de trace são inicializados
   - Se um **ponto de entrada** *"called rule"* é definido, ele será executado nesse passo
2. Fase de **Combinação**
   - Usando os padrões de origem (**from**) de *regras de combinação,* elementos são selecionados no modelo de origem (cada combinação deve ser **única**)
   - Via o *padrão de destino* (**para**) elementos correspondentes são criados no modelo destino (para cada combinação existe a mesma quantia de elementos destino criados como padrões destino usado)
   - Informações de rastreabilidade são armazenadas
3. Fase de **Inicialização do Destino**
   - Os elementos no modelo de destino são inicializados baseados nas ligações (**<-**)
   - A função resolveTemp é avaliada, baseada nas informações de rastreabilidade
   - O Código imperativo (**do**) é executado, incluindo possíveis chamadas para *called rules*

Exemplo #2

Solução Alternativa com Called Rule e Bloco de Ação (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);
}
```


Exemplo #2

Solução Alternativa com Called Rule e Blocos de Ação (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

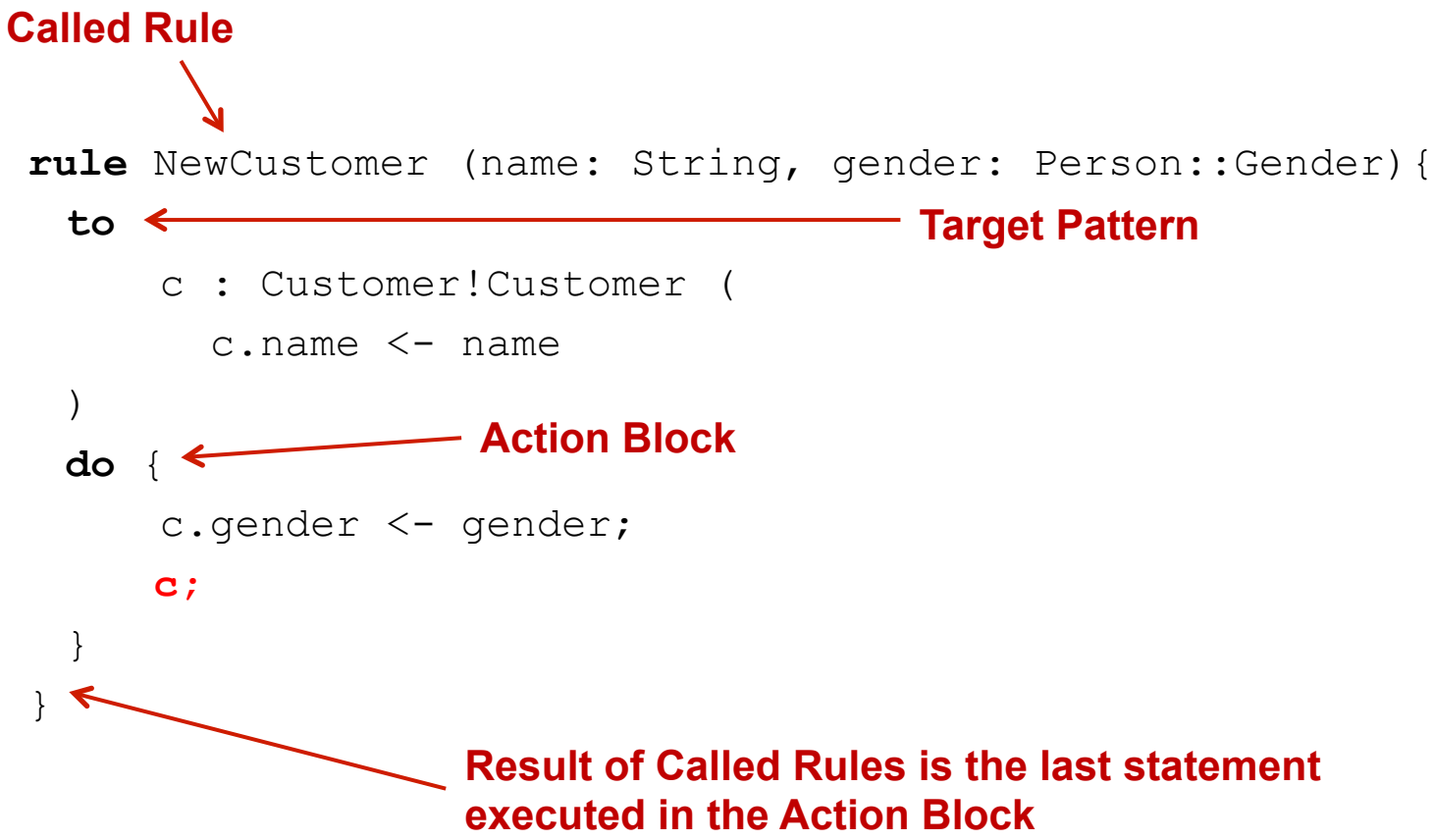
Action Block

Binding

Called Rule Call

Exemplo #2

Solução Alternativa com Called Rule e Blocos de Ação (2/3)



Exemplo #2

Solução Alternativa com Lazy Rule e Bloco de Ação (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

Exemplo #2

Solução Alternativa com Lazy Rule e 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

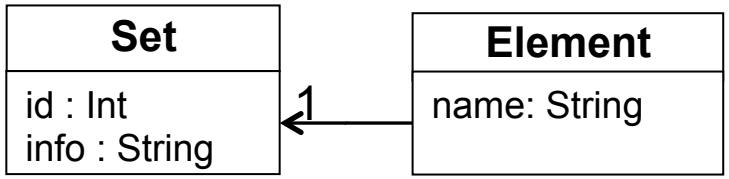
Target Pattern

Result of Lazy Rules is the first specified target pattern element

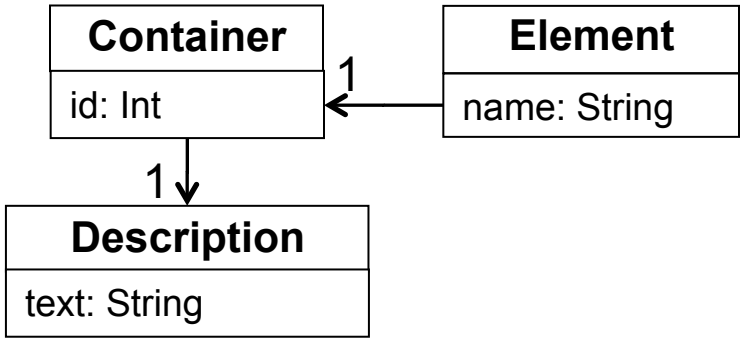
Exemplo #3

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

Source Metamodel



Target Metamodel



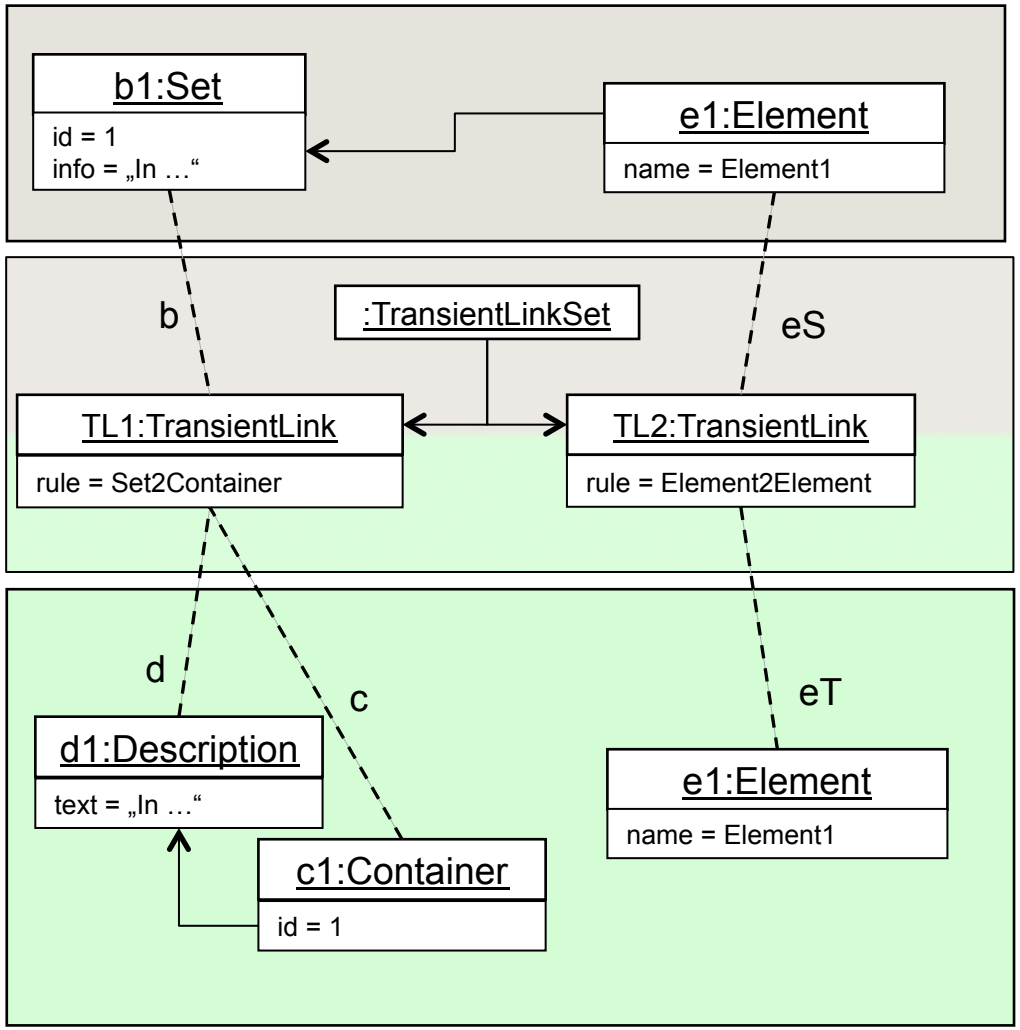
Transformation

```
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')
   )
}
```


Exemplo #3

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



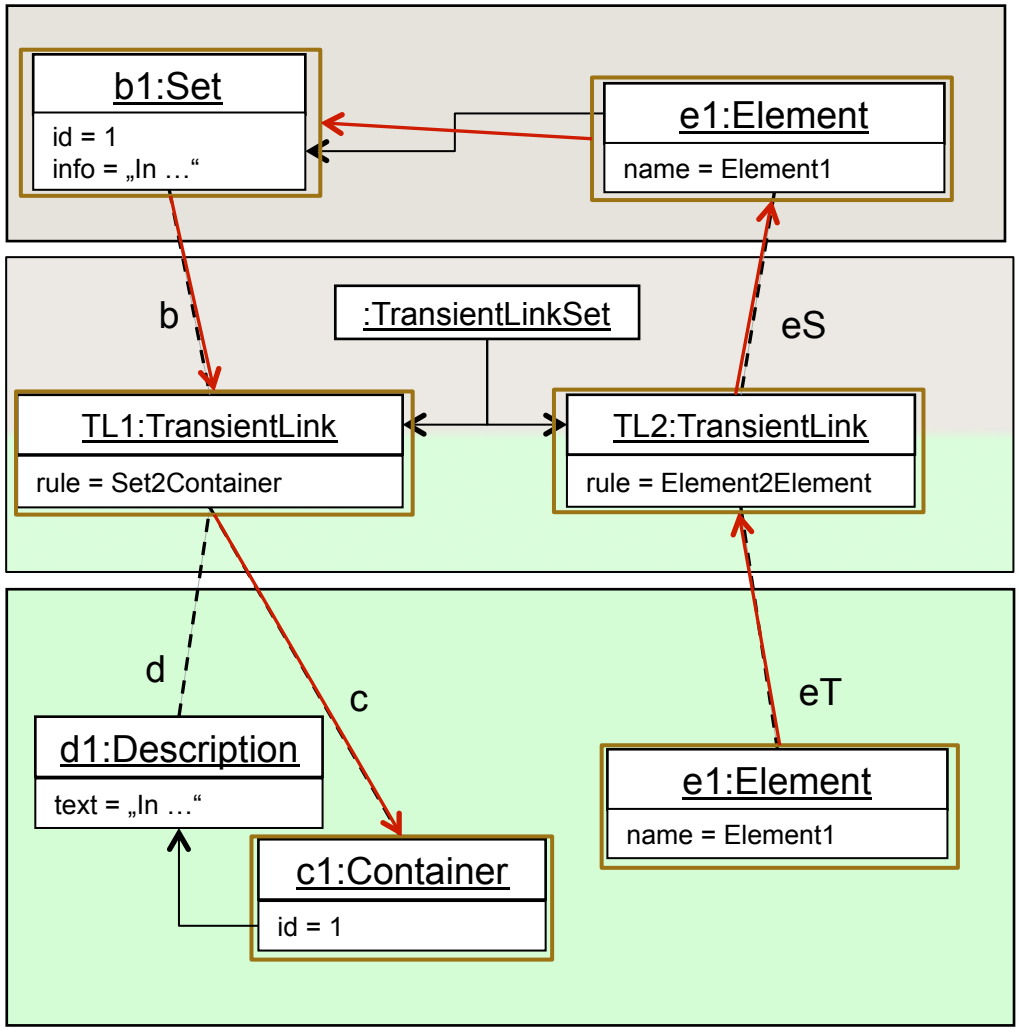

```
rule Set2Container {
    from
        b : source!Set
    to
        d : target!Description(
            text <- b.info
        )
        c : target!Container(
            id <- b.id,
            description <- d

        )
}
```

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


Exemplo #3

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



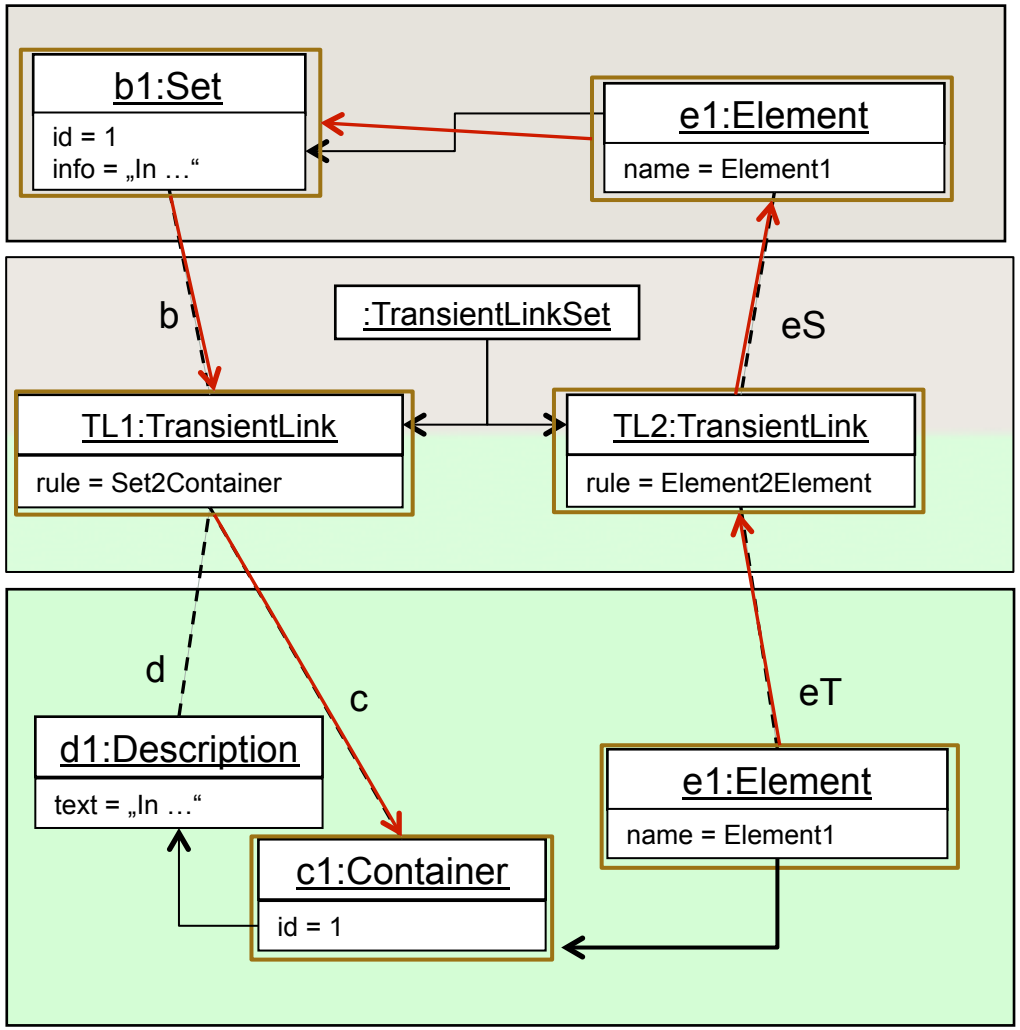

```
rule Set2Container {
    from
        b : source!Set
    to
        d : target!Description(
            text <- b.info
        )
        c : target!Container(
            id <- b.id,
            description <- d

        )
}
```

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


Exemplo #3

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



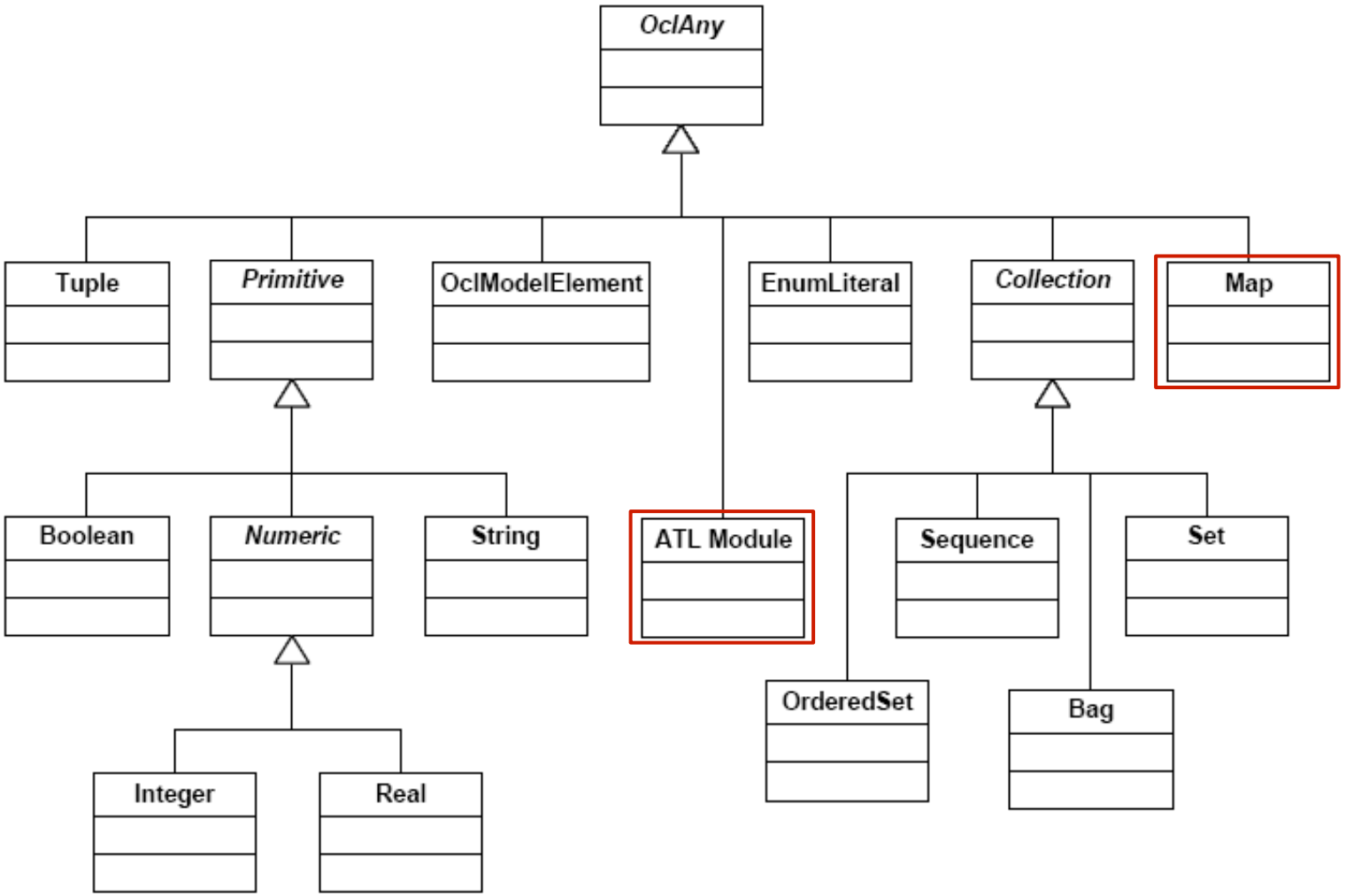

```
rule Set2Container {
    from
        b : source!Set
    to
        d : target!Description(
            text <- b.info
        )
        c : target!Container(
            id <- b.id,
            description <- d

        )
}
```

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


Tipos de Dados ATL

Operações OCL para cada tipo



Regra de herança

- Regras de herança permitem a **reutilização** de regras de transformação

- Uma regra filha combina um **subconjunto** do que as regras do seu pai combinam
  - Todas as **ligações** e **filtros** do pai ainda fazem sentido para o filho
- Uma regra filha deve **especializar** elementos destino das suas regras pai
  - Inicialização de elementos existentes podem ser especializados
- Uma regra filha deve **extender** elementos destino da suas regras pai
  - Novos elementos podem ser criados
- Uma regra pai pode ser declarada como **abstrata**
  - Então, a regra não é executada, mas apenas reutilizada por seus filhos

- **Sintaxe**

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


Regra de Herança

Exemplo #1


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


Regra de Herança

Exemplo #2


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

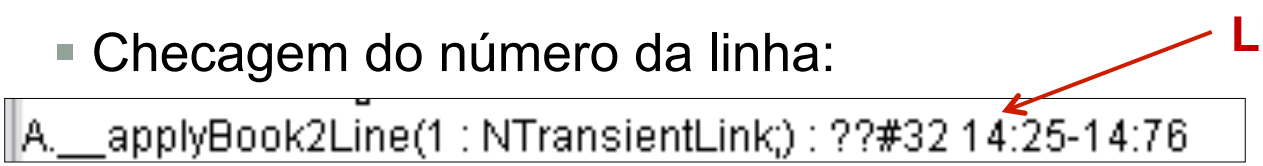
A match has to be unique!

Sugestões de Debbuging

- Rápida e Suja: Faz o uso de .debug()
- Procece em pequenos incrementos
- Testa mudanças imediatamente
- Lê Trace de Exceção
  - Visão top-down da pilha de trace „ERROR“ para encontrar uma mensagem significativa:

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

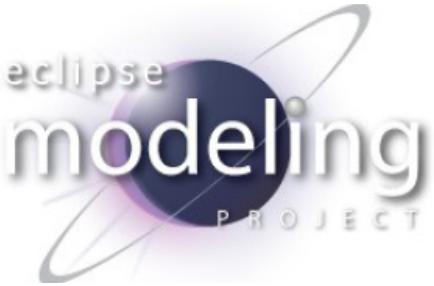
  - Checagem do número da linha:



- do-blocks podem ser usados para saída de debug temporária

ATL na prática

- Ferramentas e documentação ATL estão disponíveis em: http://www.eclipse.org/atl
  - Mecanismo de execução
    - Máquina virtual
    - Compilador ATL para byte code
  - Ambiente de Desenvolvimento Integrado (IDE) para
    - Editor com destaque de sintaxe e outline
    - Suporte de execução com configuração de launch
    - Debugger a nível de código
  - Documentação
    - Guia de inicialização
    - Manual do usuário
    - Guia do usuário
    - Exemplos básicos



Resumo

- ATL é especializada em transformações de modelo out-place
  - Problemas simples são geralmente resolvidos facilmente
- ATL suporta características avançadas
  - Navegação OCL complexa, called rules, modo de refinamento, regras de herança, etc
  - Muitos problemas complexos podem ser manipulados de forma declarativa
- ATL tem características declarativas e imperativas
  - Qualquer problema de transformação out-place pode ser tratado

- Maiores informações
  - Documentação: http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language
  - Exemplos: http://www.eclipse.org/m2m/atl/basicExamples_Patterns

Alternativas à ATL

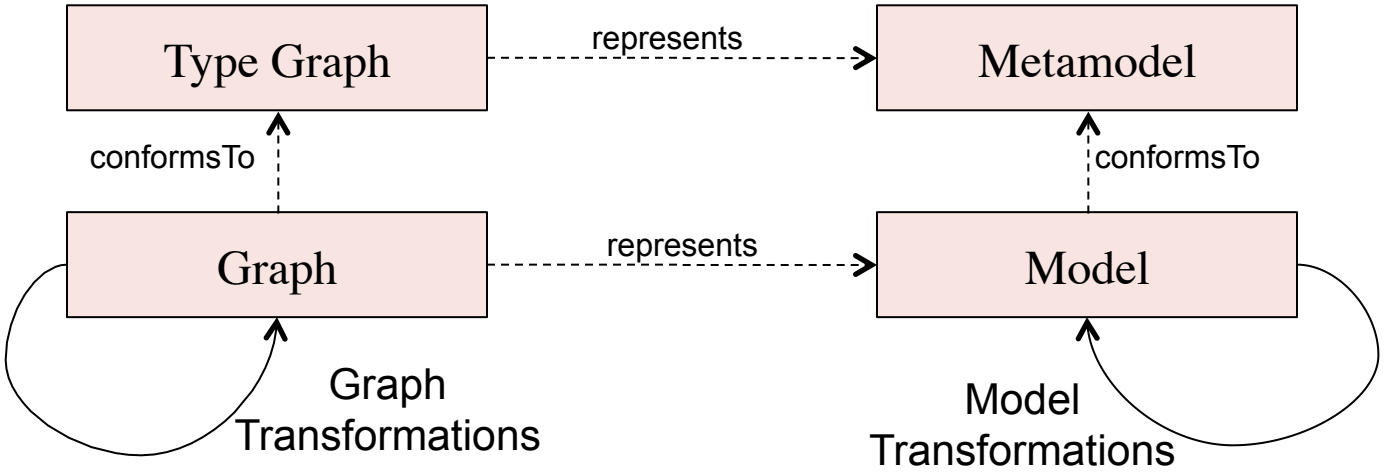
- **QVT**: Query-View-Transformation padrão da OMG
  - Linguagem Relacional QVT Declarativa
  - Linguagem Operacional QVT Imperativa
  - Linguagem Core QVT de baixo nível (nível VM)
- **TGG**: Triple Graph Grammars
  - Gráficos de correspondência entre metamodelos
  - Transforma modelos em ambas as direções, integra e sincroniza modelos
- **JTL**: Janus Transformation Language
  - Forte foco nas sincronizações de modelos pela propagação de mudanças
- **ETL**: Epsilon Transformation Language
  - Linguagem designada no framework Epsilon para transformações out-place
- **RubyTL**: Ruby Transformation Language
  - Extensão da linguagem de programação Ruby
  - Conceitos fundamentais para transformações de modelos out-place (extensíveis)
- **Muitas** outras linguagens, como VIATRA, Tefkat, Kermeta, SiTra, …

TRANSFORMAÇÕES IN-PLACE:
TRANSFORMAÇÕES DE GRAFO

Por que transformações de grafo?

- **Modelos** são **grafos**
  - Diagrama de Classe, Redes Petri, Máquinas de Estado, …
- **Tipo do Grafo**:
  - Generalização dos elementos gráficos
- **Transformações de Grafo**
  - Generalização das **evoluções** do grafo

⇒ Transformações de grafo são aplicáveis à modelos!

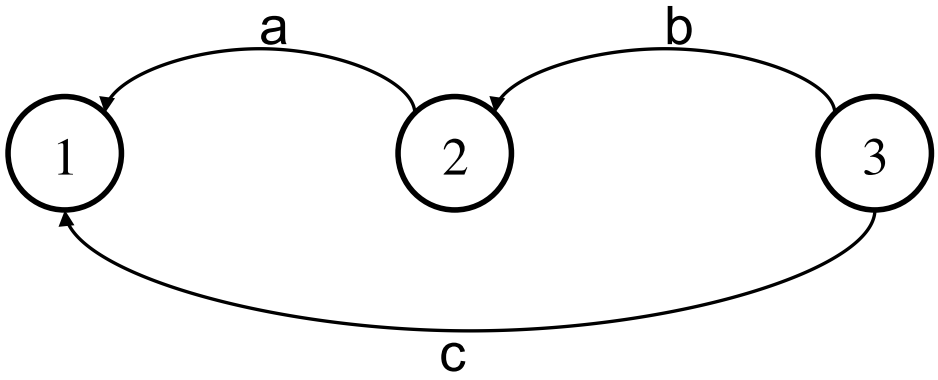


Noções básicas: Grafo direcionado

- Um **grafo direcionado** *G* consiste de dois conjuntos disjuntos
  - Vértices *V (Vértice)*
  - Arestas *E (Aresta)*
- Cada aresta tem uma vertice de origem s e um vértice de destino *t*

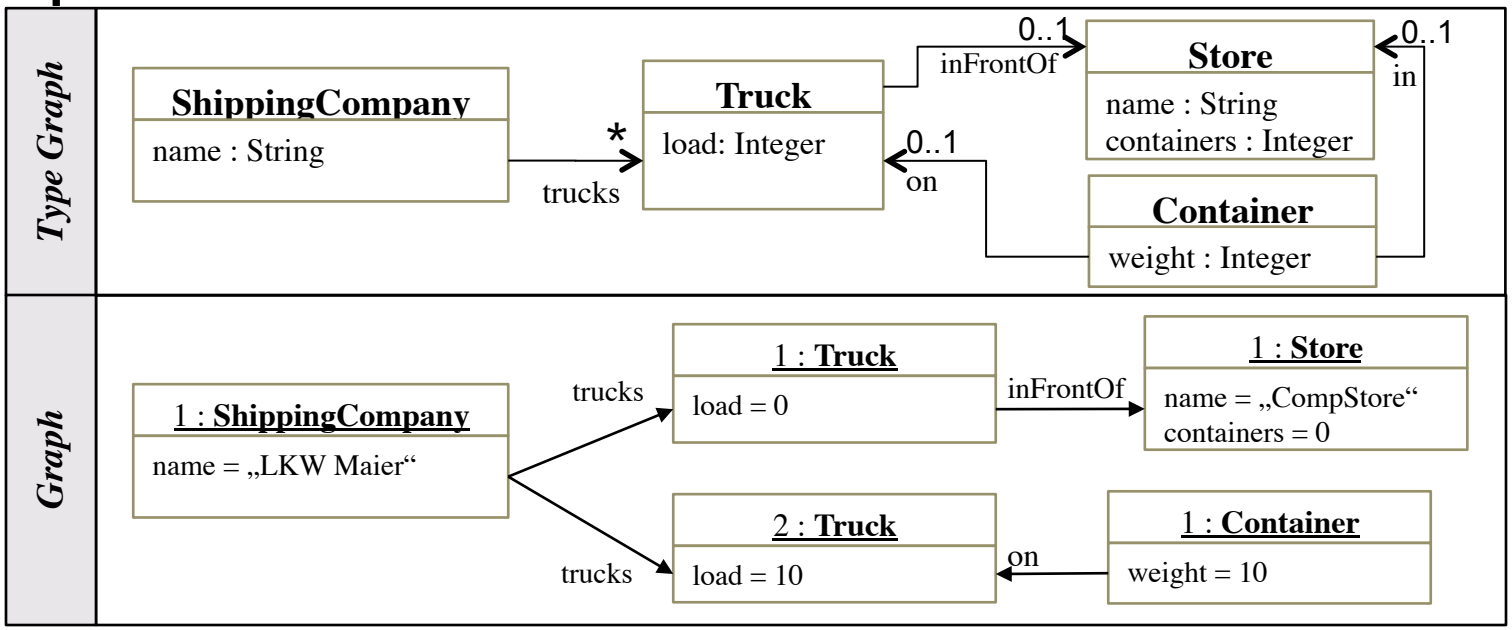
- Resumindo: $G = (V, E, s: E \rightarrow V, t: E \rightarrow V)$

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



Tipo de atributo do grafo(1/3)

- Para representar modelos, informação adicional é necessária
  - **Tipagem**: Cada vértice e cada aresta tem um **tipo**
  - **Atribuição**: Cada vértice/aresta tem um número arbitrário de pares de nomes/valores
- **Notação** para grafos *direcionados*, *'tipados'* e atribuídos
  - **Grafo** é representado como um **diagrama de objetos**
  - **Tipo de grafo** é representado como um **diagrama de classes**
- **Exemplo**

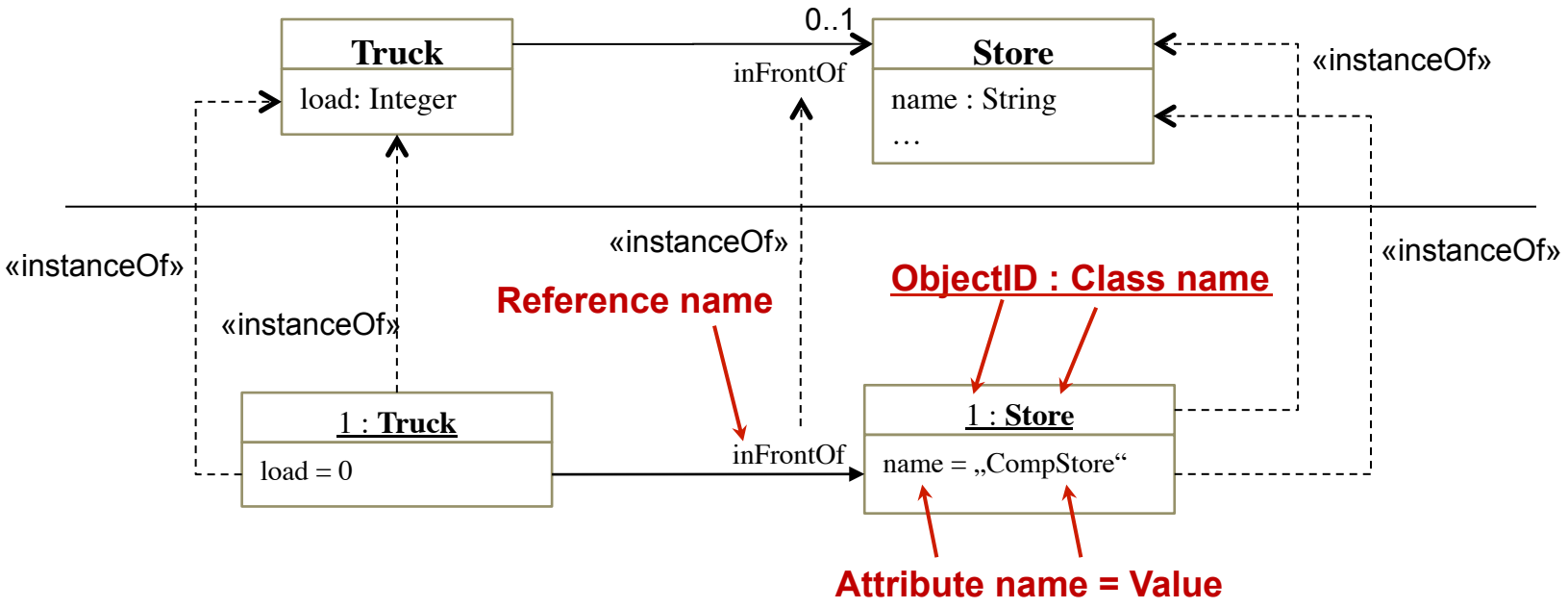


Example taken from: C. Ermel, M. Rudolf, and G. Taentzer, **The AGG approach: Language and environment**, *Handbook of Graph Grammars and Computing by Graph Transformation, Application, Languages and Tools,* volume 2, World Scientific, 1999.

Tipo de atributo do grafo(2/3)

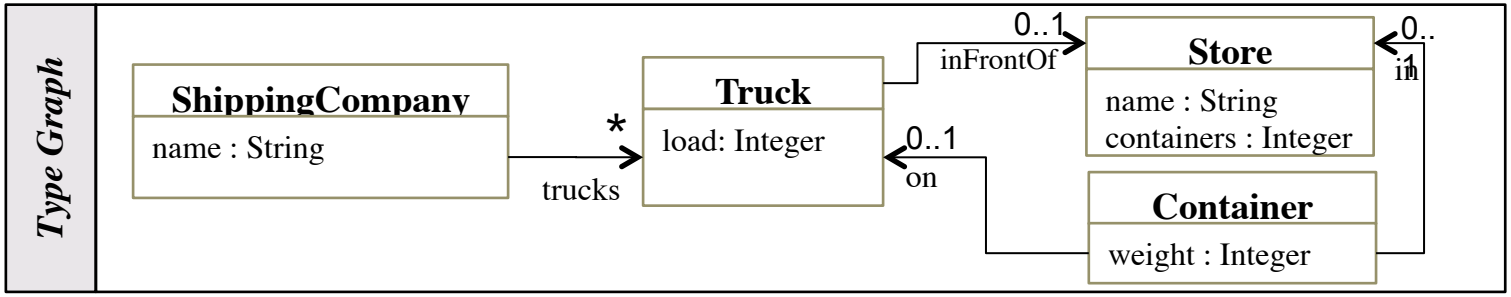
- **Diagrama de objetos**
  - Instância do diagrama de classes

- **Conceitos básicos**
  - **Objeto** : Instância da classe
  - **Valor**: Instância do atributo
  - **Link**: Instância da referência



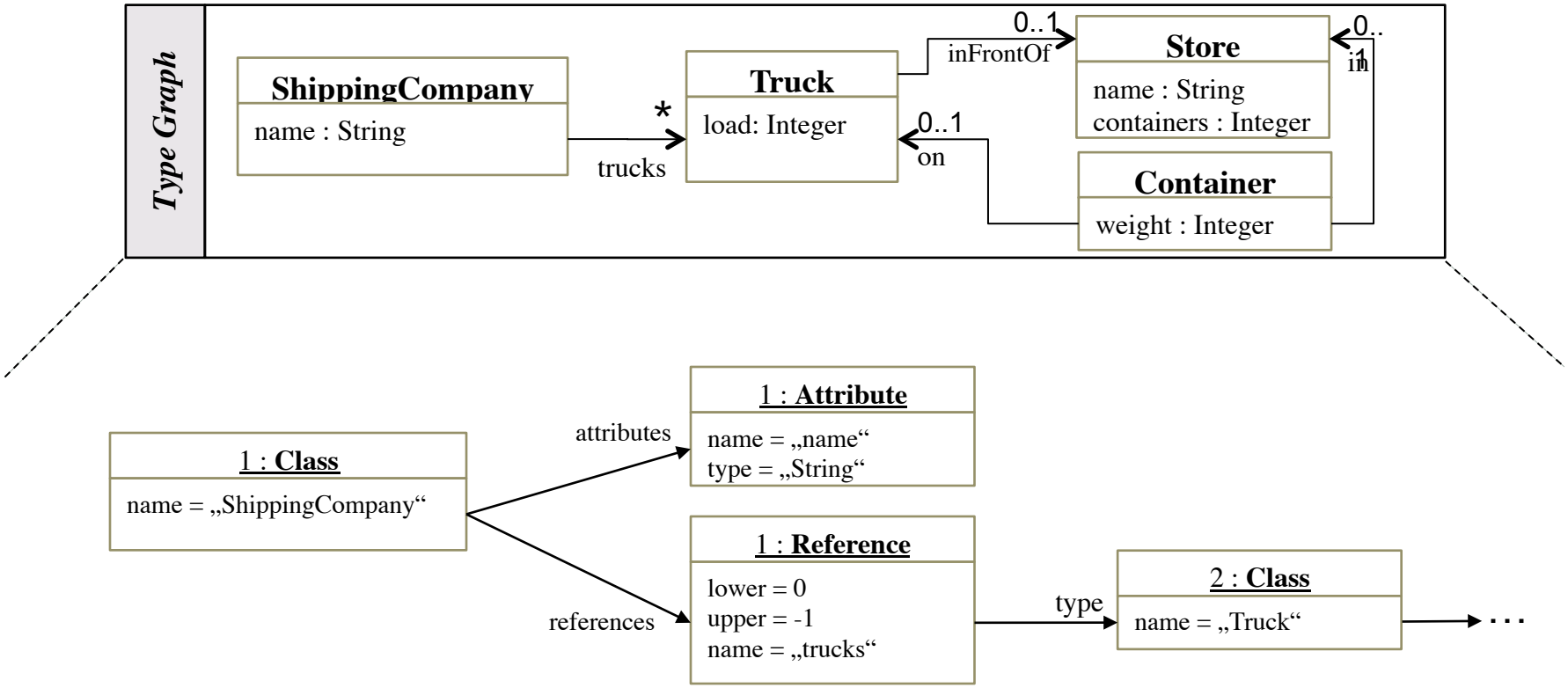
Tipo de atributo do grafo(3/3)

- Pergunta: Como fazer o tipo de grafo ser visto em forma de grafo puro (diagrama de objeto)?



Typed attributed graph (3/3)

- Pergunta: Como fazer o tipo de grafo ser visto em forma de grafo puro (diagrama de objeto)?

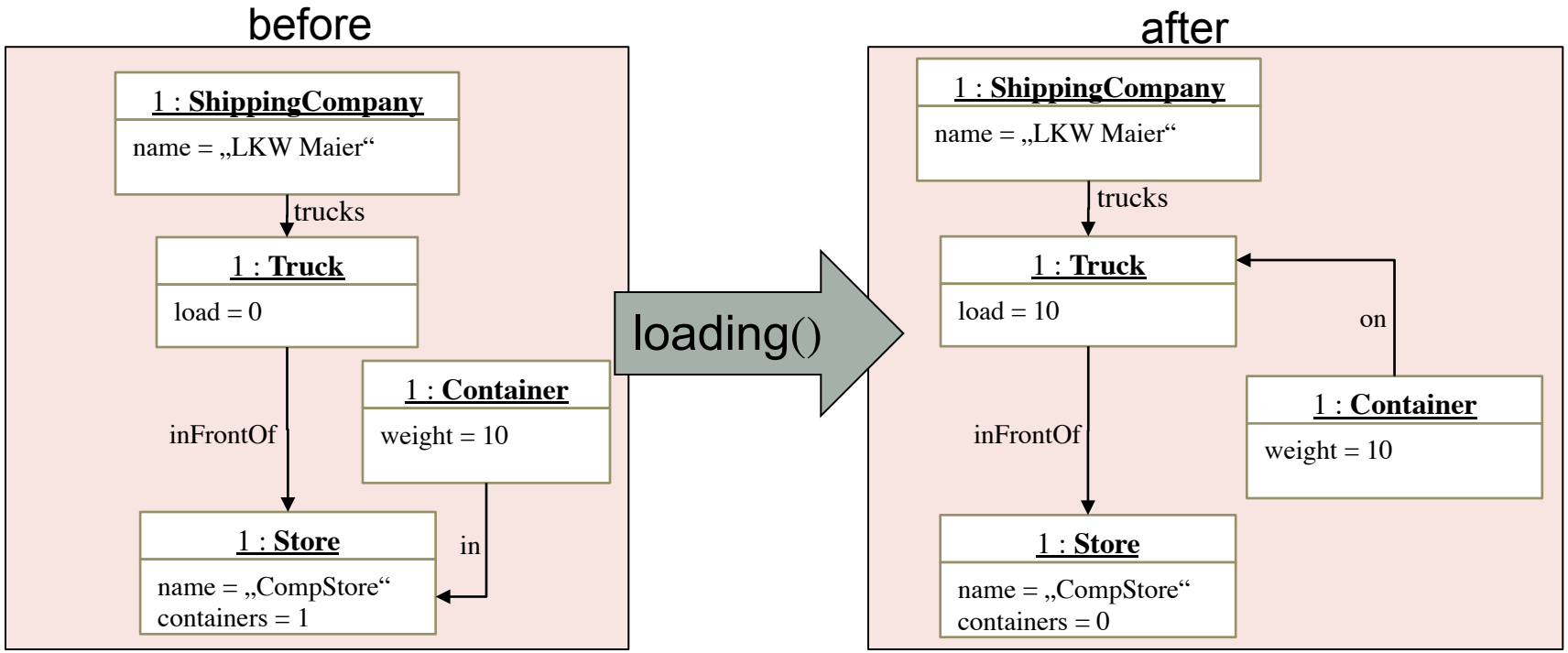


Até agora…

- …nós consideramos modelos como entidades estáticas
  - Um modelador cria um modelo usando um editor – Feito!
- *Mas oque está acerca das modificações dinâmicas de modelo?*
- Eles são **necessários** para
  - Simulação
  - Execução
  - Animação
  - Transformação
  - Extensão
  - Melhorias
  - …
- *Como os grafos podem ser modificados?*
  - Imperativo: Programa Java + Modelo API
  - Declarativo: Transformações gráficas por meio de regras de trasformação gráfica

Exemplo

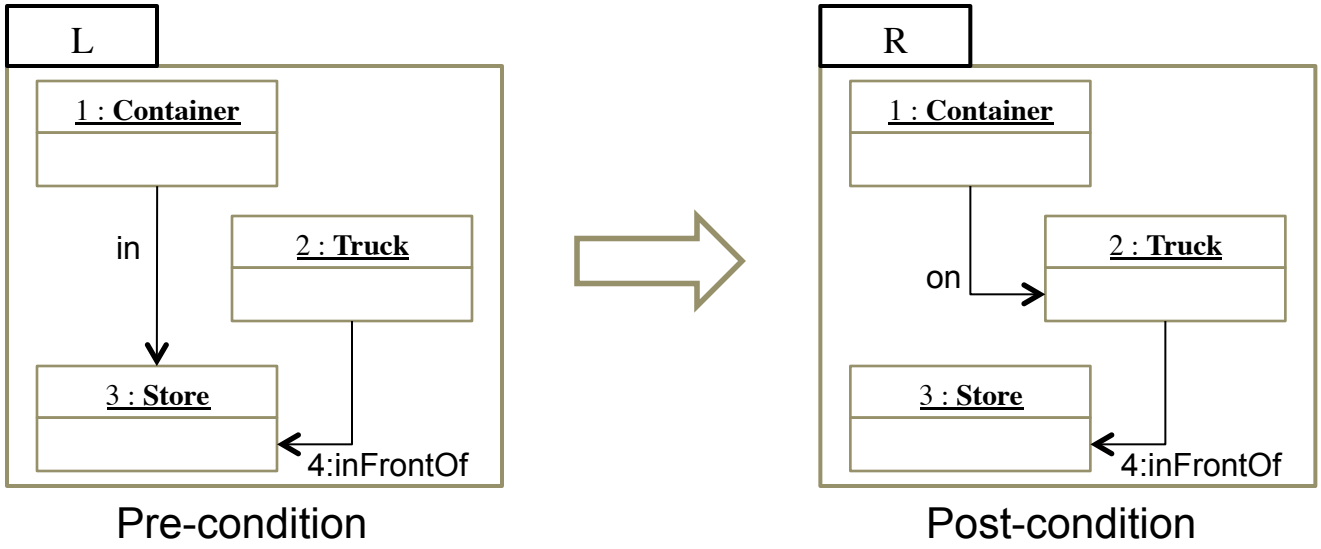
- Operação: Carregamento de um Container em um Caminhão



- Como este comportamento pode ser descrito de maneira genérica?

Regra de transformação de grafo

- Uma regra de transformação de grafo *p: L → R irá preservar a estrutura, o mapeamento parcial entre dois grafos*
  - L e R são dois grafos propriamente direcionados, tipados e atribuídos
  - Estruturas de preservação, porque vértices, arestas e valores podem ser preservados
  - Parcial, porque vértices e arestas podem ser adicionadas/deletadas
- Exemplo: Carregamento de um Container em um Caminhão



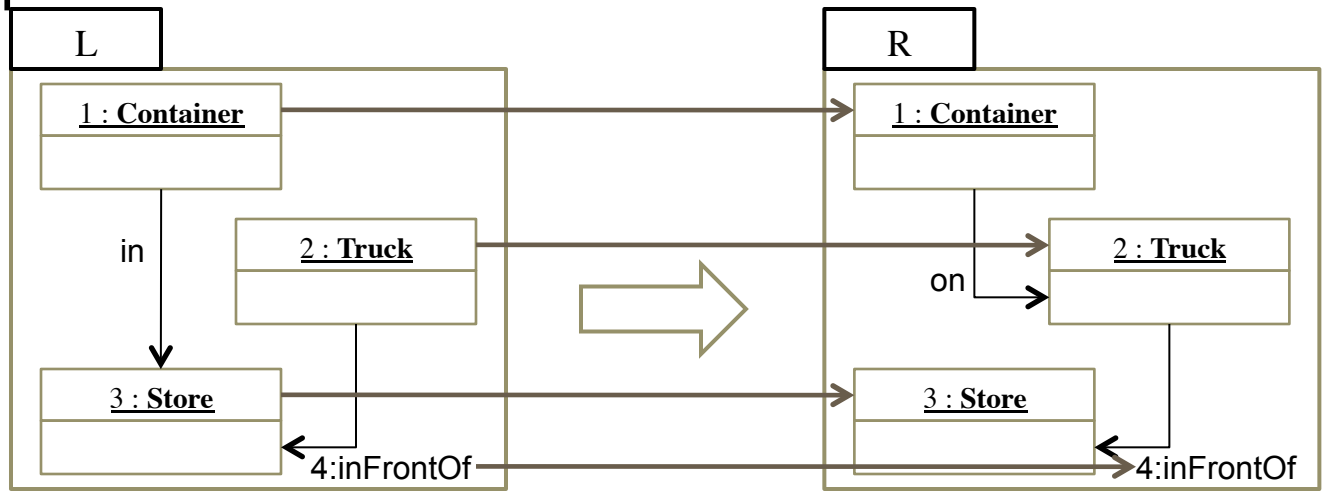
Pre-condition

Post-condition

Regra de transformação de grafo

- Preservação de estrutura
  - Todas as vértices e arestas que estão contidas no conjunto L ∩ R

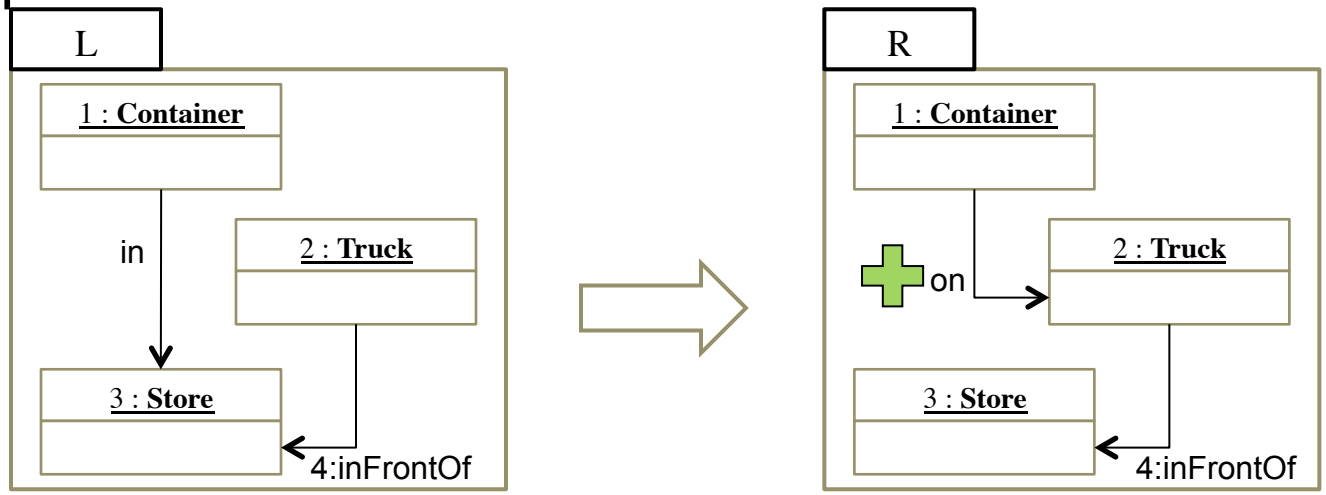
- Exemplo



Regra de transformação de grafo

- Adicionando
  - Todas as vértices e arestas que estão contidas no conjunto R \ L

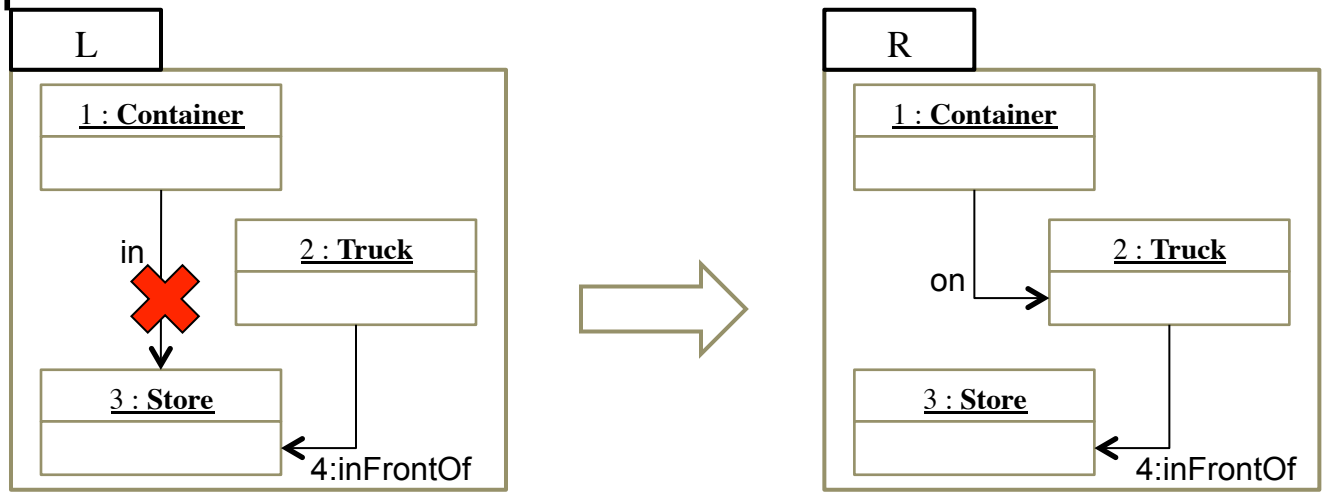
- Exemplo



Regra de transformação de grafo

- Deletando
  - Todas as vértices e arestas que estão contidas no conjunto L \ R

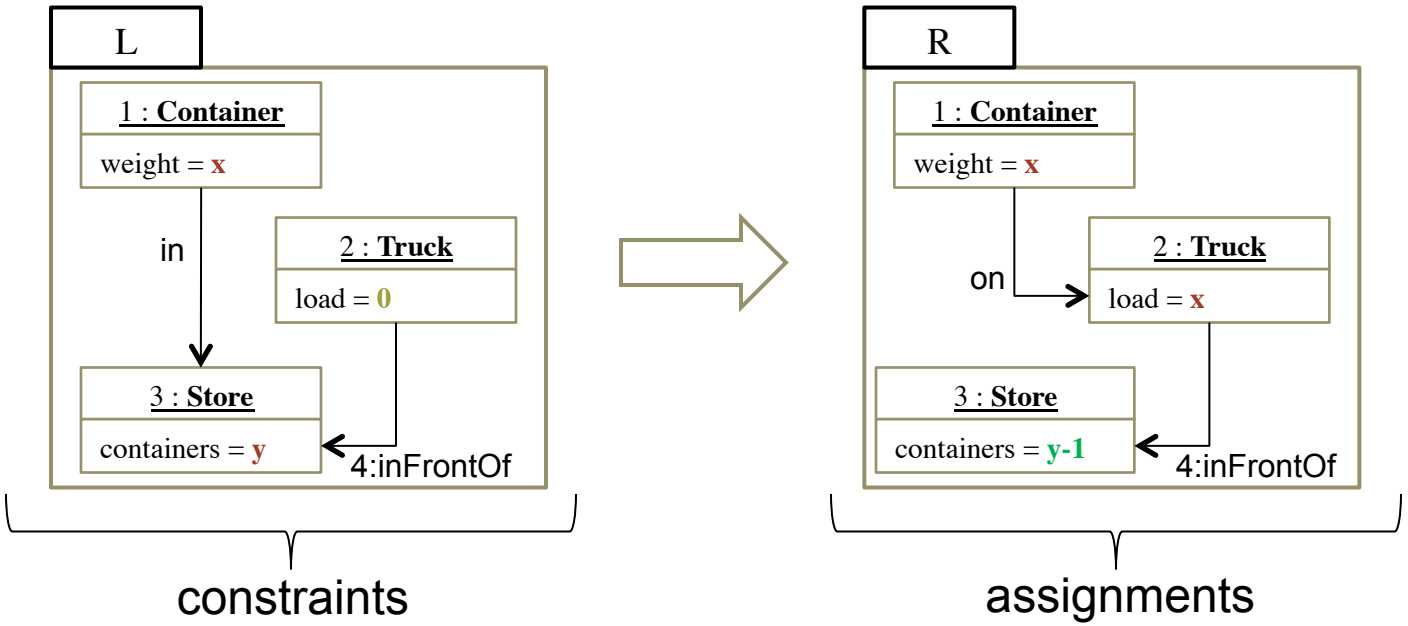
- Exemplo



Regra de transformação de grafo

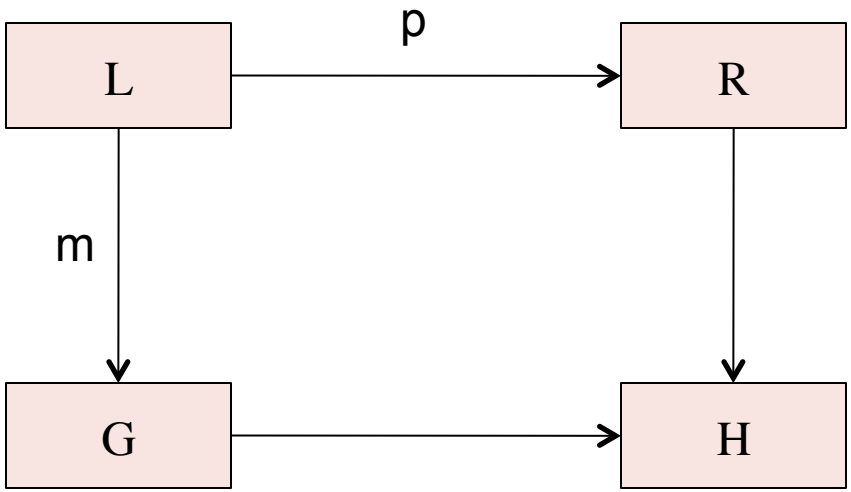
- Calculando – Fazendo uso de atributos
  - Constantes
  - Variáveis
  - Expressões (OCL & Co)

- Exemplo



Transformação de grafo

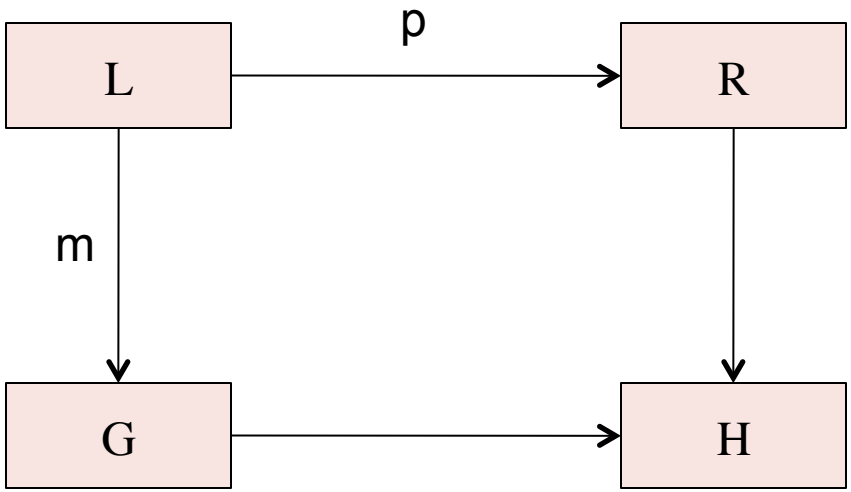
- Uma transformação de grafo *t: G → H* é o resultado da execução de uma regra de transformação de grafo *p: L → R* no contexto de *G*
  - t = (p,m) onde *m: L → G* é um morfismo grafo injetor (**match**)



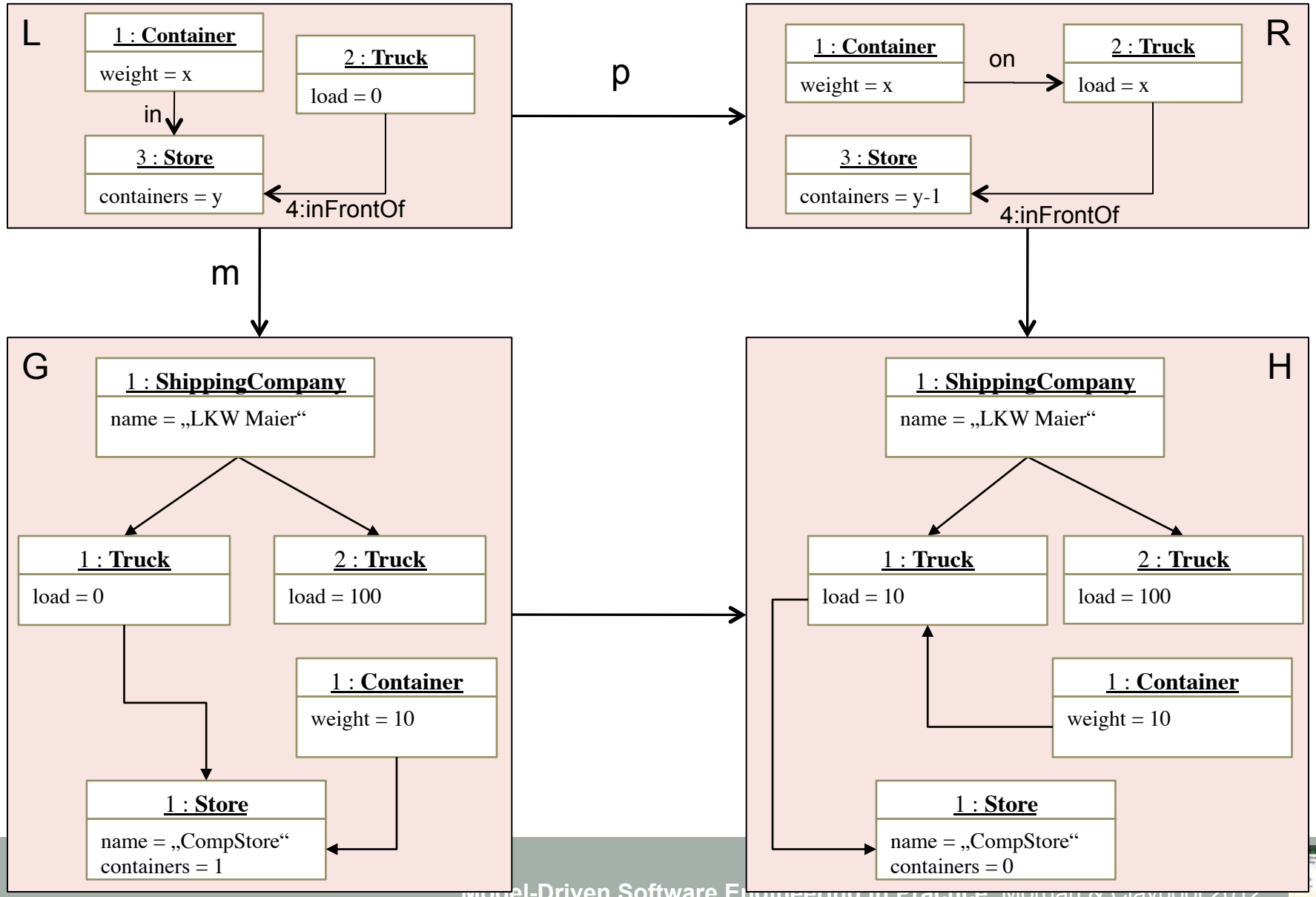
Transformação de grafo

Especificação operacional

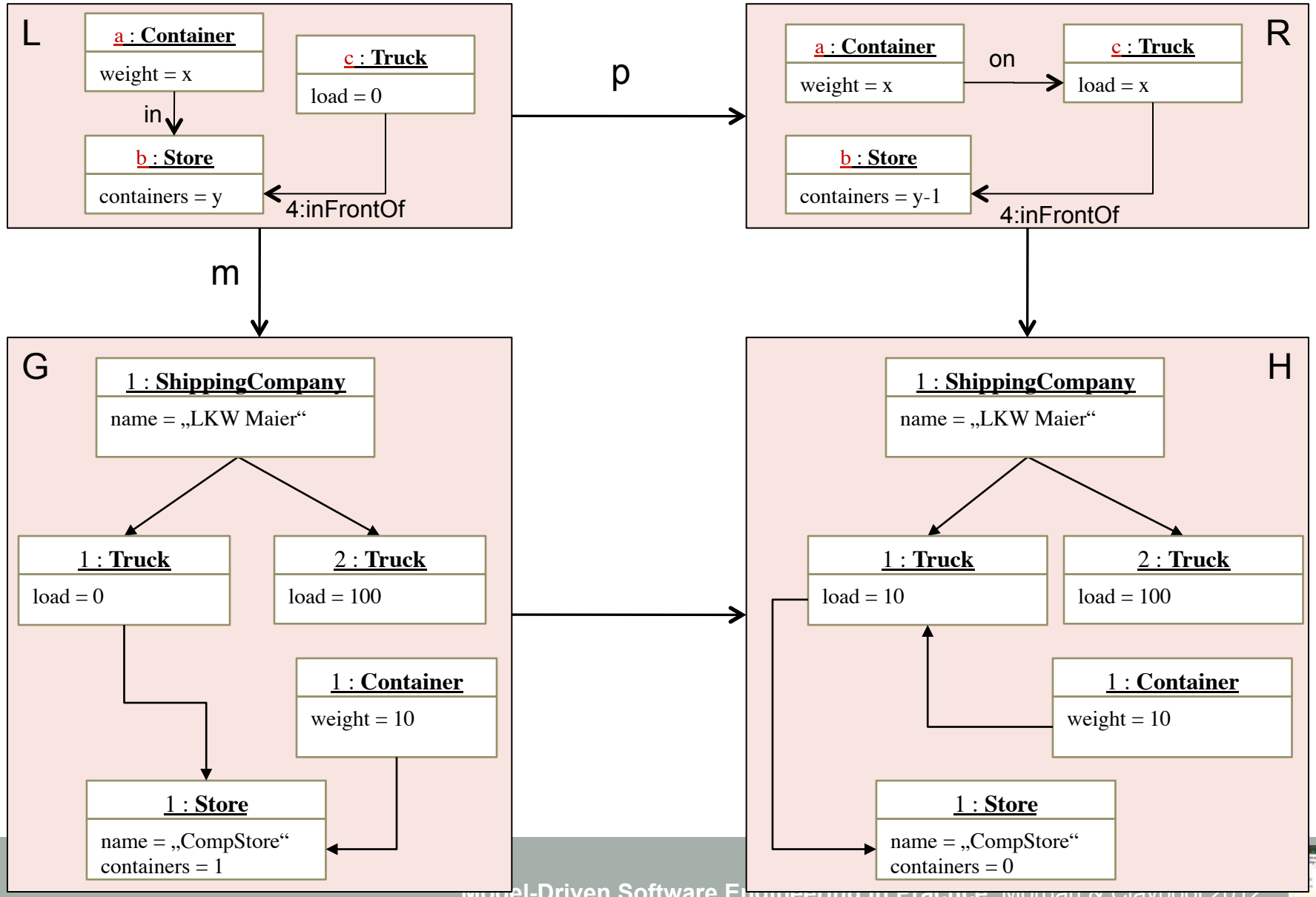
- Prepare a transformação
  - Selecione a regra p: L -> R
  - Selecione a combinação m: L -> G
- Gere um novo grafo H por
  - Deleção de L \ R
  - Adição de R \ L



Exemplo de transformação de grafo(1/2)

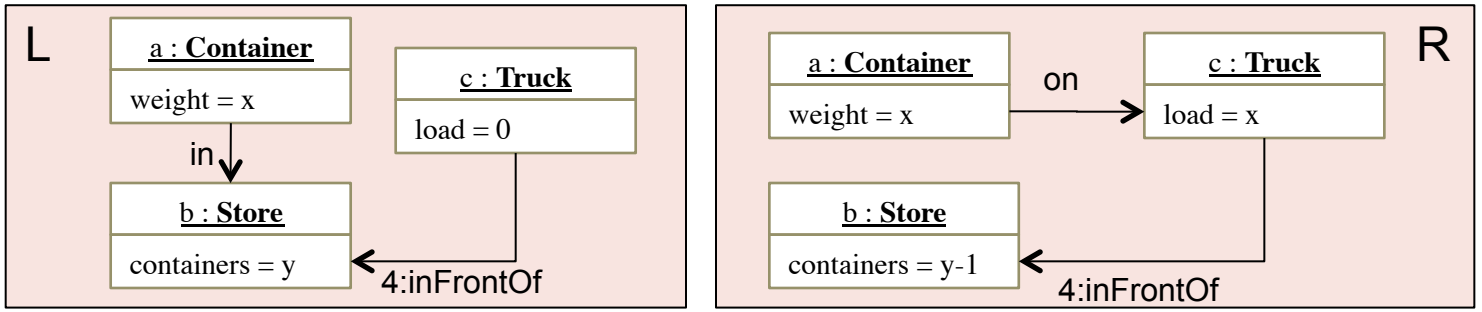


Exemplo de transformação de grafo(2/2)



Transformações de grafo em ATL

- **Transformação de grafo**



- **ATL (Modo de refinamento)**

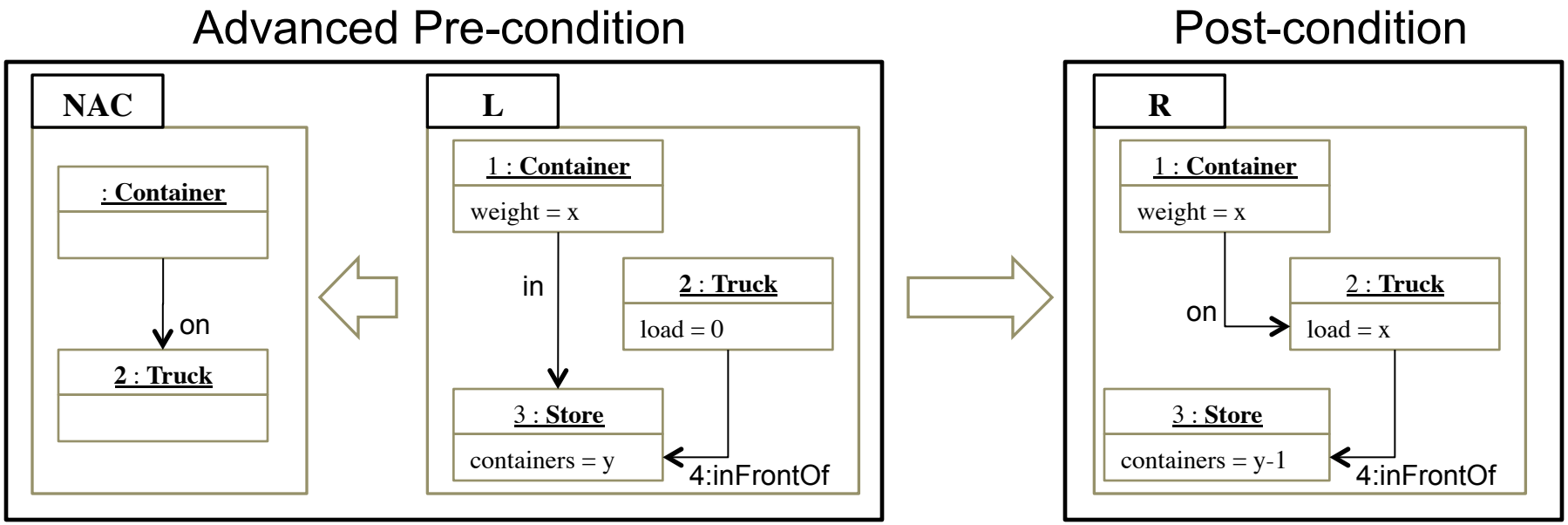
```
rule Loading {
  from
       a : Container (a.in = b)
       b : Store
       c : Truck (c.inFrontOf = b AND c.load = 0)
  to
       _a : Container(weight <- a.weight, on <- _c)
       _b : Store(containers <- b.containers - 1)
       _c : Truck(load <- a.weight, inFrontOf <- _b)
}
```


Condição de Aplicação Negativa (NAC)

- O lado esquerdo de uma regra especifica o que deve **existir** para executar a regra
  - *Condição de Aplicação*
- Muitas vezes é necessário para descrever o que **não deve existir**
  - Condição de Aplicação Negativa *(NAC)*
- NAC é um grafo que descreve uma estrutura **proibida** do sub grafo
  - Ausência de vértices e arestas específicas devem ser concedidas
- A regra de transformação de grafo é executada quando o NAC **não** é cumprido

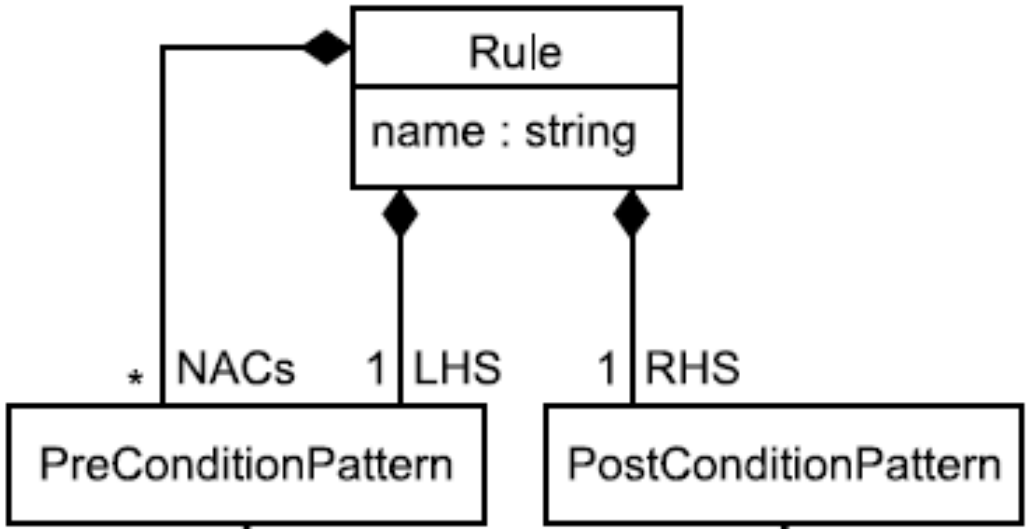
Condição de Aplicação Negativa (NAC)

- Exemplo: Um caminhão só deve ser carregado, se não houver um container no caminhão



Condição de Aplicação Negativa (NAC)

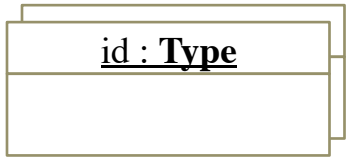
- **Multiplos** NACs para cada regra possível
- **Mas**: LHS e RHS só devem ser especificados uma vez



Multi-Object

- O **número** de objetos correspondentes nem sempre é conhecido **antecipadamente**

- O conjunto máximo de objetos que satisfazem uma determinada condição
  - Seleção de todos os objetos x de um conjunto y que satisfazem uma condição z
  - Em OCL: *y -> select(x | z(x))*
- **Notação: Multi-Object from UML 1.4**
  - Um símbolo do multi-object mapeia para uma coleção de instâncias em que cada instância está em conformidade com um determinado tipo e condições
  - Exemplo: *select(x | x.oclIsTypeOf(Type))*



Ordem de execução das regras de transformação de grafos

- Um sistema de transformação é consiste em um conjunto de regras de transformação grafo diferentes
- Em que **ordem** elas estão sendo executadas?

- Múltiplos procedimentos
  - não-deterministico
  - deterministico
    - Prioridades
    - Programas (ou “transformações gráficas programáveis”)

- Exemplo – Diagrams de atividade UML especializados
  - [falha]
  - [sucesso]

Ferramentas de transformação de grafo

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

Resumo

- Transformações de modelo são transformações de grafo
  - Type Graph = Metamodelo
  - Grafo = Modelo
- Transformações grafo programáveis permitem o desenvolvimento de cenários de evolução de grafos complexos
- Transformações out-place exógenas podem ser especificadas como transformações grafos
  - Entretanto, mais complexa com ATL
- Transformações grafo tornam-se mais relevantes na prática
  - Encontre o caminho delas para o Eclipse!

DOMINANDO TRANSFORMAÇÕES DE MODELO

HOTs

- **As transformações** podem ser consideradas como os **próprios** modelos
  - Elas são instâncias de uma **transformação de metamodelo!**
  - Esta **uniformidade** permite a reutilização de ferramentas e métodos definidos para os modelos

- Assim, um modelo de transformação pode ser criado pro ele mesmo ou manipulado por transformações, pelos chamados ***Higher Order Transformations*** (HOTs)
  - Transformações que tem como entrada uma transformação de modelo e/ou geração de transformação do modelo como saída

- **Exemplos**
  - Suporte a refatoração para transformações para melhorar a sua estrutura interna
  - Adicionando um aspecto de registo de transformações…

Transformações Bi-directionais

- Linguagens de transformação de modelo **Bi-directional**
  - **Não impõem** uma **direção** de transformação quando especifica uma transformação
  - **Permitem modos de execução diferentes** tais como transformação, integração e sincronização
- **Modo de transformação** é primeiramente dividido em transfomação **para frente** e **para trás** (origem -> destino -> origem)
- **Modo de integração** assume ter modelos de origem e destino dados e verifica se existem elementos esperados (que poderiam ser produzidos no modo de transformação) existente
- **Modo de sincronização** atualiza os modelos no caso do modo de integração tiver relatado correspondências insatisfeitos

- Linguagens que permitem transformações bi-directionais:
  - JTL, TGG, QVT Relational, …

Transformações Lazy & Incremental

- **Estratégia de execução padrão** para transformações out-place
  1) Ler o modelo de **entrada completo**
  2) Produzir o **modelo de saída a partir do zero**, aplicando **todas** as **regras** de transformação correspondentes
- Essas execuções são muitas vezes designadas transformações ***batch***

- Dois cenários podem se beneficiar de estratégias de execução alternativas
  1) Um **modelo de saída já existente** apartir de uma execução de transformação anterior para um dado modelo de entrada → transformações ***incrementais***
  2) Apenas uma **parte modelo de saída** é necessária para o consumidor → transformações ***lazy***

- Implementações experimentais para transformações lazy e incrementais estão disponíveis para ATL

Cadeias de Transformação

- **As transformações** podem ser um **processo complexo**
- **Dividir para Conquistar**
  - Use diferentes etapas de transformação para evitar uma transformação monolítica!
- **Cadeias de transformações** são técnicas de escolha para modelagem da **orquestração** de diferents transformações de modelo
- **Cadeias de transformações** são definidas com **linguagens de orquestração**
  - Forma simplista: etapas sequencias de execução das transformações
  - Formas mais complexas: ramos condicionais, laços, e outras estruturas de controle
  - Mesmo os HOTs podem produzir transformações dinamicamente utilizadas pela seqência
- **Transformações menores** focam em certos aspectos que permitem maior reusabilidade

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