APPROACHES TO MODEL TRANSFORMATION REUSE

Juan de Lara

joint work with
Esther Guerra and Jesús Sánchez Cuadrado

Modelling&Software Engineering Research Group
http://miso.es *@miso_uam*

UNIVERSIDAD AUTONOMA DE MADRID

ICMT'2016, Viena

FROM *CONCEPTS* TO A-POSTERIORI TYPING

Juan de Lara

joint work with
Esther Guerra and Jesús Sánchez Cuadrado

Modelling&Software Engineering Research Group
http://miso.es   @miso_uam

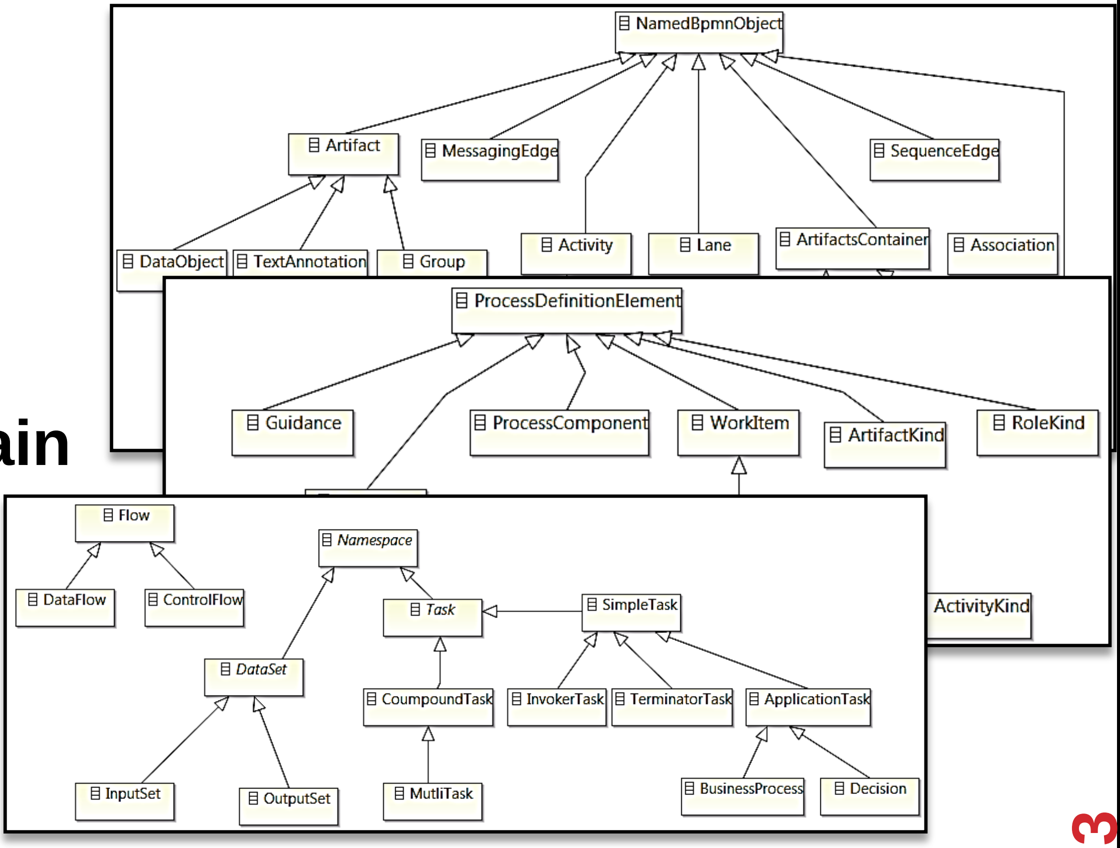
ICMT'2016, Viena

MOTIVATION

Model-transformations are the key elements for automation in model-based approaches

Domain-specific languages

Proliferation of domain meta-models

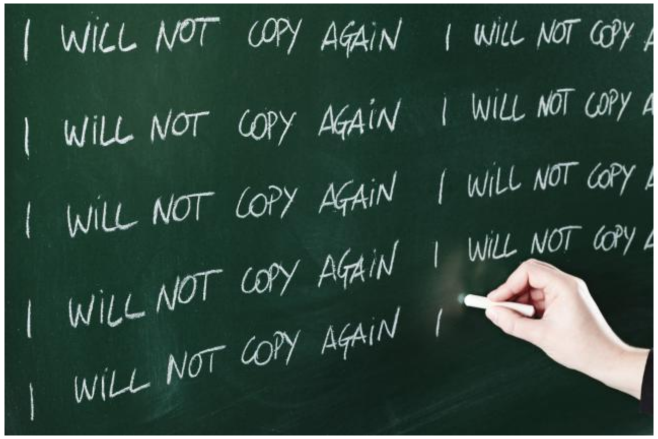


MOTIVATION

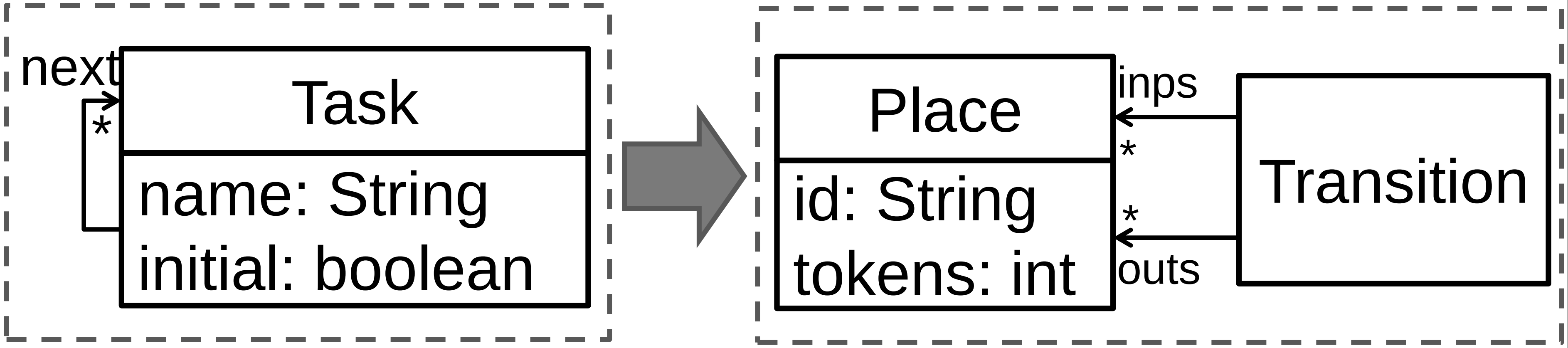
Transformations are defined over specific meta-models

Difficult to reuse for other related meta-models

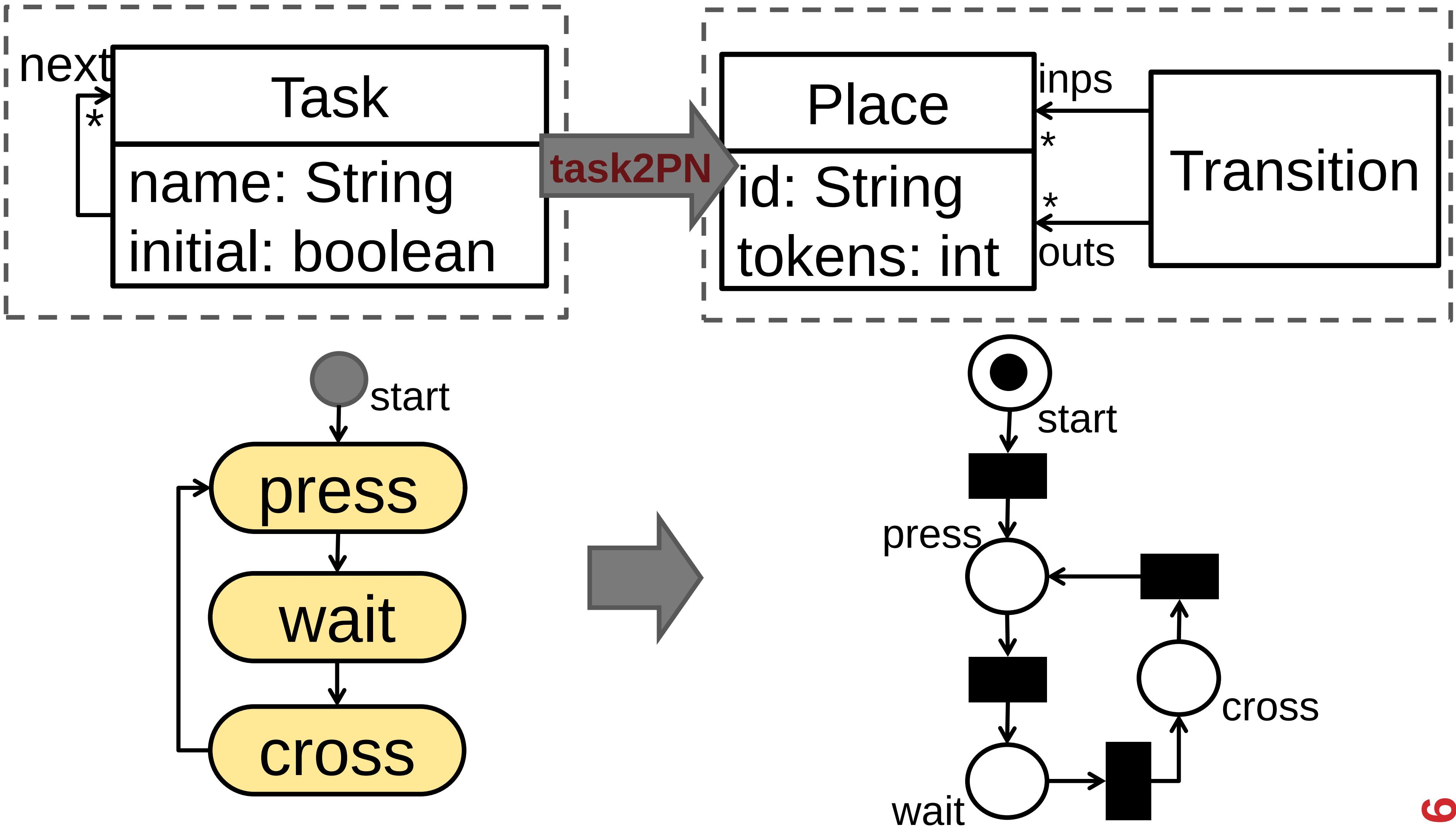
How to avoid creating (essentially) the same transformation for different meta-models?



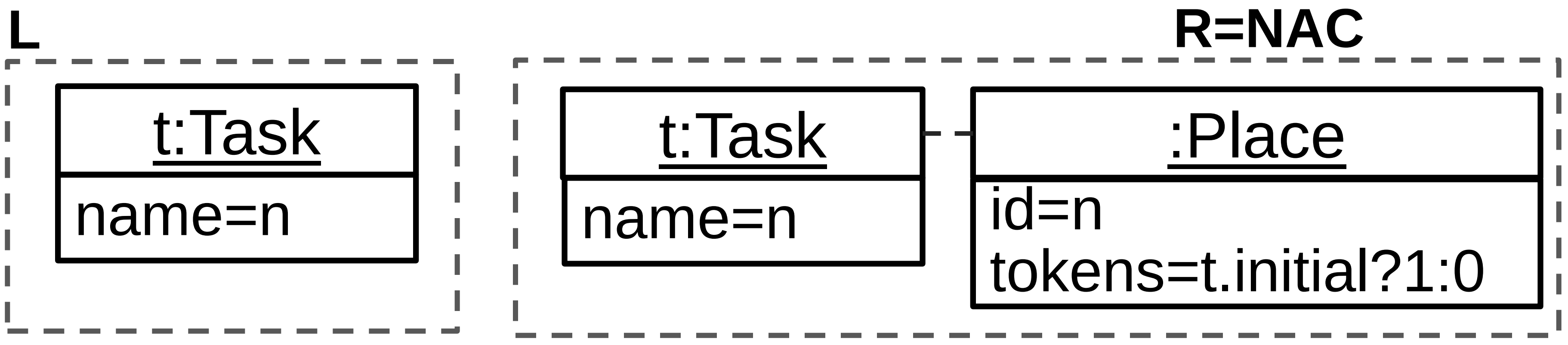
MOTIVATION:
PROCESS MODELS TO PETRI NETS

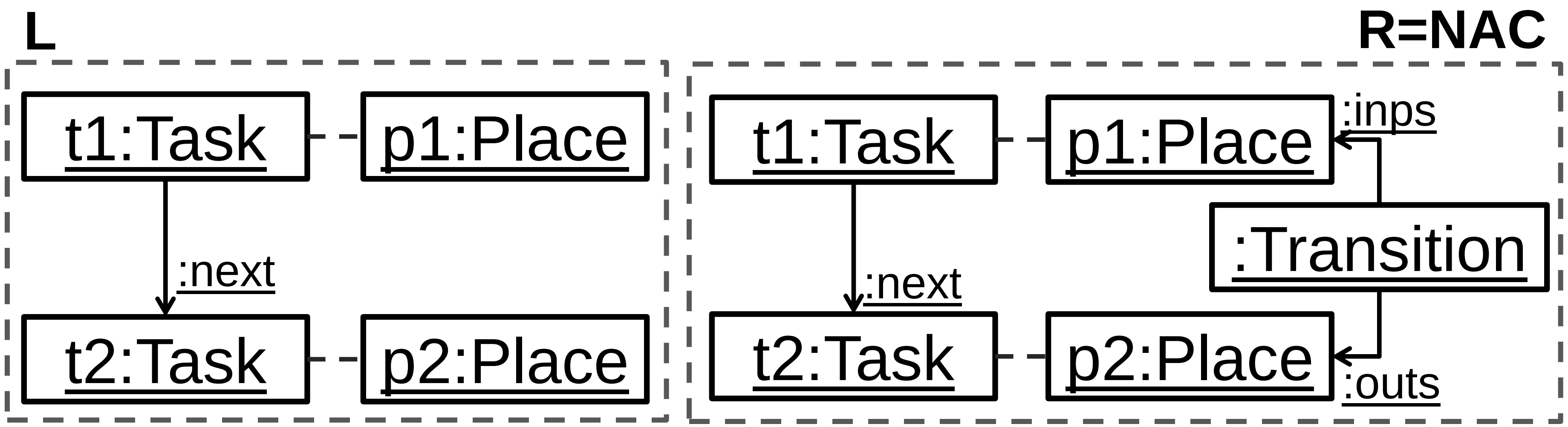


MOTIVATION: PROCESS MODELS TO PETRI NETS



TRANSFORMATION (GRAPH BASED)



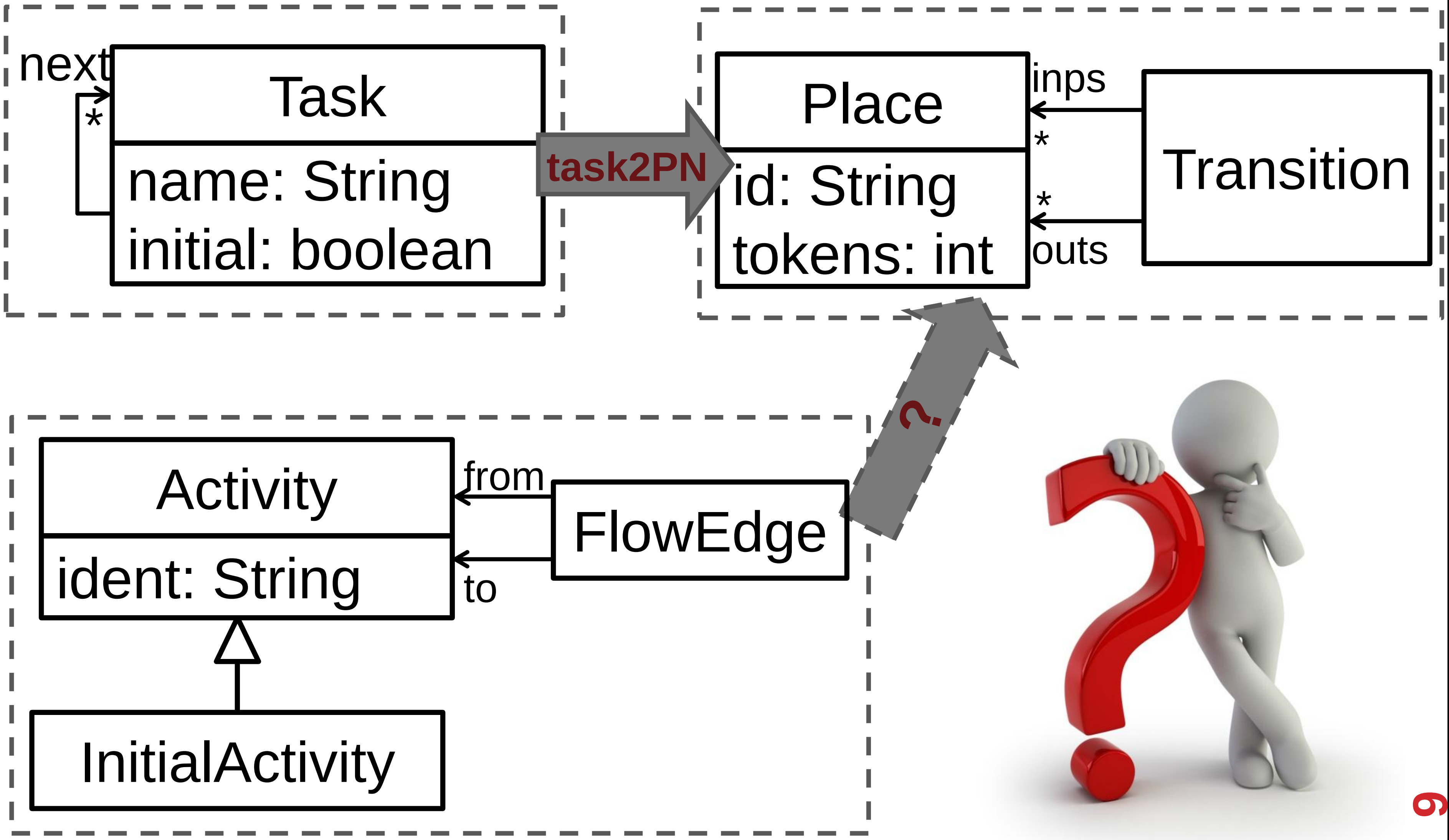


TRANSFORMATION (ATL)

```
rule Task2Place {
from t : Process!Task
to   p : PN!Place (
           id <- t.name,
           tokens <- if t.initial then 1 else 0 endif
     )
}

rule next2Transition {
from t1 : Process!Task,
     t2 : Process!Task (t1.next->includes(t2))
to   tr : PN!Transition (
            inps <- t1,
            outs <- t2
     )
}
…
```

REUSE FOR ANOTHER META-MODEL



AD-HOC REUSE (COPY+ADAPT)

```
rule Task2Place {
from t : Process!Task
to   p : PN!Place (
         id <- t.name,
         tokens <- if t.initial then 1 else 0 endif
     )
}

rule next2Transition {
from t1 : Process!Task,
     t2 : Process!Task (t1.next->includes(t2))
to   tr : PN!Transition (
            inps <- t1,
            outs <- t2
     )
}
…
```

AD-HOC REUSE (COPY+ADAPT)

```
rule Task2Place {
from t : Process!Activity
to   p : PN!Place (
     id <- t.ident,
     tokens <- if t.oclIsTypeOf(Process!InitialActivity) then
                 1 else 0 endif
     )
}
rule next2Transition {
from t1 : Process!Activity,
     t2 : Process!Activity
          (Process!FlowEdge.allInstances()->exist(e |
              e.from=t1 and e.to=t2))
to   tr : PN!Transition (
            inps <- t1,
             outs <- t2
     )
}
```

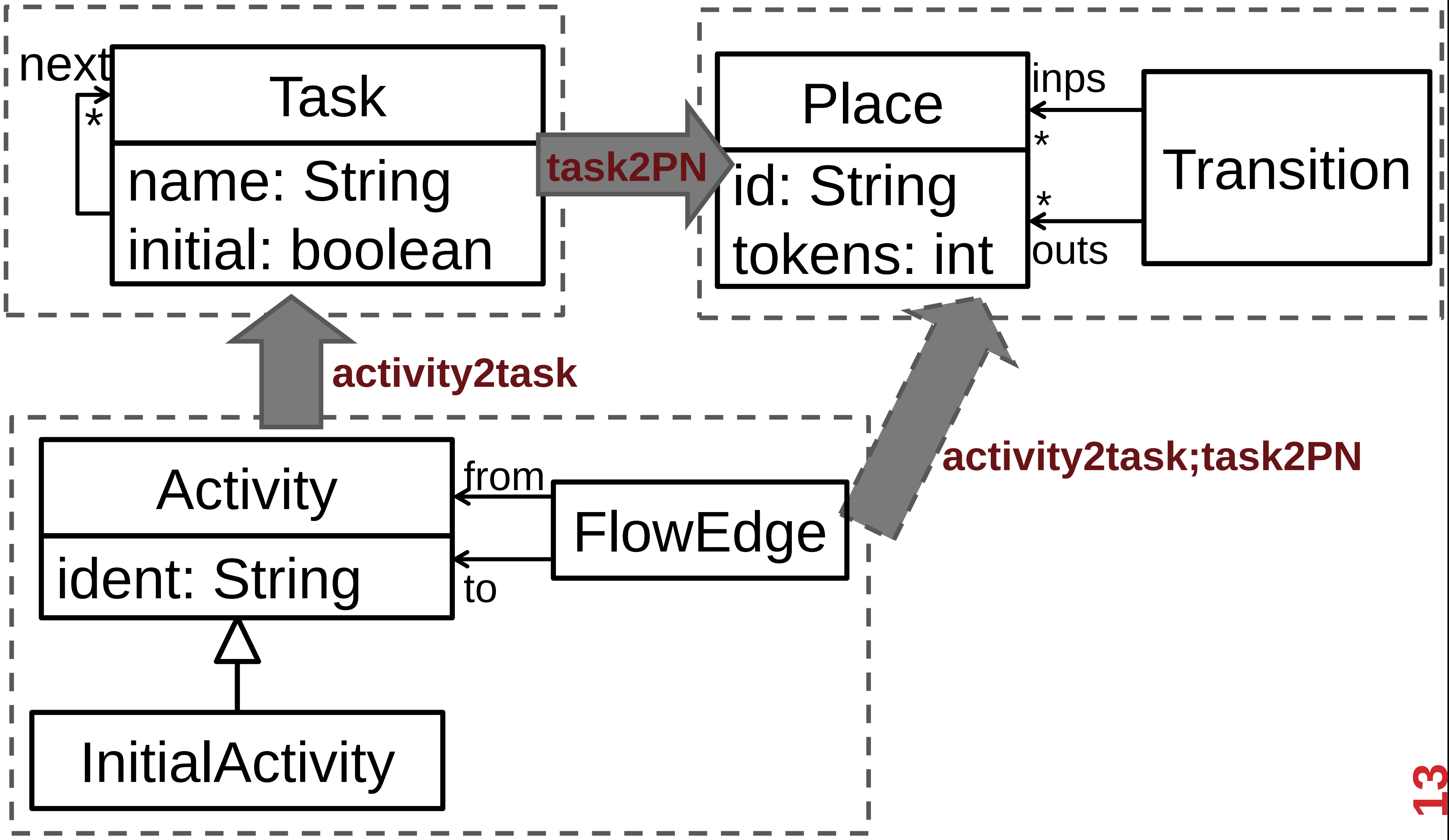
AD-HOC REUSE (COPY+ADAPT)

```
rule Task2Place {
```

- Complex and error prone
- Repetitive
- Does not scale to large transformations

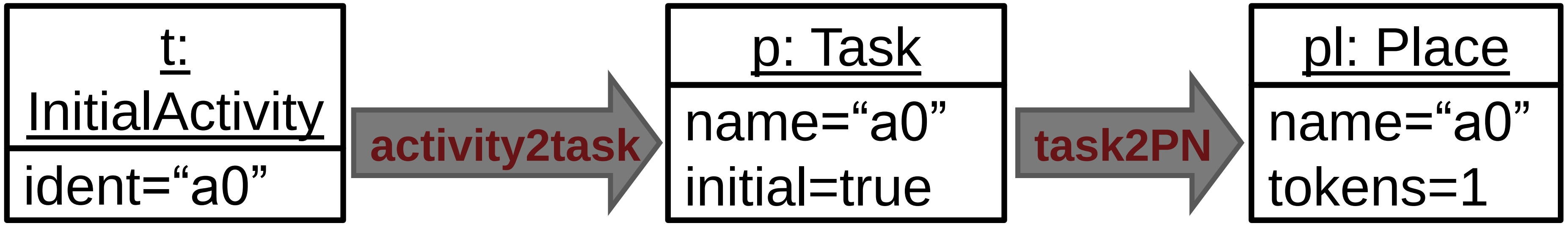
```
              outs <- t2
      )
}
```

EXPLICIT MODEL ADAPTATION



MOTIVATION

```
rule Activity2Task {
from t : Activities!Activity
to   p : Process!Task (
         id      <- t.ident,
         initial <- t.oclIsKindOf(Activities!InitialActivity),
         next    <- Activities!FlowEdge.allInstances()->
                       select (e | e.from = t)->
                       collect(e | e.to )
         )
}
```



MOTIVATION

- Three models involved, complicates traceability
- Two transformations: less efficient
- May require a transformation back to the initial model

SO WHAT?

Concept-based reuse

de Lara, Guerra. "*From types to type requirements: genericity for model-driven engineering*". SoSyM 2103.

Sánchez Cuadrado, Guerra, de Lara. "*A Component Model for Model Transformations*". IEEE Trans. Software Eng. 2014.

Multi-level based reuse

de Lara, Guerra, Sánchez Cuadrado. "*Model-driven engineering with domain specific meta-modelling languages"*. SoSyM 2015.

A-posteriori typing

de Lara, Guerra, Sánchez Cuadrado. "*A-posteriori typing for Model-Driven Engineering*". MoDELS 2015: 156-165

Transformation co-evolution

Di Ruscio, Iovino, Pierantonio. "*A Methodological Approach for the Coupled Evolution of Metamodels and ATL Transformations*". ICMT'13

Model (sub-)typing

Steel, Jezequel. "*On Model Typing*". SoSyM 2007.

Guy, et al. "*On Model Subtyping*". ECMFA'12.

SO WHAT?

Concept-based reuse

de Lara, Guerra. "*From types to type requirements: genericity for model-driven engineering*". SoSyM 2103.

Sánchez Cuadrado, Guerra, de Lara. "*A Component Model for Model Transformations*". IEEE Trans. Software Eng. 2014.

Multi-level based reuse

de Lara, Guerra, Sánchez Cuadrado. "*Model-driven engineering with domain specific meta-modelling languages"*. SoSyM 2015.

A-posteriori typing

de Lara, Guerra, Sánchez Cuadrado. "*A-posteriori typing for Model-Driven Engineering*". MoDELS 2015: 156-165

Transformation co-evolution

Di Ruscio, Iovino, Pierantonio. "*A Methodological Approach for the Coupled Evolution of Metamodels and ATL Transformations*". ICMT'13

Model (sub-)typing

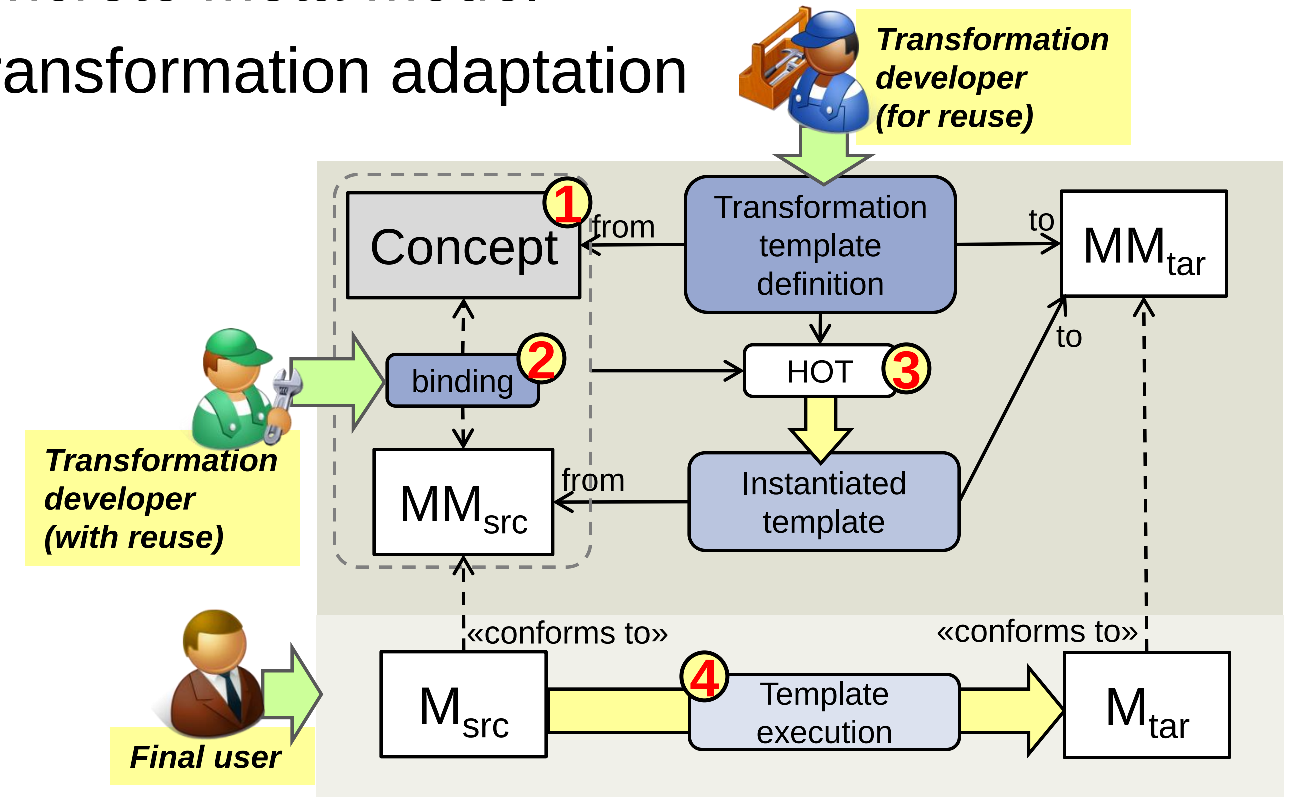
Steel, Jezequel. "*On Model Typing*". SoSyM 2007.

Guy, et al. "*On Model Subtyping*". ECMFA'12.

CONCEPT-BASED REUSE

Automate ad-hoc copy-adaptation:

- Transformations defined over "concepts"
- Binding to concrete meta-model
- Automated transformation adaptation

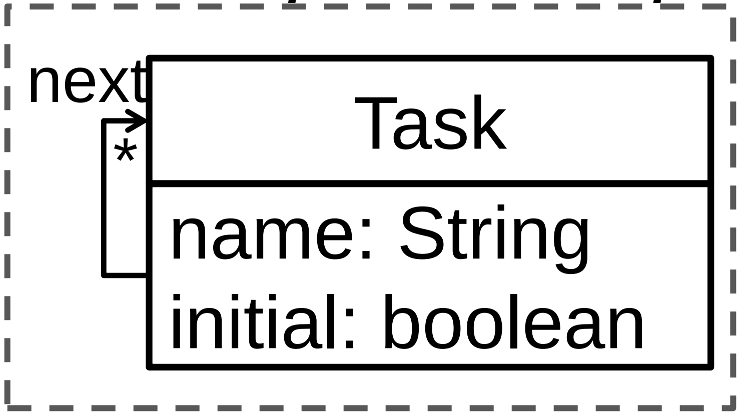


STRUCTURAL CONCEPTS

Structural Concepts.

- They gather the structure needed from meta-models for the transformation to be applicable.
- They have the form of a meta-model as well.

SimpleTask concept

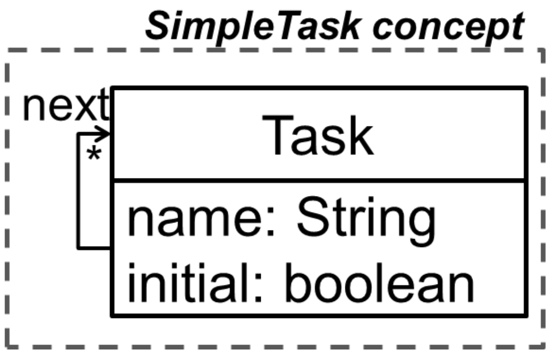


Their elements are treated as variables, which need to be bound to elements in specific meta-models.

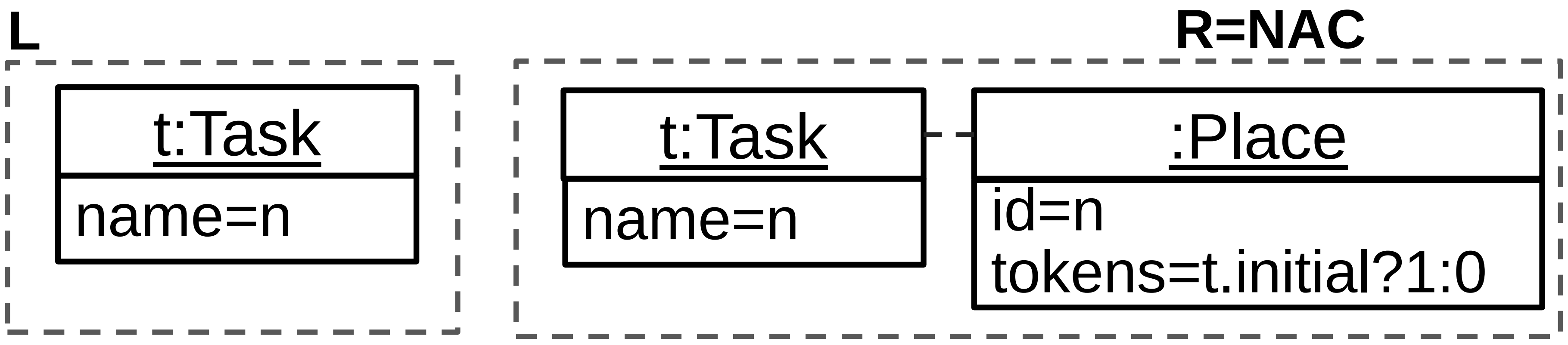
Interface of the transformation

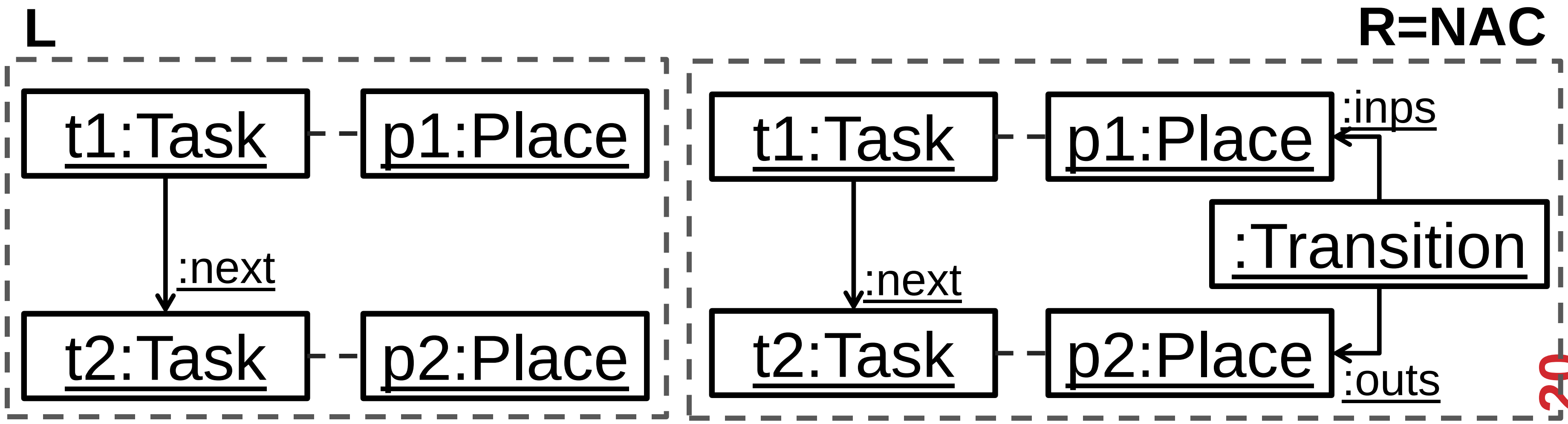
- No superflous elements
- No extra complexity
- Transformations become as simple as possible

REUSABLE TRANSFORMATIONS



Transformations defined using the types of the concept.





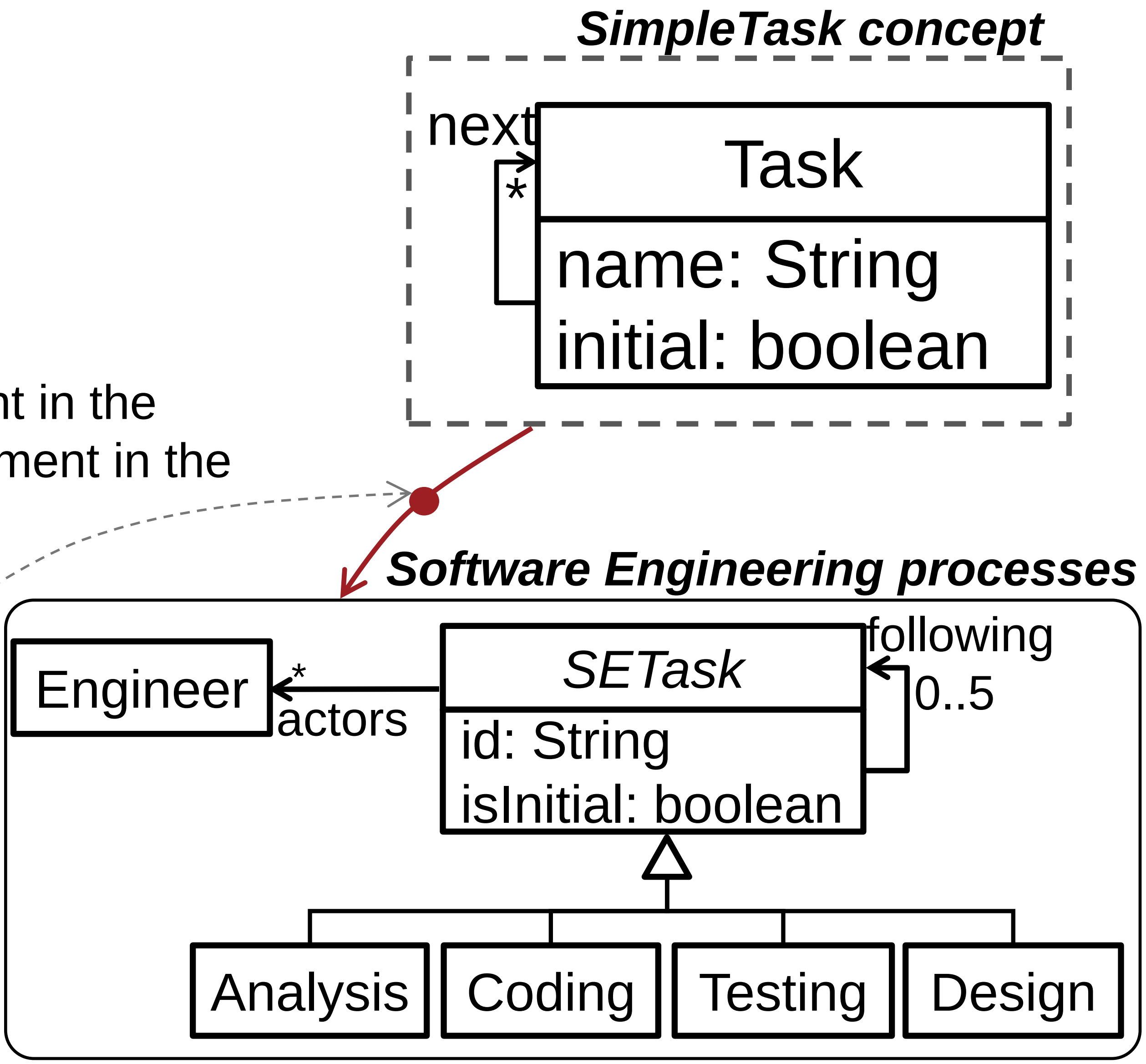
BINDING

Concept binding:

- Bind each element in the concept to an element in the meta-model.

Binding

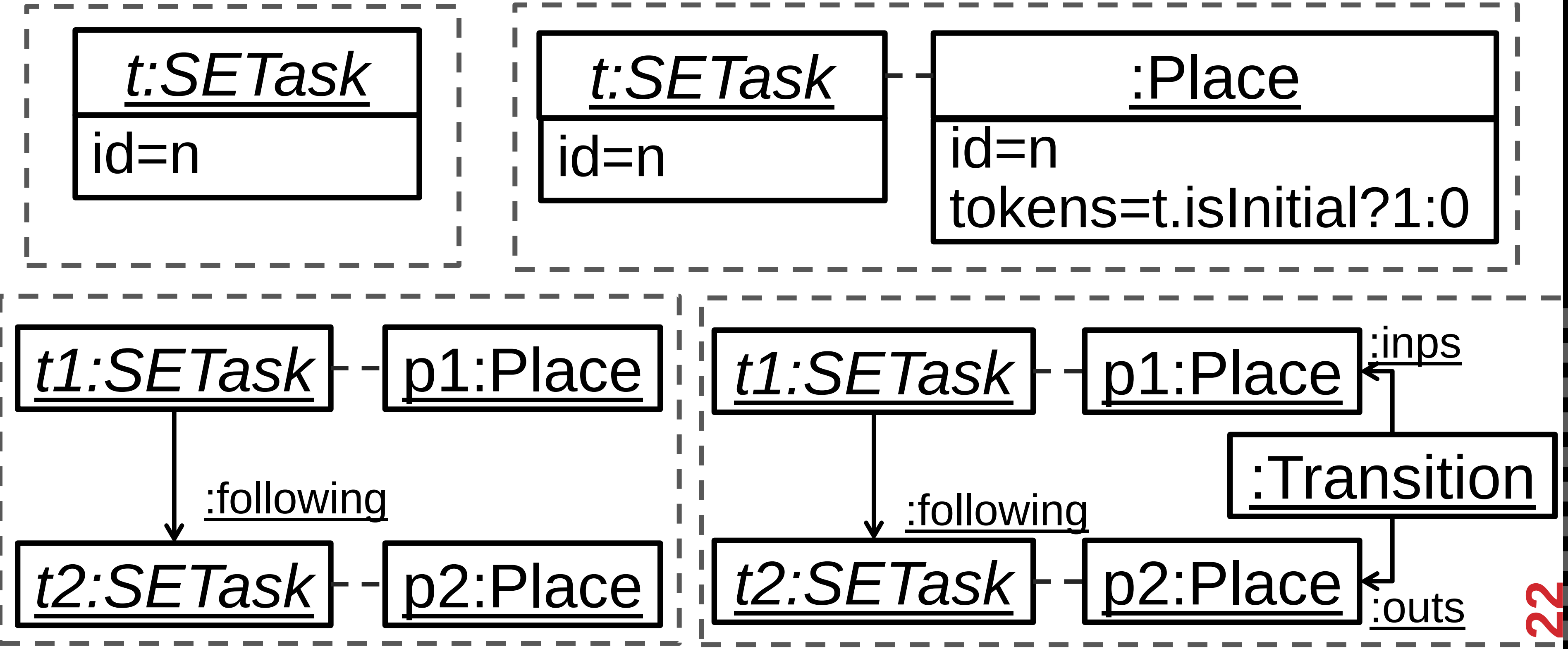
Task→ SETask
name → id
initial → isInitial
next → following



ADAPTATION

The transformation gets automatically adapted

Similar to template instantiation in generic programming



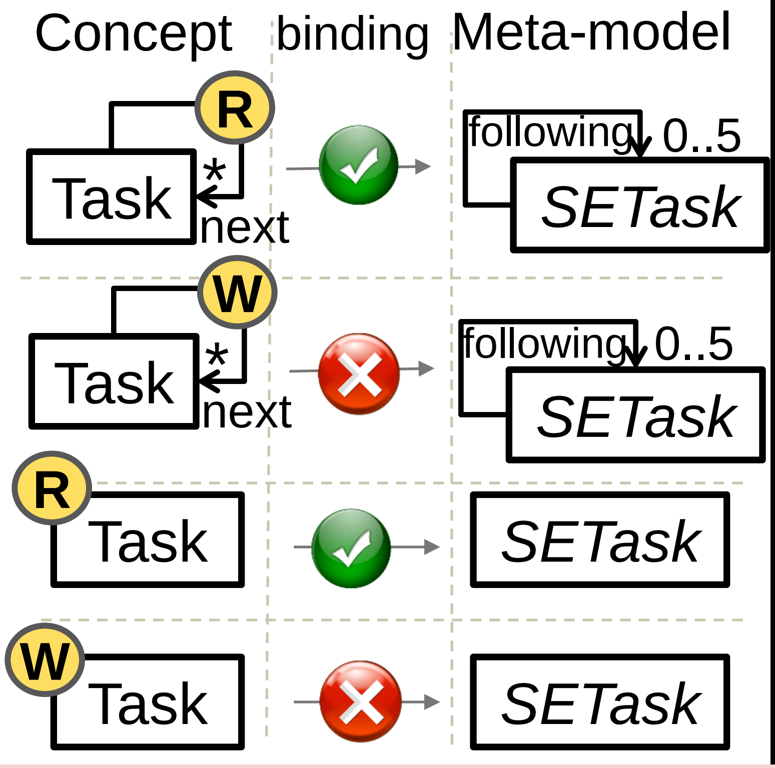
WAIT A MOMENT... IS THE RESULTING TRANSFORMATION CORRECT?

Rules for the binding

- Cardinalities
- Composition, features, subtyping, etc

Depends on what you do with the concept

- Read only (e.g., for M2M transformation)
- Write (e.g., for in-place transformation)



Sánchez Cuadrado, Guerra, de Lara. “*Flexible Model-to-Model Transformation Templates: An Application to ATL*”. JOT 2012.

WAIT A MOMENT… ISN'T THIS JUST A RENAMING? HOW FLEXIBLE IS THE BINDING?

Basic binding

- Renaming of types, attributes
- N-to-1 mappings

More flexibility is needed in practice

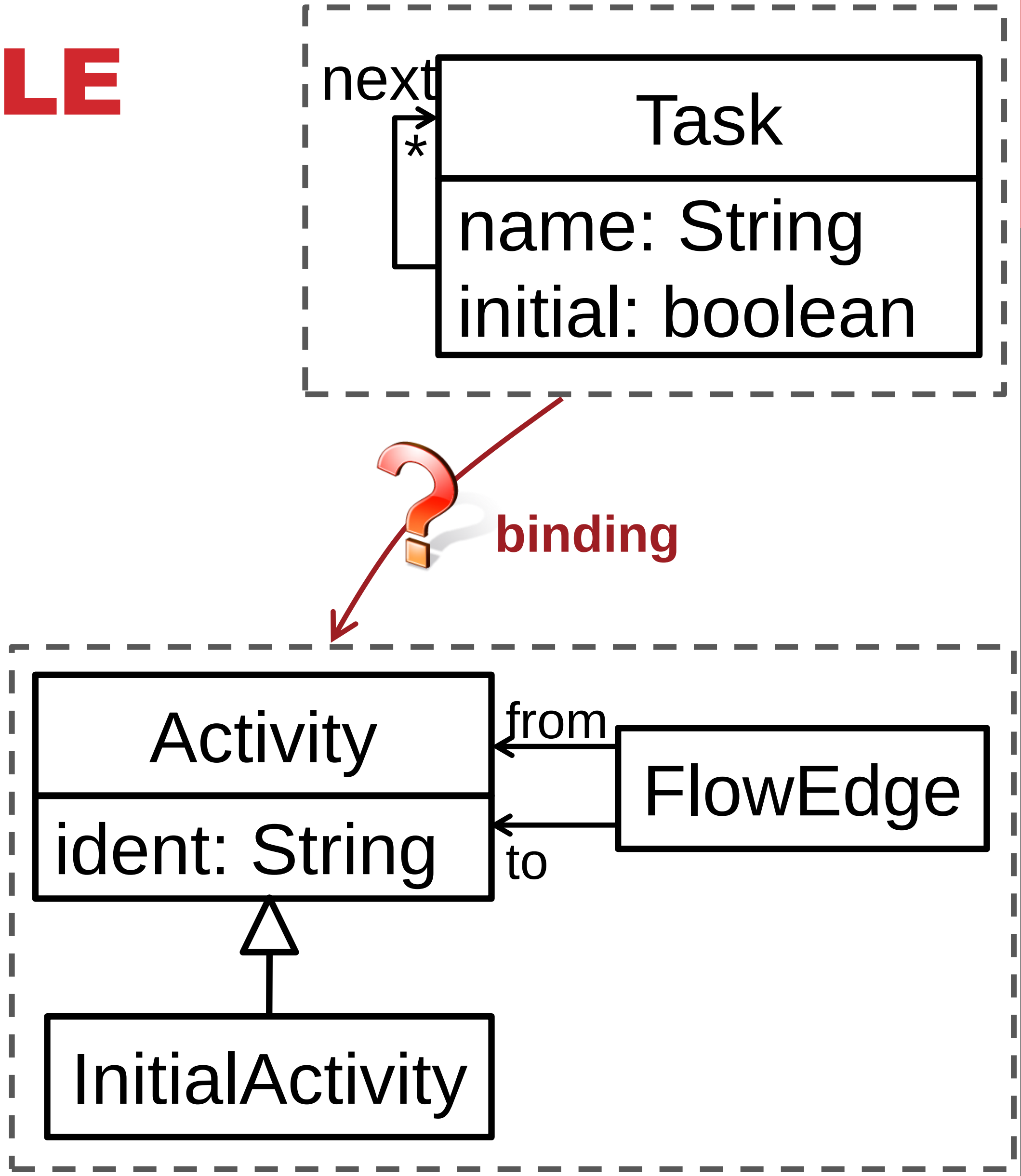
- Hybrid concepts
- Cardinalities in concept elements
- Adapters

MORE FLEXIBLE BINDINGS

Concepts express a particular design choice on how structure is organized

Different meta-models may implement the same structure differently

- Association as intermediate class
- Enumerate as subclases
- etc.



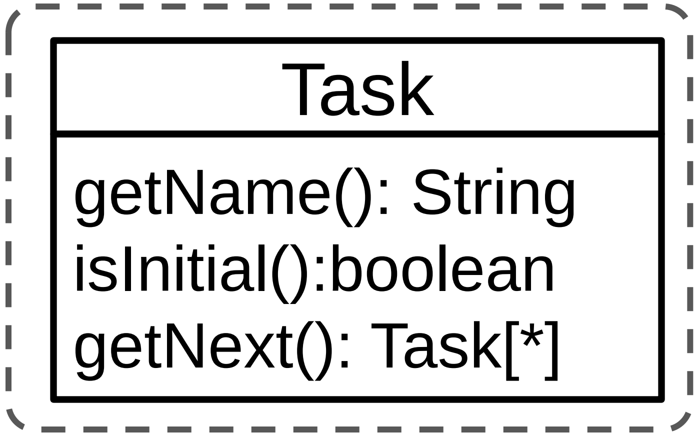
HYBRID CONCEPTS

Hide "unessential" structure behind required operations

Operations need to be implemented together with the binding

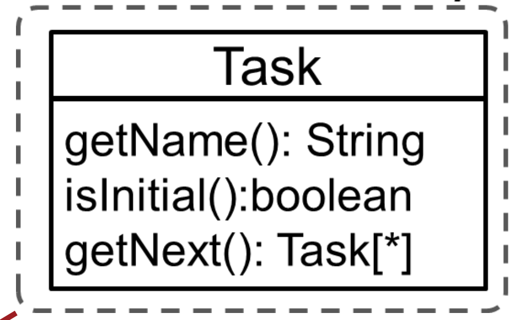
The generic transformation may use these required operations

SimpleTask hybrid concept



HYBRID CONCEPTS

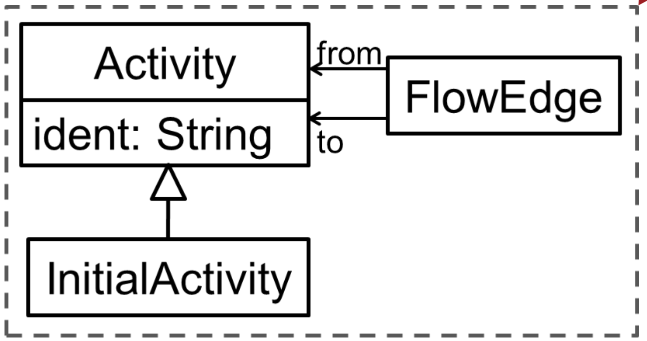
SimpleTask hybrid concept



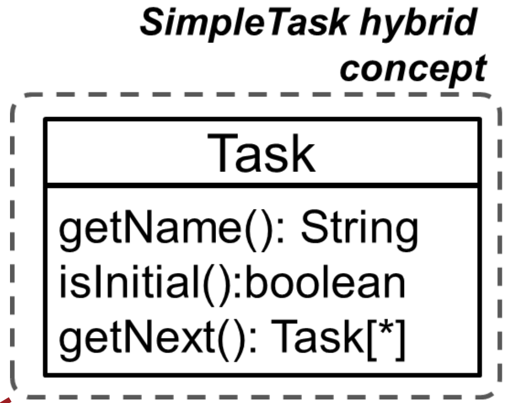
binding

Task → Activity

```
operation Activity getName() : String { return self.ident }
operation Activity isInitial() : boolean { return false; }
operation Activity getNext() : Set(Activity) {
   return FlowEdge.all.select(f|f.from=self).collect(f|f.to);
}
operation InitialActivity isInitial() : boolean { return true;}
```



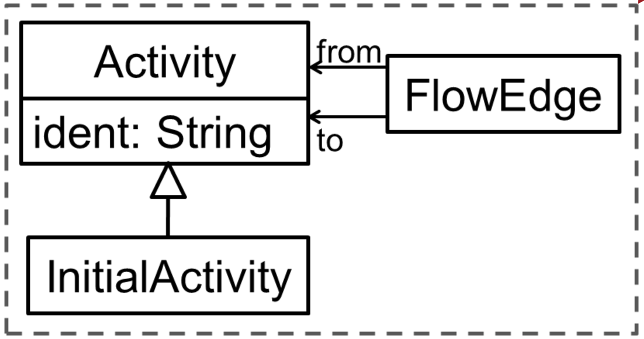
HYBRID CONCEPTS



binding

Task → Activity

```
operation Activity getName() : String { return self.ident }
operation Activity isInitial() : boolean { return false; }
operation Activity getNext() : Set(Activity) {
  return FlowEdge.all.select(f|f.from=self).collect(f|f.to);
}
operation
```



Allows bridging a large number of heterogeneities

The generic transformation needs to be written using these operations. Difficult for e.g. graph transformation approaches

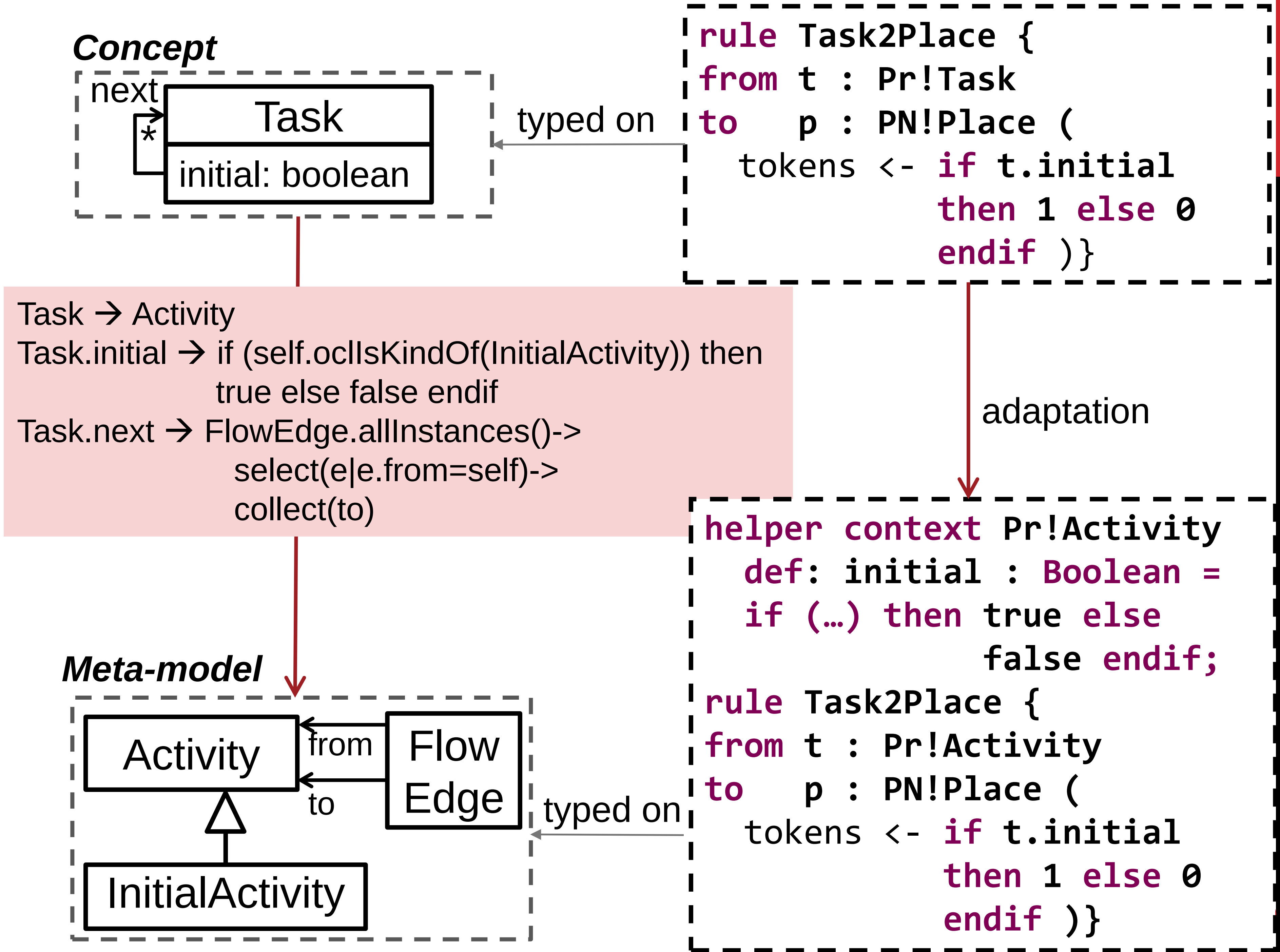
BINDING ADAPTERS

Elements in the concept are variables

- Allow binding them to expressions: adapters

Adapters resolve the heterogeneities between the concept and the meta-model

They induce an adaptation of the transformation



```
rule Task2Place {
from t : Pr!Task
to   p : PN!Place (
  tokens <- if t.initial
            then 1 else 0
            endif )}
```

Task → Activity
Task.initial → if (self.oclIsKindOf(InitialActivity)) then
 true else false endif
Task.next → FlowEdge.allInstances()->
 select(e|e.from=self)->
 collect(to)

```
helper context Pr!Activity
  def: initial : Boolean =
  if (…) then true else
              false endif;
rule Task2Place {
from t : Pr!Activity
to   p : PN!Place (
  tokens <- if t.initial
            then 1 else 0
            endif )}
```

ADAPTERS VS HYBRID CONCEPTS

Hybrid concepts

👍 Bridge a large number of heterogeneities

👎 The generic transformation needs to be written using operations

👎 Adaptation mechanism dependent on the particular transformation language (e.g., Epsilon)

Adapters

👍 Bridge a large number of heterogeneities

👍 The generic transformation remains agnostic w.r.t. the binding

👎 Adaptation mechanism dependent on the particular transformation language (e.g., ATL)

WAIT A MOMENT…

Aren’t concepts characterizing a set of “suitable” meta-models for the transformation?

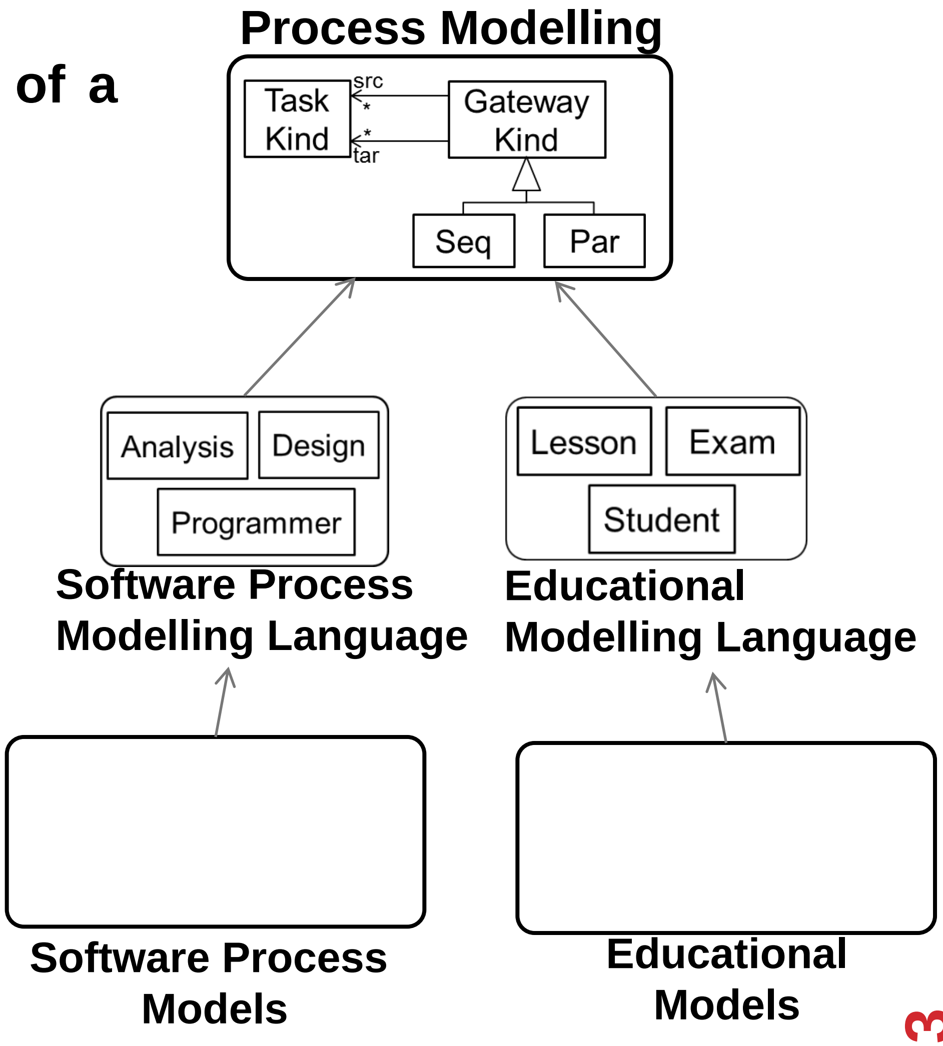
Isn’t that exactly what meta-models do (for models)?

Concepts as **meta**-meta-models

Rely on typing: no need to adapt the transformation

MULTI-LEVEL BASED REUSE

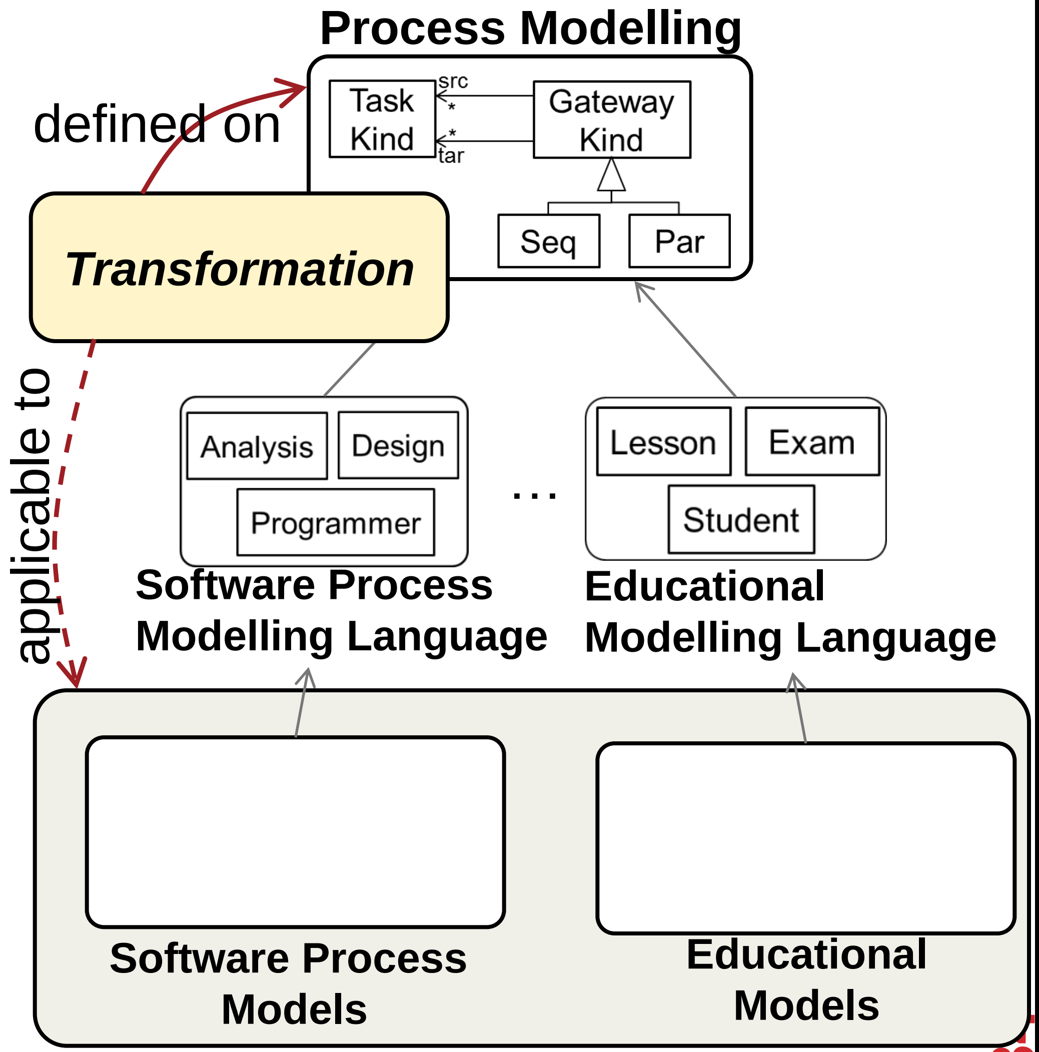
Language families as instances of a common meta-model



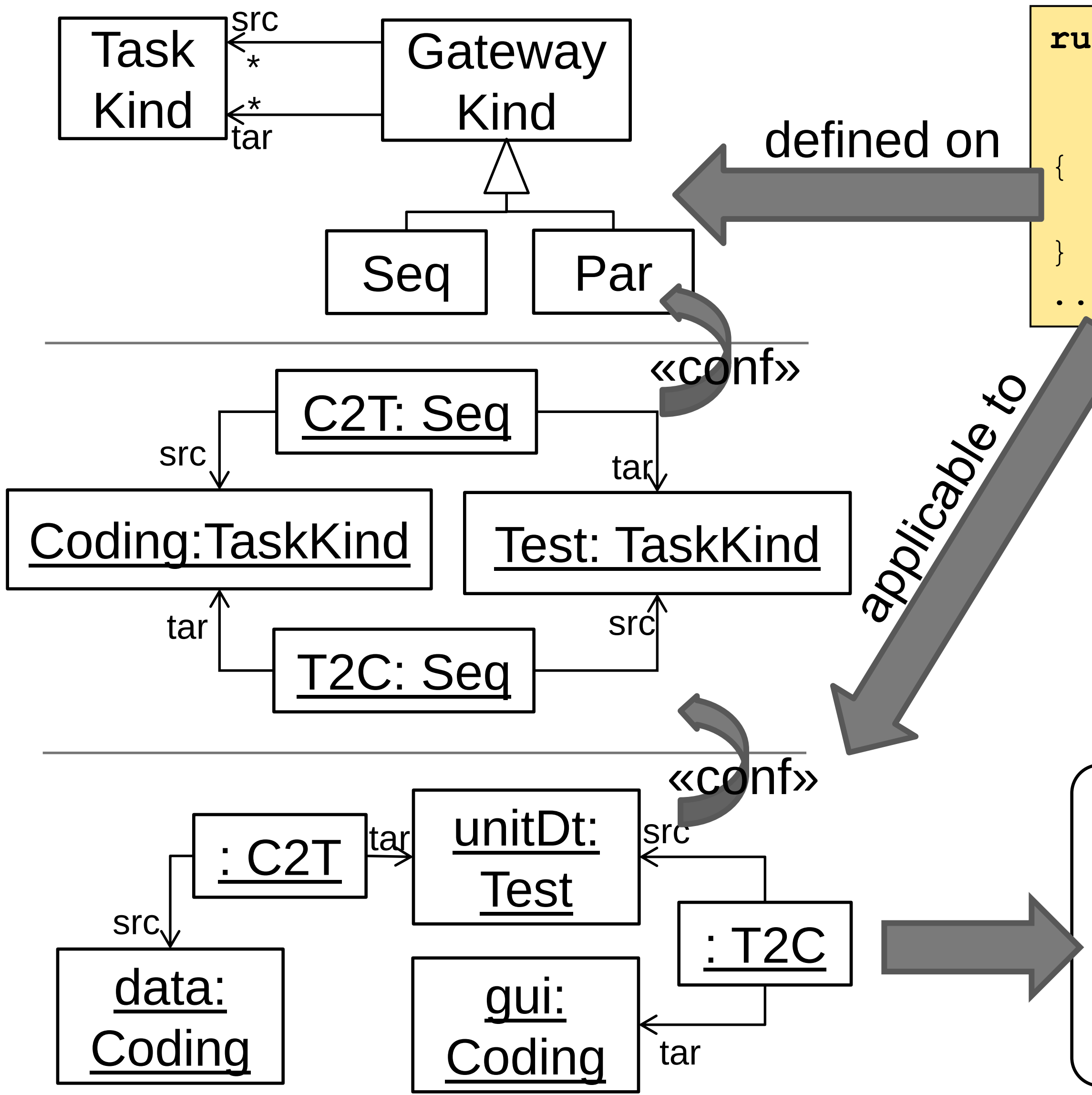
MULTI-LEVEL BASED GENERICITY

Transformations can be defined at the top meta-level

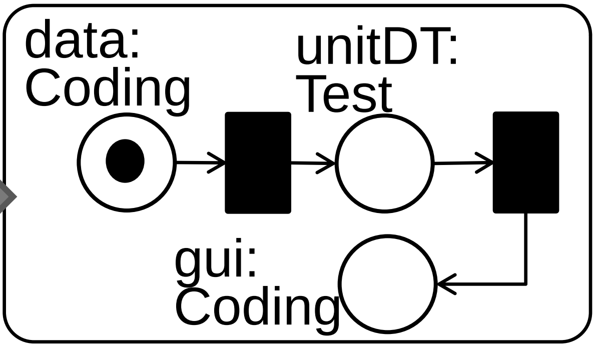
Applicable to instances of all languages in the family



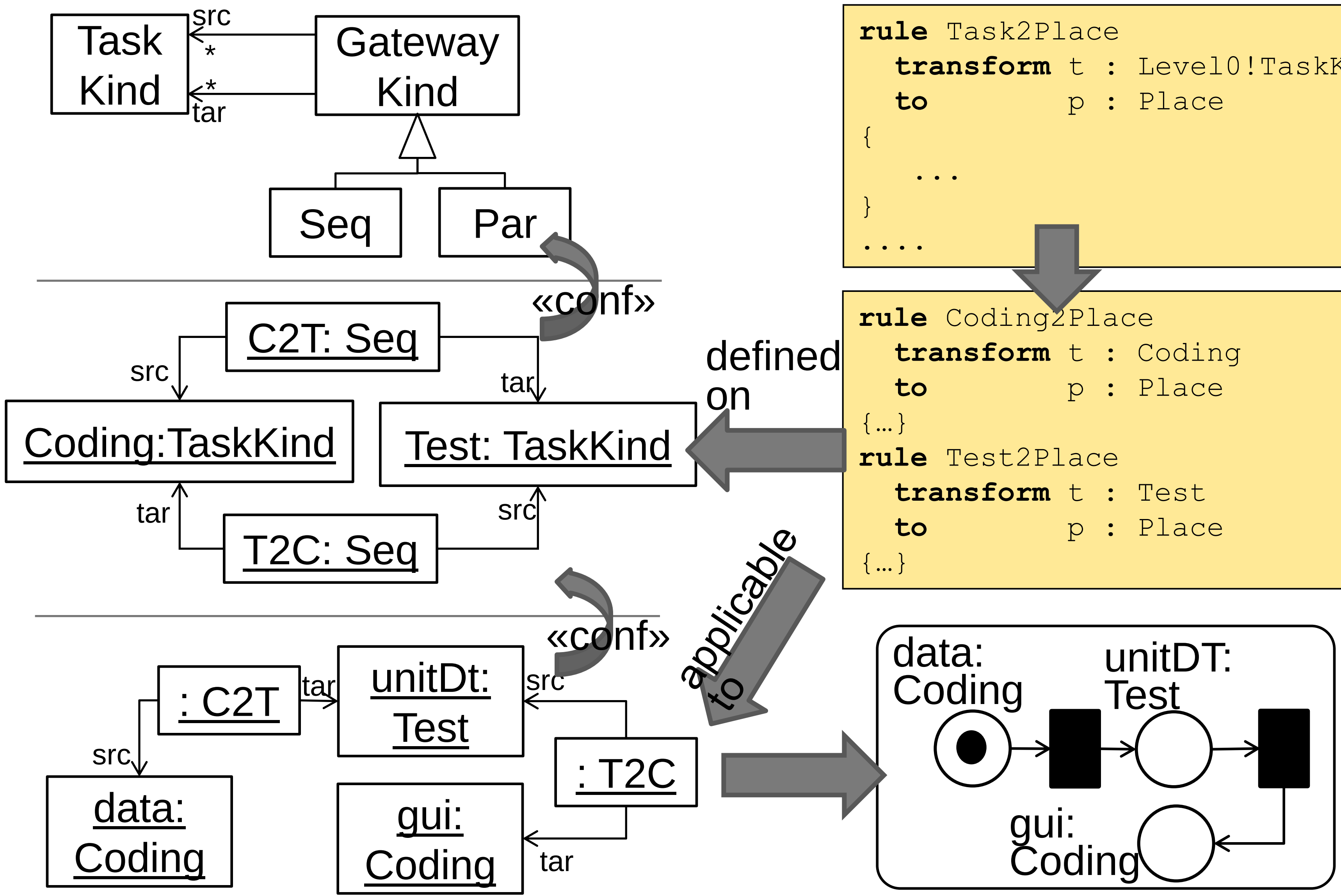
MULTI-LEVEL BASED GENERICITY



```
rule Task2Place
  transform t : Level0!TaskKind
  to        p : Place
{
   ...
}
....
```



MULTI-LEVEL BASED GENERICITY

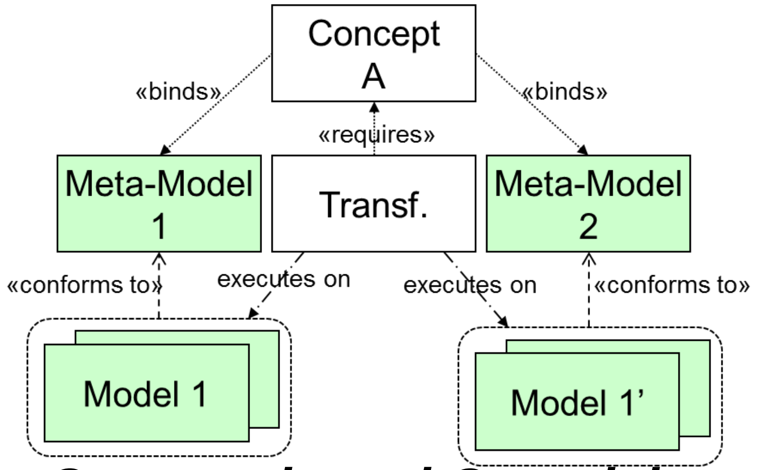


```
rule Task2Place
  transform t : Level0!TaskKind
  to        p : Place
{
   ...
}
....
```

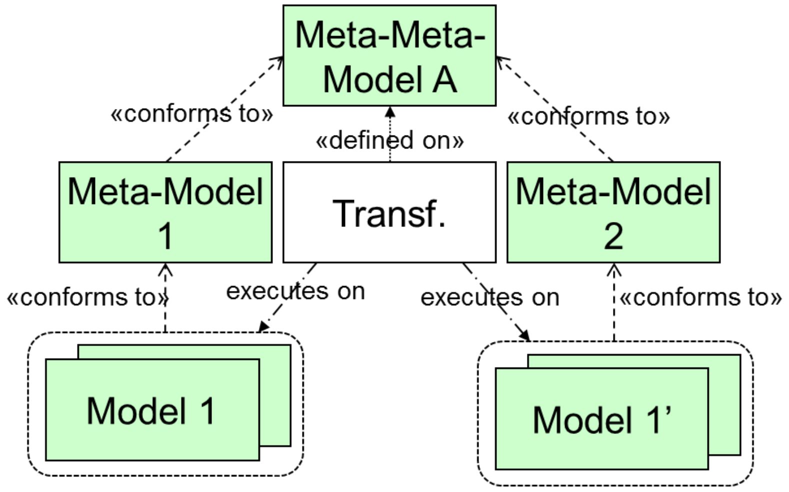
```
rule Coding2Place
  transform t : Coding
  to        p : Place
{…}
rule Test2Place
  transform t : Test
  to        p : Place
{…}
```

MULTI-LEVEL VS CONCEPT-BASED GENERICITY

- A posteriori binding: meta-models may exist first.
- Adapters and hybrid concepts to solve heterogeneities.



Concept-based Genericity



Multi-level-based Genericity

- A priori: meta-meta-model should exist first.
- Ability to apply an operation several meta-levels below.
- Domain-Specific meta-modelling.

CAN WE GET THE BEST OF BOTH APPROACHES?

Concept-based reuse

- **A posteriori** binding: meta-models may exist first.
- **Adapters** or Hybrid concepts to solve heterogeneities.

Multi-level reuse

- **Independence** of the **transformation language**

CAN WE GET THE BEST OF BOTH APPROACHES?

Let's make the typing relation:

- A-posteriori
- As flexible as adapters

A-POSTERIORI TYPING

A more flexible typing mechanism for MDE

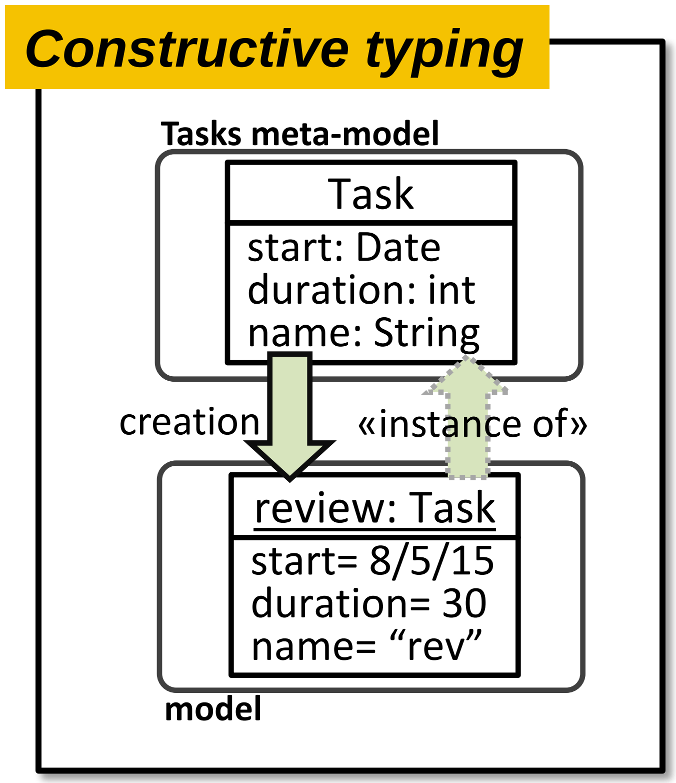
Decouple instantiation from classification

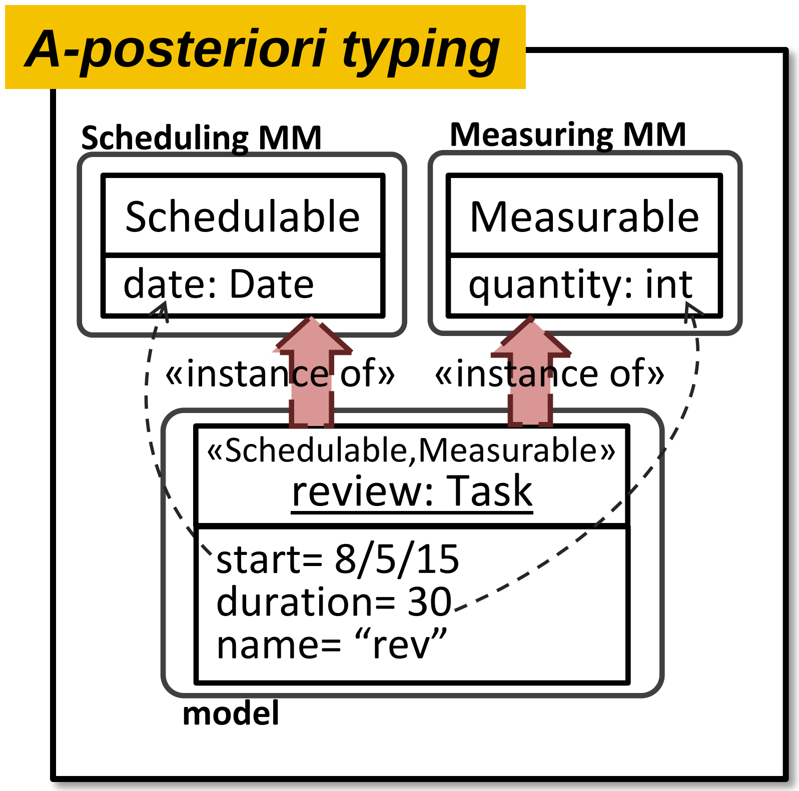
- Interfaces in object-oriented programming
- Roles in role-based programming languages

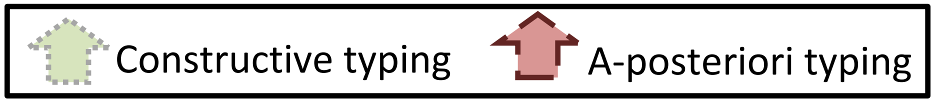
Allow dynamic typing and multiple classifiers for objects

Type and instance-level reclassification specifications

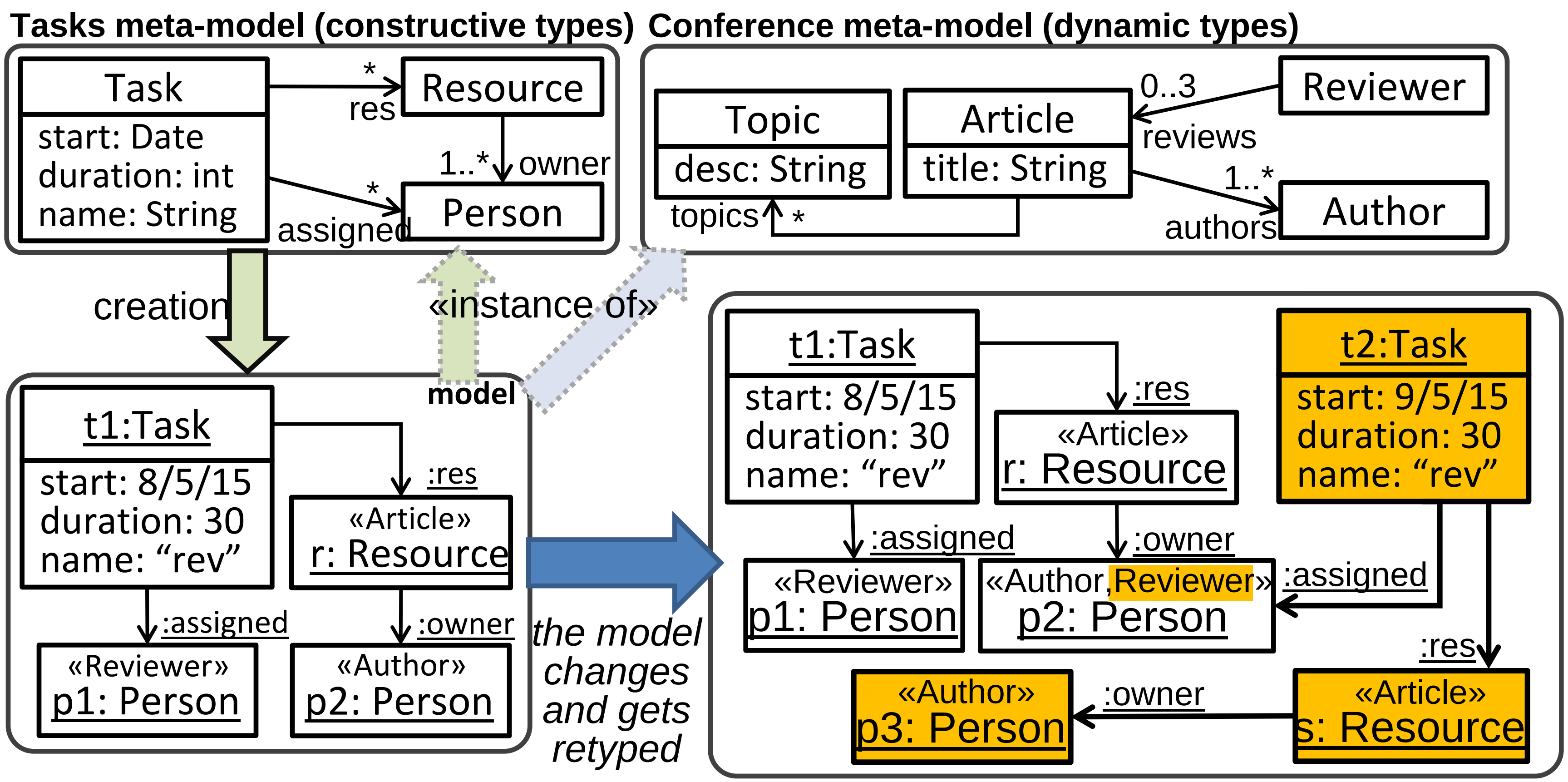
CONSTRUCTIVE VS A-POSTERIORI TYPING





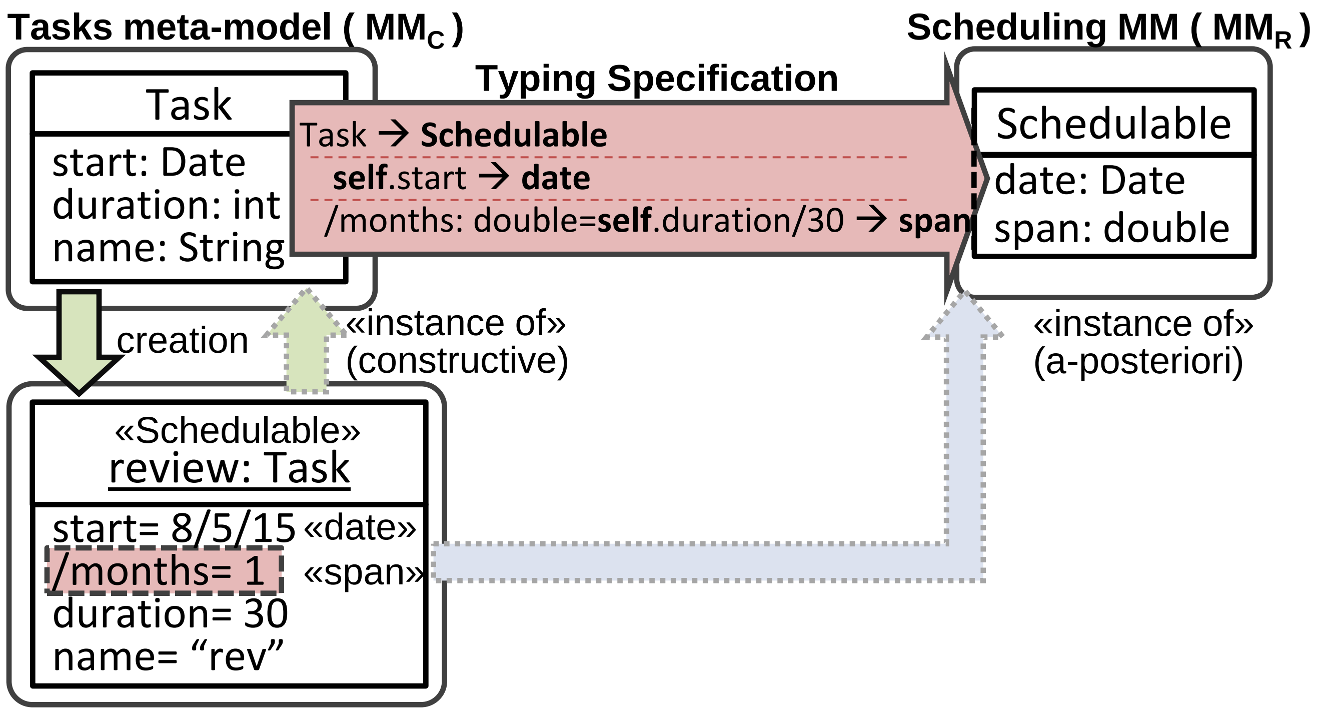


A-POSTERIORI TYPING: FLEXIBLE REUSE



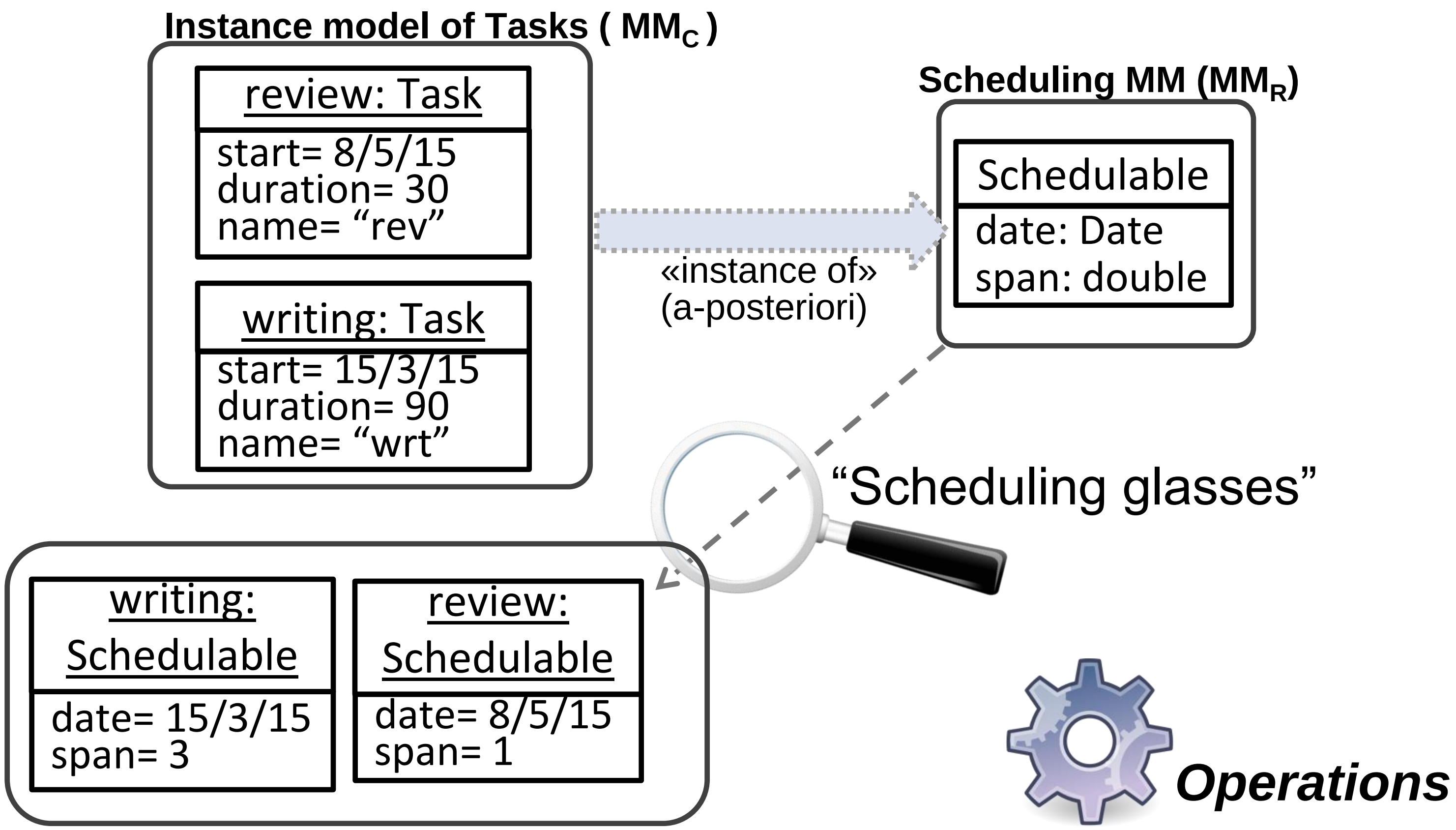
- A Person (constructive type) is only a Reviewer (a posteriori type) when some condition is met.

SPECIFYING AP TYPINGS AT THE TYPE-LEVEL



- Derived attributes (months) defined in the typing specification.
- ***Typing rules***: ensure syntactic correctness

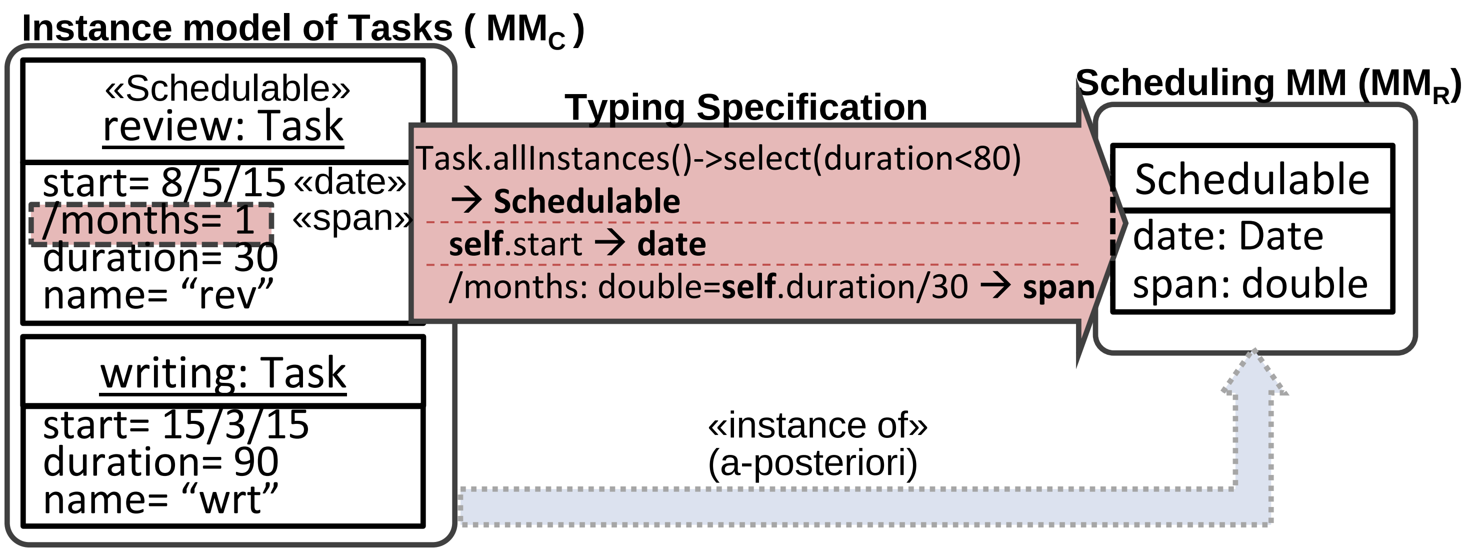
TYPE-LEVEL AP TYPING



{review, writing}← - - - - - - - - `Schedulable.allInstances()`

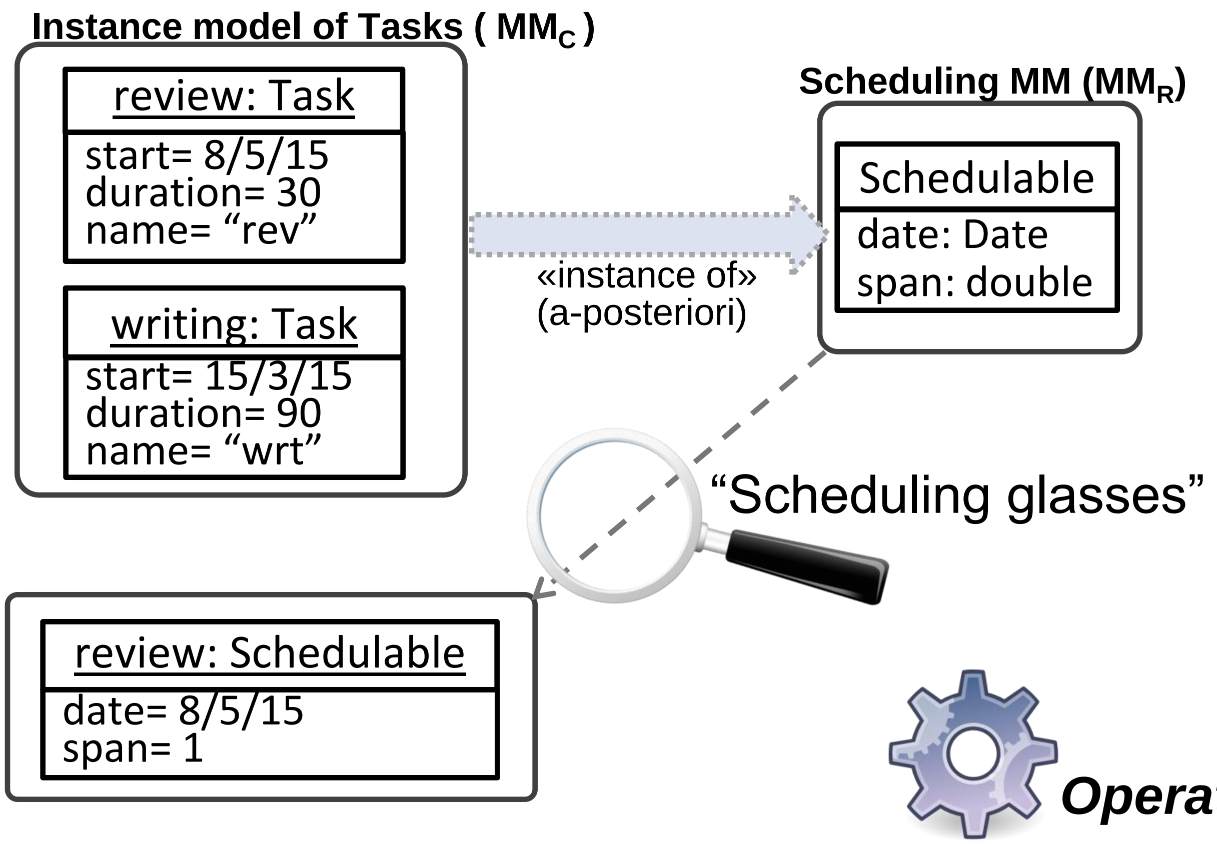
{1, 3} ← - - - - - - - - `Schedulable.allInstances()->collect(span)`

SPECIFYING AP TYPINGS AT THE INSTANCE LEVEL



- Typing defined by queries.
- Derived attributes defined by queries as well.

INSTANCE-LEVEL AP TYPING

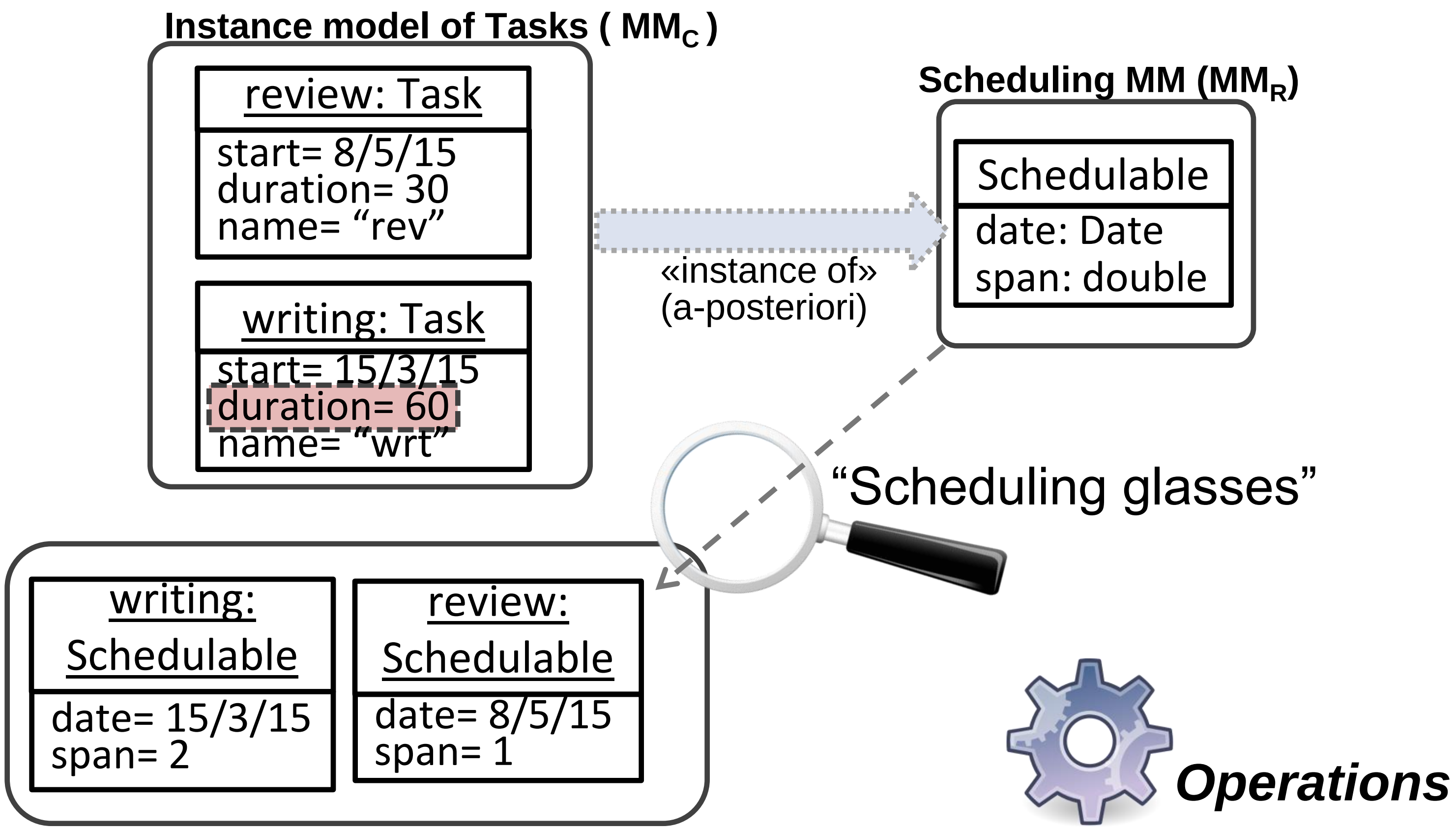


Operations

```
{review} <--------  Schedulable.allInstances()

{1}      <--------  Schedulable.allInstances()->
                    collect(span)
```

INSTANCE-LEVEL AP TYPING

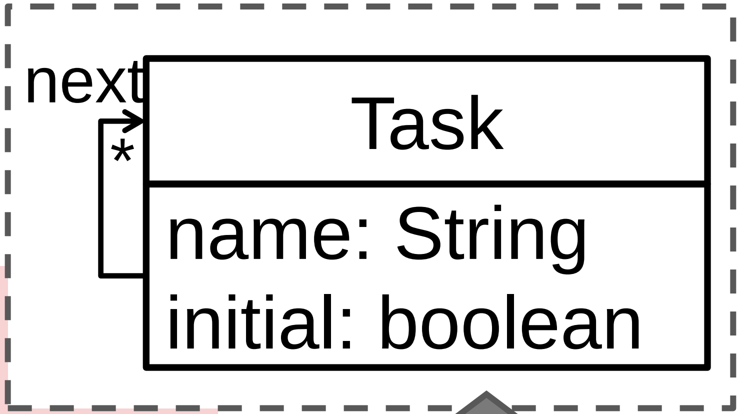


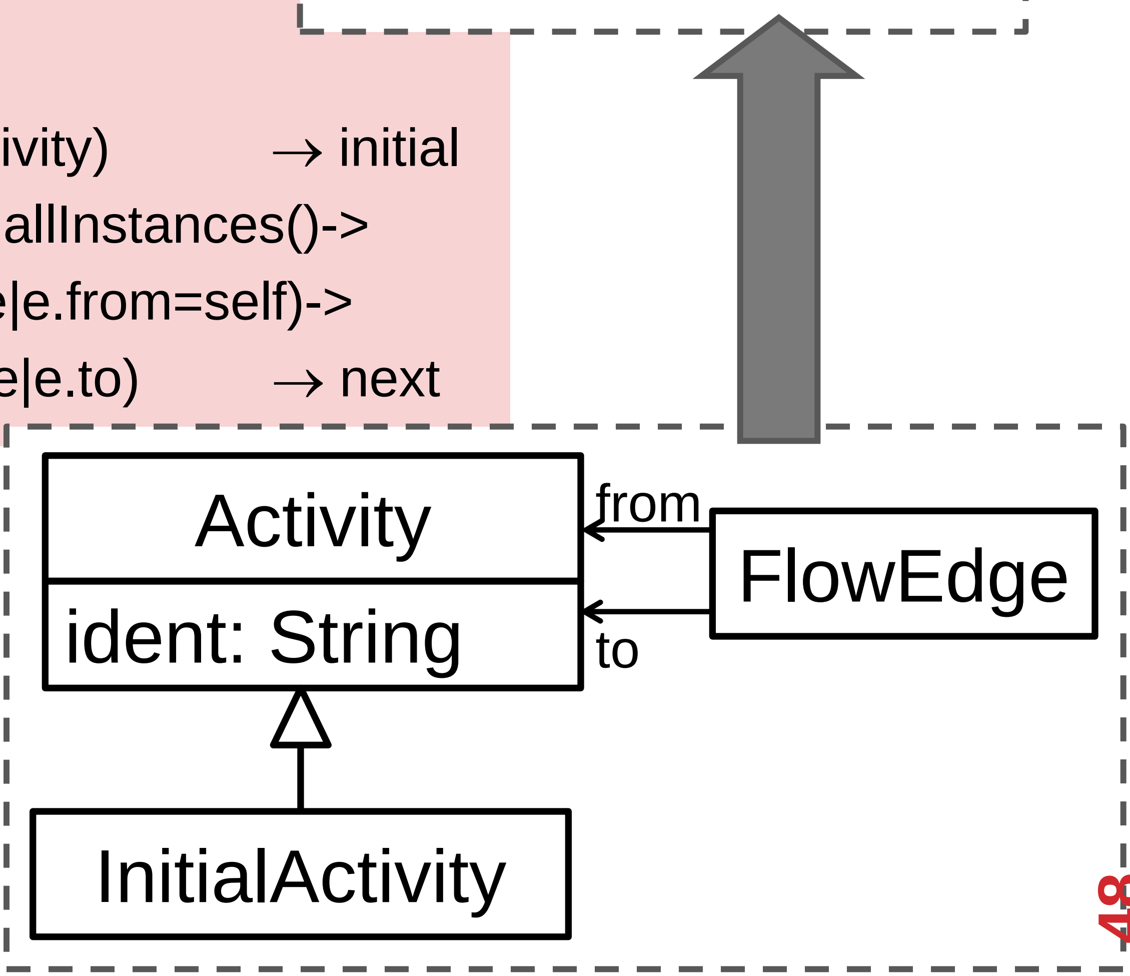
{review, writing} ←-------- `Schedulable.allInstances()`

{1, 2} ←-------- `Schedulable.allInstances()->collect(span)`

EXAMPLE

```
Activity → Task
  ident → name
  /init : boolean =
     self.oclsKindOf(InitialActivity)        → initial
  /follow: Task[*] = FlowEdge.allInstances()->
                       select(e|e.from=self)->
                       collect(e|e.to)        → next
```





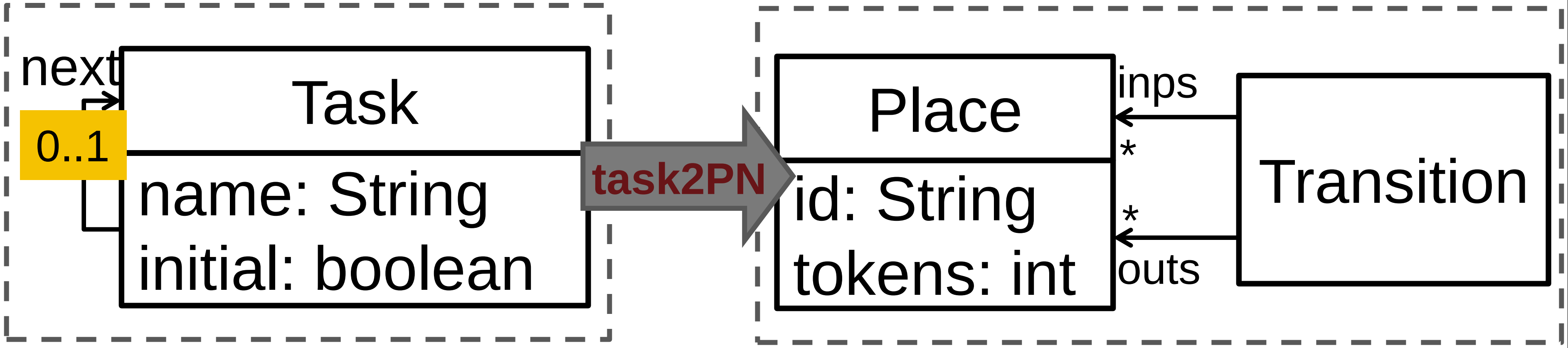
WAIT A MOMENT…

Aren’t retypings now a kind of transformation?

Producing a view of a given model

How expressive are retypings considered as model transformations?

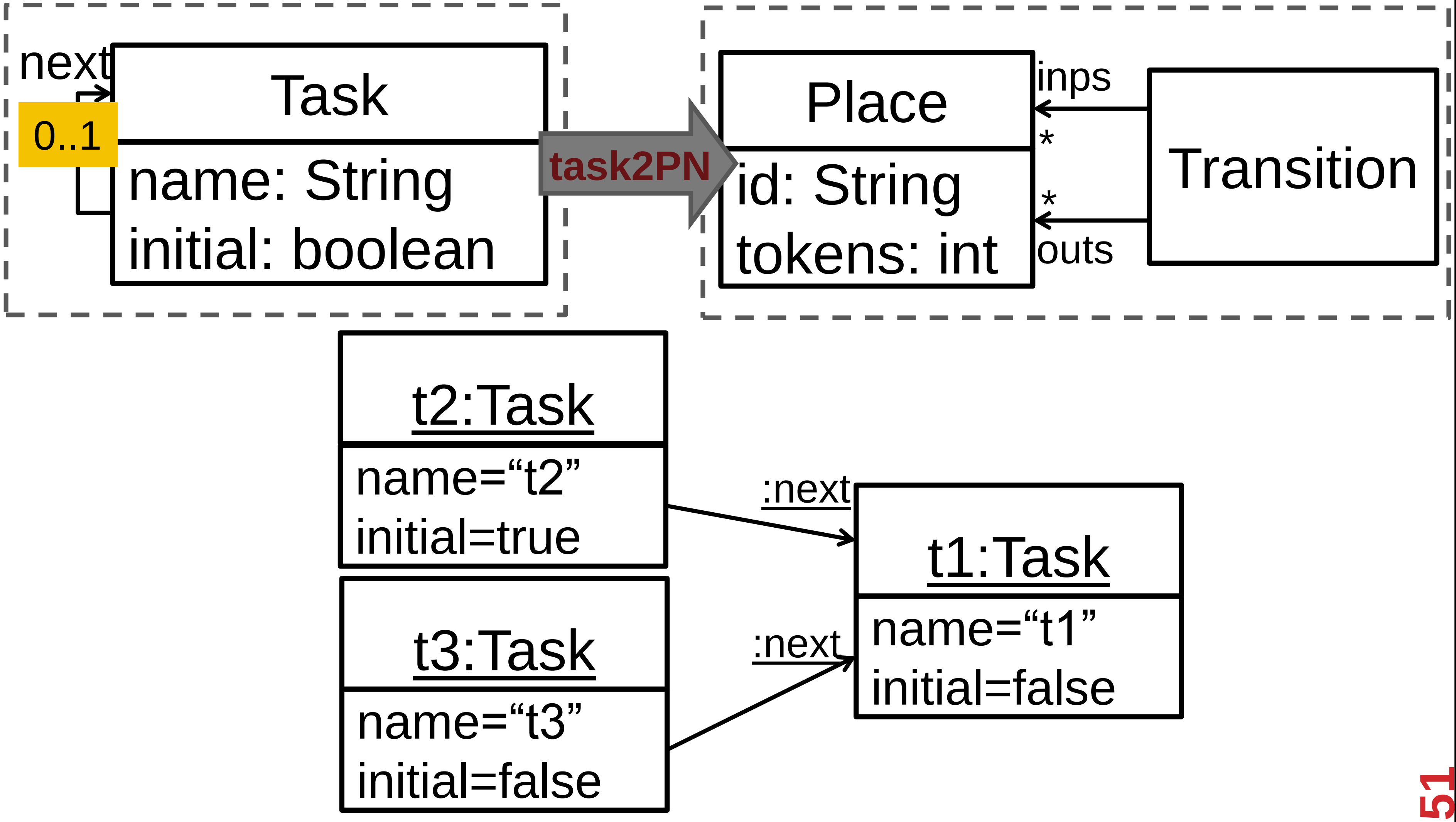
A-POSTERIORI TYPING AS A TRANSFORMATION



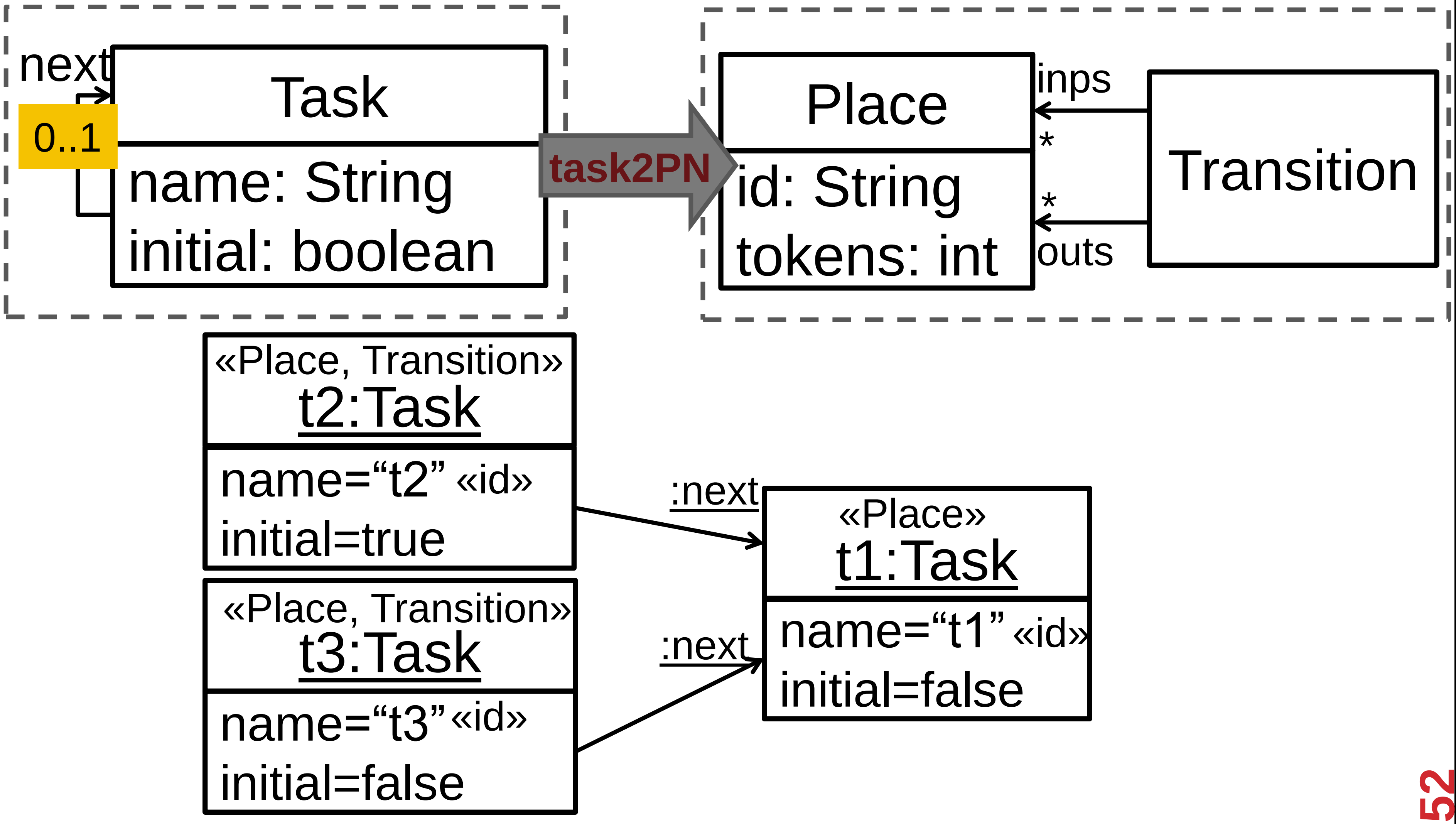
Task.allInstances() → Place
  name → id
  /toks : int = if (self.initial) then 1 else 0 → tokens

Task.allInstances()->select(t|t.next.isDefined()) → Transition
  /prevs : Task[*] = Set{self} → inps
  /nexts : Task[*] = Set{self.next} → outs

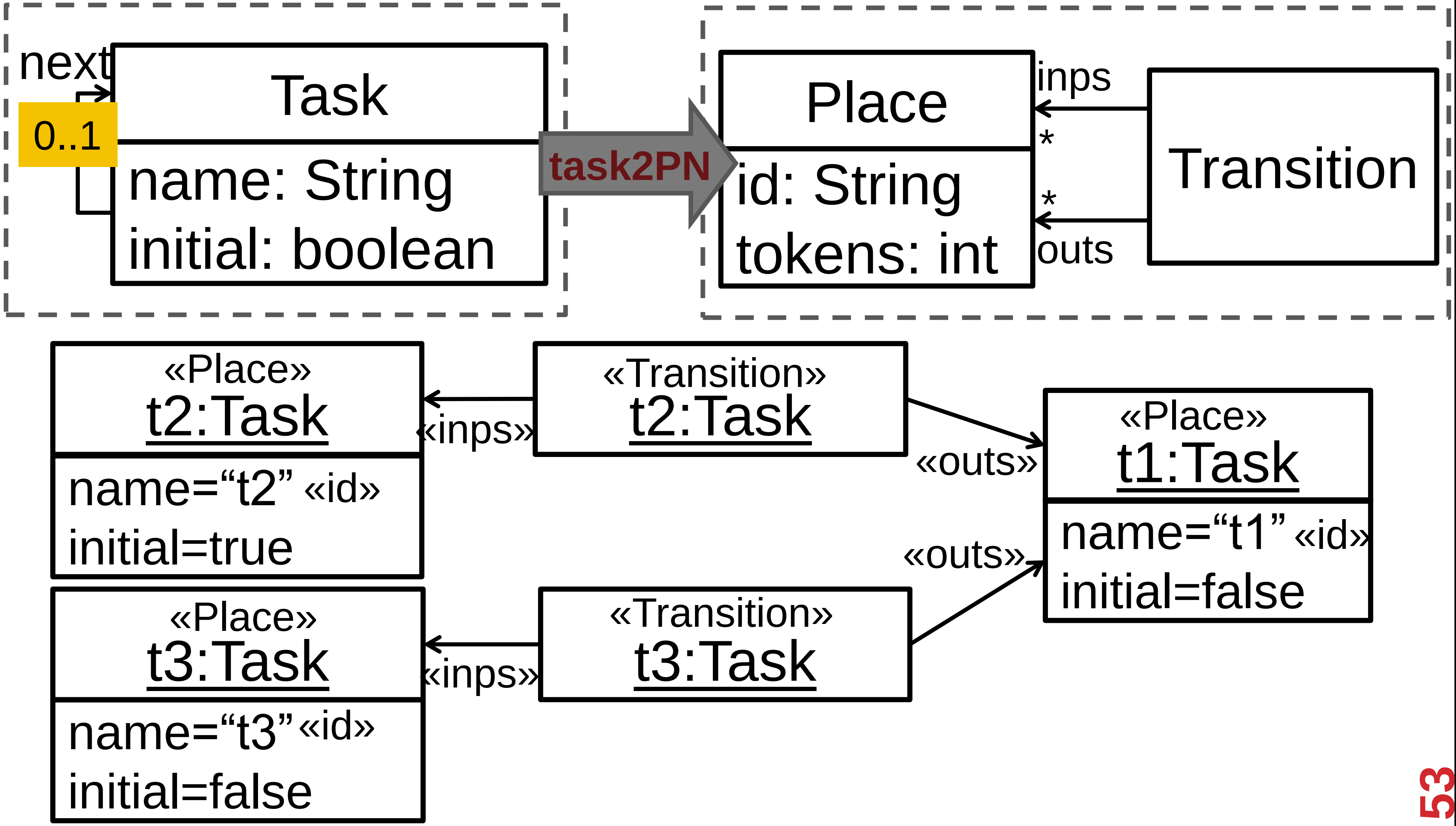
A-POSTERIORI TYPING AS A TRANSFORMATION



A-POSTERIORI TYPING AS A TRANSFORMATION



A-POSTERIORI TYPING AS A TRANSFORMATION



TYPING AS TRANSFORMATION: THE GOOD

No objects need to be created

- Just retype existing elements

Incrementality (forwards)

- For free

Bidirectionality

- For restricted cases of type-level specifications
- Incrementality (backwards) for (more) restricted cases

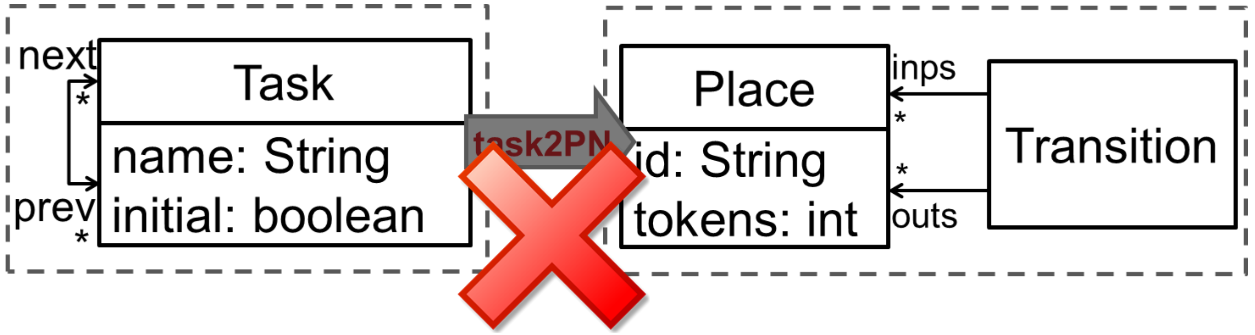
Analysis

- Executability
- Totality
- Surjectivity

TYPING AS TRANSFORMATION: THE BAD

Limited expressivity

- 1-to-1 or N-to-1 mappings
- 1-to-N and N-to-M only in certain cases



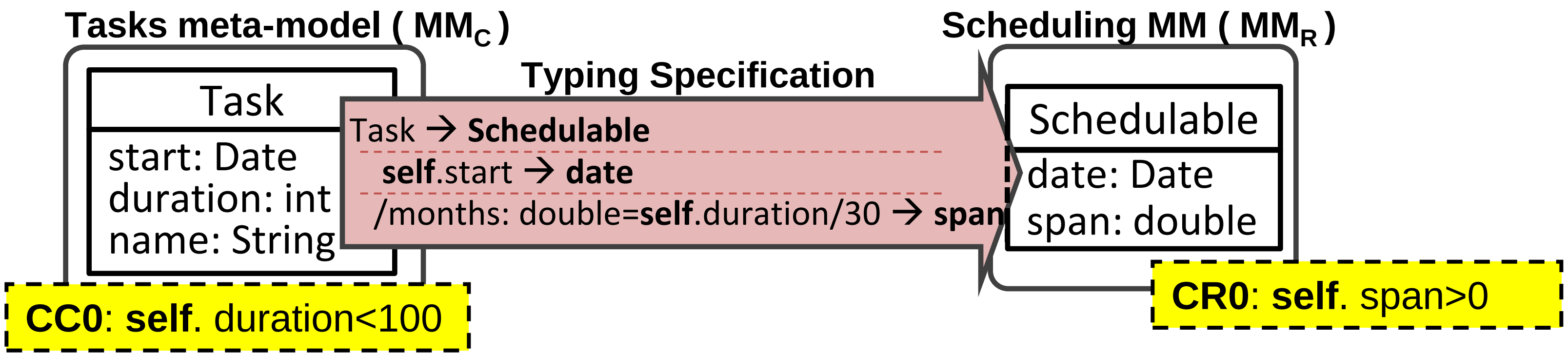
No replacement for regular transformation languages

- Helpers, auxiliary operations, etc

Efficiency operations for instance-level specifications

- T.allInstances() needs to compute all objects belonging to T
- Type caches and “smart” update policies

ANALYSIS OF A-POSTERIORI RETYPING



- Creation and role meta-models may have OCL constraints.
- **Executability:** Can ***some*** "Tasks" models become valid Scheduling models?.
- **Totality:** Can ***all*** Tasks models become valid Scheduling model?.
- **Surjectivity:** Can every Scheduling instance be obtained via retyping some Tasks instance?

SO WHAT? WHAT HAVE WE GAINED?

Concept-based reuse

- The transformation is adapted
- Adapters to solve heterogeneities
- **Automate ad-hoc reuse**

Multi-level reuse

- Language families
- Independence of the transformation language
- Transformation applicable across levels

Retyping-based reuse

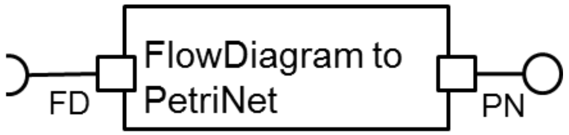
- Independence of the transformation language
- Heterogeneities can be resolved
- Typing becomes dynamic and multiple
- **Improve model-adaptation-based reuse**

Tool support

TOOL SUPPORT

bentō (http://www.miso.es/tools/bento.html)

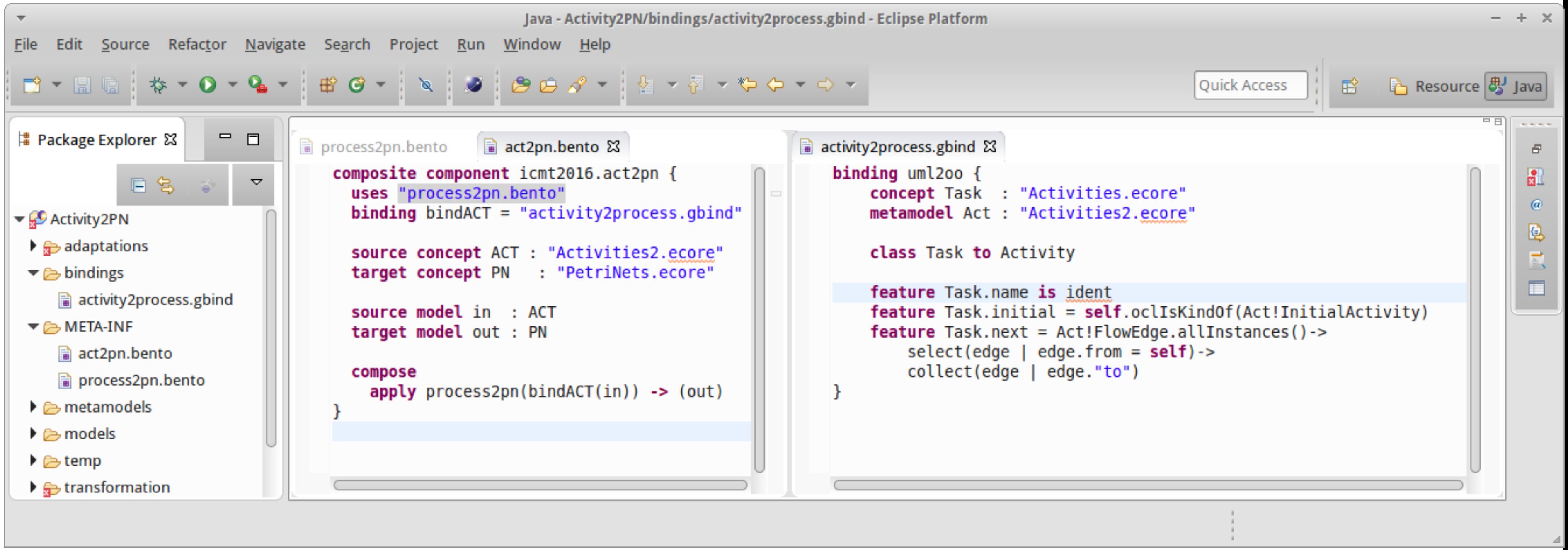
- Component model for transformations
- Concepts
- Domain specific language for bindings
- ATL transformations



metaDepth (http://metadepth.org)

- Multi-level textual modelling
- Integrated with the Epsilon languages (EOL, ETL, EGL)
- Concepts (structural and hybrid)
- Multi-level based reuse
- A-posteriori typing

BENTŌ



```
composite component icmt2016.act2pn {
  uses "process2pn.bento"
  binding bindACT = "activity2process.gbind"

  source concept ACT : "Activities2.ecore"
  target concept PN   : "PetriNets.ecore"

  source model in  : ACT
  target model out : PN

  compose
    apply process2pn(bindACT(in)) -> (out)
}
```

```
binding uml2oo {
    concept Task   : "Activities.ecore"
    metamodel Act  : "Activities2.ecore"

    class Task to Activity

    feature Task.name is ident
    feature Task.initial = self.oclIsKindOf(Act!InitialActivity)
    feature Task.next = Act!FlowEdge.allInstances()->
        select(edge | edge.from = self)->
        collect(edge | edge."to")
}
```

Adaptation of ATL transformations according to the binding

Support for refactoring meta-models into concepts

Sánchez Cuadrado, Guerra, de Lara. "*A Component Model for Model Transformations*". IEEE TSE 2014
Sánchez Cuadrado, Guerra, de Lara. "*Reusable Model Transformation Components with bentō*". ICMT' 15

METADEPTH

Multi-level textual modelling

Concepts

- Over every Epsilon language
- Hybrid concepts

A-posteriori typing

- Over every Epsilon language

Analysis of type-level specs

- Integration with the USE validator
- Bidirectional reclassification
- Reclassification totality and surjectivity

```
// Meta-model
Model Tasks {
  Node Task {
    start : Date;
    duration : int;
    name : String;
  }
}
```

```
// Model
Tasks someTasks {
  Task t0 {
    start = "30/04/2015";
    duration = 30;
    name = "coding";
  }
}
```

CONCEPT-BASED REUSE

```
concept SimpleTasks(&M,
                    &T,
                    &initial)
{
   Model &M {
       Node &T {
         &initial : boolean;
         name : String;
       }
   }
}
```

```
bind SimpleTasks(
   SEProcess,
   SEProcess::SETask,
   SEProcess::SETask.isInitial
)
```

ETL transformation

```
rule Task2Place
transform task : Source!&T
to place : Target!Place
{
   place.name := task.name;
   if (task.&initial=true)
      place.tokens:=1
   else
      place.tokens:=0
}
```

MULTI-LEVEL BASED REUSE

```
Model ProcessModel@2 {
   Node Task {
     name@1  : String[0..1];
     initial : boolean = false;
   }
}
```

```
ProcessModel SoftwareProcess {
   Task Analysis {
     name = “reqs, analysis”;
   }
}
```

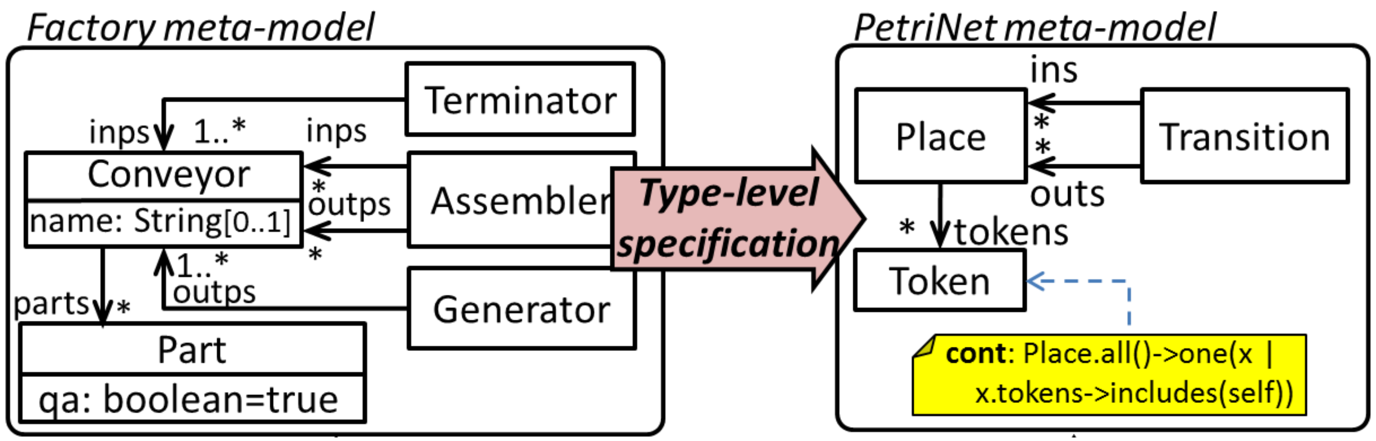
```
SoftwareProcess aSoftProcess{
   Analysis a {
     initial = true;
   }
}
```

ETL transformation

```
@model(potency=0)
rule Task2Place
transform task : Source!Task
to place : Target!Place
{
   place.name := task.name;
   if (task.initial=true)
      place.tokens:=1
   else
      place.tokens:=0
}
```

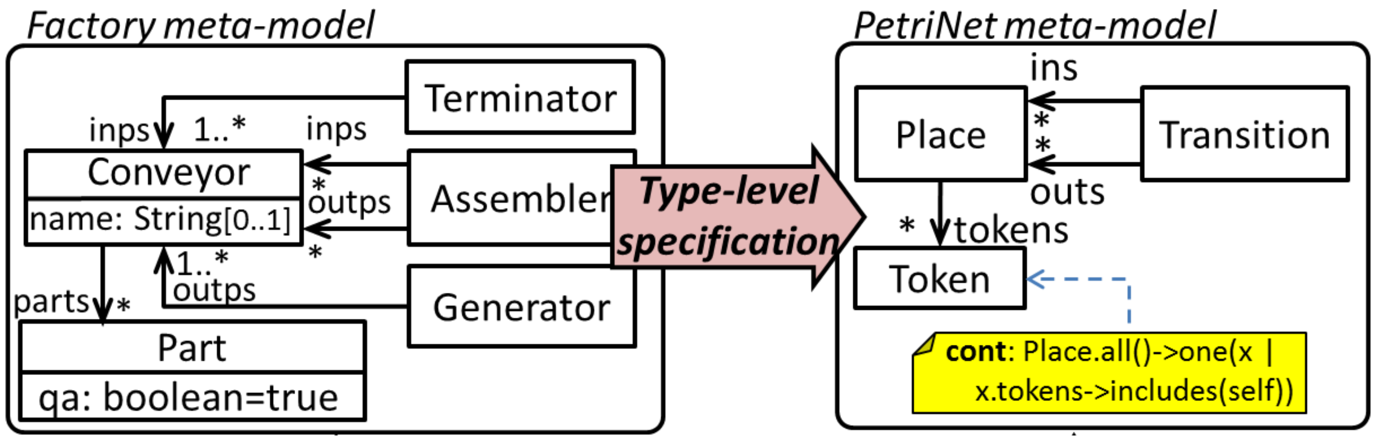
A-POSTERIORI TYPING:
BX MODEL TRANSFORMATIONS

Simple bidirectional model transformations by reclassification



A-POSTERIORI TYPING: BX MODEL TRANSFORMATIONS

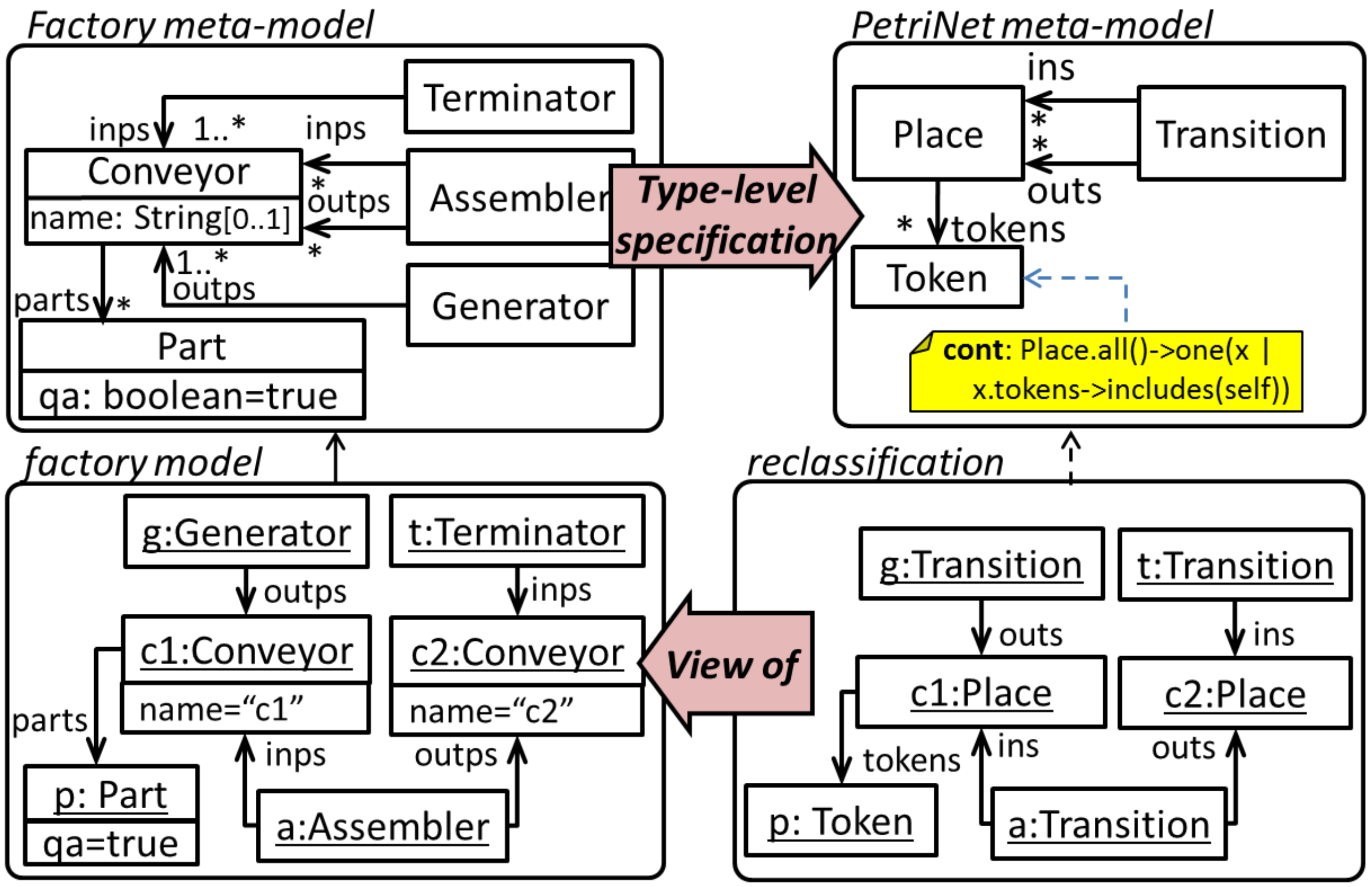
Simple bidirectional model transformations by reclassification



```
type Factory PetriNet {
  Conveyor::parts > Place::tokens,
  Generator::outps > Transition::outs,  Terminator::inps > Transition::ins,
  Assembler::inps > Transition::ins, Assembler::outps > Transition::outs,
}
```

A-POSTERIORI TYPING: BX MODEL TRANSFORMATIONS

Simple bidirectional model transformations by reclassification



Forward reclassification

A-POSTERIORI TYPING: BX MODEL TRANSFORMATIONS

The typing specification can also be used backwards!

Might yield multiple AP typings for a model

```
MetaDepth console
> dump example as Factory
```

```
PetriNet example {
  Place p {}
  Transition t { ins = [p]; }
}
```

```
Factory example { // 1st typing
   Conveyor p {}
   Terminator t { inps= [p]; }
}
Factory example { // 2nd typing
   Conveyor p {}
   Assembler t { inps= [p]; }
}
```

A-POSTERIORI TYPING: BX MODEL TRANSFORMATIONS

The reclassification is not total

- Equivalently, the backwards reclassification is not surjective

The solver produces a witness

- A Factory model that cannot be reclassified into a Petri net

```
// Model witness with no Petri net equivalent
Factory noRefinementWitness {
  Assembler assembler2 { outps= [conveyor2, conveyor1]; }
  Conveyor conveyor1 { name= "string1"; }
  Conveyor conveyor2 { name= "string1"; }
  Generator generator2 { outps= [conveyor1]; }
  Part part2 {
    qa = true;
  } // This part would become a token outside any Place
}
```

APPLICATIONS: REUSE OF OPERATIONS

Simulator for Petri nets can be reused “as is” for Factory

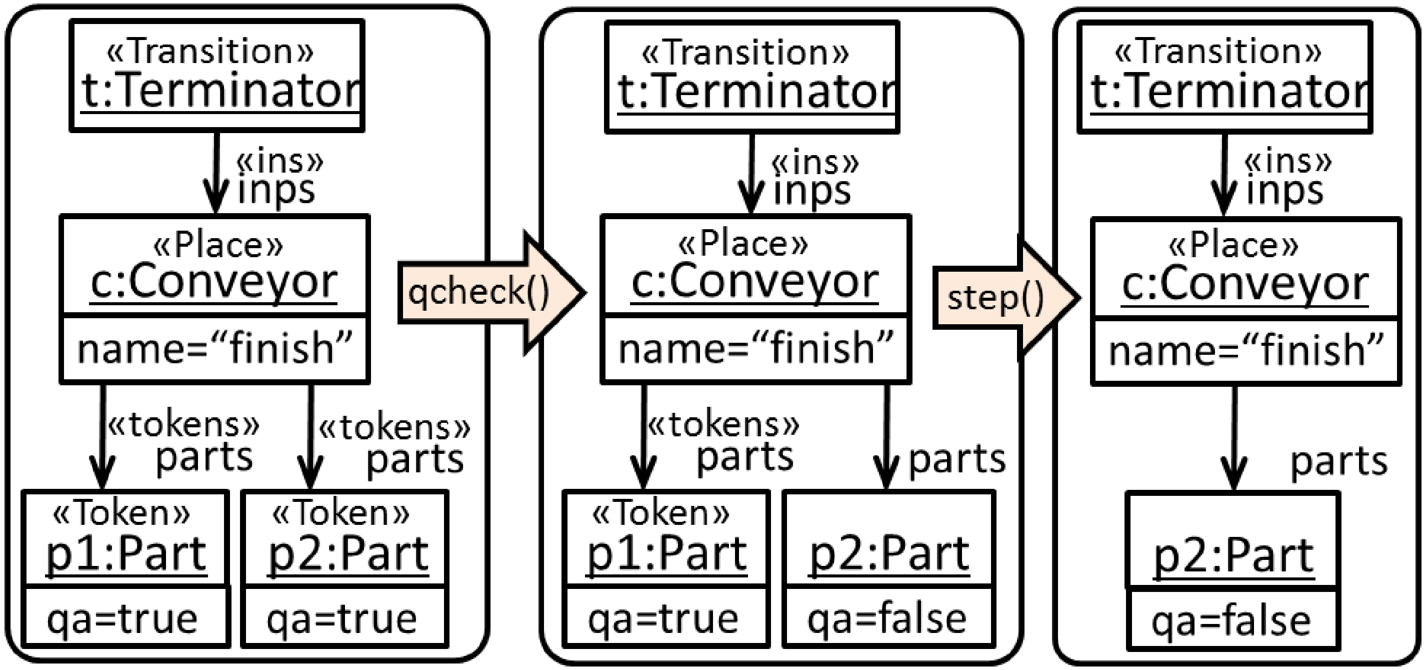
- Provide an AP typing from Factory to Petri nets
- Factory models become typed as Petri nets

```
// EOL excerpt of the simulator
operation Transition enabled() : Boolean {
 return self.ins.forAll(p| p.tokens.size()>0);
}

operation step() : Boolean {
  var enabled : Set(Transition) := Transition.all.select( t | t.enabled());
  ... // fire one random Transition from enabled
}
```

APPLICATIONS: FLEXIBLE REUSE

```
// Type parts as tokens only if QA is passed
type Factory PetriNet inst {
   $Conveyor.all$ > Place with {
      /sp : Token[*] = $self.parts.select(p|p.qa=true)$ > tokens
   }
   $Part.all.select( p | p.qa = true )$ > Token
}
```



SUMMARY

Approaches to reuse in model transformation

- Reuse transformations for other meta-models

Concept-based

- Ideas from generic programming
- Hybrid concepts, adapters

Multi-level modelling

- Families of languages

A-posteriori typing

- Provide additional types via retyping specifications
- Dynamic typing

FUTURE WORK

Syntactic vs Semantic reuse correctness

- Transformation intents

Improve tool support for retypings and reuse

- Repositories of reusable transformations
- Reuse recommenders

Exploit structural typing in MDE

Explore multiple typings for MDE

Explore more in detail retypings as bx transformations

TAKE-HOME LESSON

if you put water into a cup, it becomes the cup

TAKE-HOME LESSON

if you put water into a cup, it becomes the cup

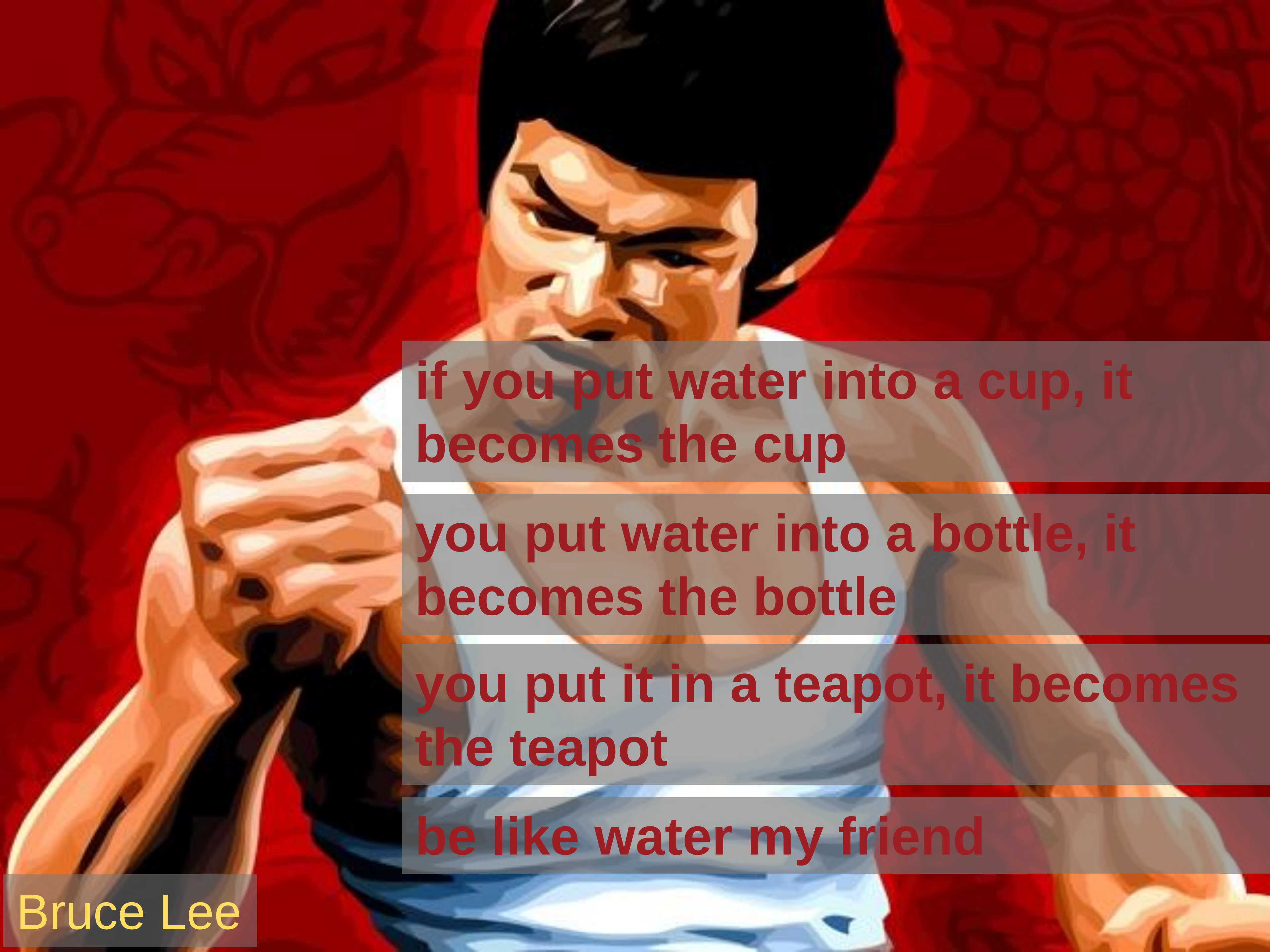
you put water into a bottle, it becomes the bottle

TAKE-HOME LESSON

if you put water into a cup, it becomes the cup

you put water into a bottle, it becomes the bottle

you put it in a teapot, it becomes the teapot



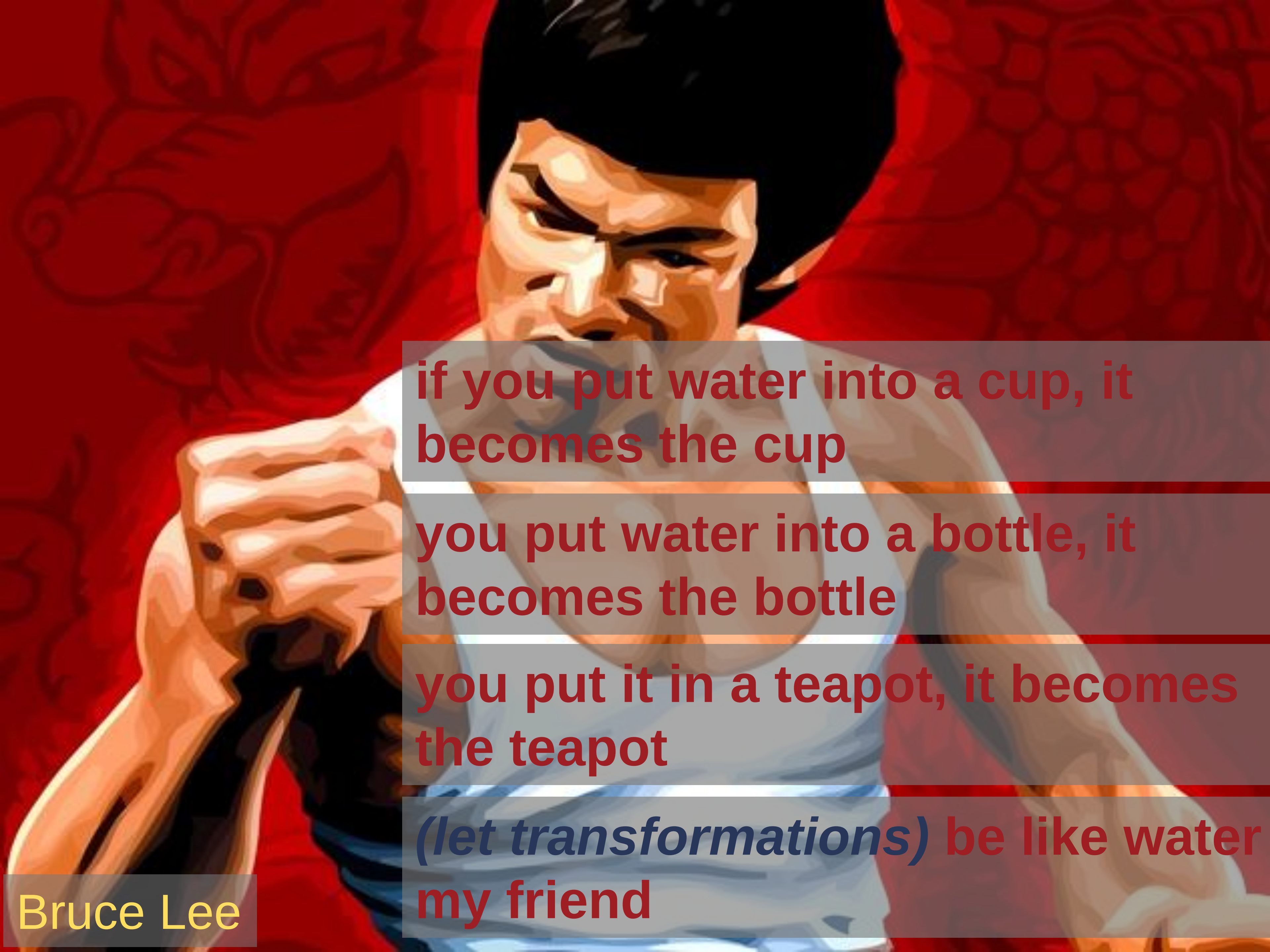
if you put water into a cup, it becomes the cup

you put water into a bottle, it becomes the bottle

you put it in a teapot, it becomes the teapot

be like water my friend

Bruce Lee



if you put water into a cup, it becomes the cup

you put water into a bottle, it becomes the bottle

you put it in a teapot, it becomes the teapot

(let transformations) be like water my friend

Bruce Lee

THANKS!

Juan.deLara@uam.es

@miso_uam

joint work with
E. Guerra and J. Sánchez Cuadrado

miso

http://www.miso.es