2II65 – Metamodeling and Interoperability

Authors:

1. Mihai Popovici 0924605
2. Rudiansen Gunawan 0924857

2014

Table of Contents

Task 1

a) For modeling communicating component, we extend the classical Petri Nets by introducing interface places. We make Generic Place class as the generalization class of Place, InputPort and OutputPort classes. OutputPort and InputPort are used to send/receive messages to/from partners. It is appropriate for modeling communicating component because in the communication process, we usually use ports as a medium for sending/receiving messages. The ports are used by both parties i.e. sender and receiver. These ports are indicated with specialization of place in Petri Nets.

We define the extended classical Petri Nets as N = [P,T,F,I,O,W,M_0) consists of Petri Nets [P,T,F,W,M_0] together with

- An interface (I ∪ O) ⊆ P defined as two disjoint sets I of InputPort place and O of OutputPort place such that •p = ∅ for any p ∈ I and p• = ∅ for any p ∈ O

In the initial marking and the final marking the interface places are not marked, i.e. m(p) = 0, for all p ∈ I ∪ O.

Here are the assumptions and constraints for the metamodel we created.

1. OutputPort is only connected to OutputArc class. It shall only has OutputArc coming from a transition and pointed to it.
2. InputPort is only connected to InputArc class. It shall only has InputArc which is an arc outgoing from it and pointing to a transition or more.

Below is the metamodel for the extended classical Petri Nets.

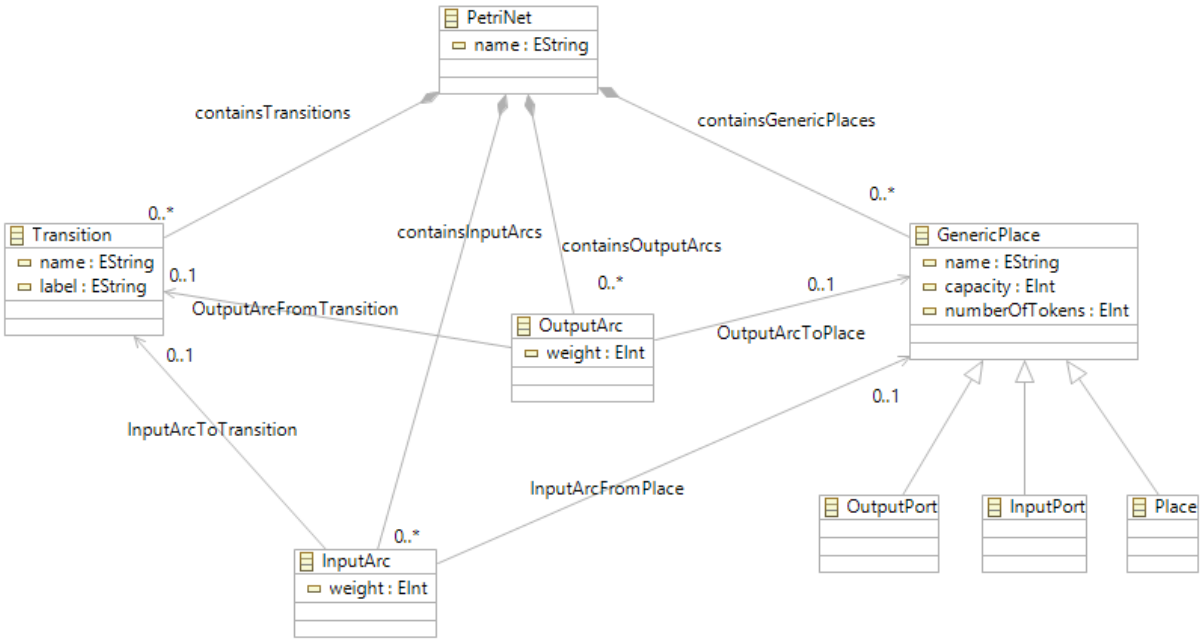

Figure 1.1 Extended Classical Petri Nets Metamodel for Modeling Communicating Component

We include *label* attribute in the Transition model element so that we could differentiate between internal and external transitions if two instance models of a metamodel would participate in the synchronous product transformation of two extended Petri Nets. The difference between internal transitions and external transitions is that the former do not have the label attribute initialized and the later do.

The source files for the graphical editor of the extended classical Petri Nets are sent separately.

b) We develop model transformation for transforming model from the extended classical Petri Net of task 1a) to the classical Petri Net using ATL below.

```
module ExtendedPN2ClassicalPN;
create OUT : PetriNets from IN : ExtendedPetriNets;

rule PN2PN {
   from
          pn1 : ExtendedPetriNets!PetriNet
   to
          pn2 : PetriNets!PetriNet (
                name <- pn1.name,
                containsTransitions <- pn1.containsTransitions,
                containsPlaces <- pn1.containsGenericPlaces,
                containsInputArcs <- pn1.containsInputArcs,
                containsOutputArcs <- pn1.containsOutputArcs
          )
}

rule InputArc2InputArc{
   from
          input_arc1:ExtendedPetriNets!InputArc
   to
          input_arc2:PetriNets!InputArc(
                weight <- input_arc1.weight,
                InputArcToTransition <- input_arc1.InputArcToTransition,
                InputArcFromPlace <- input_arc1.InputArcFromPlace

          )
}

rule OutputArc2OutputArc{
   from
          output_arc1:ExtendedPetriNets!OutputArc
   to
          output_arc2:PetriNets!OutputArc(
                weight <- output_arc1.weight,
                OutputArcFromTransition <-
output_arc1.OutputArcFromTransition,
                OutputArcToPlace <- output_arc1.OutputArcToPlace

          )
}
```

```
rule Place2Place{
   from

   place1:ExtendedPetriNets!Place(place1.refImmediateComposite().containsGener
icPlaces->includes(place1))
   to
          place2:PetriNets!Place(
                 name <- place1.name,
                 capacity <- place1.capacity,
                 numberOfTokens <- place1.numberOfTokens
          )
}

rule InputPort2Place{
   from

   place1:ExtendedPetriNets!InputPort(place1.refImmediateComposite().containsG
enericPlaces->includes(place1))
   to
          place2:PetriNets!Place(
                 name <- place1.name,
                 capacity <- place1.capacity,
                 numberOfTokens <- place1.numberOfTokens
          )
}

rule OutputPort2Place{
   from

   place1:ExtendedPetriNets!OutputPort(place1.refImmediateComposite().contains
GenericPlaces->includes(place1))
   to
          place2:PetriNets!Place(
                 name <- place1.name,
                 capacity <- place1.capacity,
                 numberOfTokens <- place1.numberOfTokens
          )
}

rule Transition2Transition{
   from

   transition1:ExtendedPetriNets!Transition(transition1.refImmediateComposite(
).containsTransitions->includes(transition1))
   to
          transition2:PetriNets!Transition(
                 name <- transition1.name
          )
}
```

The source files of ATL model transformation above are sent separately.

Task 2

a) In this extended classical Petri Nets for modeling resources, we introduce interface Place as the generalization of places. There is one additional 'Resource' class which is a specialization of 'GenericPlace' class as interface place. It is appropriate to model resources using Place because Place class' attributes represent the characteristics what are needed for resources. Resources require name, whether it is a role name for human resource or name for machine. numberOfTokens attribute represents the number of resources available for doing certain activities. While the number of resources needed for doing certain activities can be denoted by the weight of Arc.

Below is the metamodel of the extended classical Petri Nets for modeling resources.

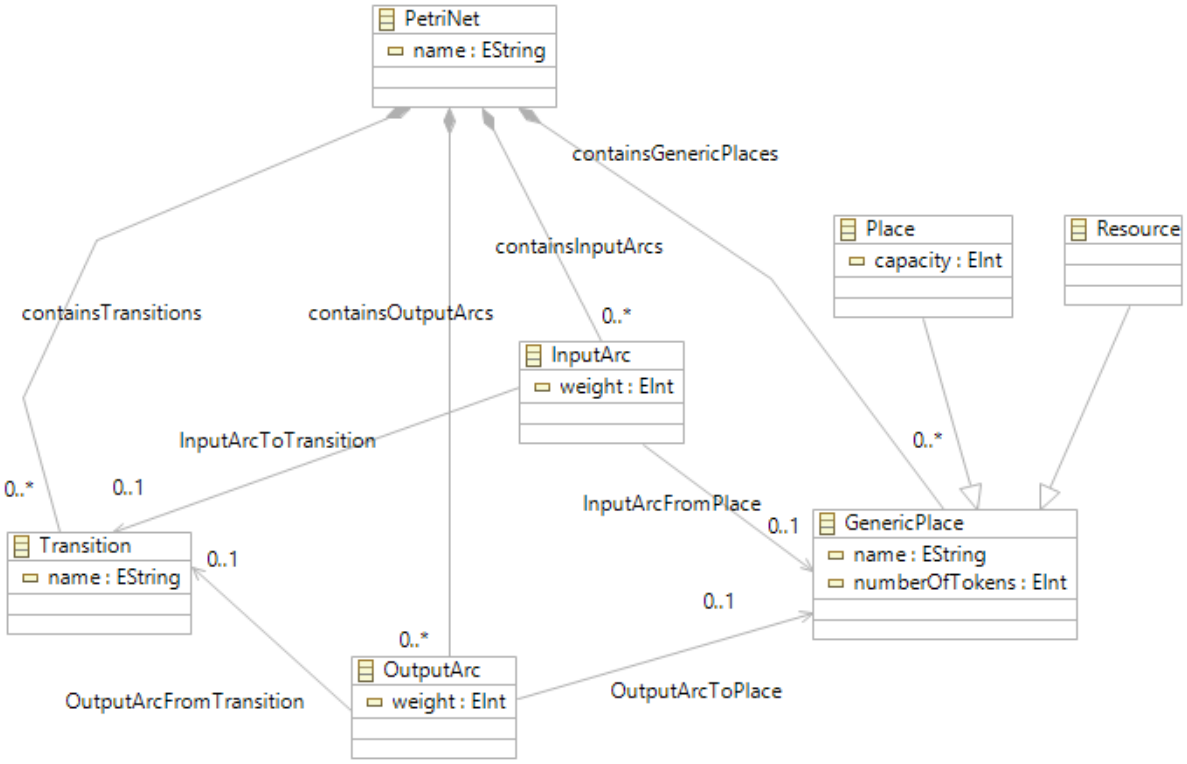

Figure 2.1 Extended Classical Petri Nets Metamodel for Modeling Resources

Here are the assumptions and constraints for the metamodel we created.

1. Resource can be connected with InputArc and OutputArc
2. Resource only has attributes name as role name and numberOfTokens as the number indicated the available resources. Resource does need capacity attribute.

b) Below is the metamodel for role model.

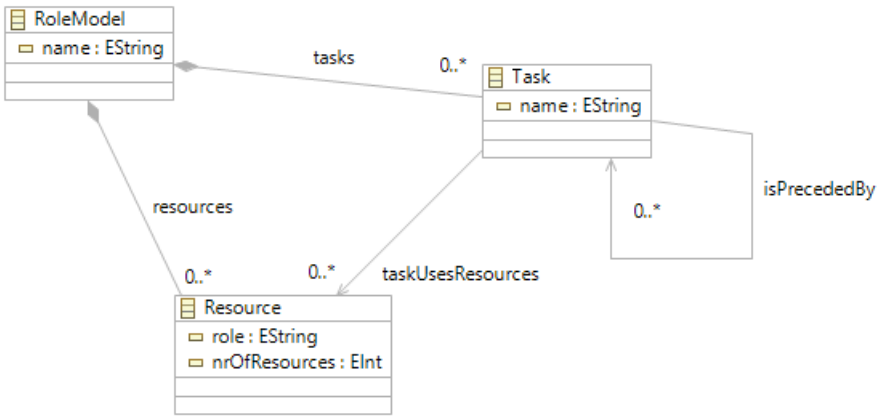

Figure 2.2 Role Model Metamodel

Role Model consists of Tasks and Resources. Every task requires resources to execute. It is indicated by the association of *taskUsesResources* with the cardinality 0..*. From this association, we know which resources (role) are necessary for which tasks. The association *isPrecededBy* depicts the relation of dependency between tasks. Task B, for instance, must be preceded by Task A in order to be able to be executed.

Below are the assumptions and constraints for the role model metamodel.

1. Role model can be an empty role model (without Resource and Task)
2. Resource only need two attributes i.e. role as a role name and nrOfResources as the number of available resource.

c) Below are the model transformation for building role model as defined in the task 2b) from the extended classical Petri Nets from task 2a) using ATL.

```
module PetriNet2RoleModel;
create OUT : RoleModel from IN : PetriNet;

helper context PetriNet!Transition def: places: Sequence(PetriNet!Place)=
   (PetriNet!InputArc.allInstances() -> select(r |
r.InputArcToTransition=self)) -> collect(e | e.InputArcFromPlace);

rule PN2RM {
   from
         pn : PetriNet!PetriNet
   to
         rm : RoleModel!RoleModel (
               name <- pn.name,
               tasks <- pn.containsTransitions,
               resources <- pn.containsGenericPlaces

         )
}

rule Resource2Resource{
   from

   resource1:PetriNet!Resource(resource1.oclIsKindOf(PetriNet!Resource))
   to
         resource2:RoleModel!Resource(
               role <- resource1.name,
               nrOfResources <- resource1.numberOfTokens
         )
}

rule Transition2Task{
   from
         transition:PetriNet!Transition
   to
         task:RoleModel!Task(
               name <- transition.name,
               isPrecededBy <- ( PetriNet!OutputArc.allInstances() ->
select(r | r.OutputArcFromTransition<>transition and transition.places ->
includes(r.OutputArcToPlace) ) ) -> collect(e | e.OutputArcFromTransition),
               taskUsesResources <- (((PetriNet!InputArc.allInstances() ->
select(r | r.InputArcToTransition=transition)) -> collect(p |
p.InputArcFromPlace))) -> select(r | r.oclIsKindOf(PetriNet!Resource))
               )
}
```

The source files for the graphical editor of the extended classical Petri Nets, for the role model metamodel and the ATL model transformation are sent separately.

Task 4

In the Declare, there are two main elements i.e. Activities, which is denoted by Rectangle and Constraint, which is denoted by arrow for connecting from one activity to another activity. In the Declare, constraints have many various types. There are twenty types of constraint e.g. Responded Existence, Co-Existence, Response, Precedence, Succession and many other types as we can see on the metamodel below. In the metamodel, these various type of constraints, we make it as specialization classes of Constraint.

Constraint has cardinality 0..1 for the constraint outgoing from activity class and 0..* for the constraint incoming to activity class.

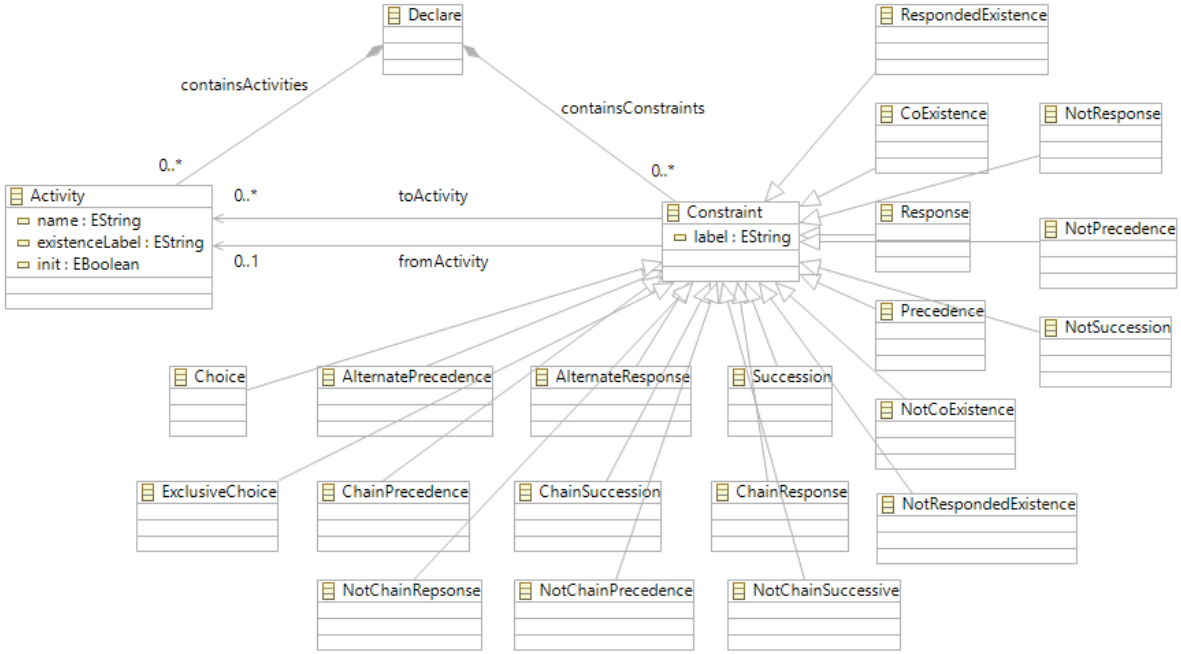

Figure 4.1 Declare Metamodel

Here are the assumptions and constraints for the Declare metamodel above.

1. Declare can be an empty Declare (without Activity and Constraint)
2. Every constraint has exactly one destination activity, except for Choice and ExclusiveChoice. It has more than one destination activities
3. Every constraint come from exactly one activity.

The source files for the graphical editor of the Declare metamodel are sent separately.