Formal Specification and Testing of Model Transformations

Manuel Wimmer, Loli Burgueño, Lars Hamann, Martin Gogolla, Antonio Vallecillo,

Universidad de Málaga/Univertät Bremen

http://www.lcc.uma.es/~av

Universität Bremen

ma

UNIVERSIDAD DE MÁLAGA

Introduction

Progress Bar

MDE

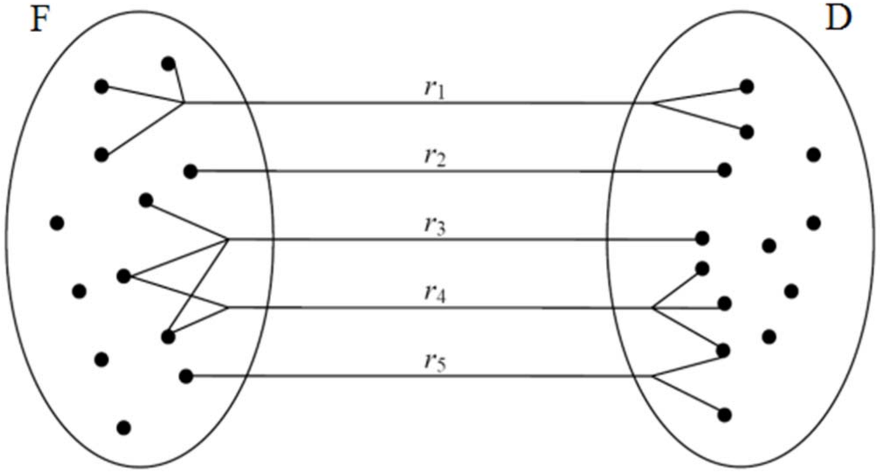
- MDE is about formulating SE activities in terms of **Models** and **Model Transformations** between them
- Models describe different views of the (same) system, at different levels of abstraction
  - Structural models, analysis models, behavioural models, run-time models, …
- Model Transformations describe the relationships between these views
  - Refinement relations, development relations, abstraction relations, mapping relations, …

Model Transformation

A **model transformation** is

(1) *The specification of the relationship between one set of source models and one set of output models*

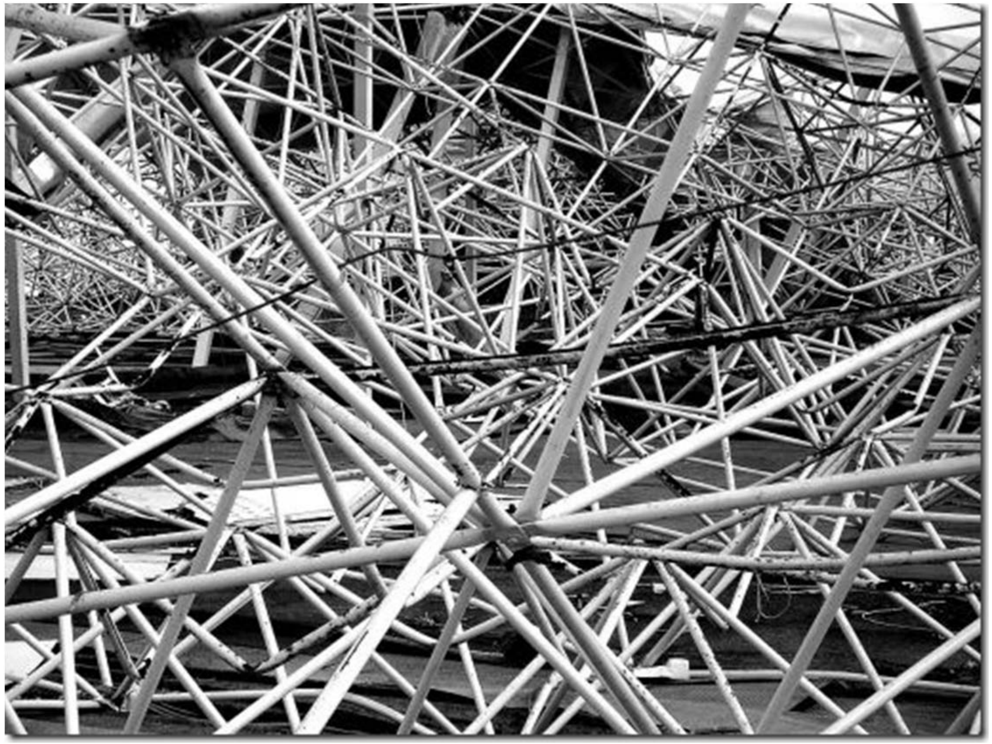
(2) *The process that generates such relationship*



- Krzysztof Czarnecki, Simon Helsen: "Feature-based survey of model transformation approaches". IBM Systems Journal 45(3): 621-646 (2006)
- Davide di Ruscio, Romina Eramo, Alfonso Pierantonio: "Model Transformations". Proc. of SMF'12, LNCS 7320, 91-136, 2012.

Model Transformations

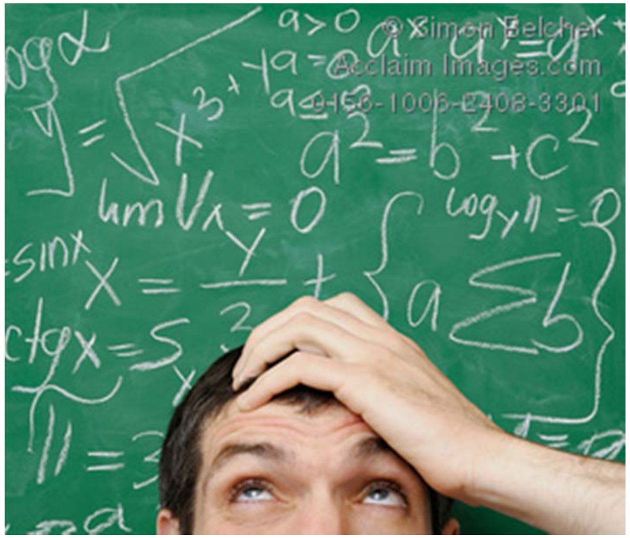
Although specified at a very high level of abstraction, **model transformations are becoming very complex** as the complexity of the relations they are able to describe grows…



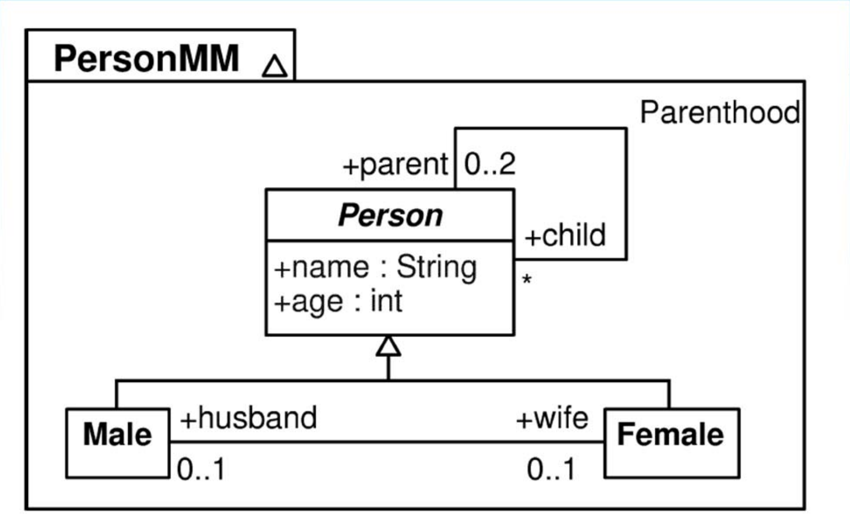
Model Transformation Challenges

- **Chains** of Transformations
  - Consider the MDA scenario: CIM -> PIM -> PSM -> Code
- **Higher Order** Transformations
  - Transformations may produce transformations that may produce transformations …
- **Underspecified** Metamodels
  - Consider the UML metamodel: many optional features, …
- **Complexity** of Input Models
  - Large graphs, combinatorial explosion how to combine model elements, …
- **No** model transformation **specifications** exist
  - Is it possible to reuse an existing model transformation for a given scenario?

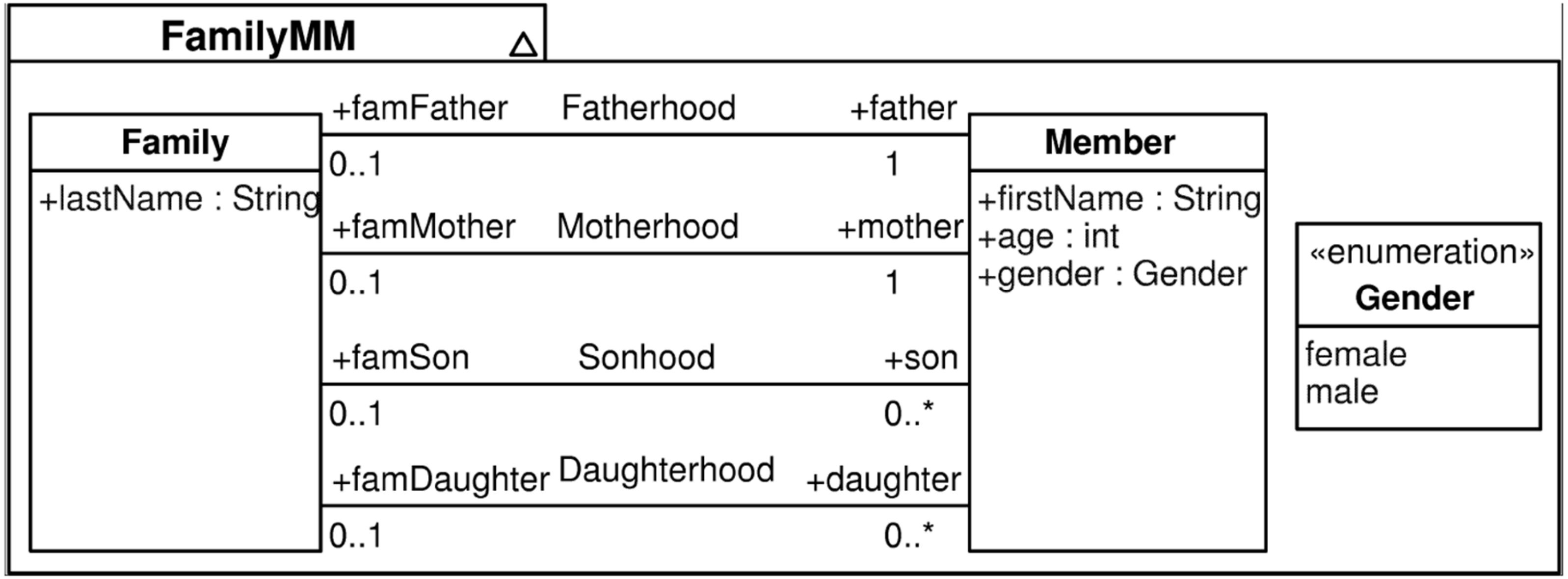
Even very simple transformations may not be that simple!



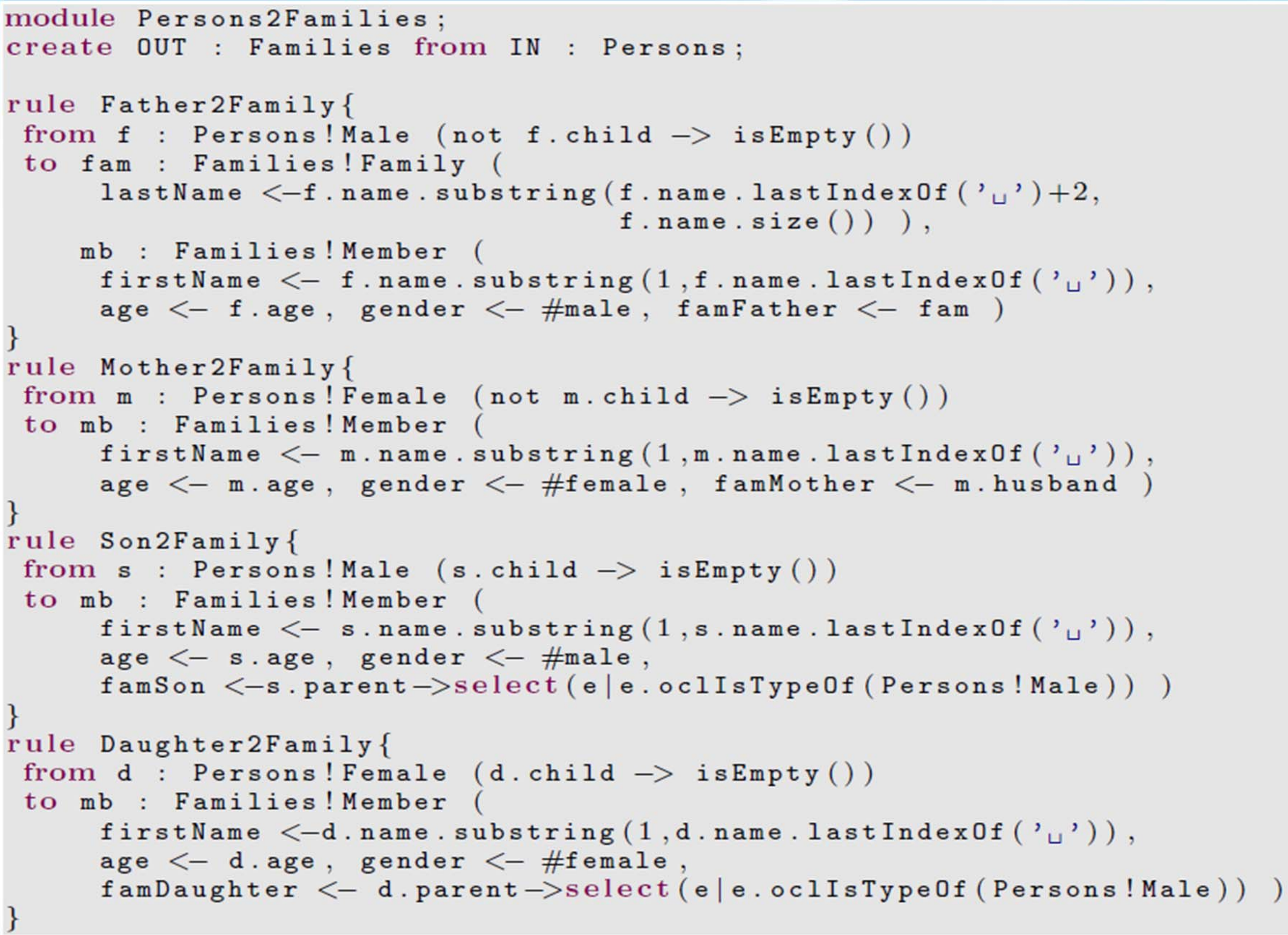
Example 1



Persons to Families



Persons2Family Transformation in ATL

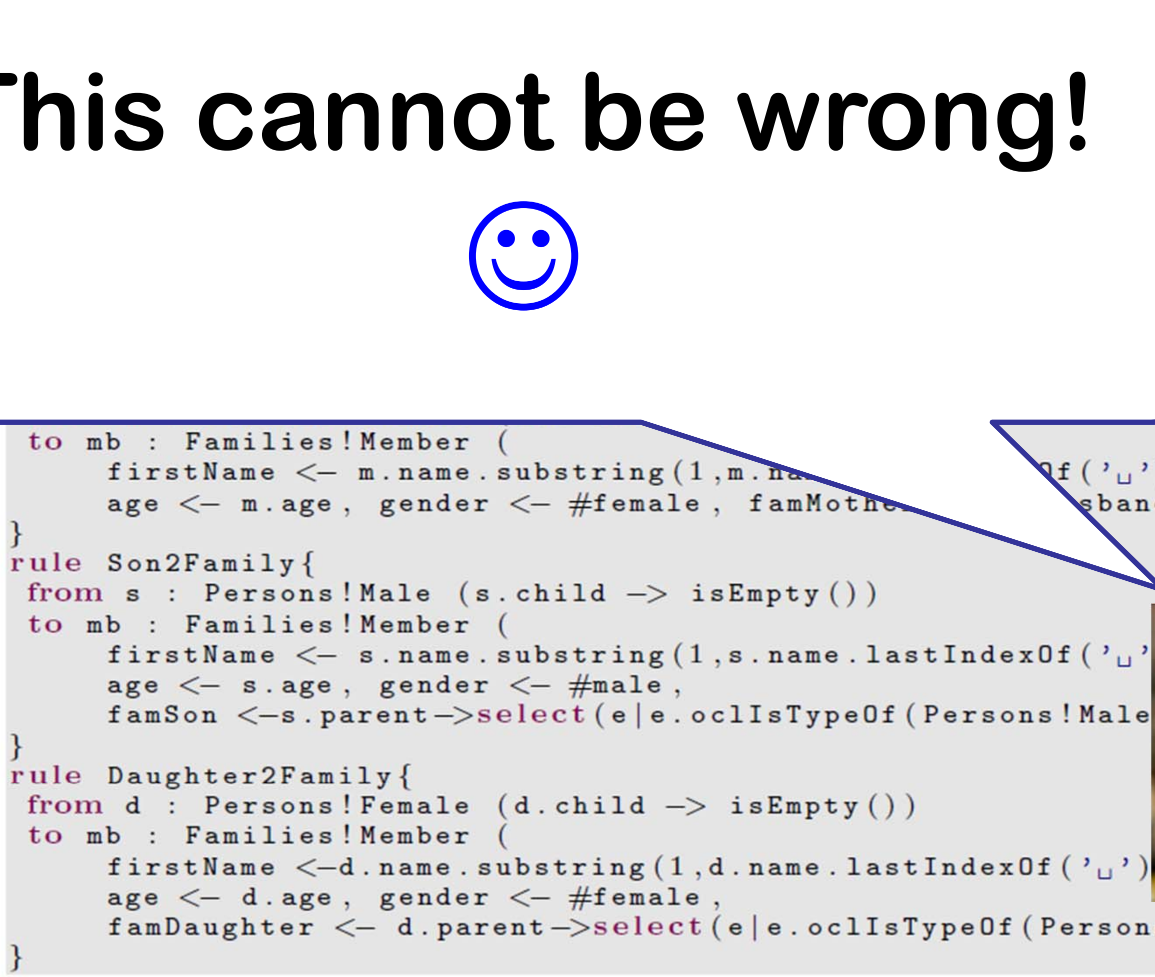


```
module Persons2Families;
create OUT : Families from IN : Persons;

rule Father2Family{
 from f : Persons!Male (not f.child -> isEmpty())
 to fam : Families!Family (
     lastName <-f.name.substring(f.name.lastIndexOf('␣')+2,
                                 f.name.size()) ),
    mb : Families!Member (
     firstName <- f.name.substring(1,f.name.lastIndexOf('␣')),
     age <- f.age, gender <- #male, famFather <- fam )
}
rule Mother2Family{
 from m : Persons!Female (not m.child -> isEmpty())
 to mb : Families!Member (
     firstName <- m.name.substring(1,m.name.lastIndexOf('␣')),
     age <- m.age, gender <- #female, famMother <- m.husband )
}
rule Son2Family{
 from s : Persons!Male (s.child -> isEmpty())
 to mb : Families!Member (
     firstName <- s.name.substring(1,s.name.lastIndexOf('␣')),
     age <- s.age, gender <- #male,
     famSon <-s.parent->select(e|e.oclIsTypeOf(Persons!Male)) )
}
rule Daughter2Family{
 from d : Persons!Female (d.child -> isEmpty())
 to mb : Families!Member (
     firstName <-d.name.substring(1,d.name.lastIndexOf('␣')),
     age <- d.age, gender <- #female,
     famDaughter <- d.parent->select(e|e.oclIsTypeOf(Persons!Male)) )
}
```

Persons2Family Transformation in ATL

This cannot be wrong!

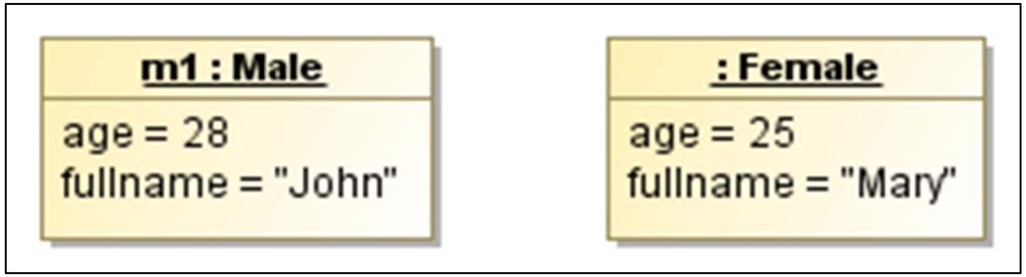
☺



```
 to mb : Families!Member (
     firstName <- m.name.substring(1,m.na              Of('␣')),
     age <- m.age, gender <- #female, famMothe          sband )
}
rule Son2Family{
 from s : Persons!Male (s.child -> isEmpty())
 to mb : Families!Member (
     firstName <- s.name.substring(1,s.name.lastIndexOf('␣'
     age <- s.age, gender <- #male,
     famSon <-s.parent->select(e|e.oclIsTypeOf(Persons!Male
}
rule Daughter2Family{
 from d : Persons!Female (d.child -> isEmpty())
 to mb : Families!Member (
     firstName <-d.name.substring(1,d.name.lastIndexOf('␣')
     age <- d.age, gender <- #female,
     famDaughter <- d.parent->select(e|e.oclIsTypeOf(Persons!Male)) )
}
```

Persons2Family Transformation in ATL

Have you tried this?

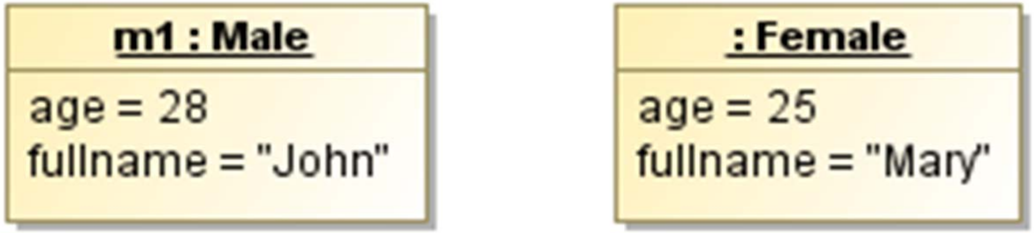


m1 : Male
age = 28
fullname = "John"

: Female
age = 25
fullname = "Mary"

```
                 r <— #female , famMother <— m.husband )
}
r
          ersons!Male (s.child —> isEmpty())
     . Families!Member (
     firstName <— s.name.substring(1,s.name.lastIndexOf('␣')),
     age <— s.age, gender <— #male,
     f  Son <—s.parent—>select(e|e.oclIsTypeOf(Persons!Male)) )

          hter2Family{
           Persons!Female (d.child —> isEmpty())
          Families!Member (
          tName <—d.name.substring(1,d.name.lastIndexOf('␣')),
          <— d.age, gender <— #female,
          aughter <— d.parent—>select(e|e.oclIsTypeOf(Persons!Male)) )
```

Not as easy as one might have thought



m1 : Male

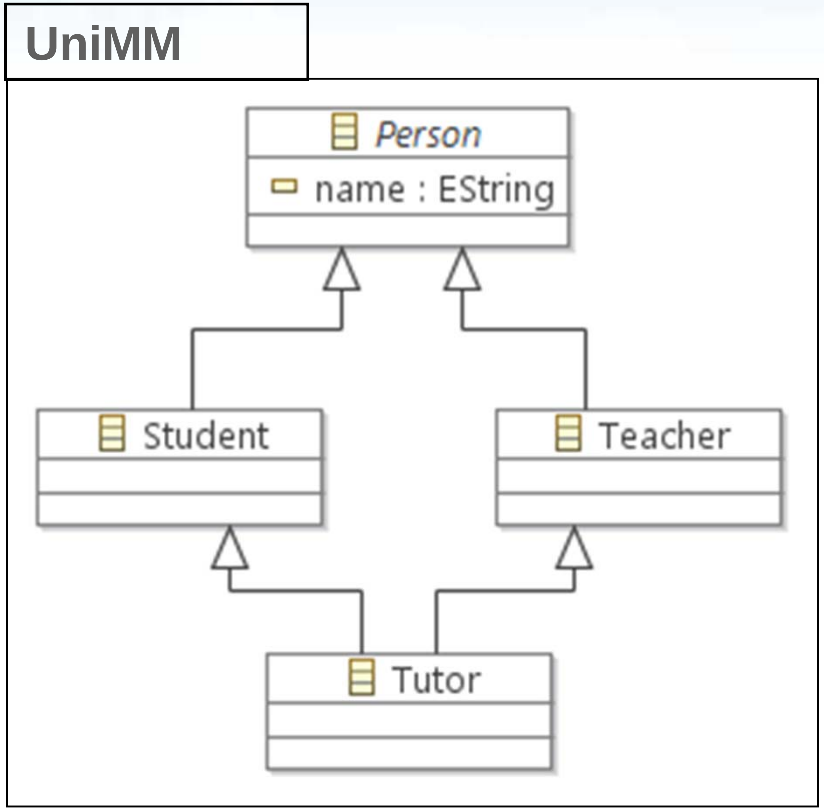
age = 28
fullname = "John"

: Female

age = 25
fullname = "Mary"

```
An internal error occurred during:
  "Launching Persons2Families".

java.lang.ClassCastException:
  org.eclipse.m2m.atl.engine.emfvm.lib.OclUndefined cannot be cast to
  org.eclipse.m2m.atl.engine.emfvm.lib.HasFields
```

Example 2



Copy Transformation

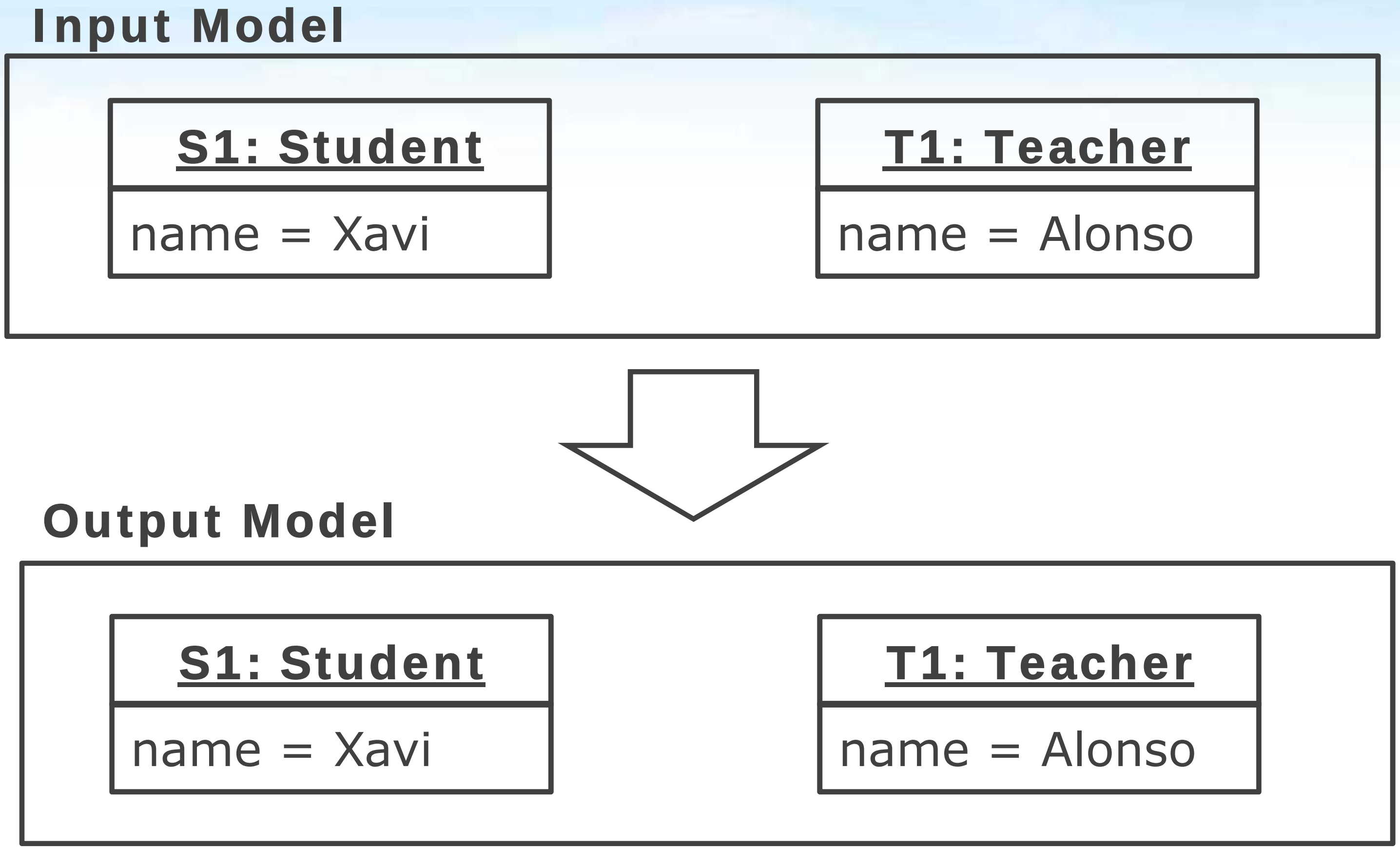
Create a copy of a model by using a model transformation

Copy Transformation in ATL

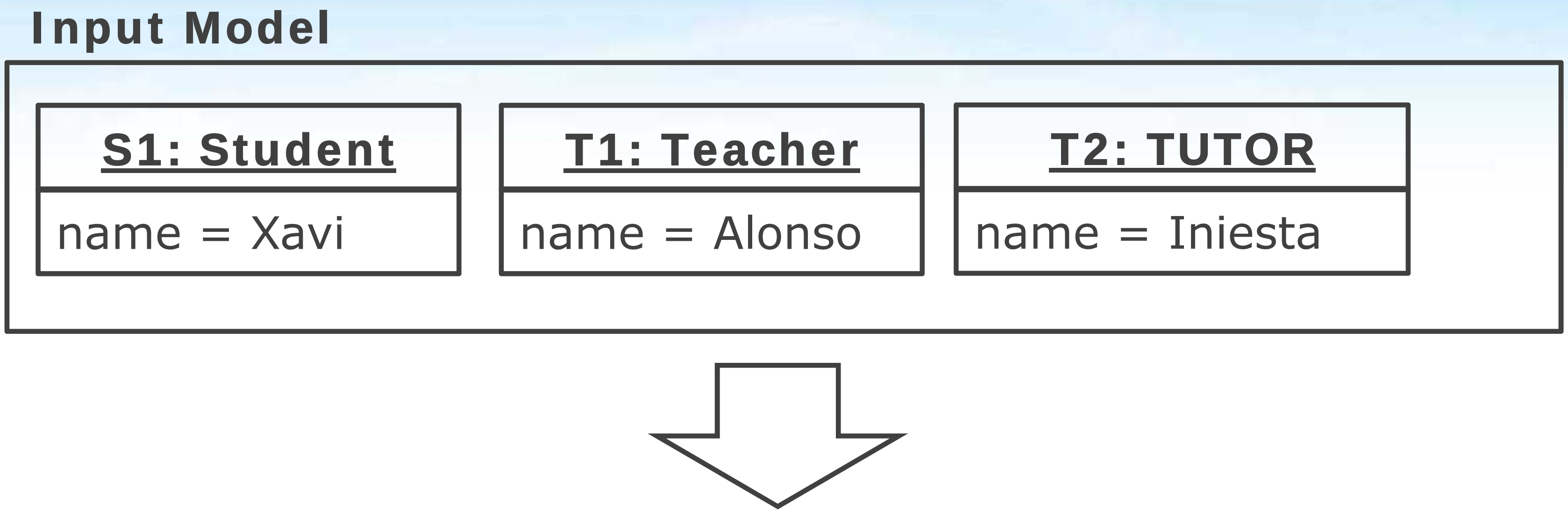
```
module Copier;
create OUT : MM2 from IN : MM1;

rule Student {
  from s1 : MM1!Student
  to  s2 : MM2!Student (
      name <- u1.name
    )
}

rule Teacher {
  from t1 : MM1!Teacher
  to t2 : MM2!Teacher (
     name <- u1.name
    )
}

rule Tutor {
  from t1 : MM1!Tutor
  to t2 : MM2!Tutor (
    name <- u1.name
  )
}
```

It is so simple - this cannot be wrong!

Let us try it out!



Not as easy as one might have thought

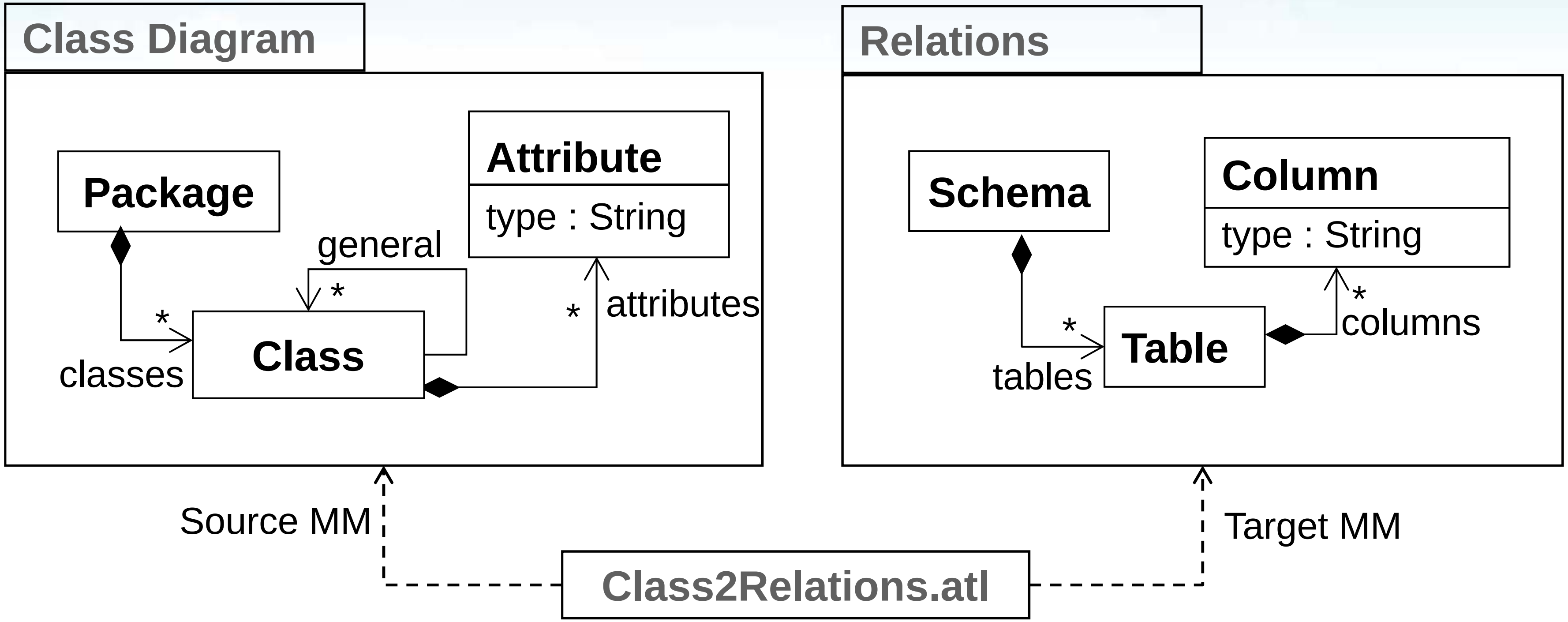

Input Model

An internal error occurred during: "Launching Copier".

org.eclipse.m2m.atl.engine.emfvm.VMException: Trying to register several rules as default for element DynamicEObjectImpl@T2 (eClass: EClassImpl@1cebdb2a (name: **Tutor**) …) : **Student and Teacher**

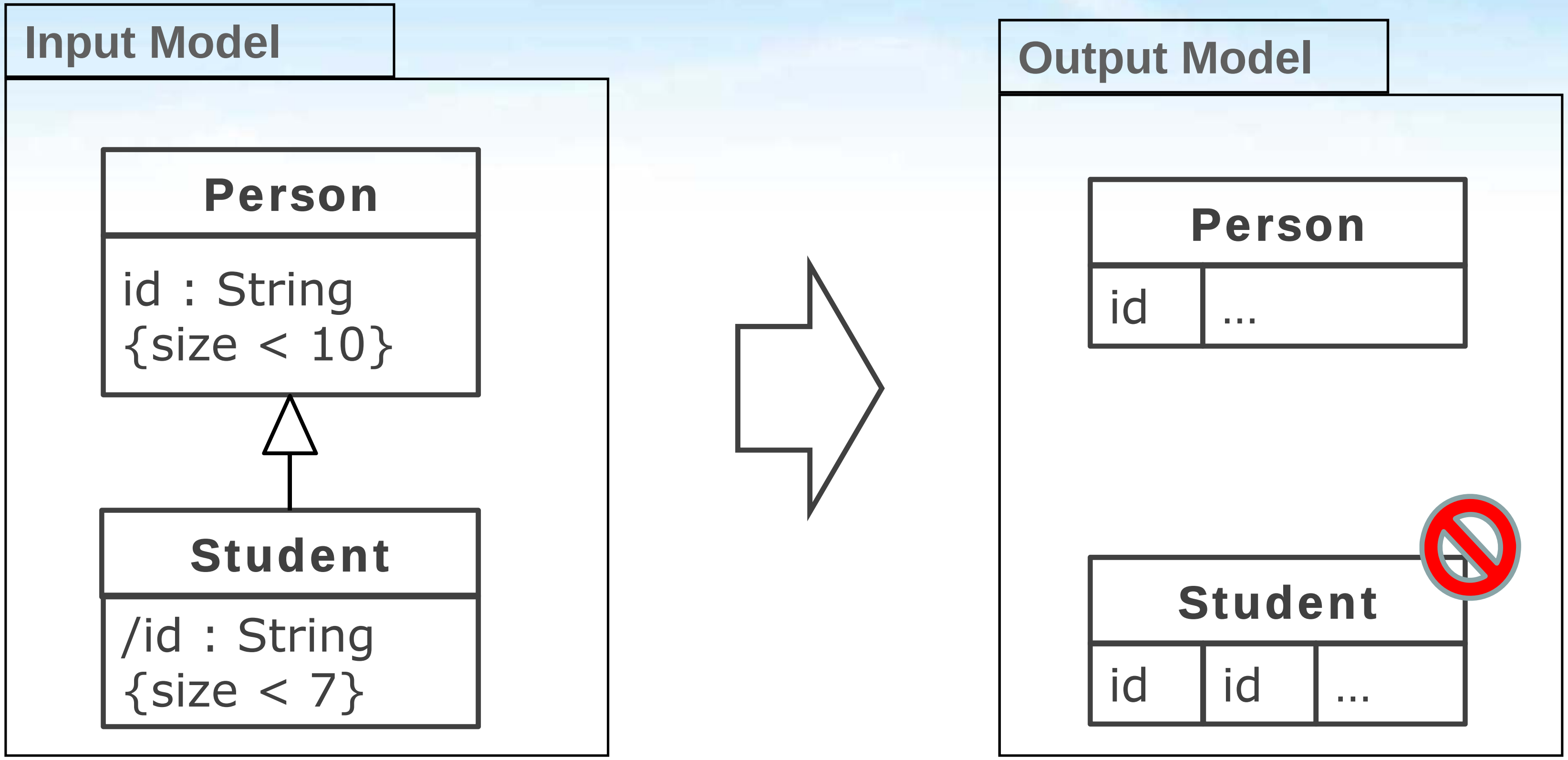
Example 3

Class Diagrams 2 Relations Transformation



- **Reuse** a transformation that implements the "*One Table per Class*" strategy. But no other documentation is available…

Not as easy as one might have thought



- It turns out that **derived attributes** are not properly treated by the transformation, i.e., invalid target models are produced

Specification and Testing of Model Transformations

Some questions

- What is testing?
- What should be tested on a model transformation?
- Should all properties to the tested treated equally?
- Which are those properties?
- Should we always aim for the best?

MT Testing Landscape

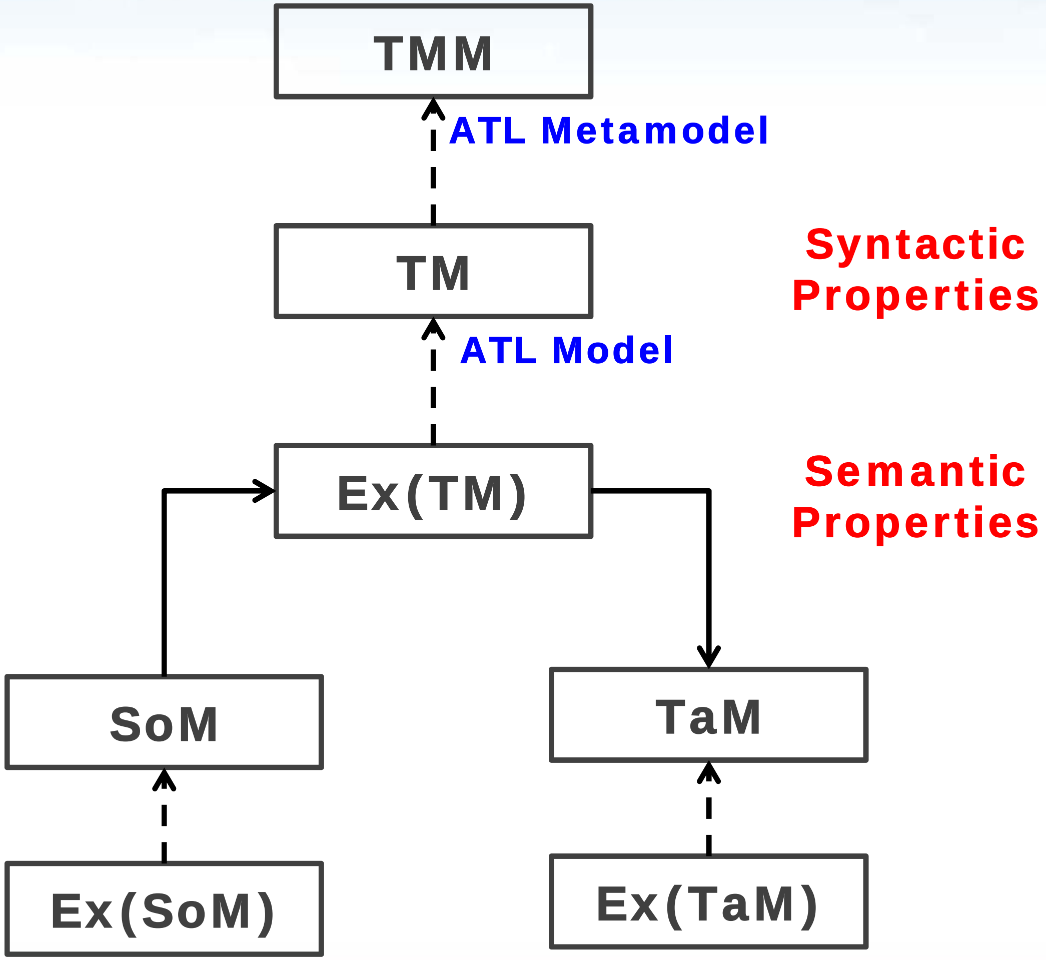
- **Plethora** of testing approaches for MTs ranging from
  - **full verification** to
  - **lightweight certification**

Two classification dimensions

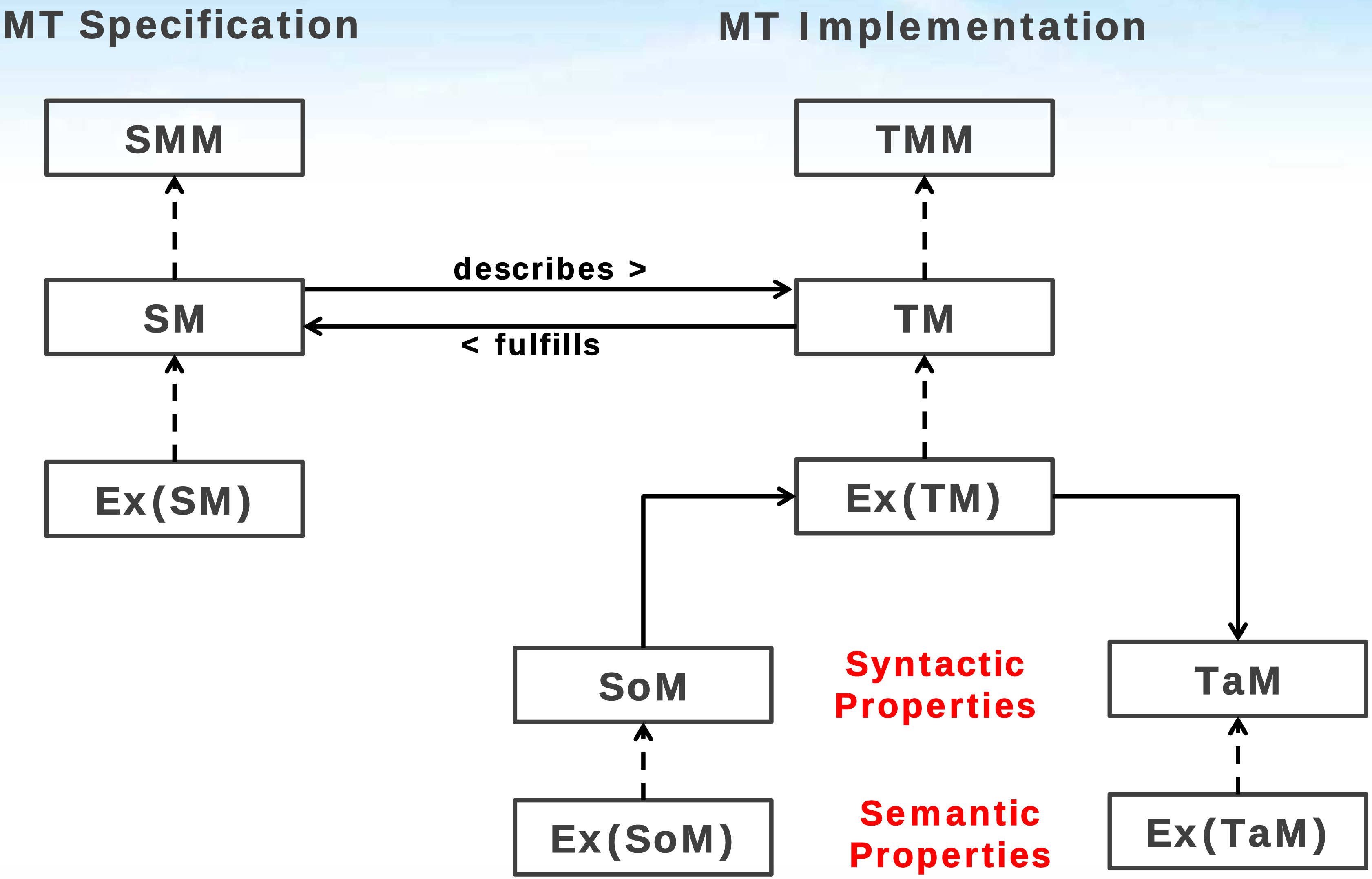
- **Level** on which they are defined
  - **General** ones are usable for all transformations
  - **Specific** ones have to be defined for each transformation
- Related to **syntax** or **semantics**
  - “Syntactic” properties are **checked** on **specifications**
    - Conformance (G), Correct output models (S)
  - “Semantic” properties are **checked** on **executions**
    - E.g., Rule confluence (G), termination (G), preservation of some properties (S), etc.

General Transformation Properties

MT Implementation



Specific Transformation Properties

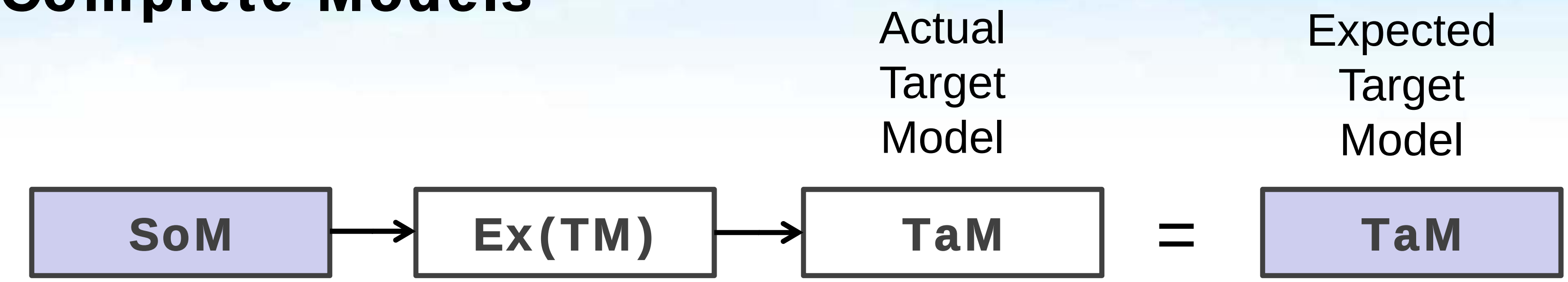


Defining Specific, Syntactical Properties

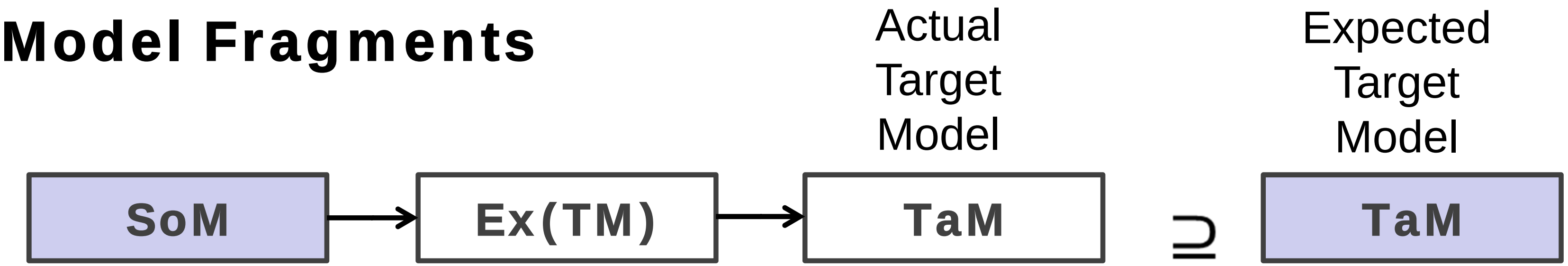
- **Model** Level
  - Complete Models
  - Model Fragments
- **Metamodel** Level
  - Graphical Constraint Languages
  - Textual Constraint Languages

Model Level Specification

- **Complete Models**



- **Model Fragments**

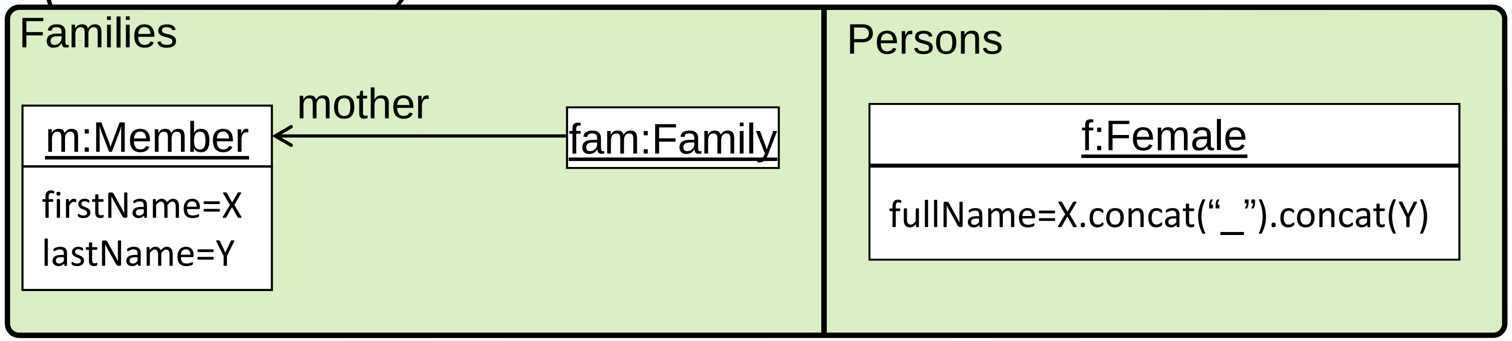


Pros: Modeling Languages are enough to specify test cases
Cons: Have to be defined for each new test source model

Metamodel Level Specification

Graphical Constraint Languages

P(Mother2Female)



Textual Constraint Language

```
context MFDS inv Src_Trg_Mother2Female:
Female.allInstances−> forAll( f |
   Family.allInstances−> exists ( fam |
      fam.mother.firstName.concat('␣').concat(fam.lastName)
      = f.fullName
   )
)
```

Specifying and Testing Model Transformations: The Tracts Approach

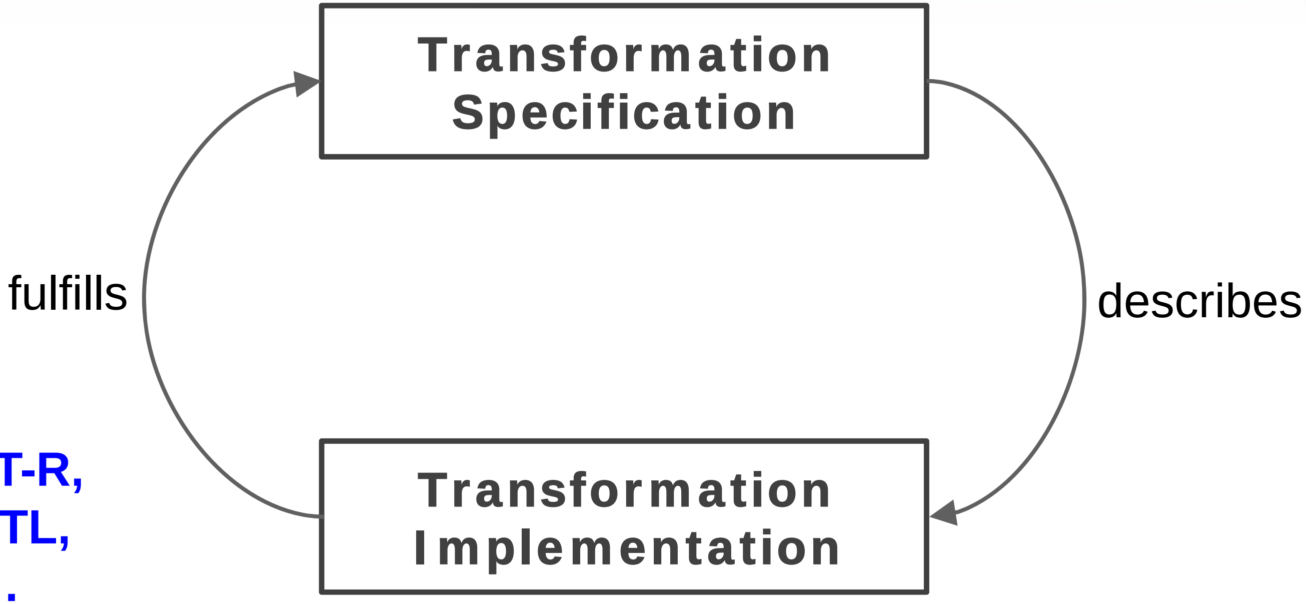
Motivation for Tracts

- In general it is very **difficult** and **expensive** (time and computational complexity-wise) to validate in full the correctness of a model transformation (even the simplest ones).

- We propose a *cost-effective* MT testing approach based on the concept of **Tract**, which is a generalization of the concept of Model Transformation Contract.

Contracts as Specifications

Specification: A document that specifies, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or component… [IEEE Standard Computer Dictionary]



Tracts

What?

vs.

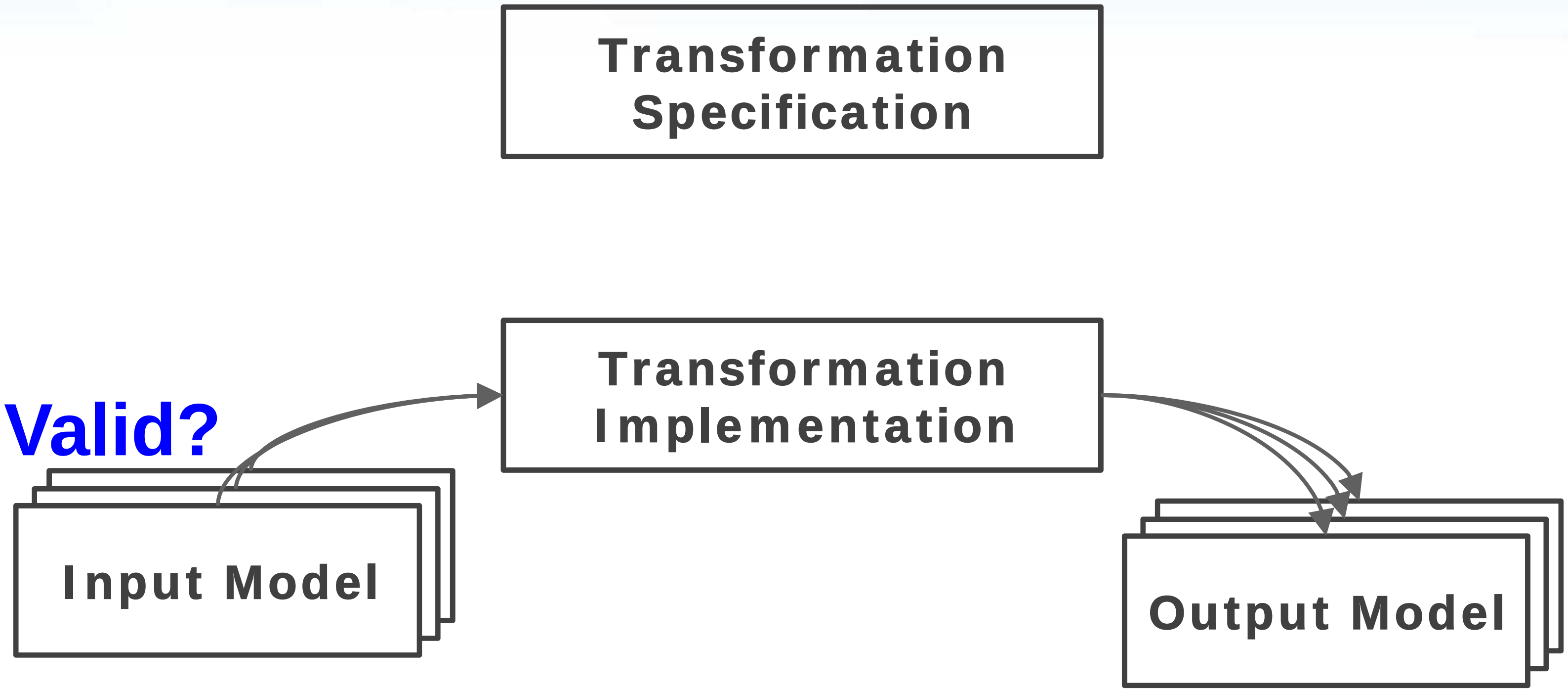
How?

ATL, ETL, QVT-R, QVT-O, RubyTL, TGG, GT, …

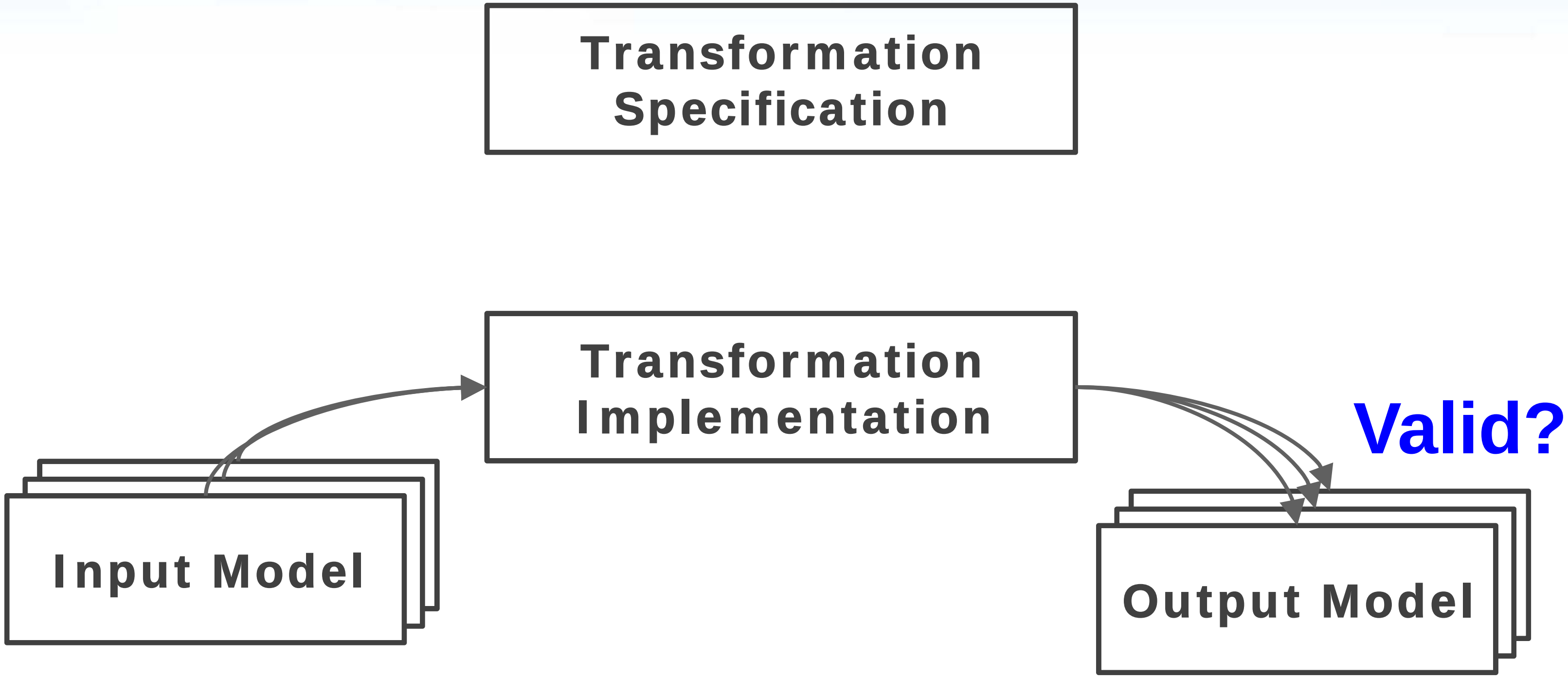
Implementation:
(1) The process of translating a design into hardware components, software components, or both.
(2) The result of the process in **(1)**

[IEEE Standard Computer Dictionary]

Contracts as Filters



Contracts as Oracles



Transformation Specification

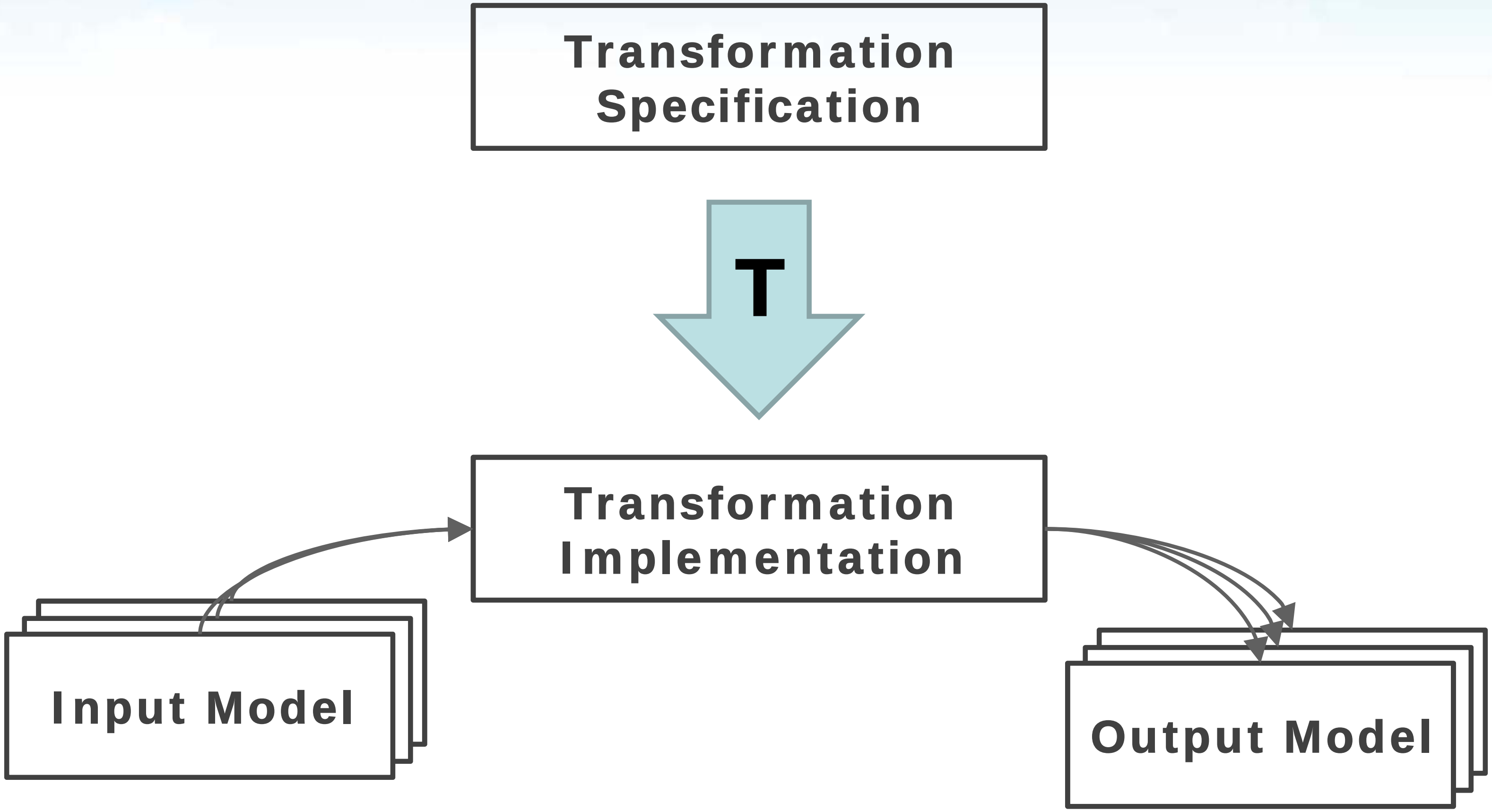
Transformation Implementation

Valid?

Input Model

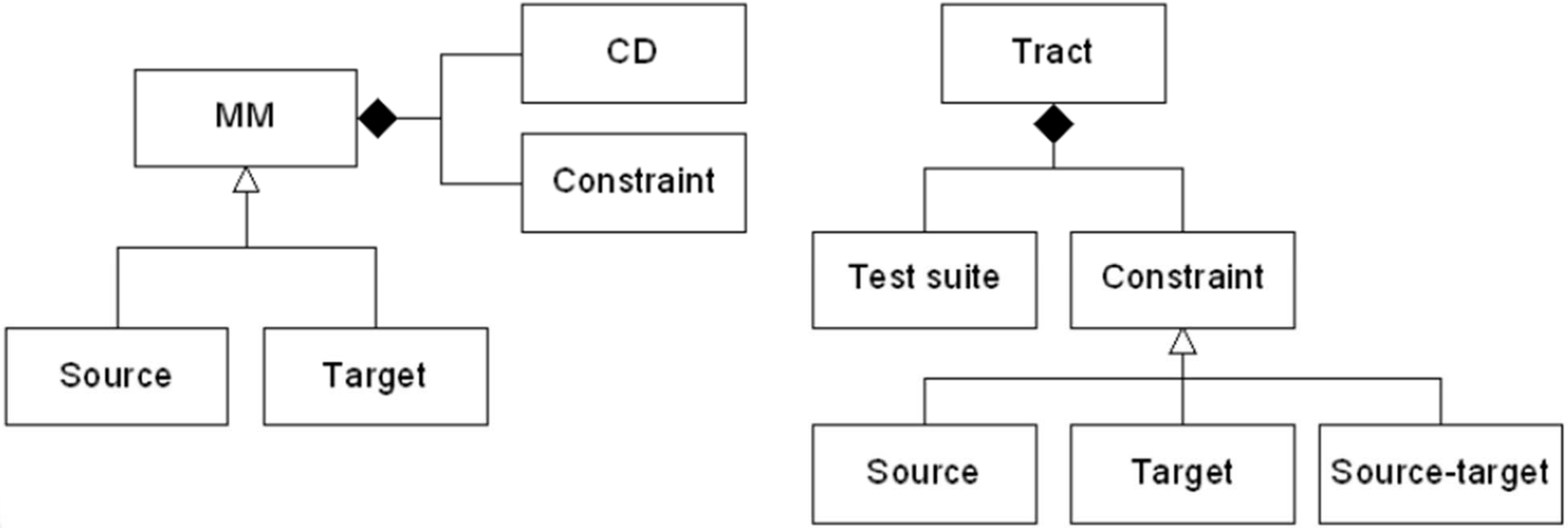
Output Model

Contracts as Implementation

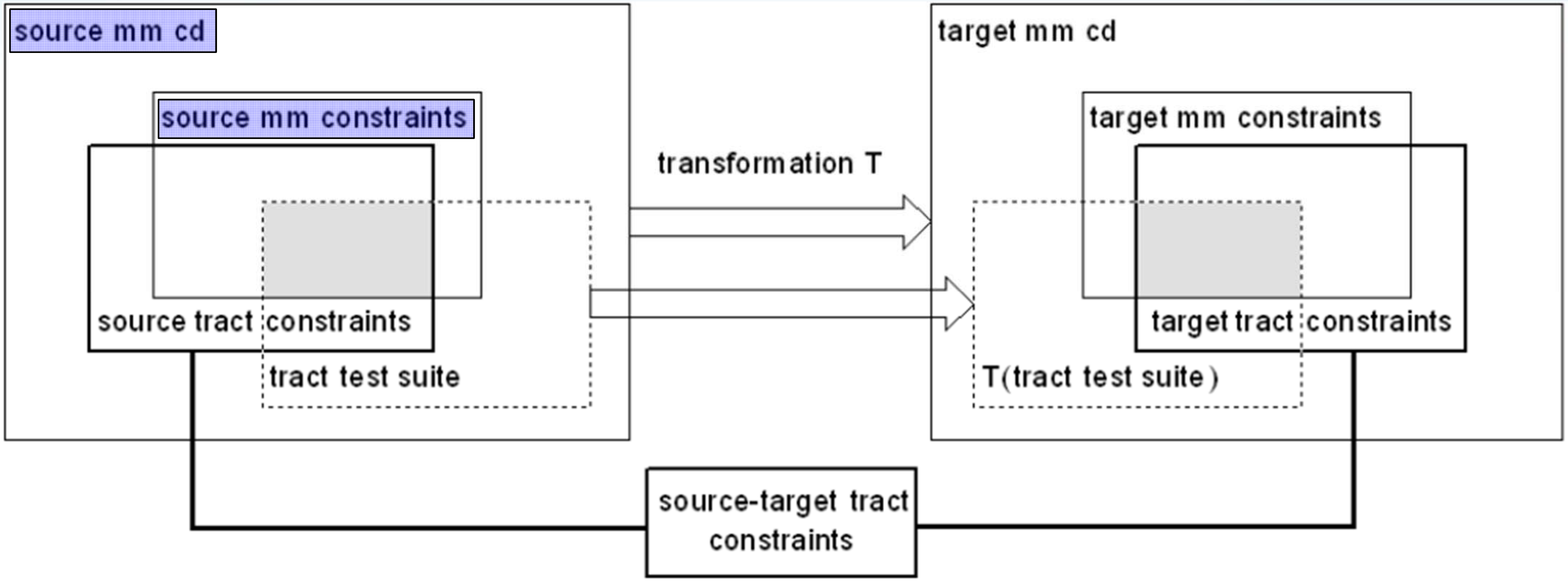


Tracts

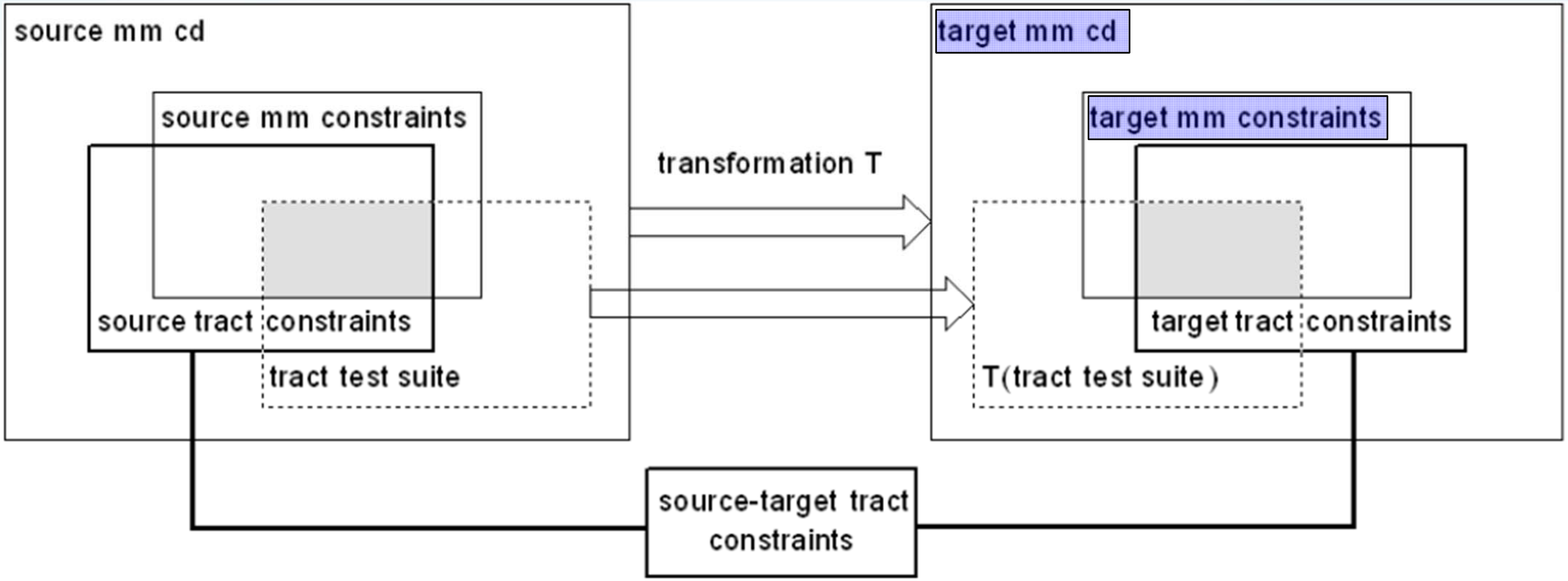
- A Tract defines
  - a set of **constraints on the source and target metamodels**,
  - a set of **source-target constraints**, and
  - a **tract test suite** (a collection of source models satisfying the source constraints)



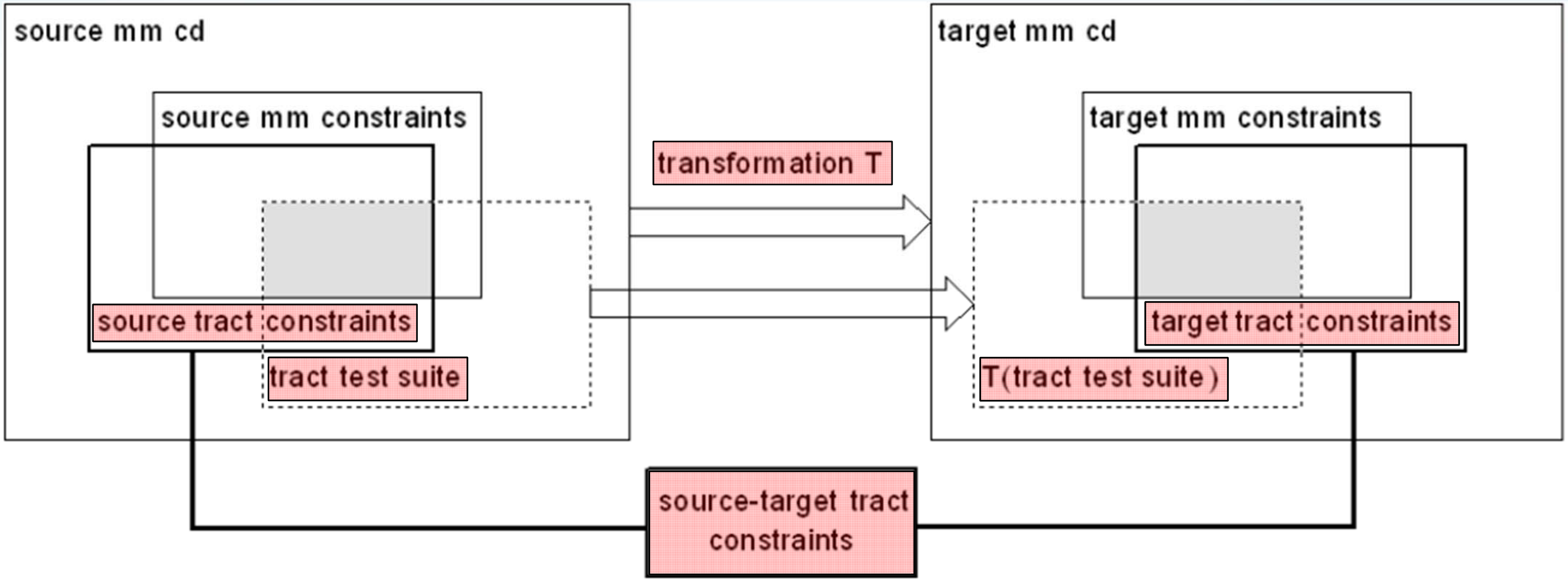
Set-Theory based View on *Tracts*



Set-Theory based View on *Tracts*



The elements of a *Tract*

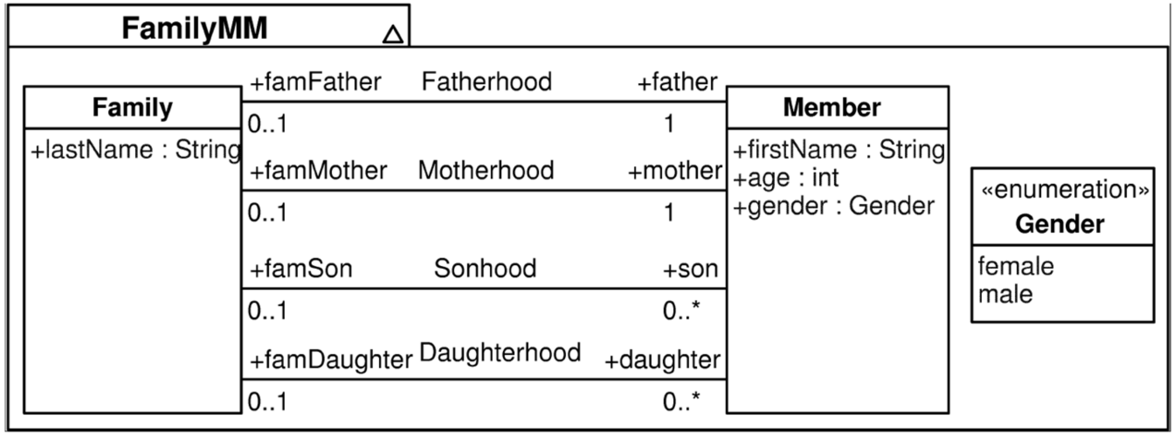


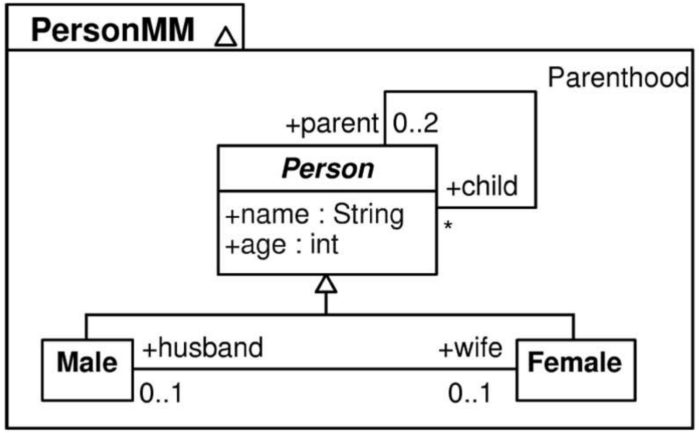
Black-box testing of MTs

- For each tract
  - Input test suite models are automatically generated using ASSL
  - Input models are transformed into output models by the transformation under test
  - The results are checked with the USE tool against the constraints defined for the transformation
- Different tracts are defined for every transformation
  - Each one defines either a *use case* or a *special condition* or a *negative test case*

Tracts for the Families 2 Persons MT

- **Source Metamodel: Family**
- **Target Metamodel: Person**

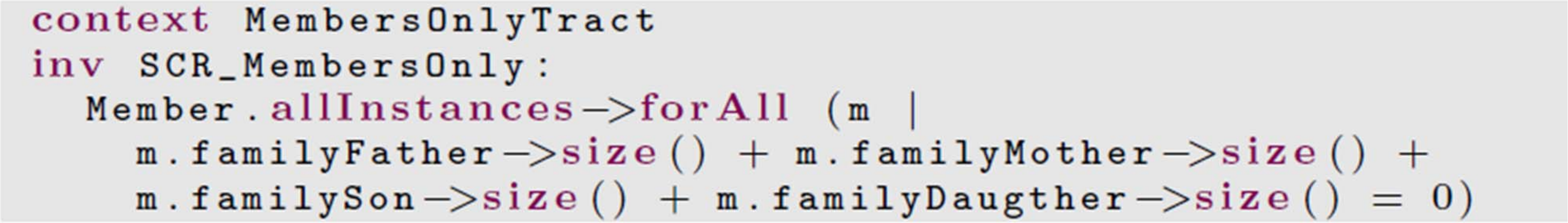




Example of *Tract*: “Members only”

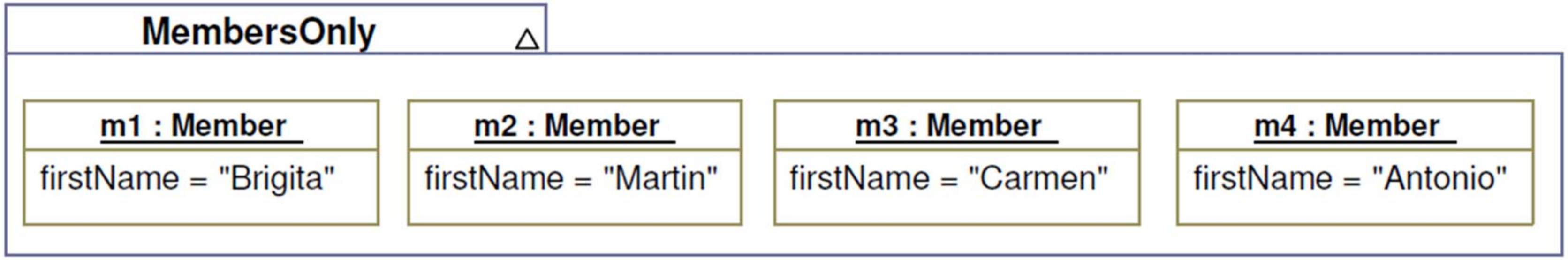
- **Tract: Members only -** interested in families consisting only of members

- Precondition



```
context MembersOnlyTract
inv SCR_MembersOnly:
  Member.allInstances->forAll (m |
    m.familyFather->size() + m.familyMother->size() +
    m.familySon->size() + m.familyDaugther->size() = 0)
```

- Test Source Model

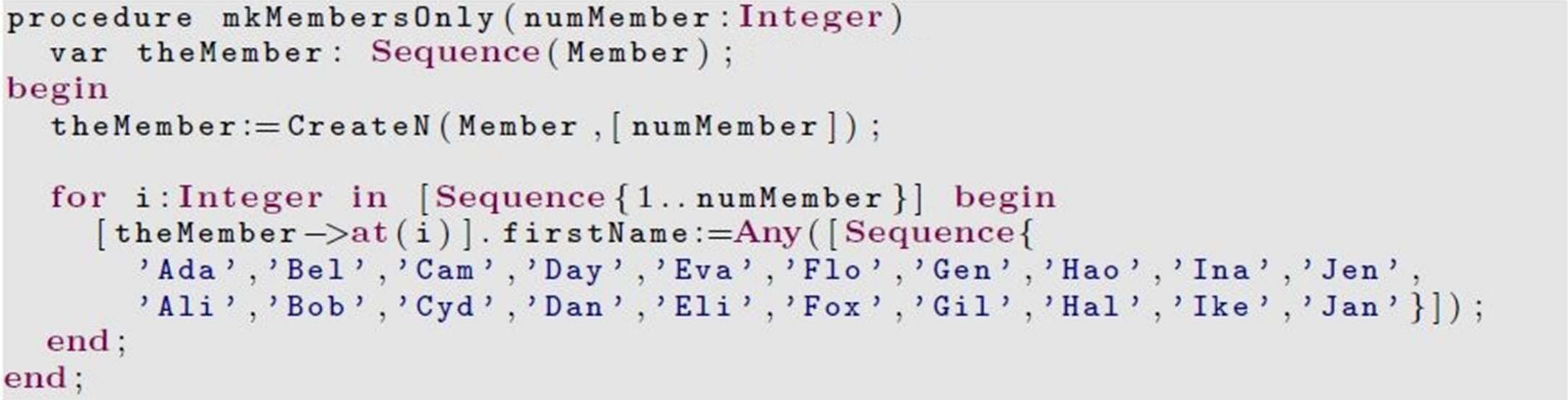


Example of *Tract*: “Members only”

- Generation of source models is done by means of ASSL (A Snapshot Sequence Language)
- ASSL allows to generate Object Diagrams for Class Diagrams
- ASSL is an imperative programming language with features for randomly choosing attribute values or association ends
- ASSL supports backtracking for finding object diagrams with particular properties

Example of *Tract*: “Members only”

- **ASSL Code for Input Model Generation**

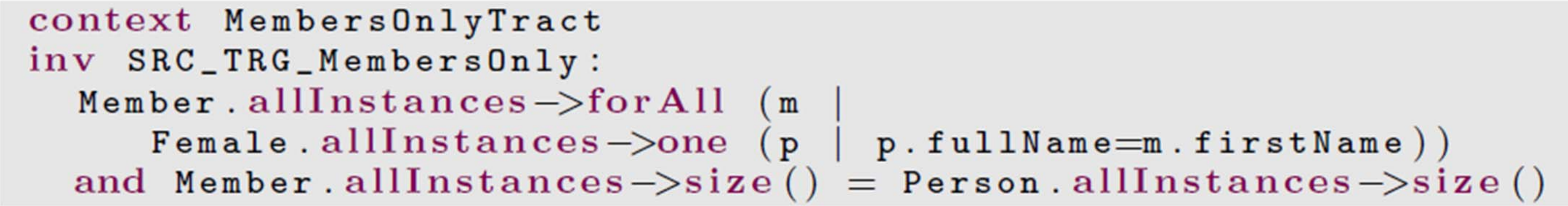


```
procedure mkMembersOnly(numMember:Integer)
  var theMember: Sequence(Member);
begin
  theMember:=CreateN(Member,[numMember]);

  for i:Integer in [Sequence{1..numMember}] begin
    [theMember->at(i)].firstName:=Any([Sequence{
      'Ada','Bel','Cam','Day','Eva','Flo','Gen','Hao','Ina','Jen',
      'Ali','Bob','Cyd','Dan','Eli','Fox','Gil','Hal','Ike','Jan'}]);
  end;
end;
```

Example of *Tract*: "Members only"

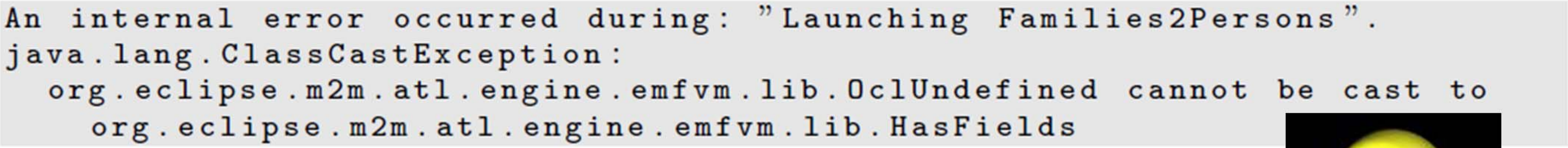
- **Tract: Members only -** interested in families consisting only of members

- Postcondition



```
context MembersOnlyTract
inv SRC_TRG_MembersOnly:
  Member.allInstances->forAll (m |
     Female.allInstances->one (p | p.fullName=m.firstName))
  and Member.allInstances->size() = Person.allInstances->size()
```

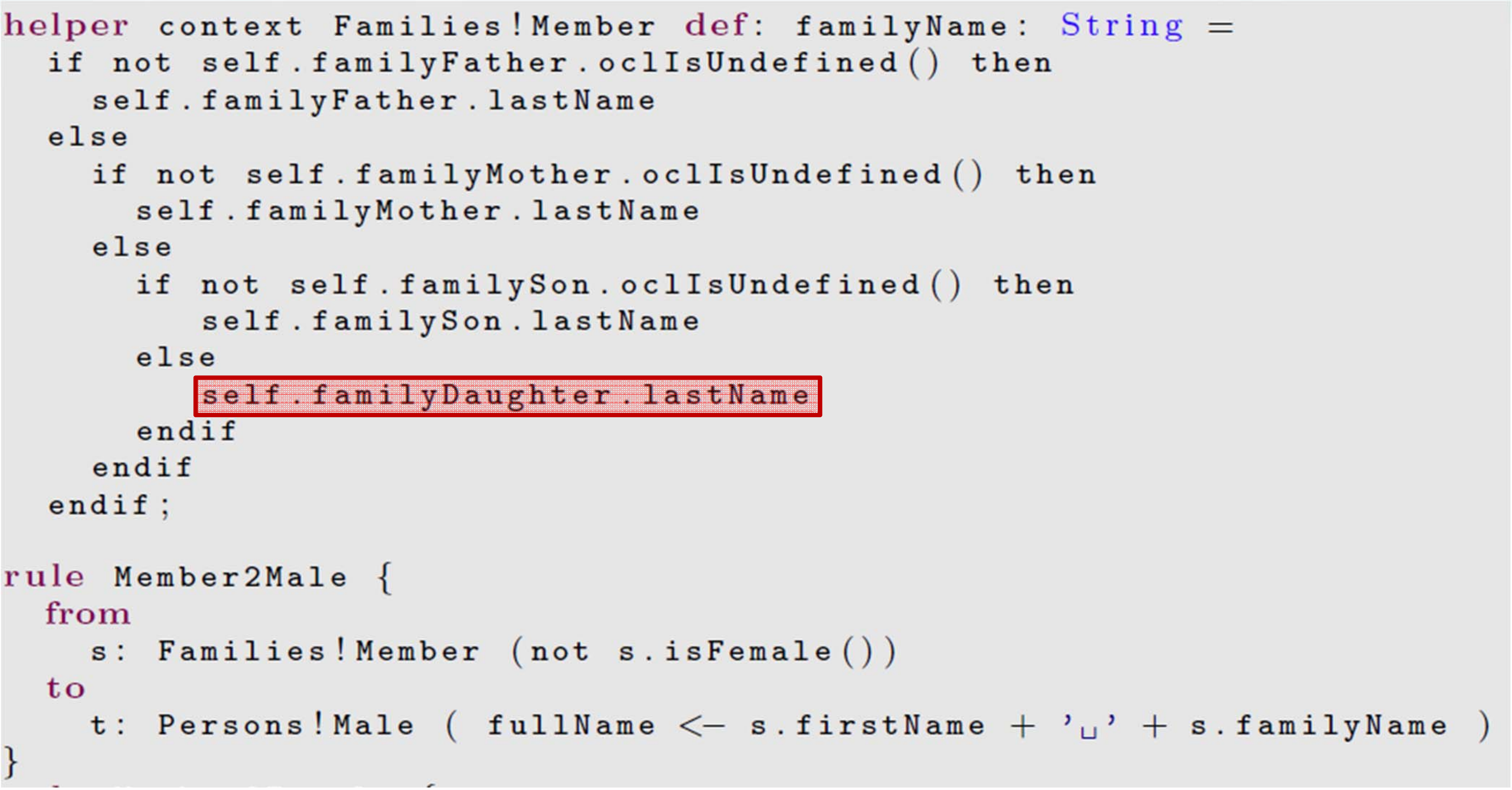
- Transformation Result



```
An internal error occurred during: "Launching Families2Persons".
java.lang.ClassCastException:
  org.eclipse.m2m.atl.engine.emfvm.lib.OclUndefined cannot be cast to
    org.eclipse.m2m.atl.engine.emfvm.lib.HasFields
```

Example of *Tract*: “Members only”

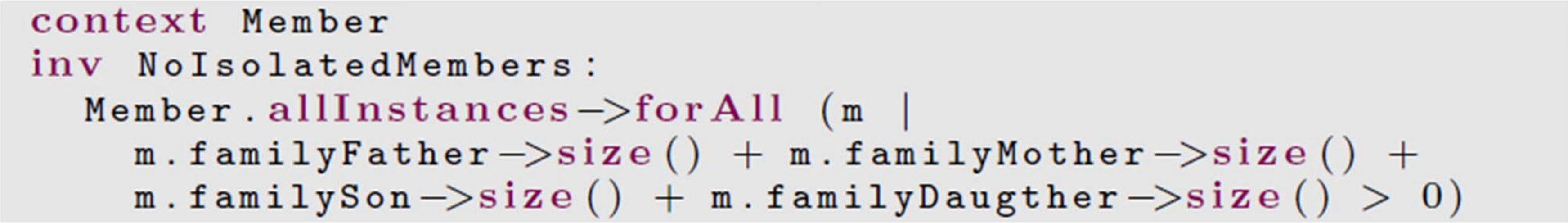
Issue in Transformation



```
helper context Families!Member def: familyName: String =
  if not self.familyFather.oclIsUndefined() then
    self.familyFather.lastName
  else
    if not self.familyMother.oclIsUndefined() then
      self.familyMother.lastName
    else
      if not self.familySon.oclIsUndefined() then
         self.familySon.lastName
      else
         self.familyDaughter.lastName
      endif
    endif
  endif;

rule Member2Male {
  from
    s: Families!Member (not s.isFemale())
  to
    t: Persons!Male ( fullName <- s.firstName + '␣' + s.familyName )
}
```

Example of *Tract*: “Members only”

- **Possible Solution**: Stronger Precondition

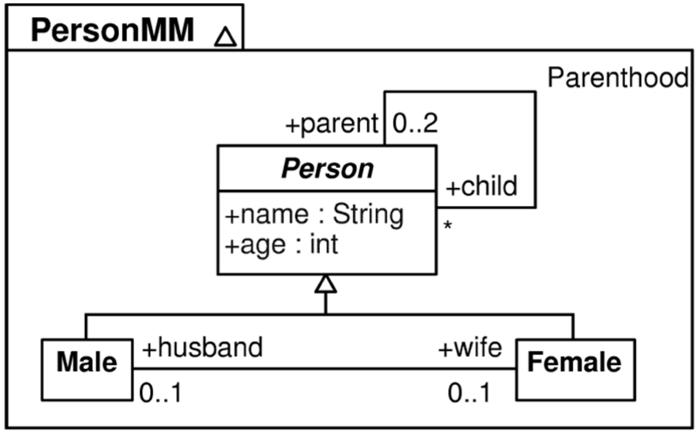


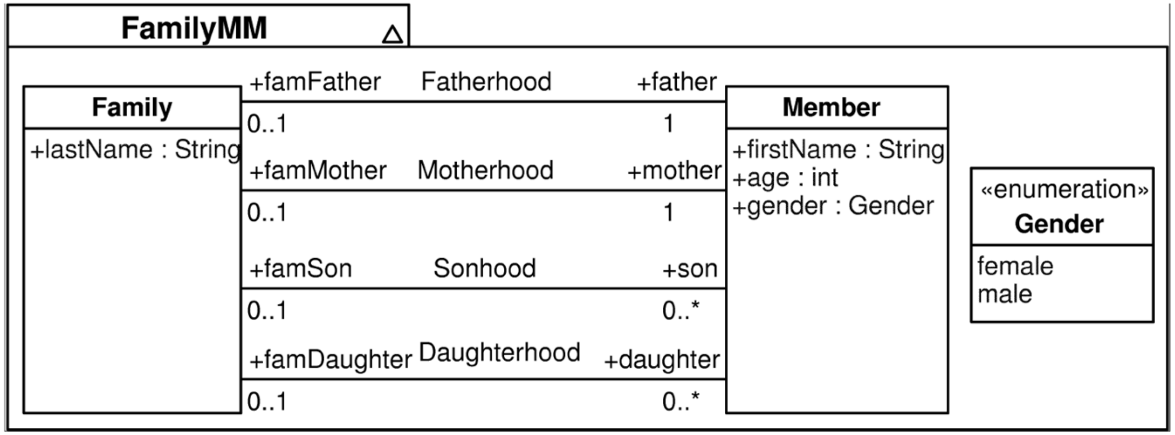
```
context Member
inv NoIsolatedMembers:
  Member.allInstances->forAll (m |
     m.familyFather->size() + m.familyMother->size() +
     m.familySon->size() + m.familyDaugther->size() > 0)
```

- **Other Solutions?**

Tracts for the Person 2 Family MT

- **Source Metamodel: Person**
- **Target Metamodel: Family**

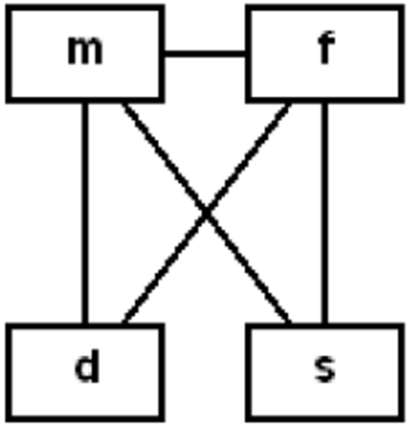




Example of *Tract*: “mfds”

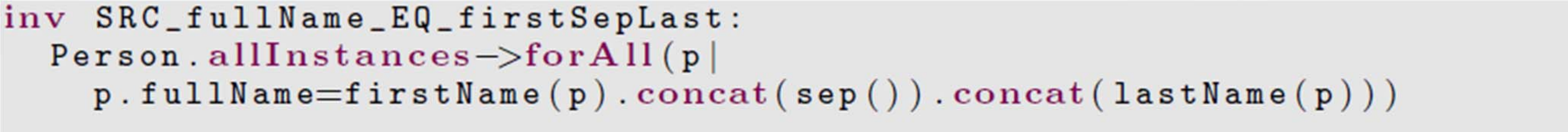
- **Tract: mfds -** interested in families consisting of one mother, father, daughter, and son

mother(m)-father(f)-daughter(d)-son(s)



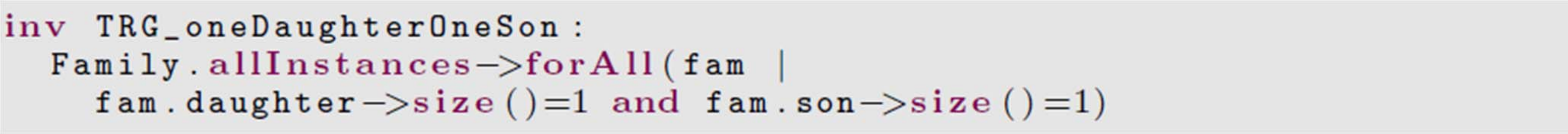
Tract constraints (1/2)

- Src_Constraints



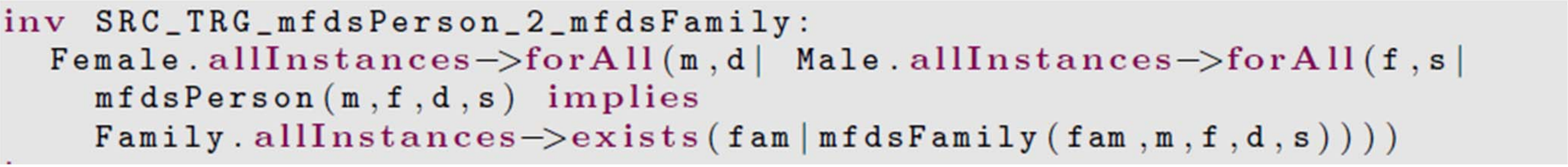
```
inv SRC_fullName_EQ_firstSepLast:
  Person.allInstances->forAll(p|
    p.fullName=firstName(p).concat(sep()).concat(lastName(p)))
```

- Trg_Constraint



```
inv TRG_oneDaughterOneSon:
  Family.allInstances->forAll(fam |
    fam.daughter->size()=1 and fam.son->size()=1)
```

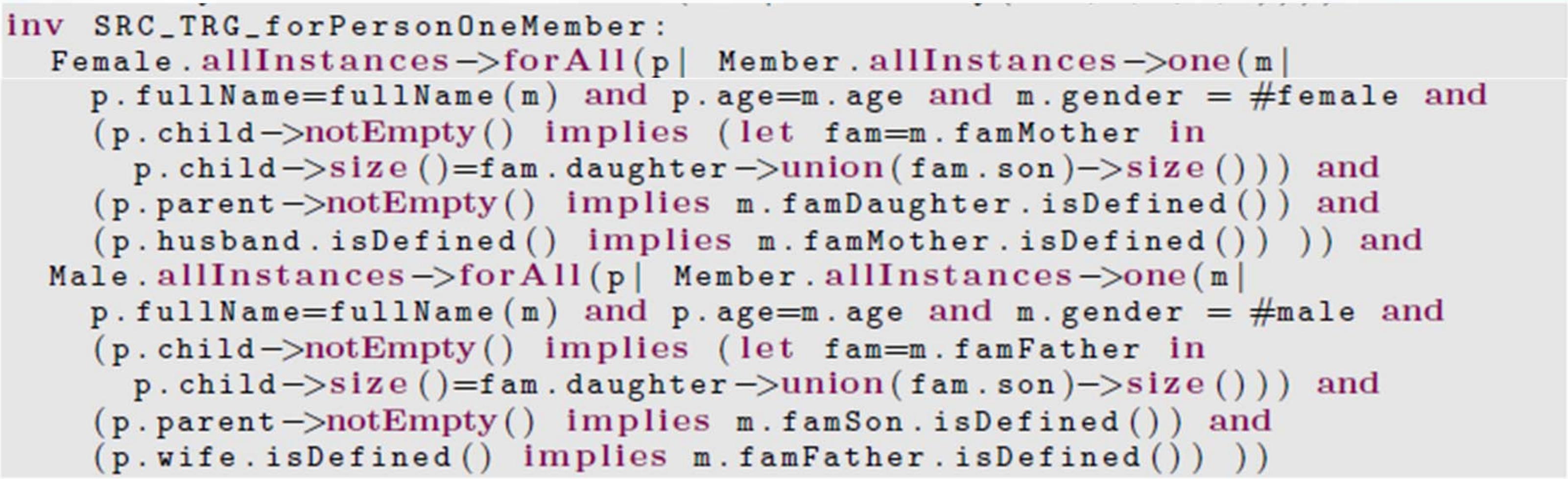
- Src_Trg_Constraint



```
inv SRC_TRG_mfdsPerson_2_mfdsFamily:
  Female.allInstances->forAll(m,d| Male.allInstances->forAll(f,s|
    mfdsPerson(m,f,d,s) implies
    Family.allInstances->exists(fam|mfdsFamily(fam,m,f,d,s))))
```

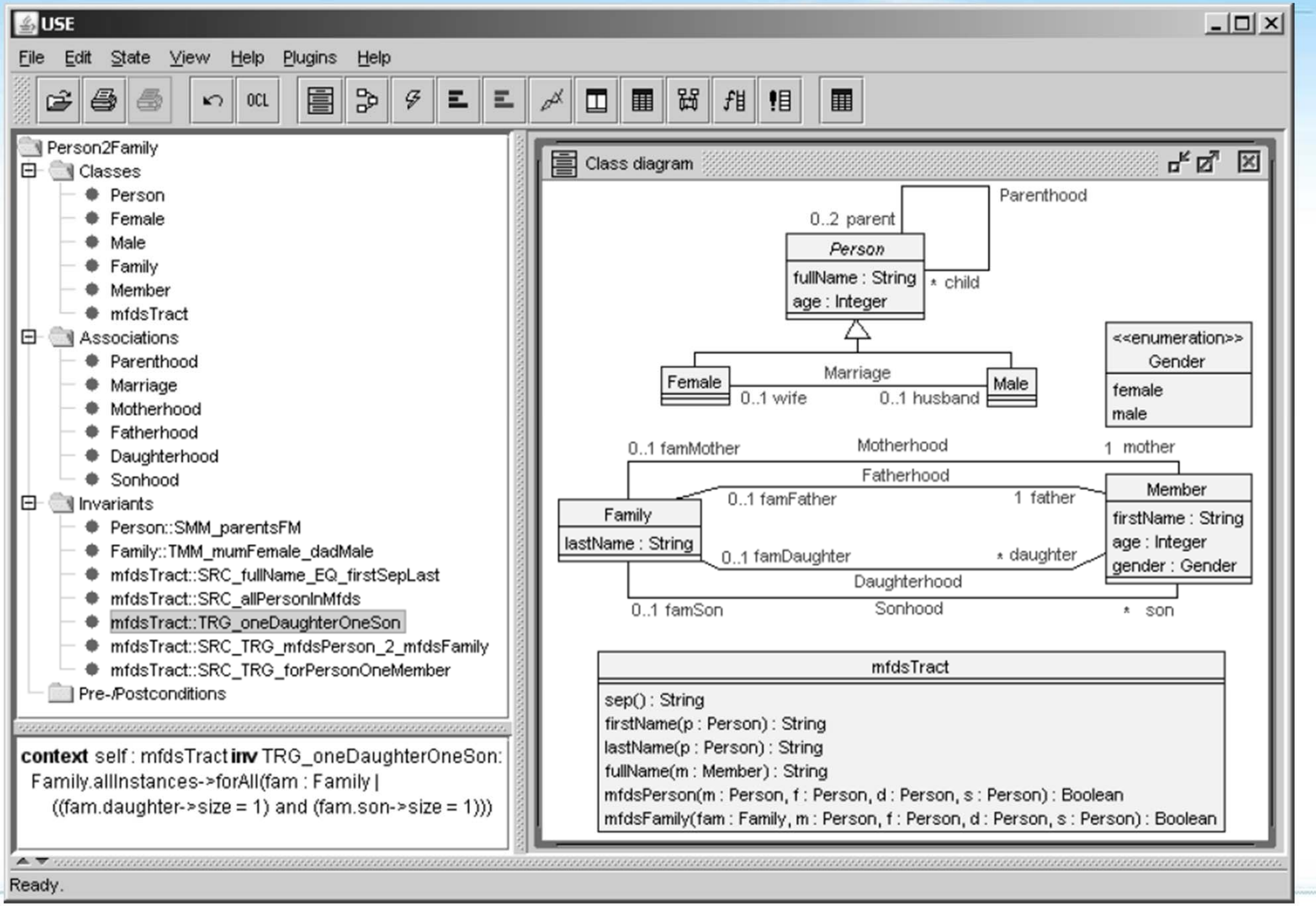
Tract constraints (2/2)

- Src_Trg_Constraint



```
inv SRC_TRG_forPersonOneMember:
  Female.allInstances->forAll(p| Member.allInstances->one(m|
    p.fullName=fullName(m) and p.age=m.age and m.gender = #female and
    (p.child->notEmpty() implies (let fam=m.famMother in
      p.child->size()=fam.daughter->union(fam.son)->size())) and
    (p.parent->notEmpty() implies m.famDaughter.isDefined()) and
    (p.husband.isDefined() implies m.famMother.isDefined()) )) and
  Male.allInstances->forAll(p| Member.allInstances->one(m|
    p.fullName=fullName(m) and p.age=m.age and m.gender = #male and
    (p.child->notEmpty() implies (let fam=m.famFather in
      p.child->size()=fam.daughter->union(fam.son)->size())) and
    (p.parent->notEmpty() implies m.famSon.isDefined()) and
    (p.wife.isDefined() implies m.famFather.isDefined()) ))
```

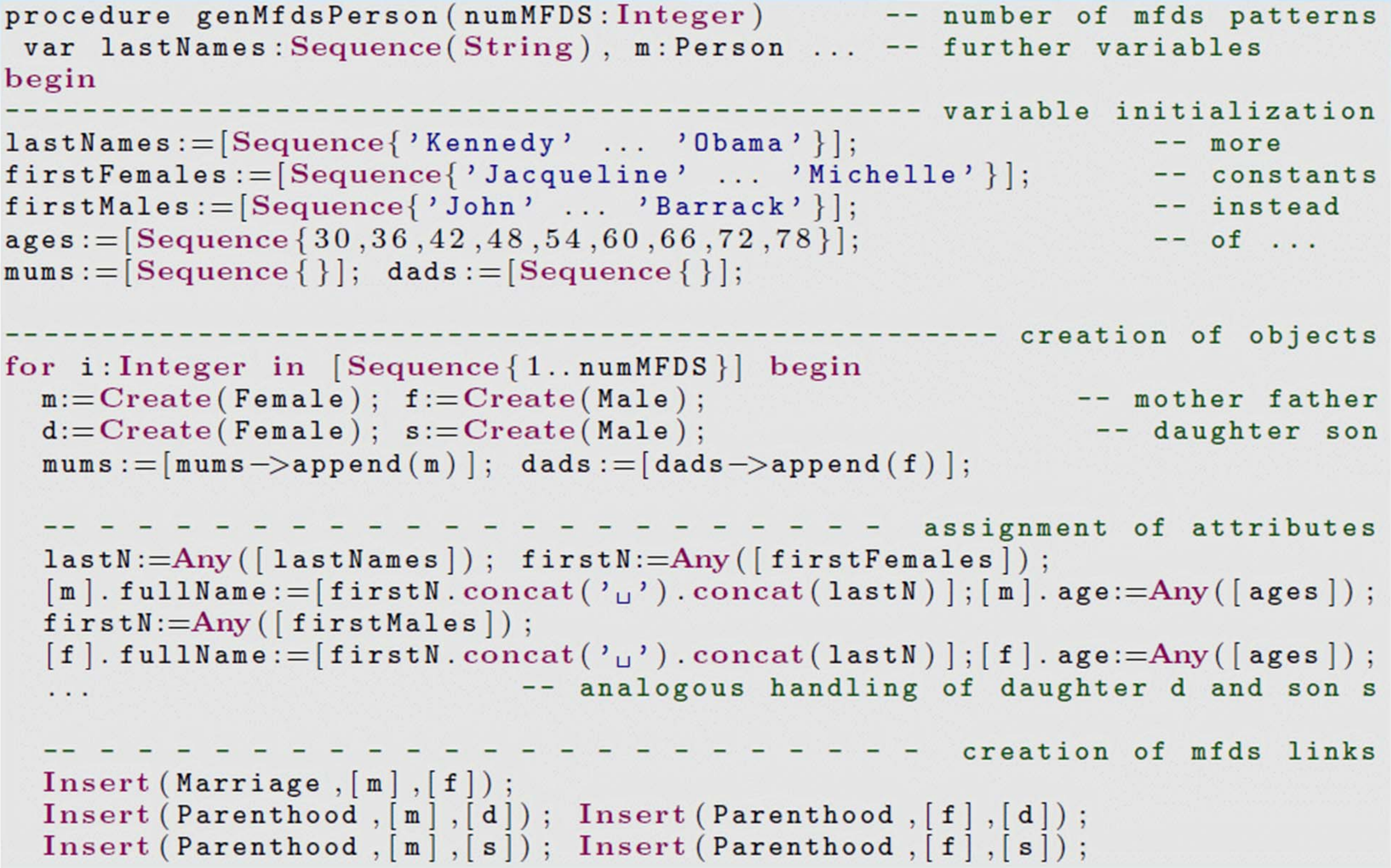
Mdfs Tract



- Person2Family
  - Classes
    - Person
    - Female
    - Male
    - Family
    - Member
    - mfdsTract
  - Associations
    - Parenthood
    - Marriage
    - Motherhood
    - Fatherhood
    - Daughterhood
    - Sonhood
  - Invariants
    - Person::SMM_parentsFM
    - Family::TMM_mumFemale_dadMale
    - mfdsTract::SRC_fullName_EQ_firstSepLast
    - mfdsTract::SRC_allPersonInMfds
    - mfdsTract::TRG_oneDaughterOneSon
    - mfdsTract::SRC_TRG_mfdsPerson_2_mfdsFamily
    - mfdsTract::SRC_TRG_forPersonOneMember
  - Pre-/Postconditions

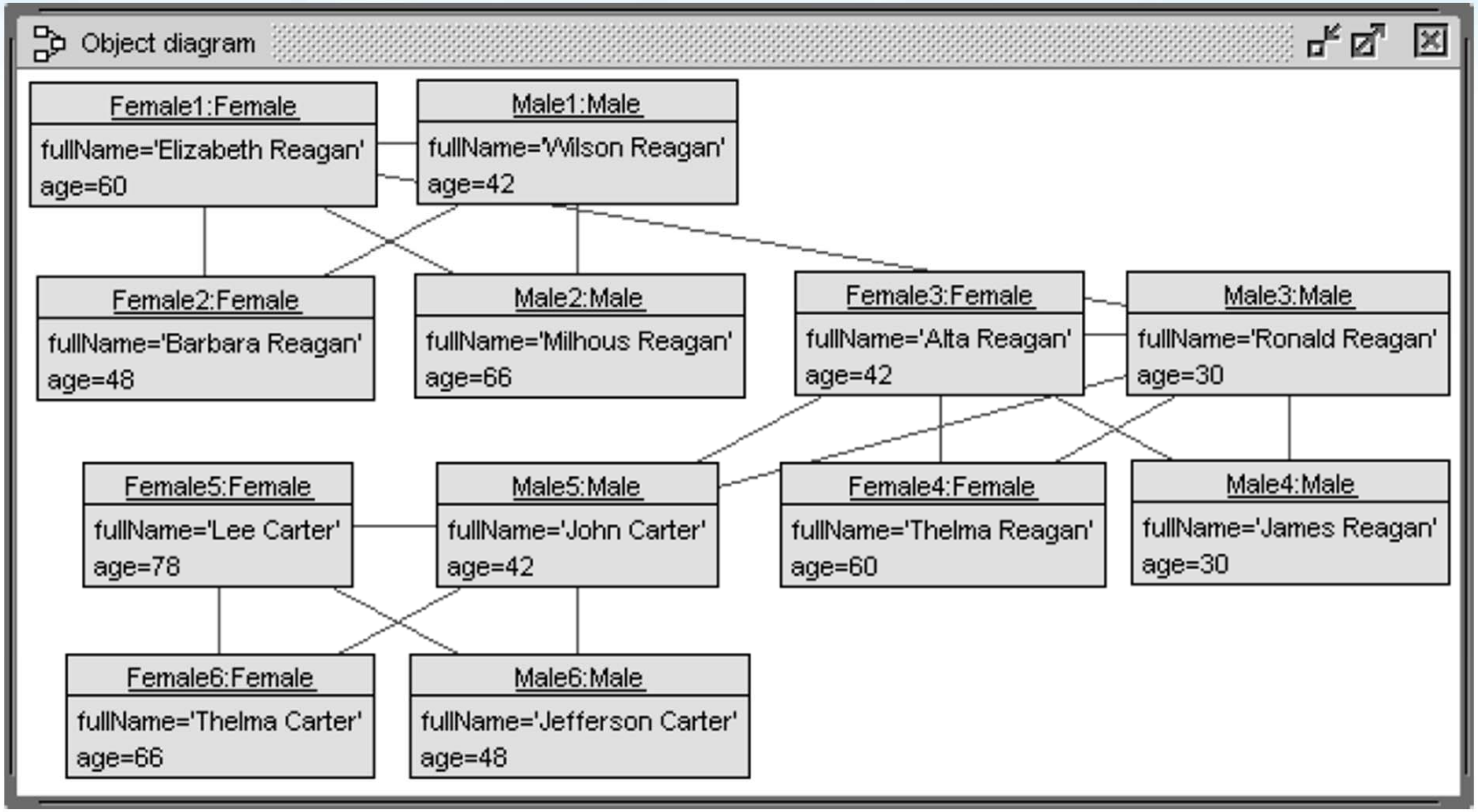
```
context self : mfdsTract inv TRG_oneDaughterOneSon:
  Family.allInstances->forAll(fam : Family |
    ((fam.daughter->size = 1) and (fam.son->size = 1)))
```

ASSL to generate the input models



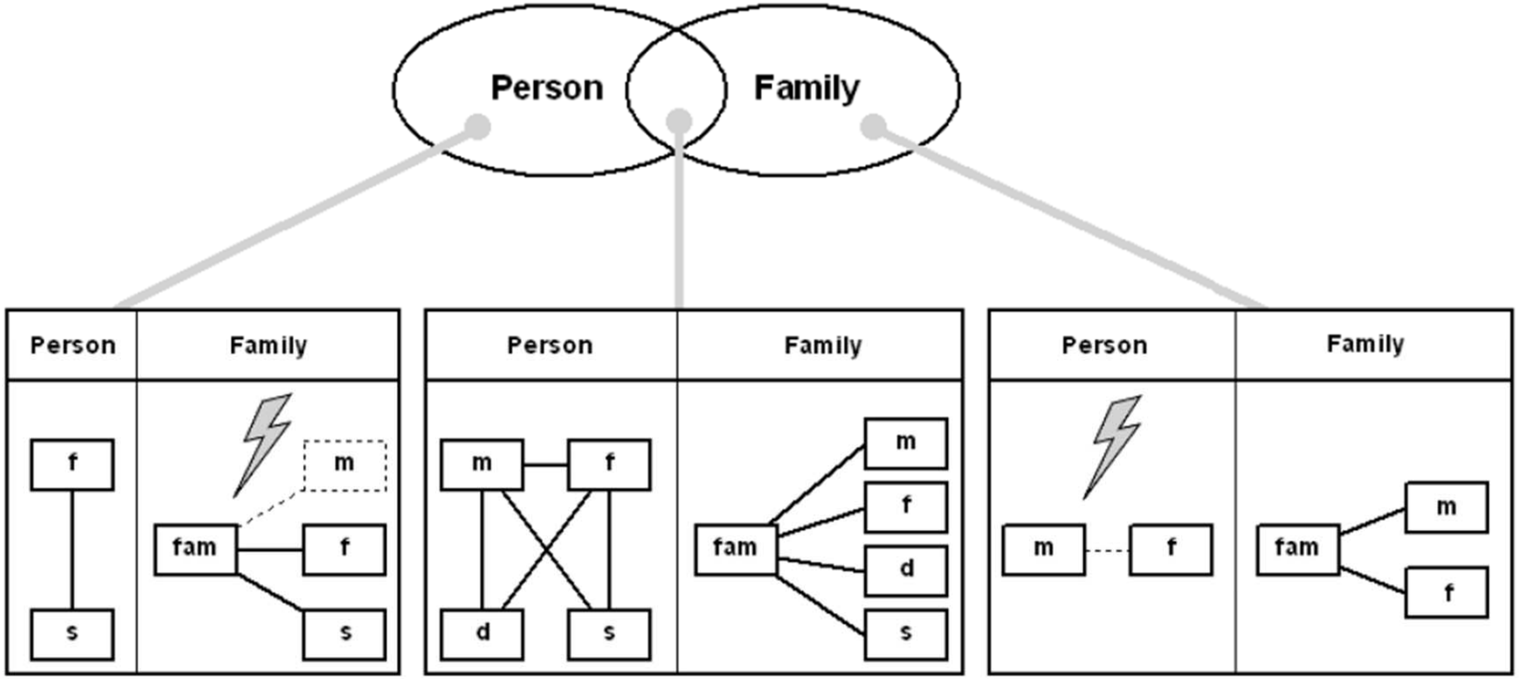
```
procedure genMfdsPerson(numMFDS:Integer)        -- number of mfds patterns
 var lastNames:Sequence(String), m:Person ...   -- further variables
begin
----------------------------------------------- variable initialization
lastNames:=[Sequence{'Kennedy' ... 'Obama'}];                  -- more
firstFemales:=[Sequence{'Jacqueline' ... 'Michelle'}];         -- constants
firstMales:=[Sequence{'John' ... 'Barrack'}];                  -- instead
ages:=[Sequence{30,36,42,48,54,60,66,72,78}];                  -- of ...
mums:=[Sequence{}]; dads:=[Sequence{}];

---------------------------------------------------- creation of objects
for i:Integer in [Sequence{1..numMFDS}] begin
  m:=Create(Female); f:=Create(Male);                     -- mother father
  d:=Create(Female); s:=Create(Male);                      -- daughter son
  mums:=[mums->append(m)]; dads:=[dads->append(f)];

  -- - - - - - - - - - - - - - - - - - - - - - -  assignment of attributes
  lastN:=Any([lastNames]); firstN:=Any([firstFemales]);
  [m].fullName:=[firstN.concat('␣').concat(lastN)];[m].age:=Any([ages]);
  firstN:=Any([firstMales]);
  [f].fullName:=[firstN.concat('␣').concat(lastN)];[f].age:=Any([ages]);
  ...                       -- analogous handling of daughter d and son s

  -- - - - - - - - - - - - - - - - - - - - - - - - -  creation of mfds links
  Insert(Marriage,[m],[f]);
  Insert(Parenthood,[m],[d]); Insert(Parenthood,[f],[d]);
  Insert(Parenthood,[m],[s]); Insert(Parenthood,[f],[s]);
```

Generation of negative cases



Kinds of problems found

- Errors in the transformation code
- Errors in the *Tract* specification
- Source-target semantic gap/mismatches
  - Unmarried couples, families with a single father or mother, married couples whose members have maintained their last names,…cannot be transformed.



Summary

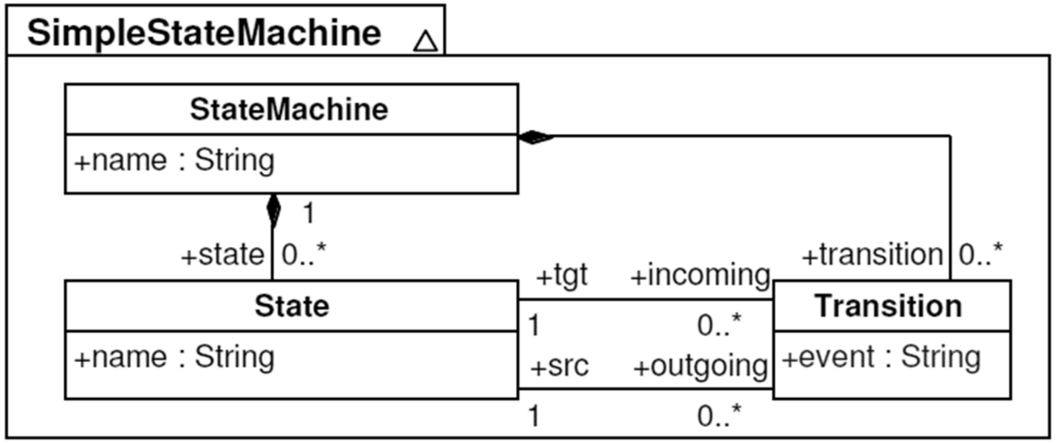
- Pros:
  - **Modular**: Allows partitioning the input space into smaller, focused behavioural units
  - **Specific**: allows defining specific tests for the input models
  - **Black-box**: Tests the MT as-is, independent from implementation
  - **Cost-effective**: Small tests are easy to define and to check

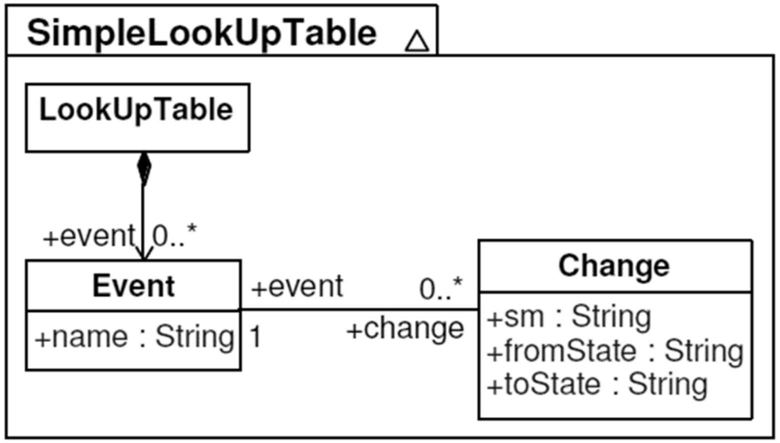
- Cons:
  - It does not guarantee full correctness (“certification” vs. full validation)
  - Completeness and coverage of input models is not guaranteed
  - Tracts are not easy to specify in general

Tracts By-Example

Tract example: SM2T (StateMachine 2 LookUpTable)

- **Source MM:** State Machine
- **Target MM:** Lookup Table

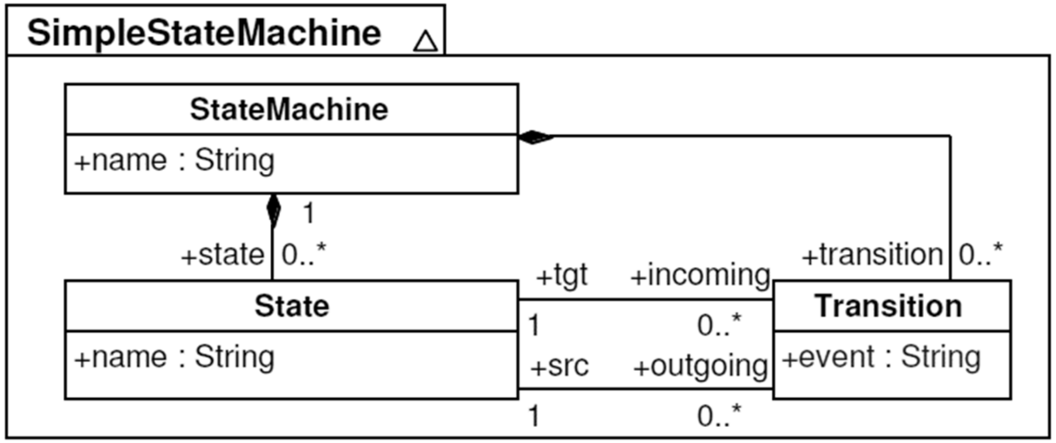


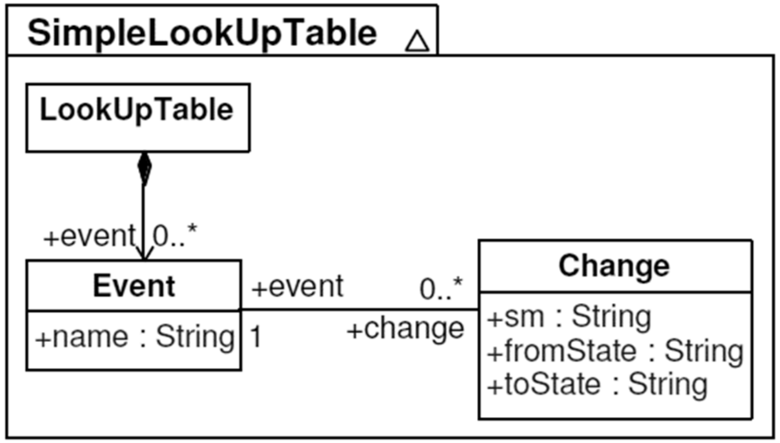


- We want only one lookup table
  - Where each entry is an event of the source model

SM2T: More restrictions

- **Source MM:** State Machine
- **Target MM:** Lookup Table

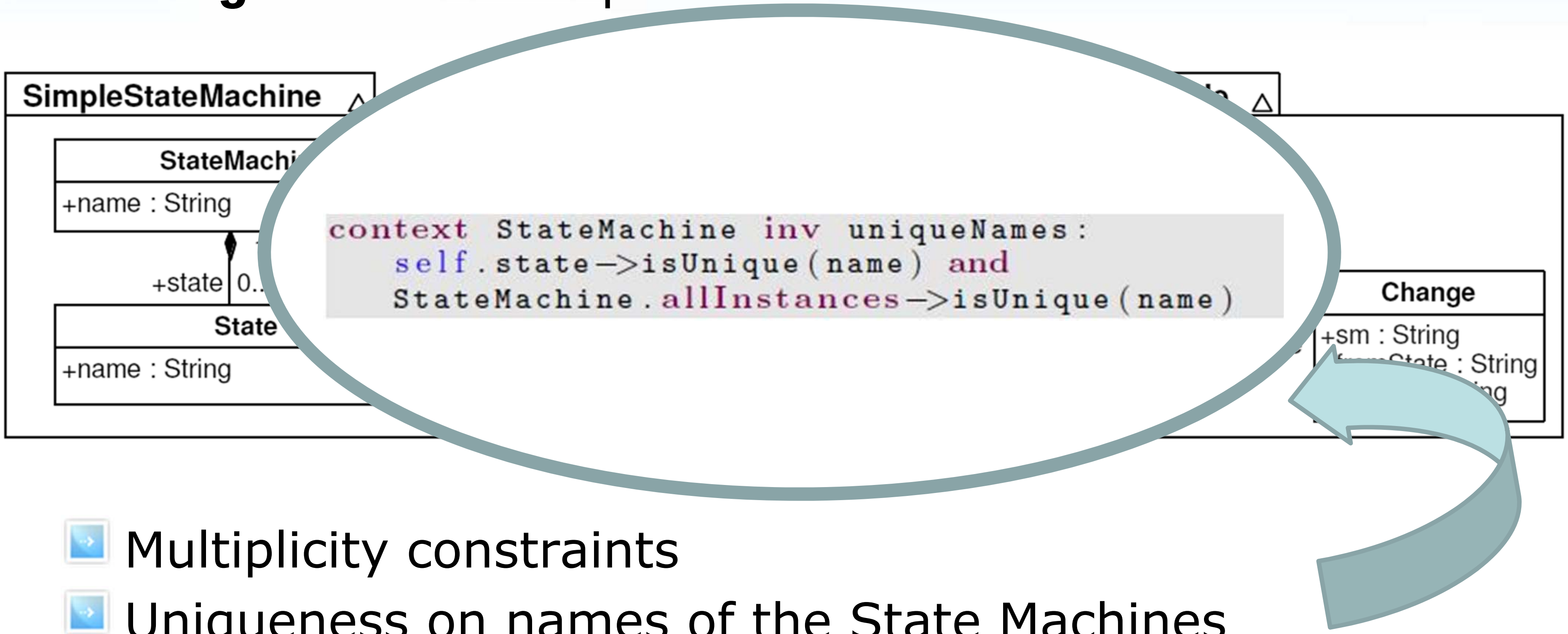




- Multiplicity constraints
- Uniqueness on names of the state machines
- Uniqueness on names of states within the same machine

SM2T: More restrictions

- **Source MM:** State Machine
- **Target MM:** Lookup Table



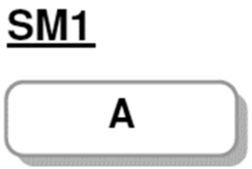
```
context StateMachine inv uniqueNames:
   self.state->isUnique(name) and
   StateMachine.allInstances->isUnique(name)
```

- Multiplicity constraints
- Uniqueness on names of the State Machines
- Uniqueness on names of states within the same machine

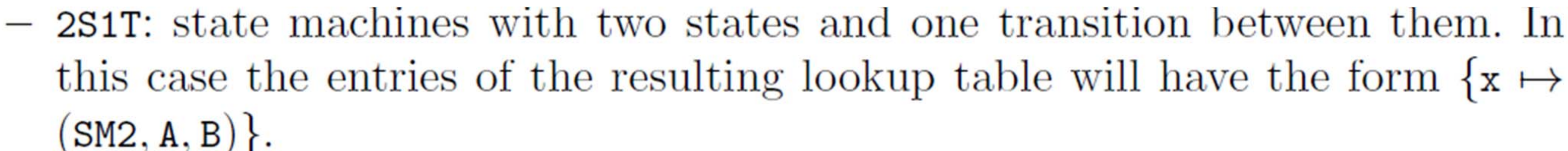
Tracts for SM2T

Six *Tracts* to start with

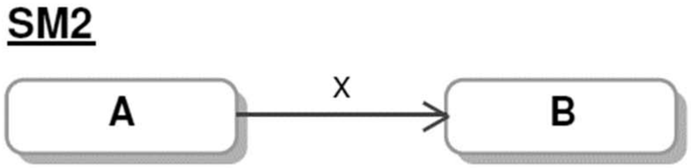
- 1S0T: state machines with single states and no transitions.



1S0T



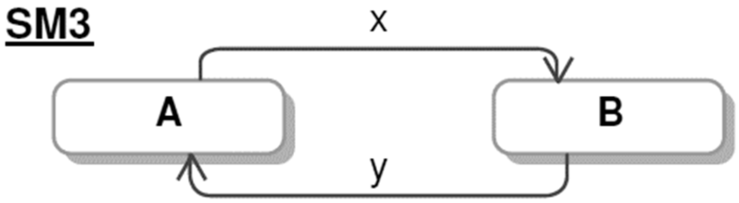
- 2S1T: state machines with two states and one transition between them. In this case the entries of the resulting lookup table will have the form $\{\mathtt{x} \mapsto (\mathtt{SM2}, \mathtt{A}, \mathtt{B})\}$.



2S1T

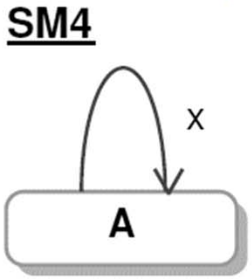
Tracts for SM2T

- 2S2T: state machines with two states and two transition between them. In this case the entries of the resulting lookup table will be of the form $\{\mathtt{x} \mapsto (\mathtt{SM3}, \mathtt{A}, \mathtt{B}), \mathtt{y} \mapsto (\mathtt{SM3}, \mathtt{B}, \mathtt{A})\}$.



2S2T

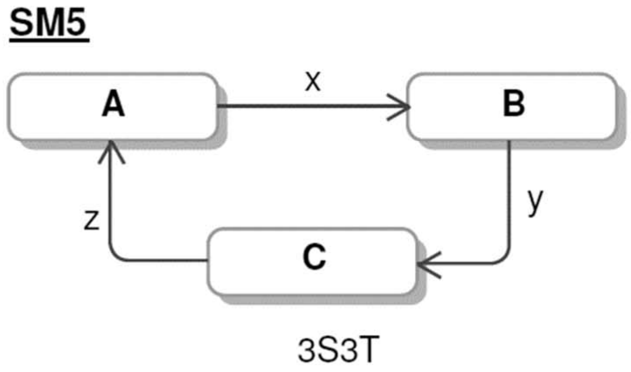
- 1S1T: state machines with single states and one transition. In this case the entries of the resulting lookup table will have the form $\{\mathtt{x} \mapsto (\mathtt{SM4}, \mathtt{A}, \mathtt{A})\}$.



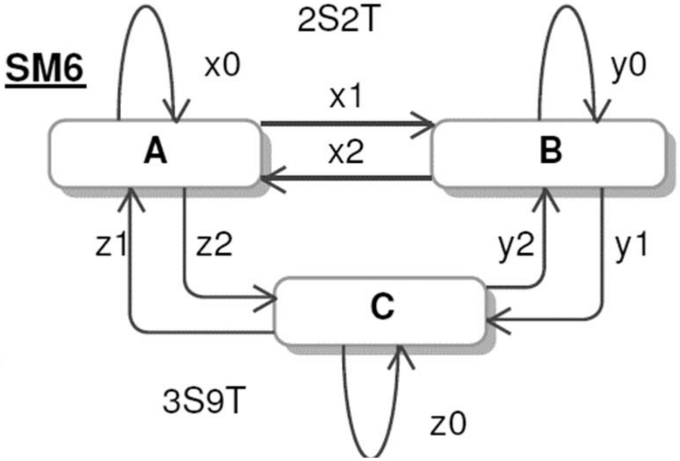
1S1T

Tracts for SM2T

- 3S3T: state machines with three states and three transitions, forming a cycle. In this case the entries of the resulting lookup table will be of the form $\{\mathtt{x} \mapsto (\mathtt{SM5}, \mathtt{A}, \mathtt{B}), \mathtt{y} \mapsto (\mathtt{SM5}, \mathtt{B}, \mathtt{C}), \mathtt{z} \mapsto (\mathtt{SM5}, \mathtt{C}, \mathtt{A})\}$.

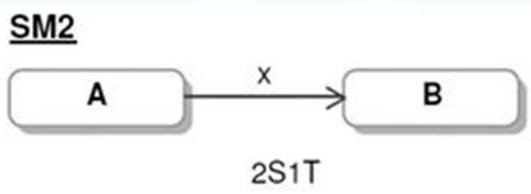


- 3S9T: state machines with three states and 9 transitions (see figure 21). In this case the entries of the resulting lookup table will have the form $\{\mathtt{x0} \mapsto (\mathtt{SM6}, \mathtt{A}, \mathtt{A}), \mathtt{x1} \mapsto (\mathtt{SM6}, \mathtt{A}, \mathtt{B}), \mathtt{x2} \mapsto (\mathtt{SM6}, \mathtt{B}, \mathtt{A}), \mathtt{y0} \mapsto (\mathtt{SM6}, \mathtt{B}, \mathtt{B}), \mathtt{y1} \mapsto (\mathtt{SM6}, \mathtt{B}, \mathtt{C}), \mathtt{y2} \mapsto (\mathtt{SM6}, \mathtt{C}, \mathtt{B}), \mathtt{z0} \mapsto (\mathtt{SM6}, \mathtt{C}, \mathtt{C}), \mathtt{z1} \mapsto (\mathtt{SM6}, \mathtt{C}, \mathtt{A}), \mathtt{z2} \mapsto (\mathtt{SM6}, \mathtt{A}, \mathtt{C})\}$.

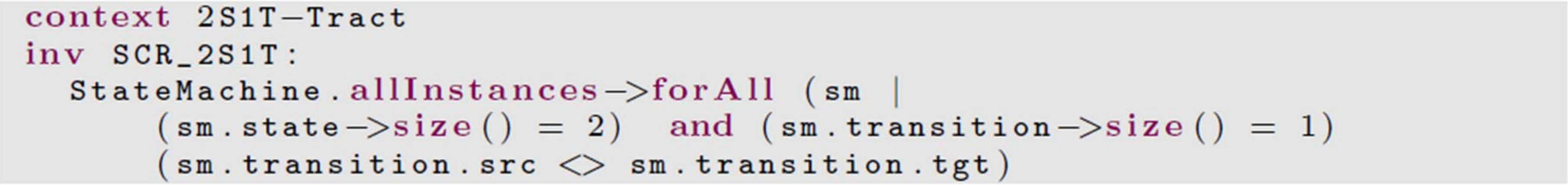


Tract example: Constraints

- ***Tract* for SM2:**

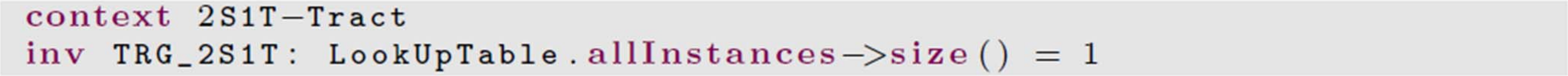


- Src_Constraints



```
context 2S1T−Tract
inv SCR_2S1T:
  StateMachine.allInstances−>forAll (sm |
      (sm.state−>size() = 2)  and (sm.transition−>size() = 1)
      (sm.transition.src <> sm.transition.tgt)
```

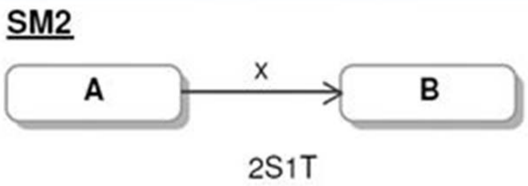
- Trg_Constraint



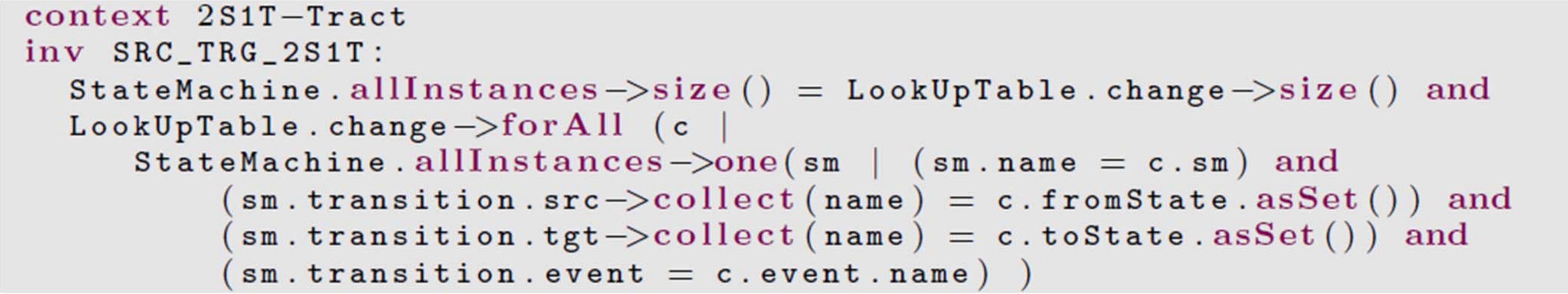
```
context 2S1T−Tract
inv TRG_2S1T: LookUpTable.allInstances−>size() = 1
```

Tract example: Constraints

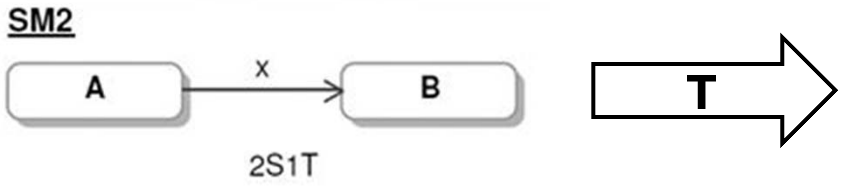
- ***Tract* for SM2:**



- Src_Trg_Constraint



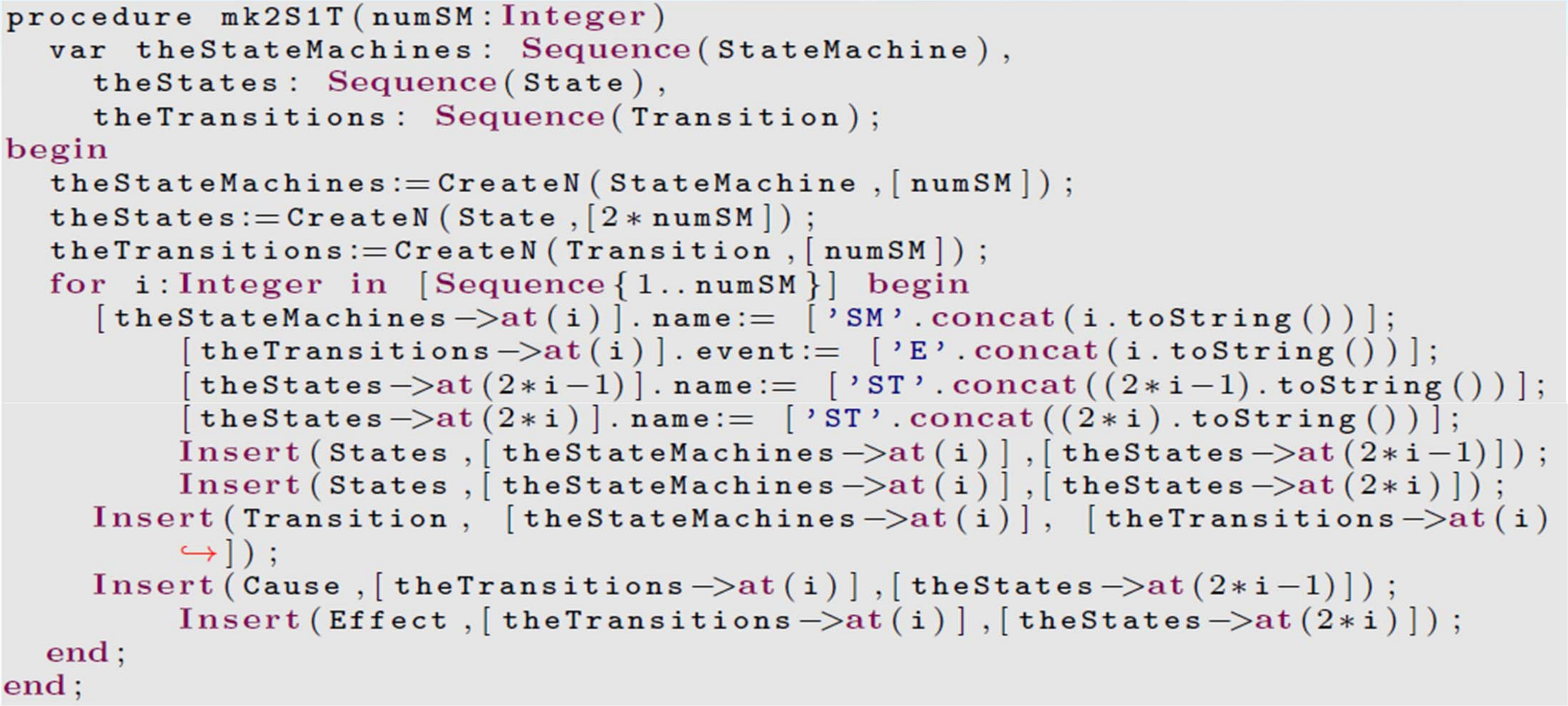
```
context 2S1T−Tract
inv SRC_TRG_2S1T:
  StateMachine.allInstances->size() = LookUpTable.change->size() and
  LookUpTable.change->forAll (c |
     StateMachine.allInstances->one(sm | (sm.name = c.sm) and
        (sm.transition.src->collect(name) = c.fromState.asSet()) and
        (sm.transition.tgt->collect(name) = c.toState.asSet()) and
        (sm.transition.event = c.event.name) )
```



LookUpTable

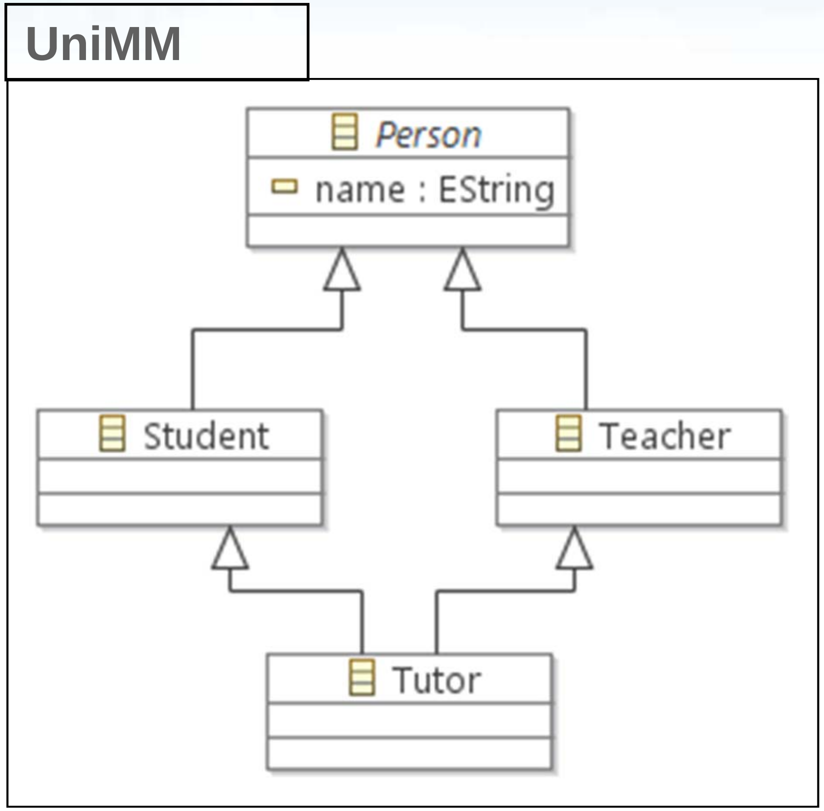
Event	Change		
	sm	fromState	toState
x	SM2	A	B

ASSL Code for SM2



```
procedure mk2S1T(numSM:Integer)
  var theStateMachines: Sequence(StateMachine),
    theStates: Sequence(State),
    theTransitions: Sequence(Transition);
begin
  theStateMachines:=CreateN(StateMachine,[numSM]);
  theStates:=CreateN(State,[2*numSM]);
  theTransitions:=CreateN(Transition,[numSM]);
  for i:Integer in [Sequence{1..numSM}] begin
    [theStateMachines->at(i)].name:= ['SM'.concat(i.toString())];
        [theTransitions->at(i)].event:= ['E'.concat(i.toString())];
        [theStates->at(2*i-1)].name:= ['ST'.concat((2*i-1).toString())];
        [theStates->at(2*i)].name:= ['ST'.concat((2*i).toString())];
        Insert(States,[theStateMachines->at(i)],[theStates->at(2*i-1)]);
        Insert(States,[theStateMachines->at(i)],[theStates->at(2*i)]);
    Insert(Transition, [theStateMachines->at(i)], [theTransitions->at(i)
        ↪ ]);
    Insert(Cause,[theTransitions->at(i)],[theStates->at(2*i-1)]);
        Insert(Effect,[theTransitions->at(i)],[theStates->at(2*i)]);
  end;
end;
```

Exercise 1: Specify and implement



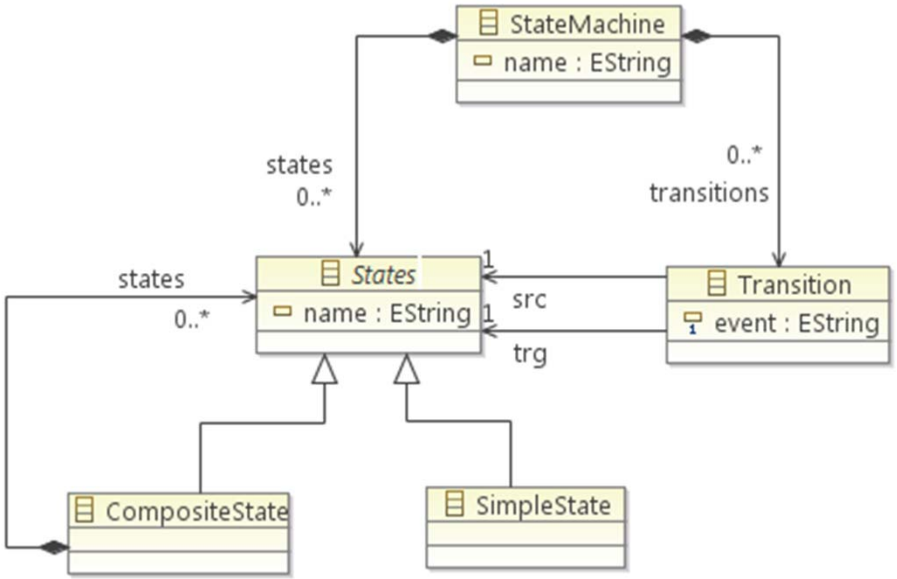
Copy Transformation

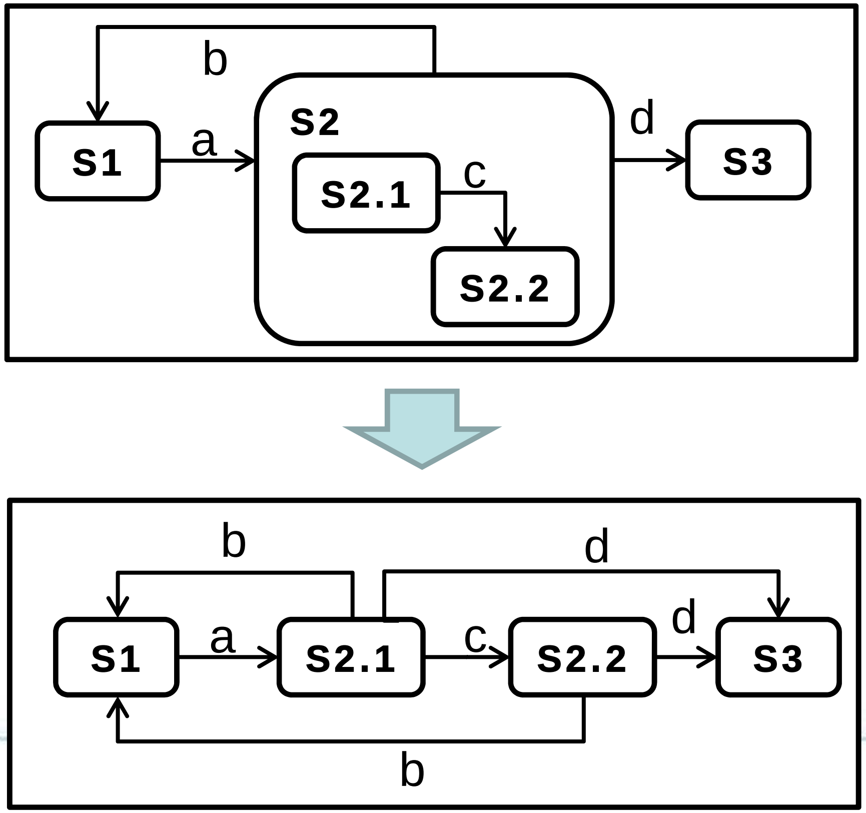
Create a copy of a model by using a model transformation

a) What tracts do we need?
b) Identify use cases
c) Write the ATL transformation
d) Prove its correctness

Exercise2: nSM2fSM

- nSM2fSM: Translate a nested state machine into a flat state machine
- Implementation: In-place Transformation
- Specification: What Tracts do we need?





Tractable Model Transformation Typing

Motivation for MT Typing

- MDE tackles complexity of large systems, but this requires to **model-in-the-large**
  - This results in **megamodels**
- Increasing need for precise and abstract mechanisms
  - Reason about the designed systems
  - Test individual components
- Assigning **types** to models and model transformations and arrange them in **type hierarchies**
- Light-weight approach to type model transformations using tracts

Definitions: Type and Subtype/Supertype

Type (of an <X>**)**: A predicate characterizing a collection of <X>s.

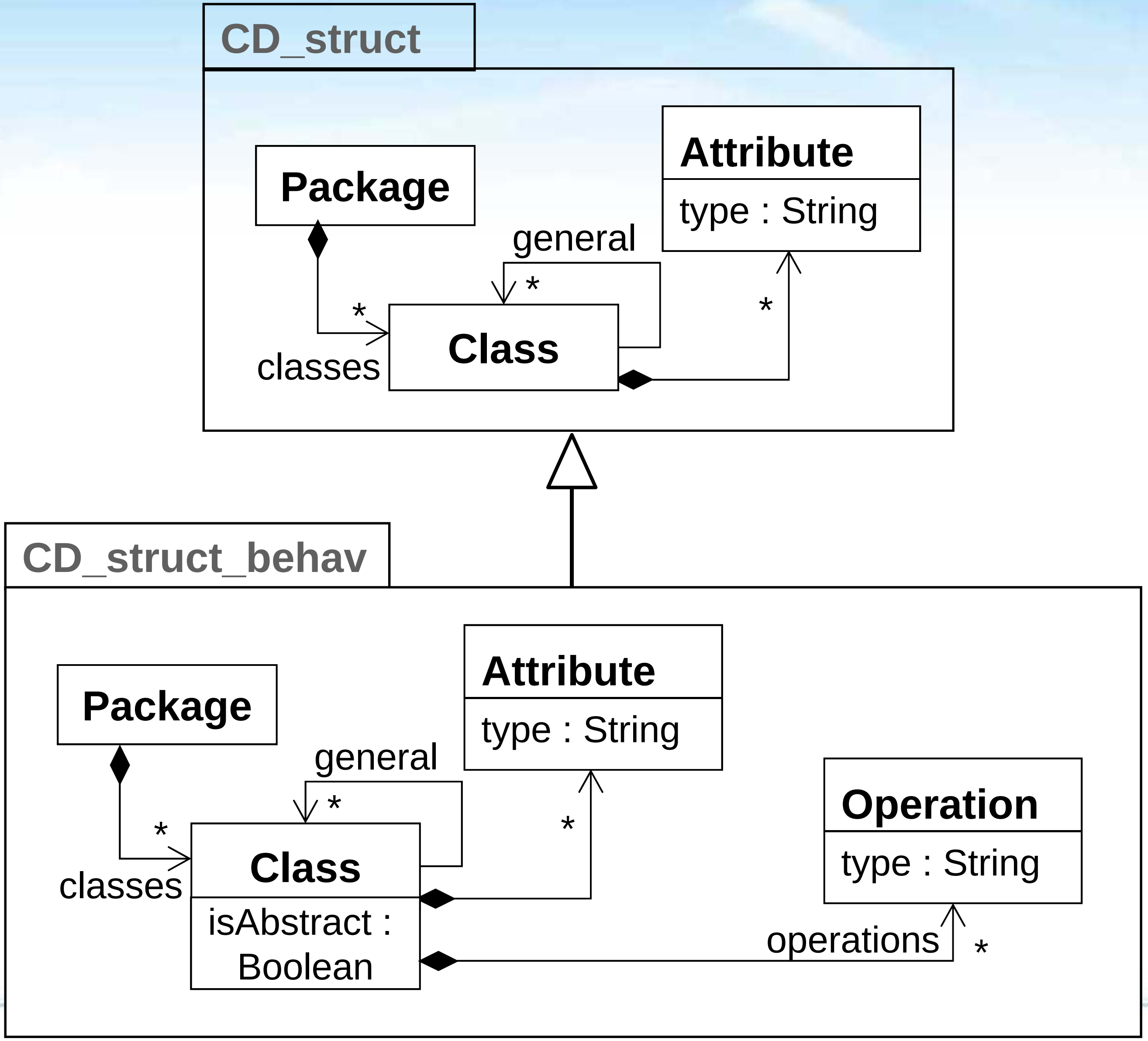
Subtype/supertype: A type A is a subtype of a type B, and B is a supertype of A, if every <X> which satisfies A also satisfies B.

[ISO/IEC 10746-2]

Typing models

- Model types are needed fo describing the signature of *model operations*
- The type of a model is basically its metamodel
- We can talk about “subtyping” and “safe replaceability”
    - M’ extends M iff M’ contains all classes, attributes and relationships of M, and M’ imposes the same or weaker constraints than M

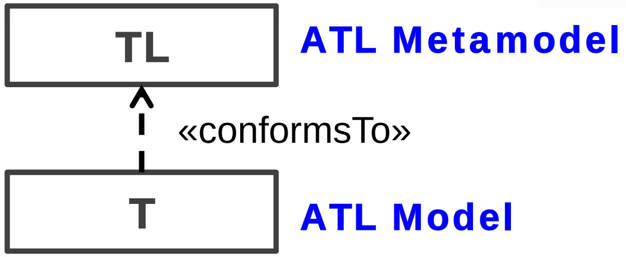
Typing Models (2/2)



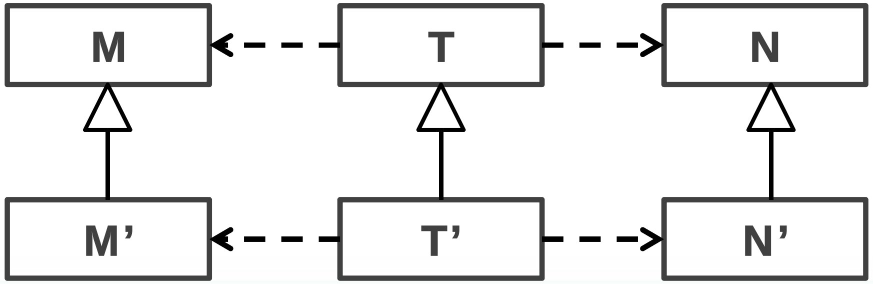
Typing MTs (1/2)

- Dual nature of model transformations: models and operations

- Naïve type 1: Models typed by meta-model of their modeling language (e.g., QVT or ATL)



- Naïve type 2: types of input and output models ("structural" type)
  - Two transformations: T : M → N and T' : M' → N'
  - T' structurally substitute of T iff (M' < M) ∧ (N' < N)

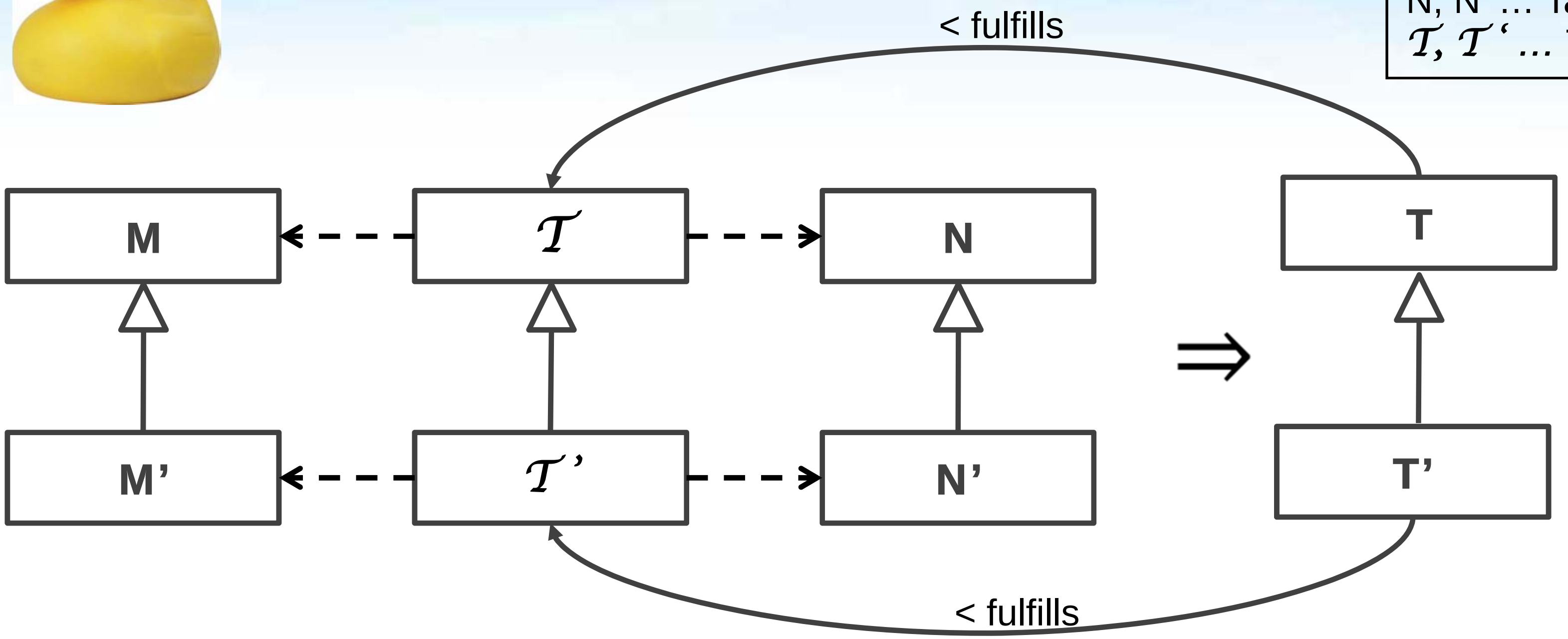


Typing MTs (2/2)

- To reason properly about subtype relationships between transformations, the **behavioural type** of a transformation must be considered

- **But**…
  - Typing model transformation as operations is difficult
  - Type of any behavioral software artefact is complex; manipulating and reasoning about behavioural types expensive
  - Current types capture full behaviour of the artefact independently from any context of use
  - Traditionally requires heavyweight reasoning techniques and tools such as theorem provers

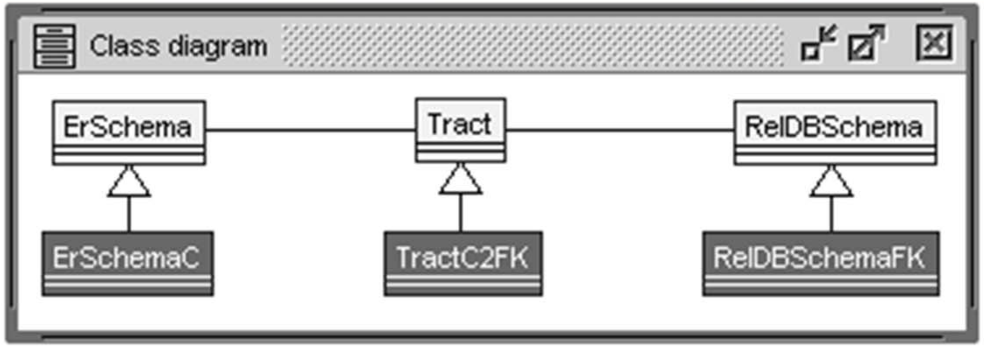
Model Transformation Typing using Tracts

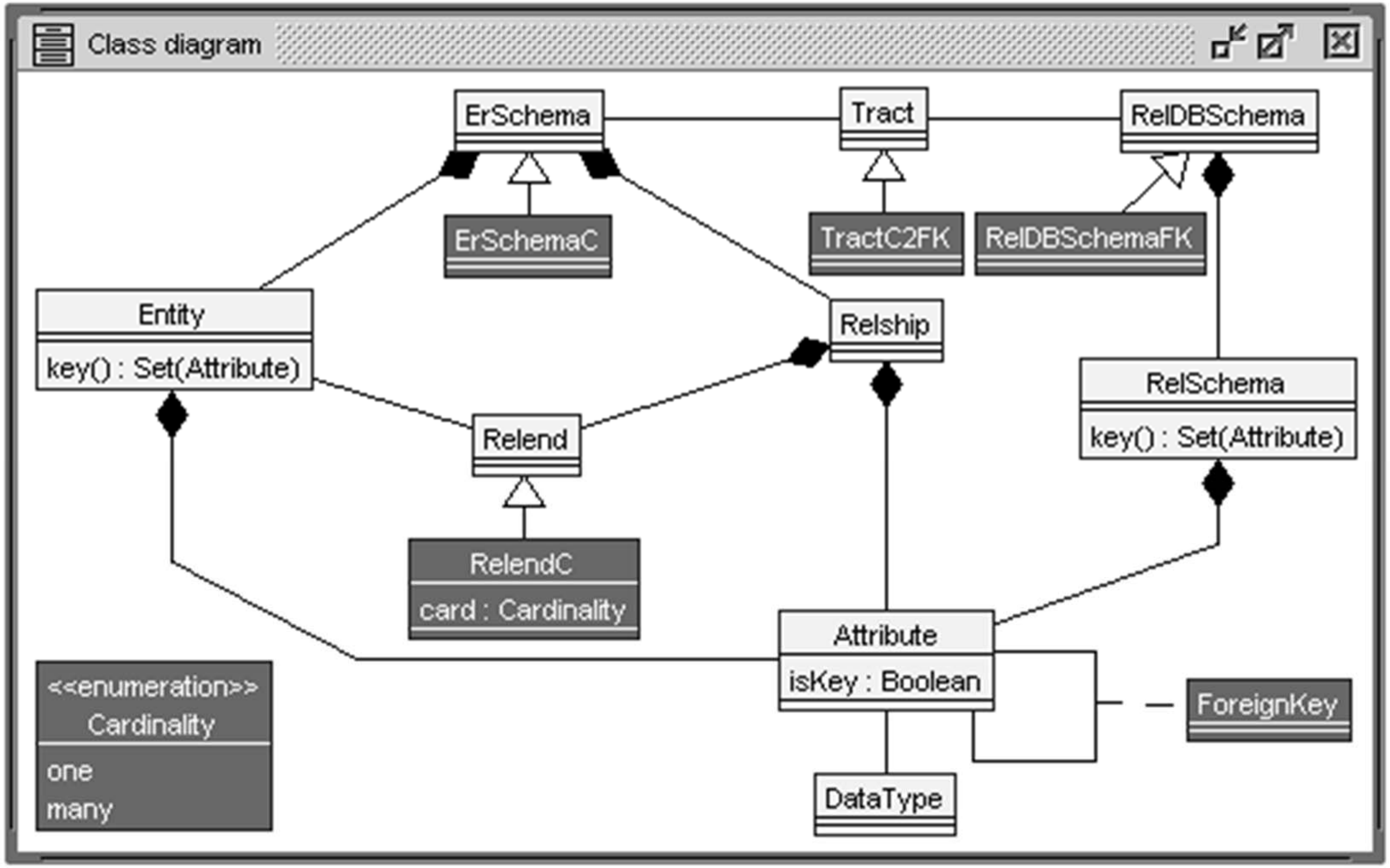


Legend:
M, M’ … Source MMs
N, N’ … Target MMs
$\mathcal{T}$, $\mathcal{T}$‘ … Tracts

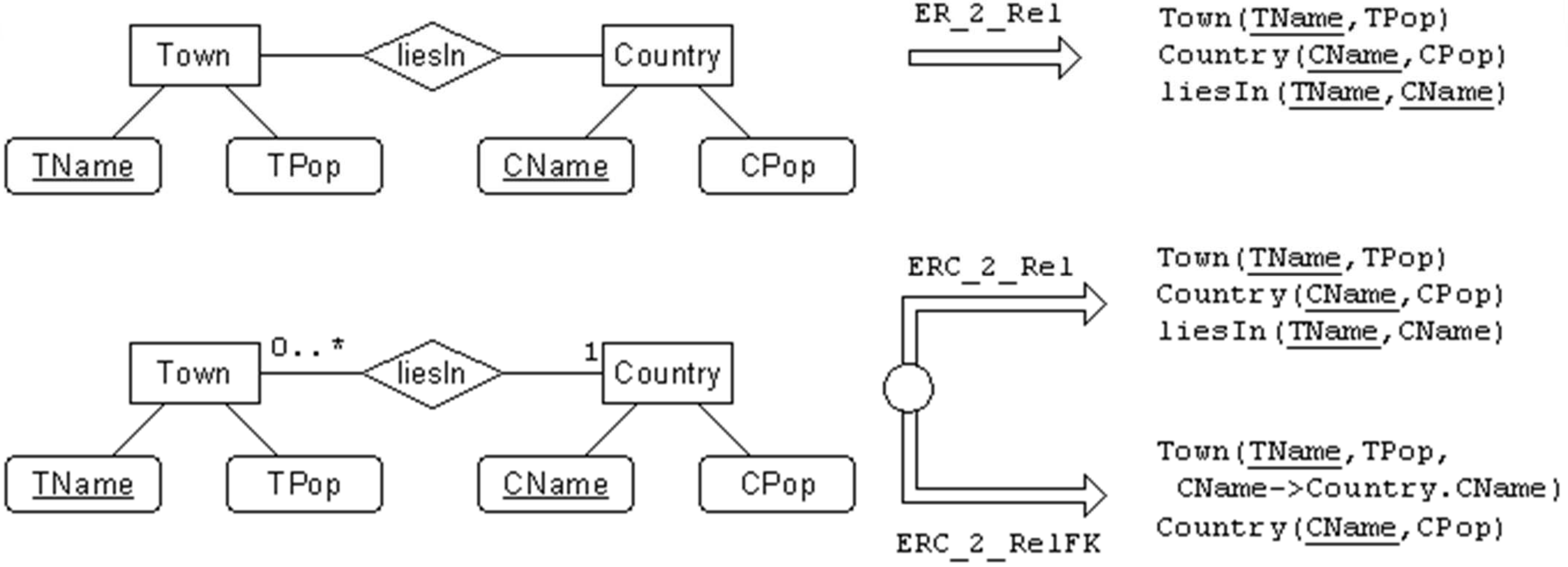
- Requirement for **behavioural** subtyping
  - **Structural** subtyping is **prerequisite**
  - Tracts consist of source conditions ($\mathfrak{H}$), target conditions ($\mathfrak{I}$), and source/target conditions ($\mathfrak{R}$)
  - $\mathcal{T}$ behavioural specification of T and $\mathcal{T}$‘ behavioural specification of T’
  - *T’* behavioural substitute of *T* iff $(\mathfrak{H} \Rightarrow \mathfrak{H}') \wedge (\mathfrak{I}' \Rightarrow \mathfrak{I}) \wedge (\mathfrak{R}' \Rightarrow \mathfrak{R})$

Model Transformation Typing by Example





Model Transformation Typing by Example
Three Transformations

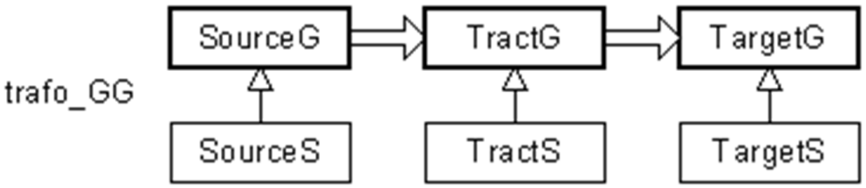


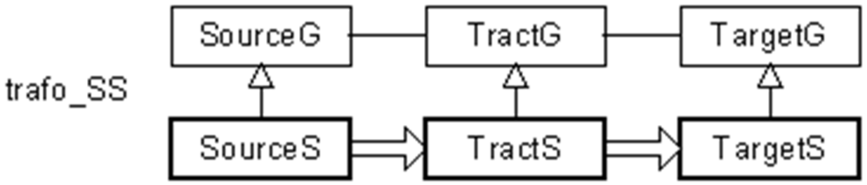
Town(TName,TPop)
Country(CName,CPop)
liesIn(TName,CName)

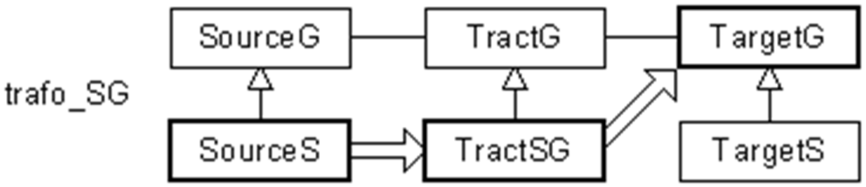
Town(TName,TPop)
Country(CName,CPop)
liesIn(TName,CName)

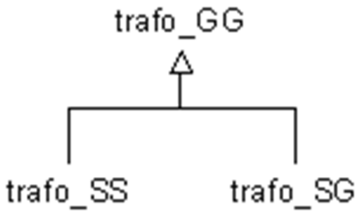
Town(TName,TPop,
 CName->Country.CName)
Country(CName,CPop)

Model Transformation Typing: Relationships between Transformations









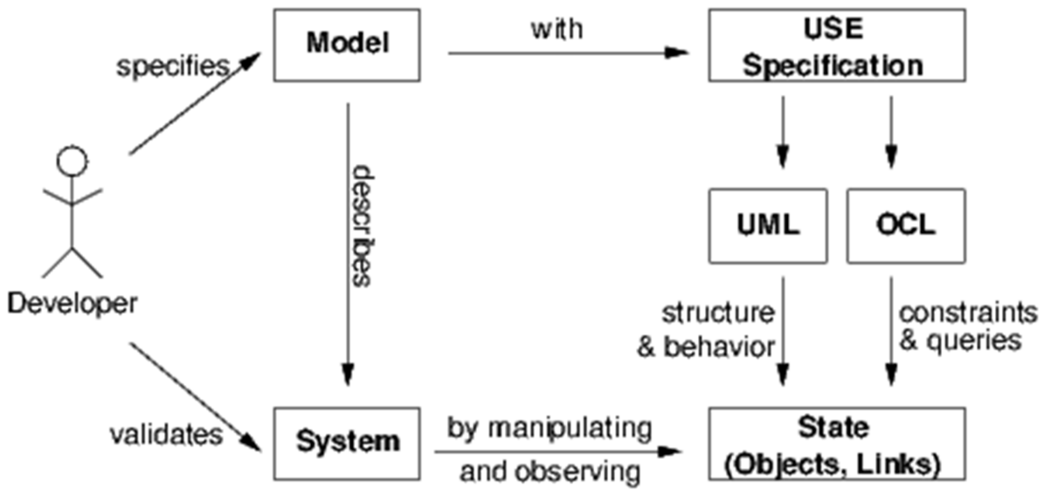
Discussion

- **Correctness** of a MT implementation
  - check that a given transformation conforms to a tract, i.e., it conforms to a certain type
- **Safe substitutability** of MTs; two step process:
  - first input models are automatically generated and then
  - for each of these we can check whether the transformation fulfils the associated tract
- **Incremental** and **systematic** transformation development
  - extend source and target metamodels by subtyping through small increments
  - accompanied by corresponding tracts including test suites;
  - benefit: rapid and direct feedback provided
- **Declarative** vs **imperative** tracts
  - only the relationship between source and target elements can be characterized;
  - but tracts also be described in an operational way when including operations mapping source elements to target elements

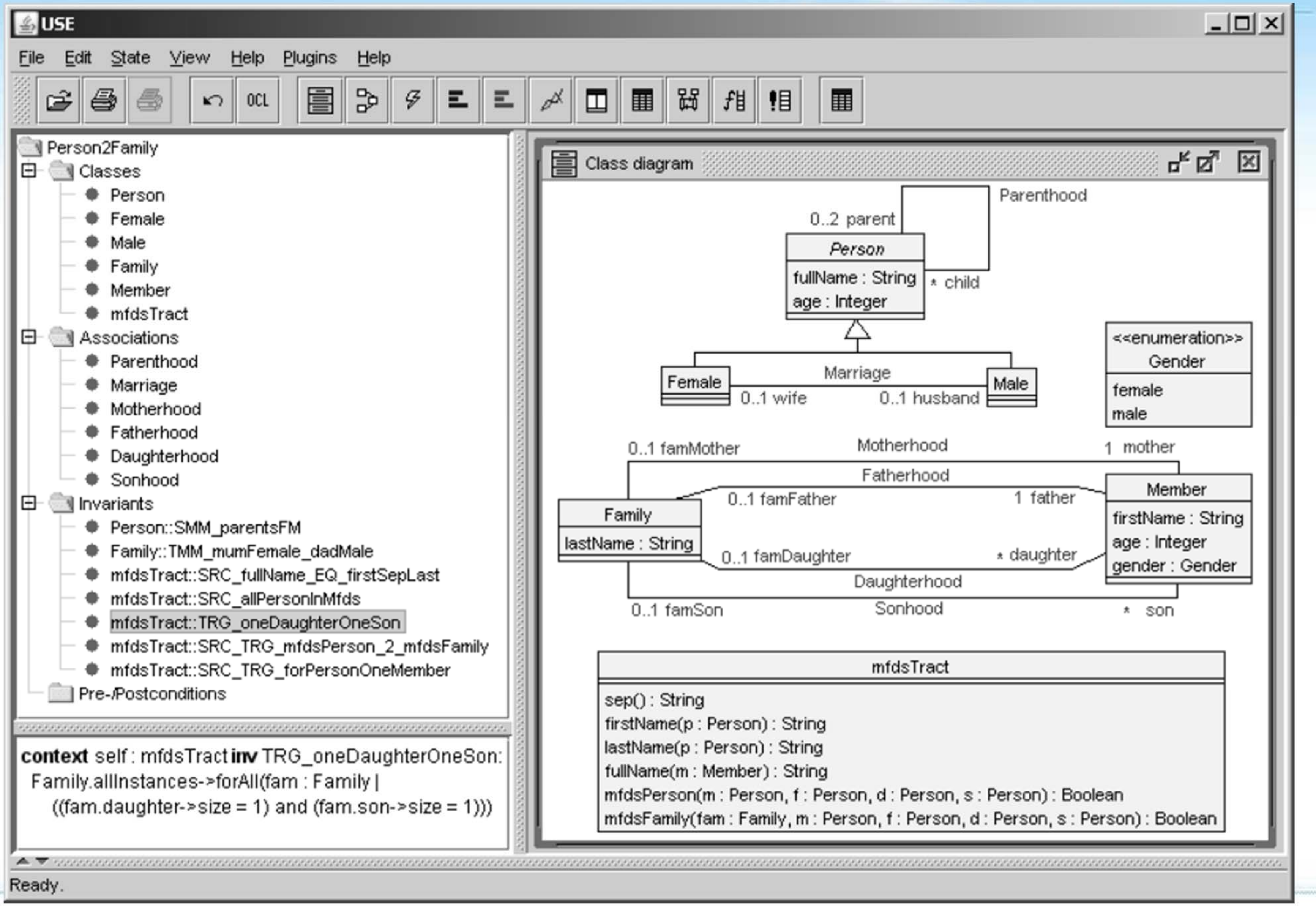
Tool Support for Tracts

Tracts Tool Support based on USE

- **USE - The UML-based Specification Environment**
  - Modeling of UML Class Diagrams and Object Diagrams
  - Support for full OCL
  - ASSL for generating Object Diagrams
  - Powerful API for validating models

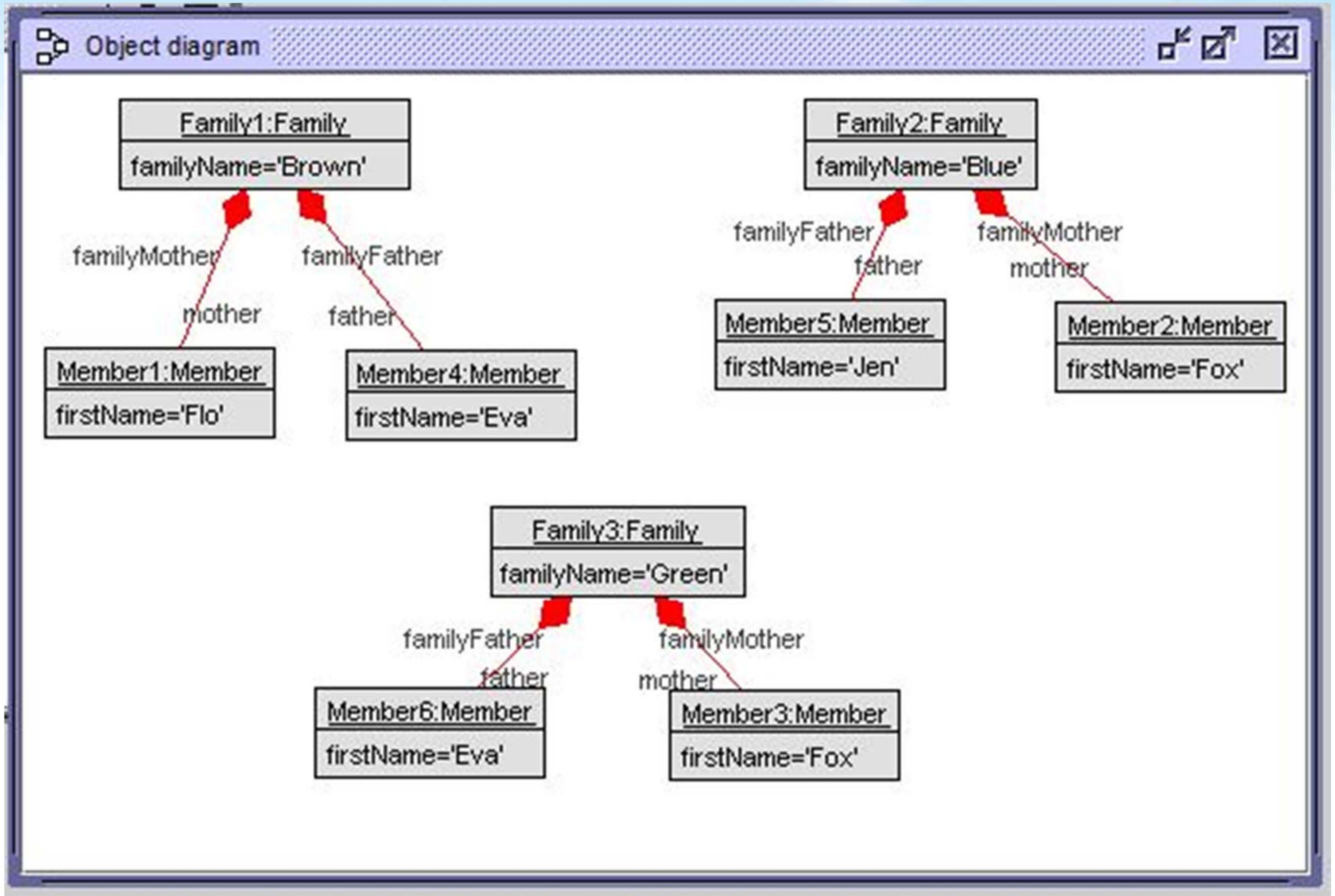


Metamodels as Class Diagrams

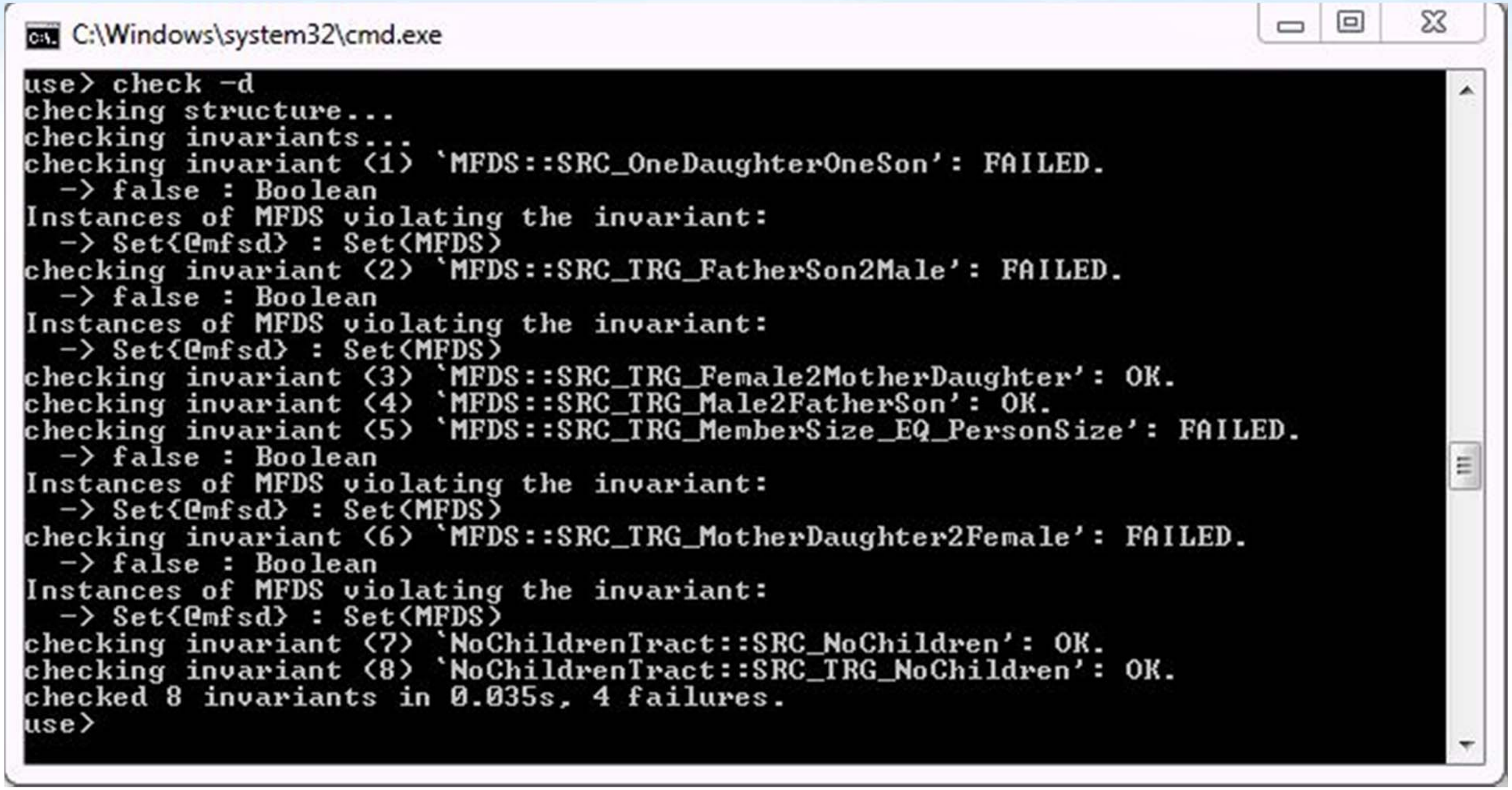


```
context self : mfdsTract inv TRG_oneDaughterOneSon:
  Family.allInstances->forAll(fam : Family |
    ((fam.daughter->size = 1) and (fam.son->size = 1)))
```

Models as Object Diagrams



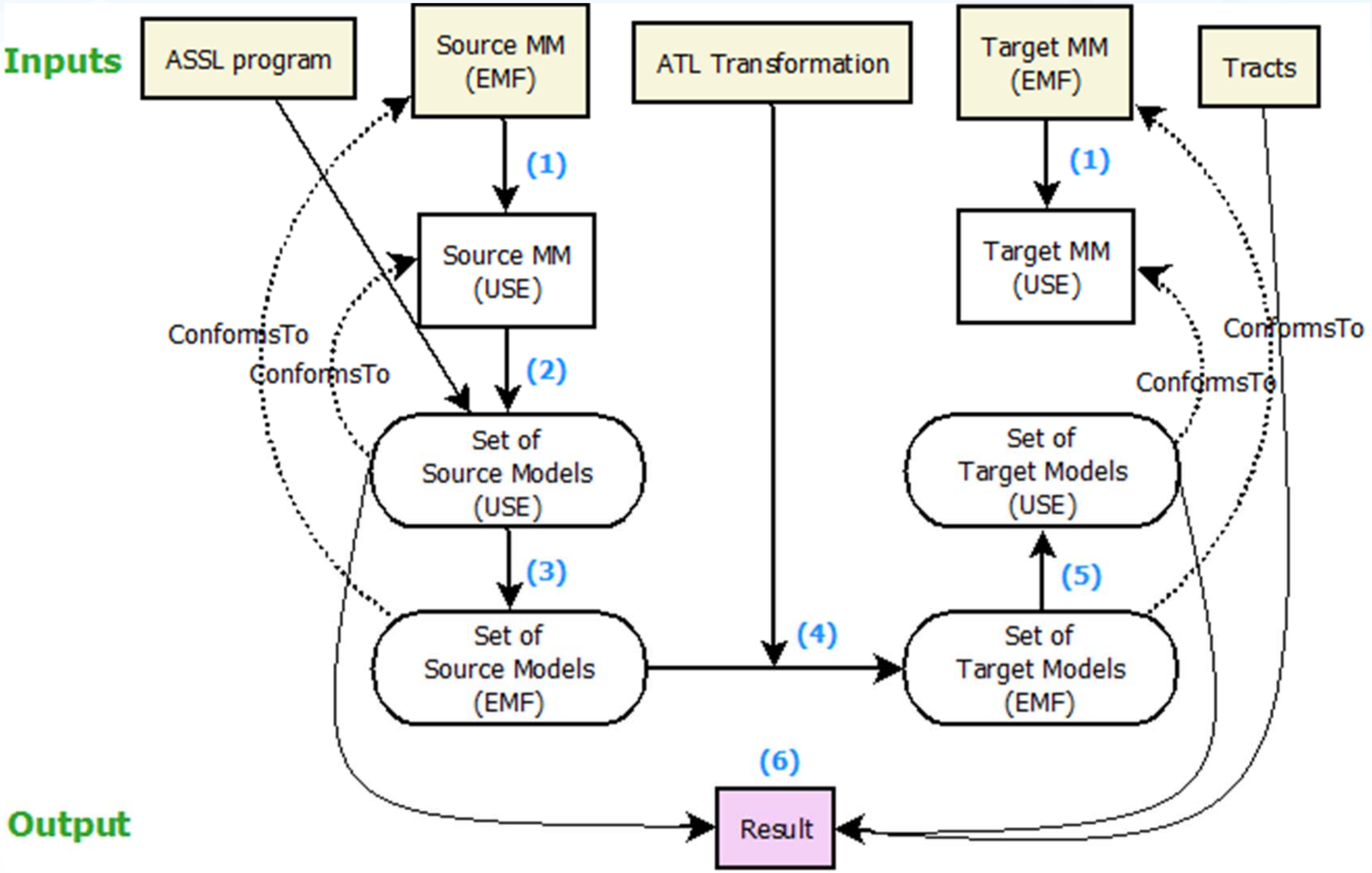
Checking the Tracts



```
use> check -d
checking structure...
checking invariants...
checking invariant (1) `MFDS::SRC_OneDaughterOneSon': FAILED.
  -> false : Boolean
Instances of MFDS violating the invariant:
  -> Set{@mfsd} : Set(MFDS)
checking invariant (2) `MFDS::SRC_TRG_FatherSon2Male': FAILED.
  -> false : Boolean
Instances of MFDS violating the invariant:
  -> Set{@mfsd} : Set(MFDS)
checking invariant (3) `MFDS::SRC_TRG_Female2MotherDaughter': OK.
checking invariant (4) `MFDS::SRC_TRG_Male2FatherSon': OK.
checking invariant (5) `MFDS::SRC_TRG_MemberSize_EQ_PersonSize': FAILED.
  -> false : Boolean
Instances of MFDS violating the invariant:
  -> Set{@mfsd} : Set(MFDS)
checking invariant (6) `MFDS::SRC_TRG_MotherDaughter2Female': FAILED.
  -> false : Boolean
Instances of MFDS violating the invariant:
  -> Set{@mfsd} : Set(MFDS)
checking invariant (7) `NoChildrenTract::SRC_NoChildren': OK.
checking invariant (8) `NoChildrenTract::SRC_TRG_NoChildren': OK.
checked 8 invariants in 0.035s, 4 failures.
use>
```

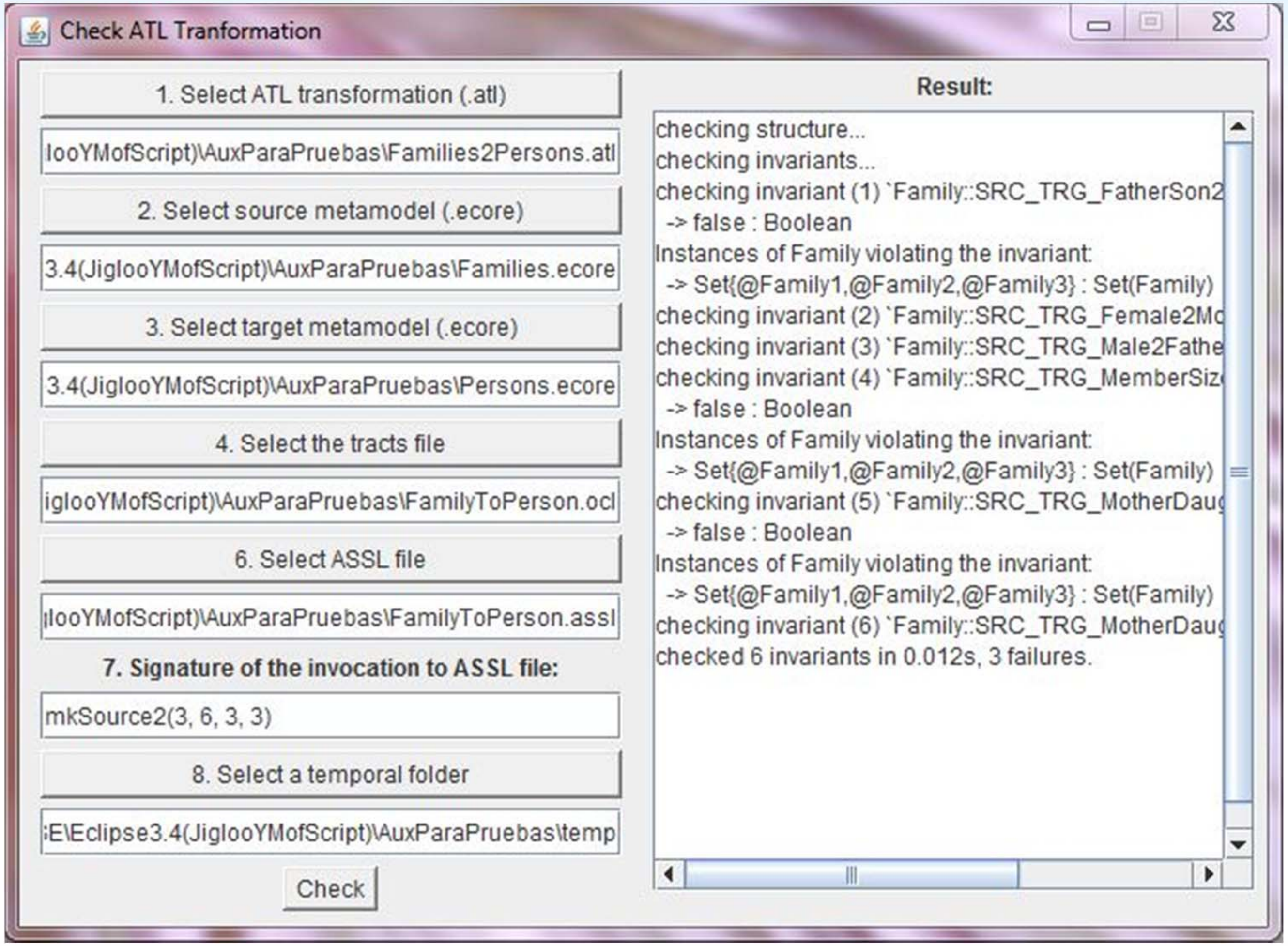
Test ATL Model Transformations for EMF-based Models

- Bridge EMF and USE



Tracts for EMF

http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Tracts


Check ATL Tranformation
1. Select ATL transformation (.atl)
2. Select source metamodel (.ecore)
3. Select target metamodel (.ecore)
4. Select the tracts file
6. Select ASSL file
8. Select a temporal folder
Check
Result:

looYMofScript)\AuxParaPruebas\Families2Persons.atl

3.4(JiglooYMofScript)\AuxParaPruebas\Families.ecore

3.4(JiglooYMofScript)\AuxParaPruebas\Persons.ecore

iglooYMofScript)\AuxParaPruebas\FamilyToPerson.ocl

looYMofScript)\AuxParaPruebas\FamilyToPerson.assl

7. Signature of the invocation to ASSL file:

mkSource2(3, 6, 3, 3)

:E\Eclipse3.4(JiglooYMofScript)\AuxParaPruebas\temp

```
checking structure...
checking invariants...
checking invariant (1) `Family::SRC_TRG_FatherSon2
  -> false : Boolean
Instances of Family violating the invariant:
  -> Set{@Family1,@Family2,@Family3} : Set(Family)
checking invariant (2) `Family::SRC_TRG_Female2Mo
checking invariant (3) `Family::SRC_TRG_Male2Fathe
checking invariant (4) `Family::SRC_TRG_MemberSiz
  -> false : Boolean
Instances of Family violating the invariant:
  -> Set{@Family1,@Family2,@Family3} : Set(Family)
checking invariant (5) `Family::SRC_TRG_MotherDau
  -> false : Boolean
Instances of Family violating the invariant:
  -> Set{@Family1,@Family2,@Family3} : Set(Family)
checking invariant (6) `Family::SRC_TRG_MotherDau
checked 6 invariants in 0.012s, 3 failures.
```

Tracts for EMF

Time for a Tool Demo!

The Future of Tracts

Next steps

- Incorporate existing works on the effective generation of input test cases, oracles, test inputs coverage
- Study tracts properties:
  - composability, subsumption, refinement,…
- Tracts for bi-directional transformations
- Improve engineering aspects
  - Visual specification of tracts
  - Diagnostics
  - Improve tool support
- Define libraries of tracts

Thanks!

Manuel Wimmer, Loli Burgueño, Lars Hamann, Martin Gogolla, Antonio Vallecillo,

Universidad de Málaga/Univertät Bremen

http://www.lcc.uma.es/~av

100 % Completed!

Universität Bremen

UNIVERSIDAD DE MÁLAGA