SURVEYING RULE INHERITANCE IN MODEL-TO-MODEL TRANSFORMATION LANGUAGES

Evaluation Examples

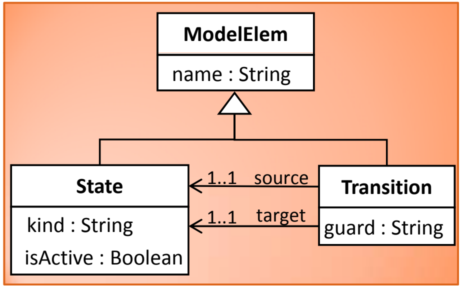
EXAMPLES

- In these tiny examples, simple state machines should be transformed into simple petri nets;
- The intent of the examples is to discover how diverse transformation languages (Kermeta, QVT-O, TGGs, TNs, ATL, ETL) interpret inheritance
- Thus, each example variation tries to reveal a certain aspect (as detailed by the goal of evaluation)

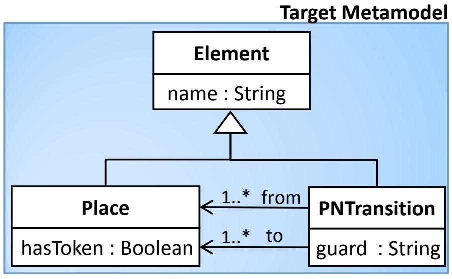
Example 1

- **Description**
  - In this example `ModelElems` should be transformed into `Elements` and `States` into `Places` by two inheriting rules in order to reuse the `name` assignment;
  - Moreover, only those `State` instances should be transformed, whose `kind` is unequal „`initial`“
- **Goal(s) of Evaluation**
  - (1) Check for which instances, a certain rule is applicable, i.e., whether indirect instances (like `Transitions` in the example) are affected
  - (2) Check, whether the assignments of a superrule are inherited by a subrule
  - (3) Check, whether elements are re-matched by a more general rule, if a condition fails

Source Metamodel

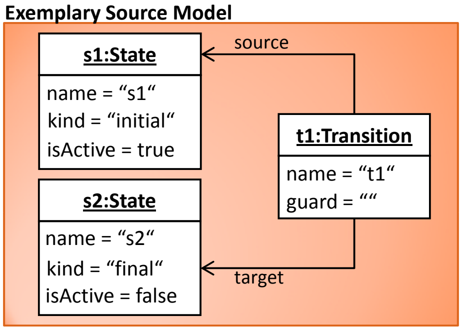


Target Metamodel

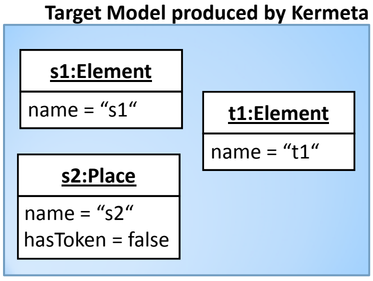


Example 1 – Kermeta (1/3)

Exemplary Source Model



Target Model produced by Kermeta



- **Results/Findings** (according to goals)
  - (1) Kermeta allows to implement type substitutability; nevertheless there is no direct support for anything, since the transformation code, representing transformation rules, must be explicitly called (so the matching as well as the rule selection is done manually); for this, the trace model has to be explicitly kept (i.e., there is no support for an automatically generated trace model)
  - (2) In Kermeta the assignments of the superrule may be inherited; nevertheless, this has to be implemented manually, again; a major problem in this respect is that the interfaces of the methods must be kept identical (i.e., neither the parameters nor the return type might be changed co- or/and contravariant); consequently type casts in subrules are required
  - (3) In Kermeta elements might be matched by a more general rule, if the condition fails; nevertheless, this is entirely under control of the developer, since the matching behavior is programmed manually

EXAMPLE 1 – KERMETA (2/3)

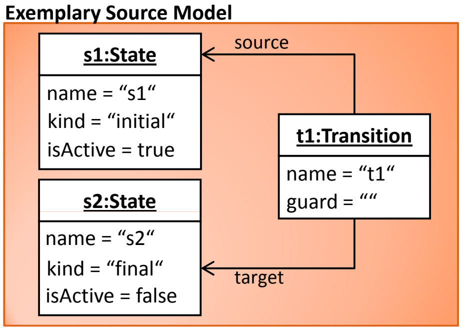
```
//transformation code for Statemachine2PetriNet
class Statemachine2PetriNet{
  operation conditionFulFilled(s : Statemachine) : kermeta::standard::Boolean is do
    result := true
  end

  operation assignments(s : Statemachine, p : PetriNet) is do
  end

  operation referenceAssignments(s : Statemachine, p : PetriNet,
  trace: Trace<Object, Object>) is do
    s.elements.each{ e |
      if trace.getTargetElem(e) != void then
        p.elements.add(trace.getTargetElem(e).asType(Element))
      end
    }
  end
}

//transformation code for ModelElem2Element
class ModelElem2Element{
  operation conditionFulFilled(m : ModelElem) : kermeta::standard::Boolean is do
    result := true
  end

  operation assignments(m : ModelElem, e : Element) is do
    e.name := m.name
  end

  operation referenceAssignments(m : ModelElem, e : Element,
  trace: Trace<Object, Object>) is do
  end
}
```

EXAMPLE 1 – KERMETA (3/3)

```
//transformation code for State2Place
class State2Place inherits ModelElem2Element{
  method conditionFulFilled(m : ModelElem) : kermeta::standard::Boolean is do
    result := super(m)
    result := result and (m.asType(State)).kind != "initial"
  end

  method assignments(m : ModelElem, e : Element) is do
    super(m,e)
    (e.asType(Place)).hasToken := (m.asType(State)).isActive
  end

  method referenceAssignments(m : ModelElem, e : Element, trace: Trace<Object, Object>) is do
    super(m,e,trace)
  end
}
```

EXAMPLE 1 – QVT-O (1/2)

Exemplary Source Model



s1:State
name = “s1“
kind = “initial“
isActive = true

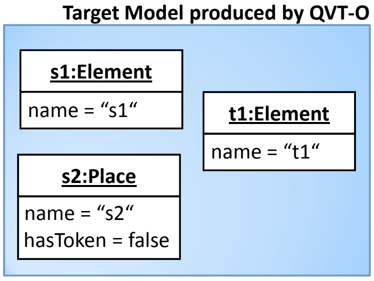
source

t1:Transition
name = “t1“
guard = ““

s2:State
name = “s2“
kind = “final“
isActive = false

target

Target Model produced by QVT-O



s1:Element
name = “s1“

t1:Element
name = “t1“

s2:Place
name = “s2“
hasToken = false

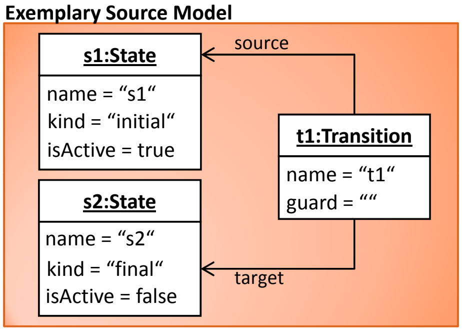
- **Results/Findings** (according to goals)
  - (1) QVT-O supports type substitutability, i.e., indirect instances are transformed by a certain rule, if no specialized rule exists, that is called before;
    This is inferred from the fact, that the instance `t1` of type `Transition` has been matched by the rule `ModelElem2Element` resulting in an instance `t1` of type `Element`
  - (2) In QVT-O the assignments of the superrule are inherited (keyword `inherits`);
    This is inferred from the fact that, e.g., instance `s2` has a corresponding name
  - (3) In QVT-O elements are re-matched by a more general rule, if the conditions fails;
    This is inferred from the fact, that an instance `s1` of type `Element` results originating from `s1` of type `State`

EXAMPLE 1 – QVT-O (2/2)

```
transformation testTrafo(in inModel : sm, out outModel : pn);
  main() {
    inModel.rootObjects()[Statemachine] -> map SM2Petri();
  }

  mapping Statemachine::SM2Petri() : PetriNet {
    //please note that specific rules must be called first!
    elements := self.elements[State] -> map State2Place();
    elements += self.elements[ModelElem] -> map ModelElem2Element();
  }

  mapping ModelElem::ModelElem2Element() : Element {
    name := self.name;
  }

  mapping State::State2Place() : Place inherits ModelElem::ModelElem2Element
  when{self.kind != 'initial'}{
    result.hasToken := self.isActive;
  }
```

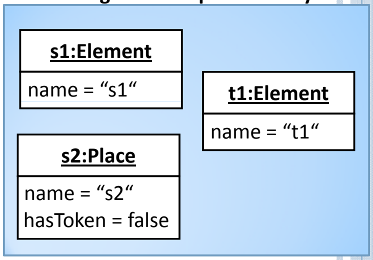
EXAMPLE 1 – TGGS (1/2)

Exemplary Source Model



Rule definitions see next slide

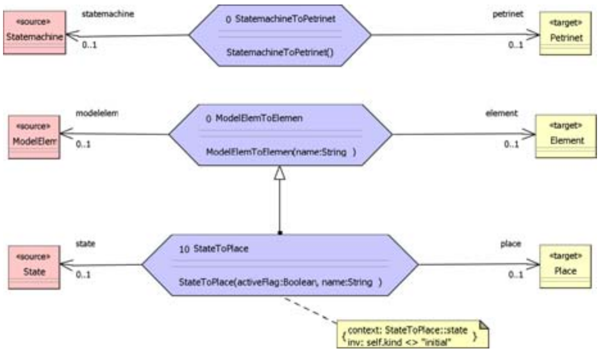
Target Model produced by TGGs



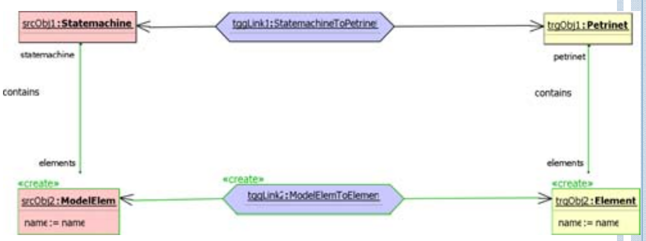
- **Results/Findings** (according to goals)
  (1) TGGs support type substitutability, i.e., indirect instances are transformed by a certain rule, if no specialized rule exists that matches;
  (2) In TGGs the assignments of the superrule are inherited (but have to be repeated explicity by a copy);
  (3) In TGGs elements are re-matched by a more general rule, if the specialized rule application fails;

EXAMPLE 1 – TGGS (2/2)

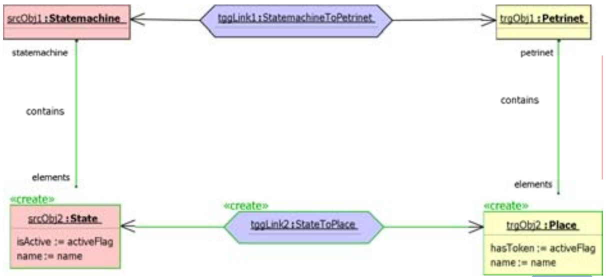
TGG-Schema (type level)



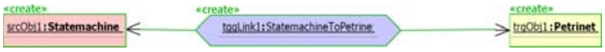
ModelElemToElemen(…)



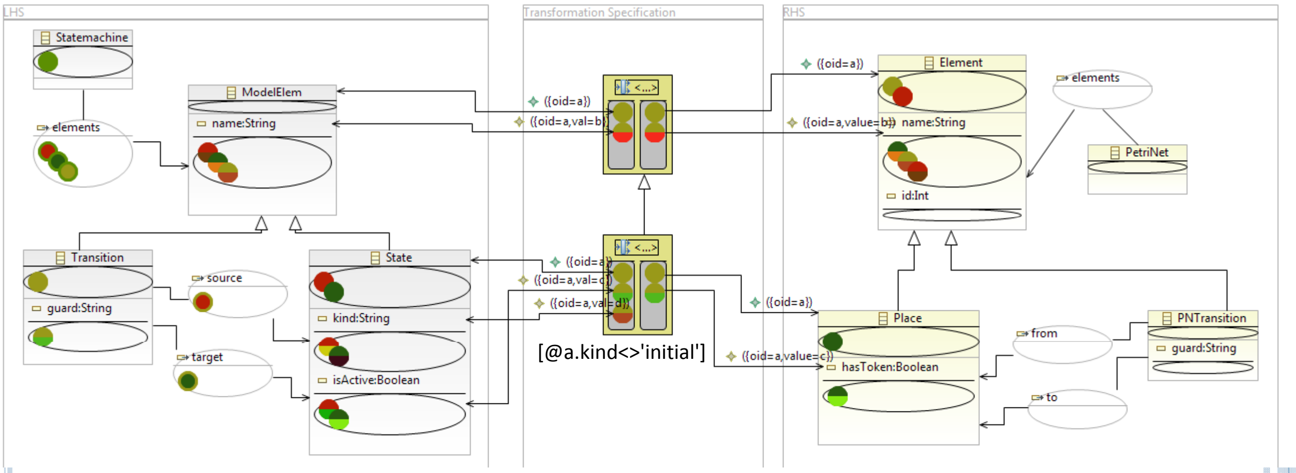
StateToPlace(…)



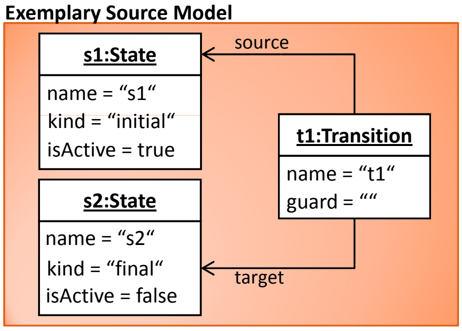
StatemachineToPetrinet(…)



EXAMPLE 1 – TNS (1/2)



Exemplary Source Model



s1:State

name = “s1“
kind = “initial“
isActive = true

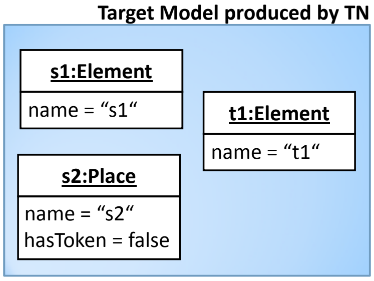
s2:State

name = “s2“
kind = “final“
isActive = false

t1:Transition

name = “t1“
guard = ““

Target Model produced by TN



s1:Element

name = “s1“

t1:Element

name = “t1“

s2:Place

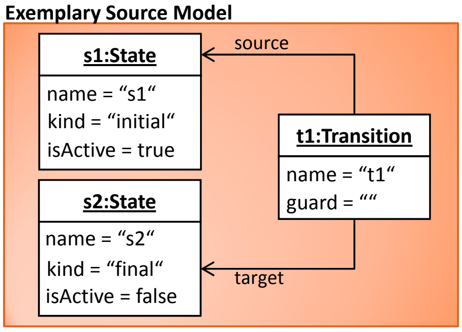
name = “s2“
hasToken = false

EXAMPLE 1 – TNS (2/2)

- **Results/Findings** (according to goals)
  - (1) TNs support type substitutability, i.e., indirect instances are transformed by a certain rule, if no specialized rule exists that matches;
    This is inferred from the fact that the instance `t1` of type `Transition` has been matched by the rule `ModelElem2Element` resulting in an instance `t1` of type `Element` (green token)
  - (2) In TNs, the assignments of the superrule are inherited;
    This is inferred from the fact, that, e.g., instance `s2` (dark green token) has a corresponding name
  - (3) In TNs, elements are re-matched by a more general rule, if the condition fails (`kind` of `s1` is `initial` and therefore not matched);
    This is inferred from the fact, that an instance `s1` (red token) of type `Element` results originating from `s1` of type `State`

EXAMPLE 1 – ATL

Exemplary Source Model



s1:State
name = “s1“
kind = “initial“
isActive = true

s2:State
name = “s2“
kind = “final“
isActive = false

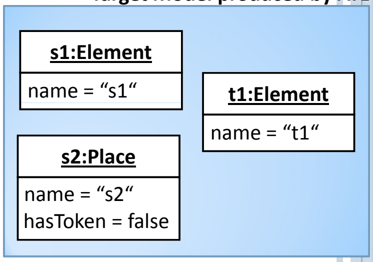
t1:Transition
name = “t1“
guard = ““

source

target

```
rule ModelElem2Element{
 from mElem : Statemachine!ModelElem
 to elem : Petrinet!Element (
   name <- mElem.name
  )
}

rule State2Place extends ModelElem2Element {
 from mElem : Statemachine!State (
   mElem.kind <> ‘initial’)
 to elem : Petrinet!Place (
   hasToken <- mElem.isActive
  )
}
```

Target Model produced by ATL



s1:Element
name = “s1“

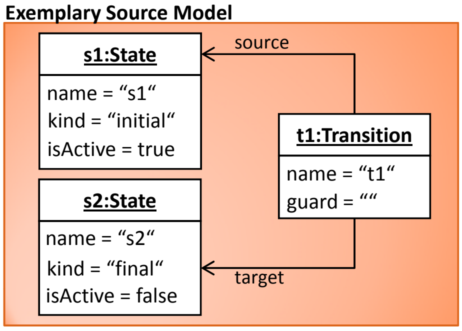
t1:Element
name = “t1“

s2:Place
name = “s2“
hasToken = false

- **Results/Findings** (according to goals)
  - (1) ATL supports type substitutability, i.e., indirect instances are transformed by a certain rule, if no specialized rule exists, that matches;
    This is inferred from the fact, that the instance `t1` of type `Transition` has been matched by the rule `ModelElem2Element` resulting in an instance `t1` of type `Element`
  - (2) In ATL the assignments of the superrule are inherited;
    This is inferred from the fact that, e.g., instance `s2` has a corresponding name
  - (3) In ATL elements are re-matched by a more general rule, if the conditions fails;
    This is inferred from the fact, that an instance `s1` of type `Element` results originating from `s1` of type `State`

EXAMPLE 1 – ETL

Exemplary Source Model



s1:State
name = “s1“
kind = “initial“
isActive = true

s2:State
name = “s2“
kind = “final“
isActive = false

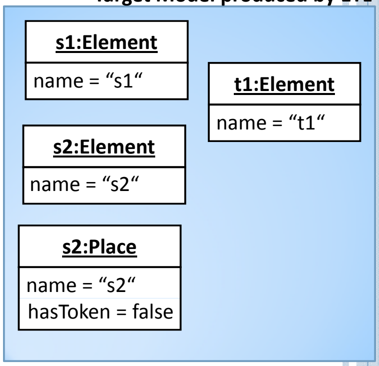
t1:Transition
name = “t1“
guard = ““

source

target

```
@greedy
rule ModelElem2Element
 transform mElem : Statemachine!ModelElem
 to elem : Petrinet!Element {
   elem.name := mElem.name;
 }

rule State2Place
 transform mElem : Statemachine!State
 to elem : Petrinet!Place
 extends ModelElem2Element {
  guard : mElem.kind <> ‘initial’
  elem.hasToken := mElem.isActive;
 }
```

Target Model produced by ETL



s1:Element
name = “s1“

t1:Element
name = “t1“

s2:Element
name = “s2“

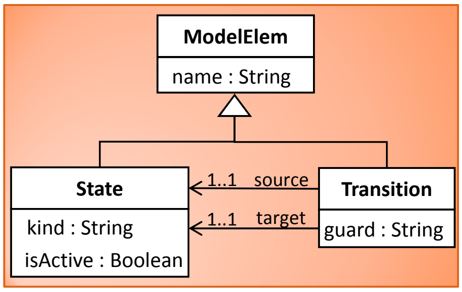
s2:Place
name = “s2“
hasToken = false

- **Results/Findings** (according to goals)
  - (1) ETL supports type substitutability, i.e., indirect instances are transformed by a rule, if the `@greedy` annotation is added; nevertheless, the interpretation is different than in ATL, since the superrule matches all direct and indirect instances irrespective whether more specific rules match them too
    This is inferred from the fact that four instances result
  - (2) In ETL the assignments of the superrule are inherited;
    This is inferred from the fact that instance `s2` has a corresponding name
  - (3) In ETL elements are never re-matched, since the elements are matched by the more general rule anyway

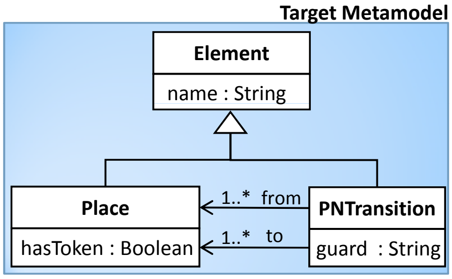
Example 2

- **Description**
  - In this example `ModelElems` should be transformed into `Elements`, but only if they exhibit a certain `name` (i.e., `not null`)
  - Furthermore, `States` should be transformed into `Places`, but again only if the `kind` is unequal „`initial`“; thereby this rule should again inherit from the base rule in order to reuse the `name` assignment;
- **Goal(s) of Evaluation**
  (1) Check how several conditions are interpreted in an inheritance hierarchy of rules; thereby it should be found out, whether conditions are inherited as well (i.e., in order to fulfill a certain condition, also the conditions of all superrules must be fulfilled)

Source Metamodel

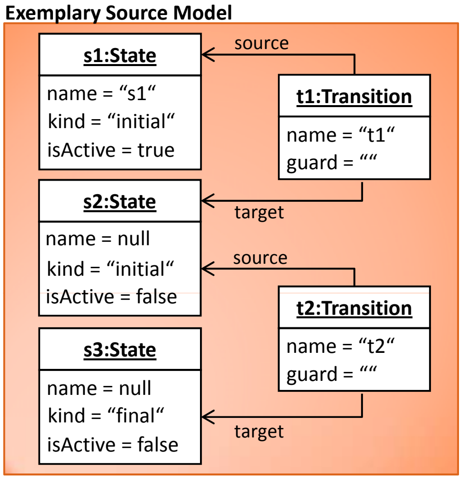


Target Metamodel

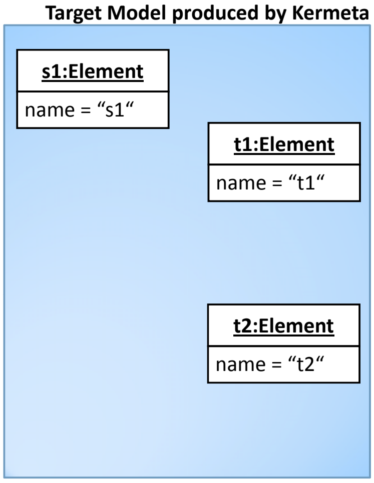


Example 2 – Kermeta (1/3)

Exemplary Source Model



Target Model produced by Kermeta



- **Results/Findings** (according to goals)
  (1) Kermeta allows to achieve the goal that conditions are interpreted as composing; nevertheless, this is again solely influenced by the programmer who has full control

EXAMPLE 2 – KERMETA (2/3)

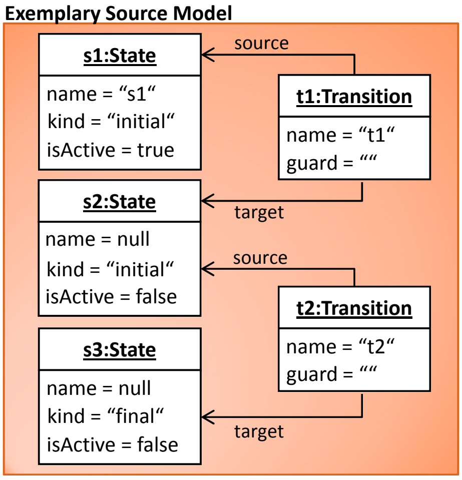
```
//transformation code for Statemachine2PetriNet
class Statemachine2PetriNet{
  operation conditionFulFilled(s : Statemachine) : kermeta::standard::Boolean is do
    result := true
  end

  operation assignments(s : Statemachine, p : PetriNet) is do
  end

  operation referenceAssignments(s : Statemachine, p : PetriNet, trace: Trace<Object, Object>) is do
    s.elements.each{ e |
      if trace.getTargetElem(e) != void then
        p.elements.add(trace.getTargetElem(e).asType(Element))
      end
    }
  end
}

//transformation code for ModelElem2Element
class ModelElem2Element{
  operation conditionFulFilled(m : ModelElem) : kermeta::standard::Boolean is do
    result := m.name != "" and m.name != void
  end

  operation assignments(m : ModelElem, e : Element) is do
    e.name := m.name
  end

  operation referenceAssignments(m : ModelElem, e : Element, trace: Trace<Object, Object>) is do
  end
}
```

EXAMPLE 2 – KERMETA (3/3)

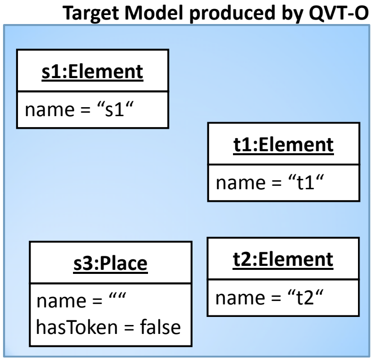
```
//transformation code for State2Place
class State2Place inherits ModelElem2Element{
  method conditionFulFilled(m : ModelElem) : kermeta::standard::Boolean is do
    result := super(m)
    result := result and (m.asType(State)).kind != "initial"
  end

  method assignments(m : ModelElem, e : Element) is do
    super(m,e)
    (e.asType(Place)).hasToken := (m.asType(State)).isActive
  end

  method referenceAssignments(m : ModelElem, e : Element, trace: Trace<Object, Object>) is do
    super(m,e,trace)
  end
}
```

EXAMPLE 2 – QVT-O (1/2)

Exemplary Source Model



Target Model produced by QVT-O



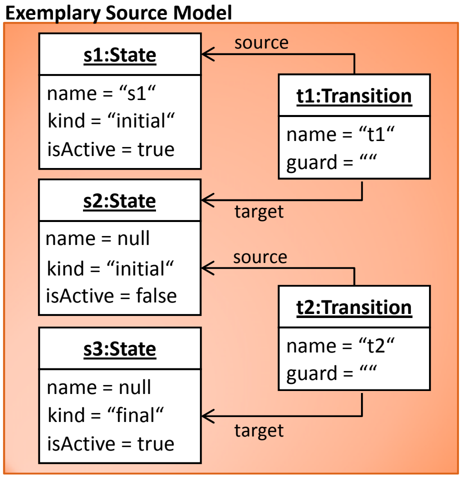
- **Results/Findings** (according to goals)
  - (1) Conditions are not inherited in QVT-O; this is inferred from the fact, that a `Place s3` has been instantiated for the `State s3;` furthermore, one might infer that the code is not executed for those model elements, which do not fulfill the condition (thus `s3` does not exhibit a `name`)

EXAMPLE 2 – QVT-O (2/2)

```
transformation testTrafo(in inModel : sm, out outModel : pn);
  main() {
    inModel.rootObjects()[Statemachine] -> map SM2Petri();
  }

  mapping Statemachine::SM2Petri() : PetriNet {
    //please note that specific rules must be called first!
    elements := self.elements[State] -> map State2Place();
    elements += self.elements[ModelElem] -> map ModelElem2Element();
  }

  mapping ModelElem::ModelElem2Element() : Element
  when{self.name != null and self.name != ''}{
    name := self.name;
  }

  mapping State::State2Place() : Place inherits ModelElem::ModelElem2Element
  when{self.kind != 'initial'}{
    result.hasToken := self.isActive;
  }
```

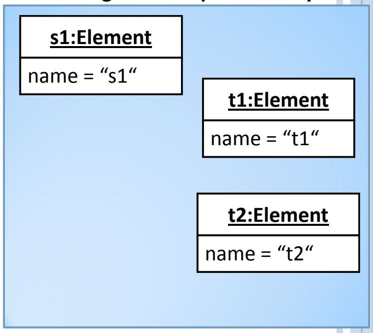
EXAMPLE 2 – TGGS (1/2)

Exemplary Source Model



Rule definitions see next slide

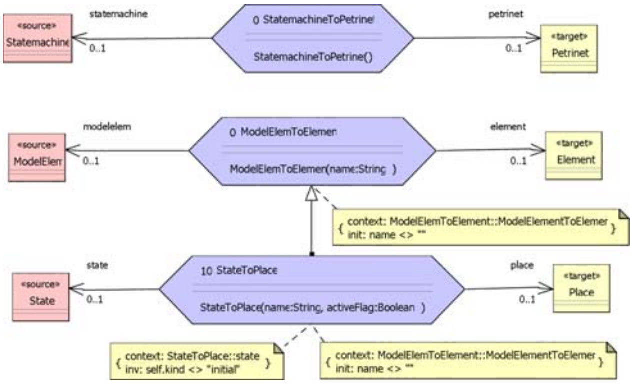
Target Model produced by TGGs



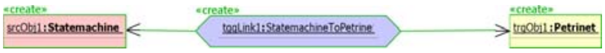
- **Results/Findings** (according to goals)
  - (1) Conditions are inherited; a subrule only matches, if its conditions and all inherited conditions are fulfilled

EXAMPLE 2 – TGGS (2/2)

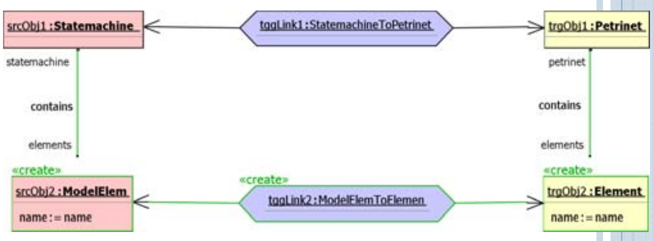
TGG-Schema (type level)



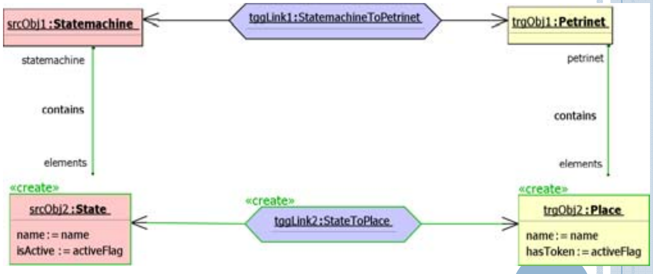
StatemachineToPetrinet(…)



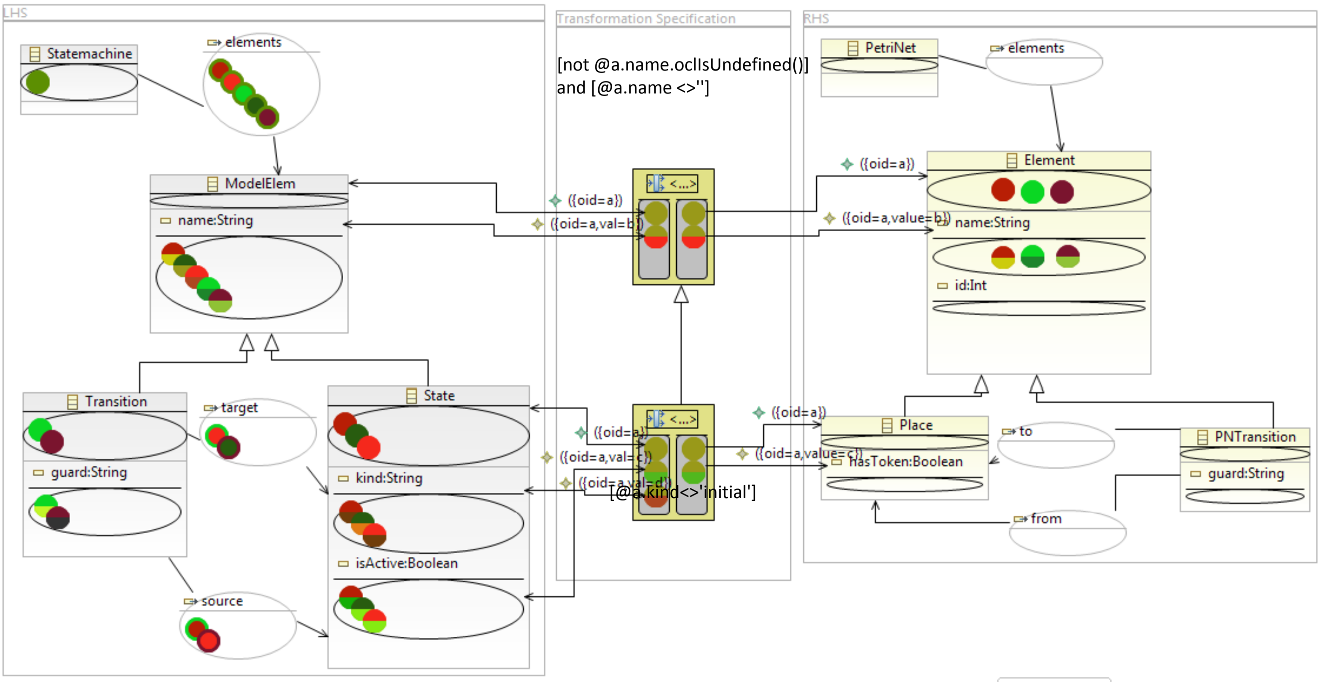
ModelElemToElemen(…)



StateToPlace(…)

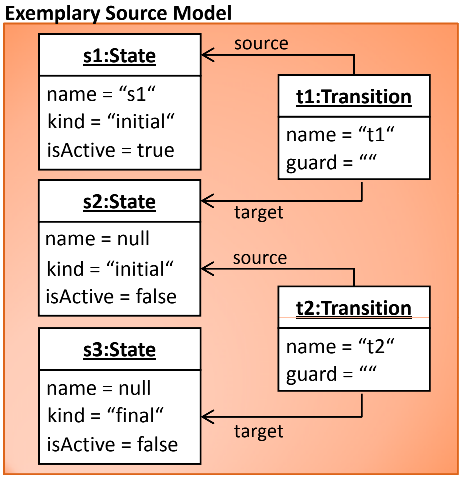


Example 2 – TNs (1/2)



Example 2 – TNs (2/2)

Exemplary Source Model



s1:State
name = "s1"
kind = "initial"
isActive = true

t1:Transition
name = "t1"
guard = ""

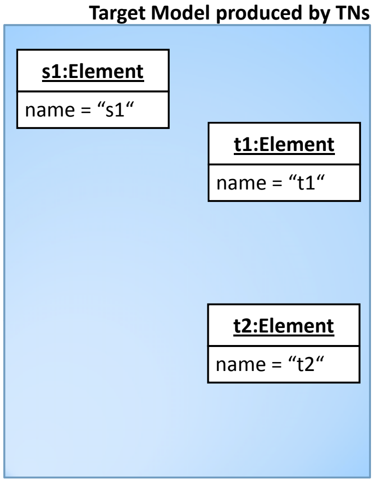
s2:State
name = null
kind = "initial"
isActive = false

t2:Transition
name = "t2"
guard = ""

s3:State
name = null
kind = "final"
isActive = false

source
target
source
target

Target Model produced by TNs



s1:Element
name = "s1"

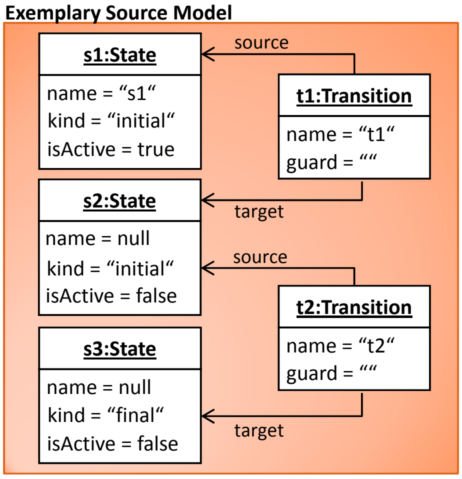
t1:Element
name = "t1"

t2:Element
name = "t2"

- **Results/Findings** (according to goals)
  (1) Conditions are inherited in TNs, i.e., a rule may transform a certain element, only, if the conditions specified by this rule and all other inherited conditions are fulfilled;
  This is inferred from the fact, that only `Element s1` results, originating from `State s1`; furthermore, the evaluation process is composing descendent-driven

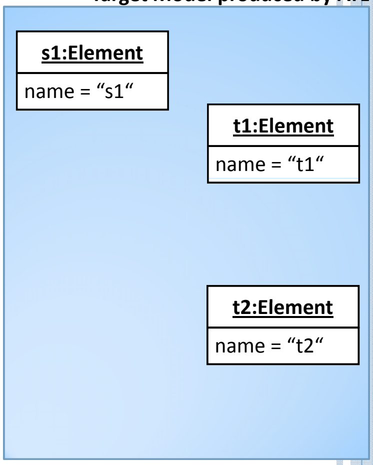
EXAMPLE 2 – ATL

Exemplary Source Model



```
rule ModelElem2Element{
  from mElem : Statemachine!ModelElem (
     mElem.name <> OclUndefined
     and mElem.name <> '')
  to elem : Petrinet!Element (
    name <- mElem.name
  )
}

rule State2Place extends ModelElem2Element {
  from mElem : Statemachine!State (
    mElem.kind <> 'initial')
  to elem : Petrinet!Place (
    hasToken <- mElem.isActive
  )
}
```

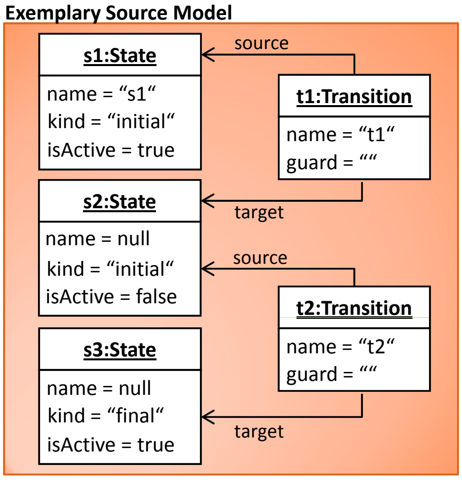
Target Model produced by ATL



- **Results/Findings** (according to goals)
  (1) Conditions are inherited in ATL, i.e., a rule may transform a certain element, only, if the conditions specified by this rule and all other inherited conditions are fulfilled;
  This is inferred from the fact, that only `Element s1` results, originating from `State s1`; moreover, short circuit evaluation takes place and the evaluation process starts at the base rule, i.e., parent-driven (as may be inferred by adding corresponding `debug()` messages)

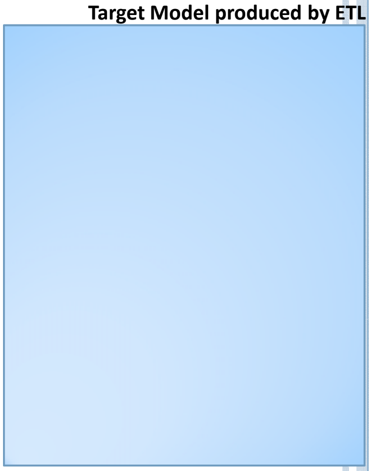
EXAMPLE 2 – ETL

Exemplary Source Model



```
rule ModelElem2Element
  transform mElem : Statemachine!ModelElem
  to elem : Petrinet!Element {
    guard : mElem.name <> null
             and mElem.name <> ''
    elem.name := mElem.name;
  }

rule State2Place
  transform mElem : Statemachine!State
  to elem : Petrinet!Place
  extends ModelElem2Element {
    guard : mElem.kind <> 'initial'
    elem.hasToken := mElem.isActive;
  }
```

Target Model produced by ETL



- **Results/Findings** (according to goals)
  (1) Conditions are inherited in ETL, i.e., a rule may only transform a certain element, only, if the conditions specified by this rule and all other inherited conditions are fulfilled;
  This is inferred from the fact that no target element has been created (`s1` and `s2` fail due to the condition on rule `State2Place`, `s3` fails due to the condition on the `ModelElem2Element` rule);
  When adding `println()` messages, one may find that the conditions of the subrule are executed first and then the conditions of the superrule (descendent-driven)