Explicit Transformation Modeling in AToM^3

Dieter De Hen

University of Antwerp, Middelheimlaan 1, B-2020 Antwerp, Belgium

Abstract

In this document we describe a practical implementation of explicit transformation modeling for traffic to Petri net transformations. We accomplish this by working in close conjunction with AToM^3[5], a tool for multi-paradigm modeling. By applying relaxation, augmentation and modification on both meta-models we obtain a traffic to Petri net transformation meta-model which we will use to compose transformation rules. We then implement a matching algorithm in Python to locate transformation rule patterns in our traffic model. Next we present a transformation algorithm which will perform left hand side to right hand side traffic to Petri net transformations. Finally we define a graph grammar which will specify in what order rules should be applied on a traffic model.

Keywords: explicit transformation modeling, model transformation, AToM^3

1. Introduction

This paper will be centered around a practical example transformation for traffic to Petri net models. A minimum of background is however needed around the concept of model transformations. Therefore we give a short definition of model transformations in section 2. In section 3 we dive into the explicit modeling of transformations. We construct traffic, Petri net and graph grammar meta-models and models. We also implement generic matching and transformation functions. In section 4 we present future work, in section 5 we conclude our findings and in section 6 we present related work.

2. Model Transformation

Model transformations generally play a key role in model driven development. Before we start explaining our implementation we should first define what we understand as model transformation. The concept is closely related to program transformation. Both have many practical uses but perhaps the most widely known is to generate lower-level models from higher-level models. The main idea is depicted in figure 1. We start from a source meta-model which we then use to create an instance of the source model. A transformation engine will then perform the rule based transformation on the source model by applying a transformation definition. The final result of this process is a target model which conforms to its corresponding meta-model.

Email address: `dieter.dehen@student.ua.ac.be` (Dieter De Hen)

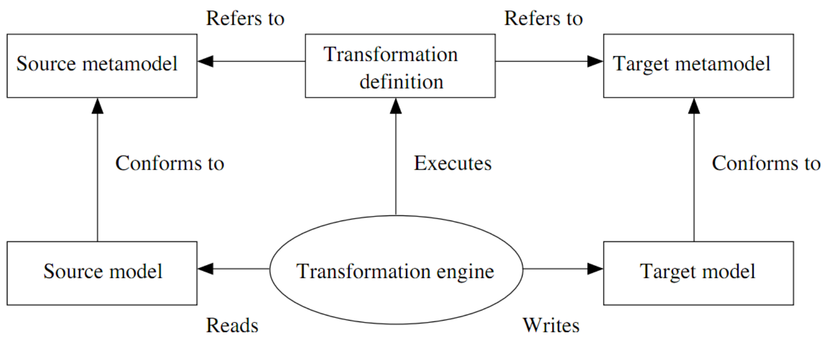


Figure 1: Concept of model transformation [1]

As models and meta-models are both typed graphs, we can transform them by applying graph rewriting. How we should perform the rewriting will need to be specified in the form of graph grammar models. Such a model can be most easily visualized as a set of rules. Each of the rules consists out of a left-hand-side (LHS) and a right-hand-side (RHS) graph. Rules will be matched on the source model. If a match between the LHS and a sub-graph of the source model is found the rule can be applied. The matched sub-graph in the source model is then replaced with the RHS of the rule. During the transformation process the intermediate models can be a blend of both the source and target notations. When a rule cannot be matched any more we proceed to the next rule and repeat this until all rules are visited. The execution of the graph grammar ends when no more rules can be matched. [6]

3. Explicit Transformation Modeling into Practice

In this section we present an explicit transformation modeling example. We will define traffic and Petri net meta-models in subsection 3.1. We create a transformation definition meta-model in subsection 3.2 and a graph grammar meta-model in 3.3. In 3.4 we construct a traffic model and in 3.5 we create the rule models which will be used by our graph grammar. Also we implement a generic matching and transformation algorithm, the two workhorses of our transformation engine. Finally in subsection 3.6 we show the result of applying our graph grammar on the traffic model.

3.1. The Traffic and Petri Net Meta-Model

As we discussed in section 2, we start with selecting a source meta-model and a target meta-model. A meta-model defines how valid models can be created. It shows the different components alongside with their connection multiplicities. By using these meta-models we will not need to rebuild a model from scratch. They promote re-usability, maintainability and validation [2]. To model the meta-model we use a meta-meta-model. Because the meta-meta-model needs to be sufficiently expressive enough we use the class diagram formalism included with AToM^3 to construct our meta-models. In the next two subsections we present both the traffic and Petri net meta-models.

3.1.1. Traffic Meta-Model

In figure 2 we present our traffic meta-model. We start with defining a *T_Place* which is an abstraction for a drivable thing. At all time a *T_Place* can only have one vehicle driving on it.

Two different *T_ Place* entities can be connected with each other with a *T_ Next* connection. Each *T_ Place* can have one incoming and one outgoing *T_ Next* connection. We currently can have one type of vehicle in our traffic model: A *T_ Car* which will only be able to drive on one place at a time. The most basic place our car will be able to drive on is a *T_ RoadSegment*. Since *T_ RoadSegment* extends from *T_ Place* it can also have *T_ Next* connections. Finally we can also split and join places by using the *T_ ForkSplit* and *T_ ForkJoin* places. A split can have one additional connection to a place and a join is allowed to have one extra incoming connection from a place.

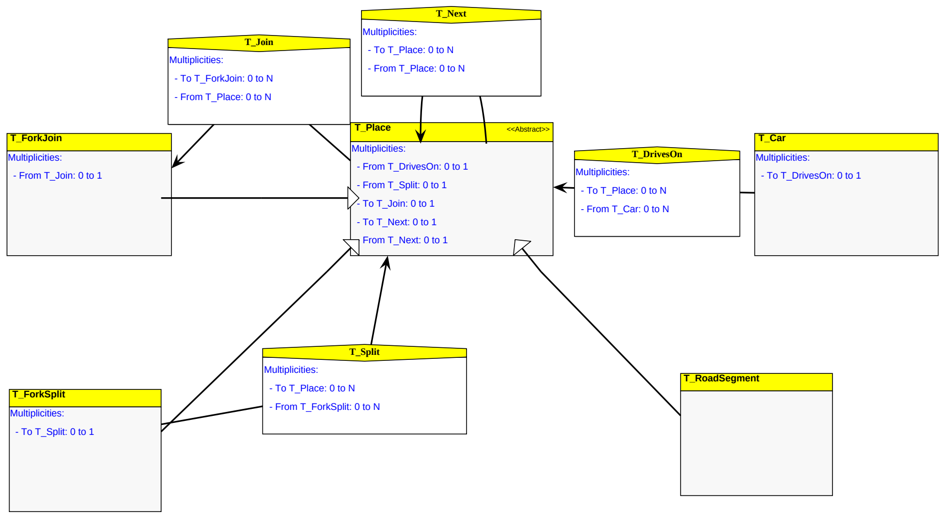


Figure 2: The traffic meta-model

One more thing we had to do is to add the ability to compile traffic models to ordinary Python[1] classes. In our transformation engine we will then easily be able to import the model and perform further operations on it. We implement our traffic compiler in Python to maintain compatibility with AToM^3 . Implementing the compiler in Python will allow us to invoke its functionality from within the AToM^3 user interface. The idea is to abstract away the compilation process behind a button. A user will then simply have to click this button to compile the model which can then easily be imported into a transformation engine (as can be seen in figure 6). Part of a compiled traffic model is shown in listing 1.

[1]Python: `http://python.org/`

```
"""
TrafficModel
"""

from ContainerFunctionality import *

class TrafficModel(object):

  def __init__(self):
    self.__model = ModelContainer()

    # Nodes

    self.__obj34 = ModelNode({'type':'T_ForkJoin'})
    self.__model.get_nodes().append(self.__obj34)

    self.__obj33 = ModelNode({'type':'T_Car'})
    self.__model.get_nodes().append(self.__obj33)

    self.__obj27 = ModelNode({'type':'T_RoadSegment'})
    self.__model.get_nodes().append(self.__obj27)

    # Shortened for readability...

    # Node connections

    self.__obj34.get_inConnections().append(self.__obj37)
    self.__obj34.get_inConnections().append(self.__obj42)
    self.__obj34.get_outConnections().append(self.__obj43)

    self.__obj33.get_outConnections().append(self.__obj35)

    self.__obj27.get_inConnections().append(self.__obj44)
    self.__obj27.get_outConnections().append(self.__obj38)

    # Shortened for readability...

  def get_model(self):
    return self.__model

```

Listing 1: A compiled traffic model

3.1.2. Petri Net Meta-Model

Our Petri net meta-model is shown in figure 3. It consists of a *PN_ Place* which can have a *name* and a certain number of *tokens*. *PN_ Place* nodes can be connected to a *PN_ Transition* which in its turn can have incoming and outgoing connections to instances of *PN_ Place*. This simple yet powerful formalism will be expressive enough to translate the logic behind our higher level traffic model.

3.2. A Traffic - Petri Net Transformation Formalism

The creation of a rule-based transformation formalism will allow us to specify how patterns in the source model will be translated to the target model. Generally speaking this boils down to

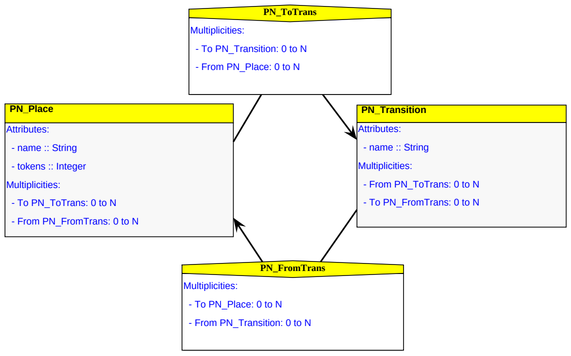


Figure 3: The Petri net meta-model

creating a pattern specification language. We can consider two popular approaches [4].

- A generic pattern specification language will be able to use the same generic for all input and output languages. Most tools using a generic pattern specification language use a UML object diagram inspired syntax.

- Customized pattern specification languages have some advantage over generic ones: They allow for customized visual representations which are adapted to the languages involved. Also a customized syntax will prevent the user from creating non-legal patterns.

Knowing this we will opt for a customized pattern specification language of which specifications can be checked for conformance. For convenience we chose to implement these conformance checks in Python. Ideally however, such constraints will be fully translated into a modeled action language [4].

To create a transformation formalism meta-model we cannot simply reuse the source and target meta-models. First of all to be able to specify useful patterns we must adapt the meta-models so that they support these, normally illegal, constructions. Secondly it should also be possible to save non-legal models to be worked on later. To accomplish this we perform relaxation, augmentation and modification on both the source and target meta-models. The resulting traffic - Petri net transformation meta-model is shown in figure 4.

Since we also need to be able to import traffic - Petri net models into our transformation engine we implement a traffic - Petri net compiler which compiles the model to an ordinary Python class.

Finally we also added an extra conformance check in Python in which constraints that are hard to define directly in the meta-model are verified. The first constraint is that there must exactly be one *TP_Separator* in a model. Secondly we check if all LHS entities are positioned on the left-hand-side of the separator and vice versa for RHS entities. Finally a check is done to make sure no colliding label attributes are found, i.e. two entities on one side of the separator can never have the same label.

3.2.1. Relaxation

A first step of relaxation will be to reduce all minimal connection multiplicities to zero. Raising all maximum multiplicities to unbounded would be an option to allow ill-formed results in intermediate steps. However in our traffic - Petri net transformation model we chose not to raise maximum multiplicities and work with generic links instead.

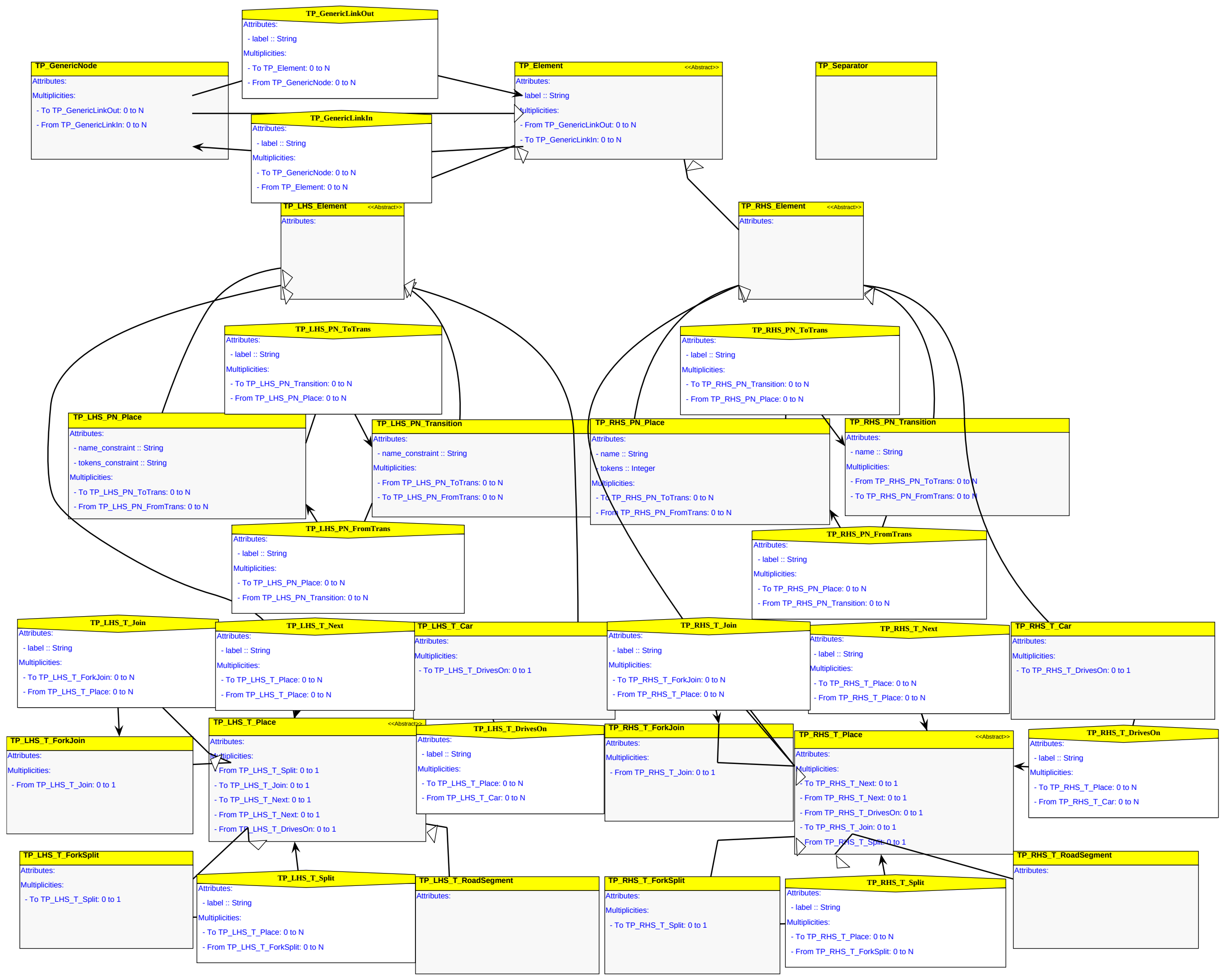

TP_GenericLinkOut
Attributes:
- label :: String
Multiplicities:
- To TP_Element: 0 to N
- From TP_GenericNode: 0 to N
TP_GenericNode
Attributes:
Multiplicities:
- To TP_GenericLinkOut: 0 to N
- From TP_GenericLinkIn: 0 to N
TP_Element
<<Abstract>>
Attributes:
- label :: String
Multiplicities:
- From TP_GenericLinkOut: 0 to N
- To TP_GenericLinkIn: 0 to N
TP_Separator
TP_GenericLinkIn
Attributes:
- label :: String
Multiplicities:
- To TP_GenericNode: 0 to N
- From TP_Element: 0 to N
TP_LHS_Element
<<Abstract>>
Attributes:
TP_RHS_Element
<<Abstract>>
Attributes:
TP_LHS_PN_ToTrans
Attributes:
- label :: String
Multiplicities:
- To TP_LHS_PN_Transition: 0 to N
- From TP_LHS_PN_Place: 0 to N
TP_RHS_PN_ToTrans
Attributes:
- label :: String
Multiplicities:
- To TP_RHS_PN_Transition: 0 to N
- From TP_RHS_PN_Place: 0 to N
TP_LHS_PN_Place
Attributes:
- name_constraint :: String
- tokens_constraint :: String
Multiplicities:
- To TP_LHS_PN_ToTrans: 0 to N
- From TP_LHS_PN_FromTrans: 0 to N
TP_LHS_PN_Transition
Attributes:
- name_constraint :: String
Multiplicities:
- From TP_LHS_PN_ToTrans: 0 to N
- To TP_LHS_PN_FromTrans: 0 to N
TP_RHS_PN_Place
Attributes:
- name :: String
- tokens :: Integer
Multiplicities:
- To TP_RHS_PN_ToTrans: 0 to N
- From TP_RHS_PN_FromTrans: 0 to N
TP_RHS_PN_Transition
Attributes:
- name :: String
Multiplicities:
- From TP_RHS_PN_ToTrans: 0 to N
- To TP_RHS_PN_FromTrans: 0 to N
TP_LHS_PN_FromTrans
Attributes:
- label :: String
Multiplicities:
- To TP_LHS_PN_Place: 0 to N
- From TP_LHS_PN_Transition: 0 to N
TP_RHS_PN_FromTrans
Attributes:
- label :: String
Multiplicities:
- To TP_RHS_PN_Place: 0 to N
- From TP_RHS_PN_Transition: 0 to N
TP_LHS_T_Join
Attributes:
- label :: String
Multiplicities:
- To TP_LHS_T_ForkJoin: 0 to N
- From TP_LHS_T_Place: 0 to N
TP_LHS_T_Next
Attributes:
- label :: String
Multiplicities:
- To TP_LHS_T_Place: 0 to N
- From TP_LHS_T_Place: 0 to N
TP_LHS_T_Car
Attributes:
Multiplicities:
- To TP_LHS_T_DrivesOn: 0 to 1
TP_RHS_T_Join
Attributes:
- label :: String
Multiplicities:
- To TP_RHS_T_ForkJoin: 0 to N
- From TP_RHS_T_Place: 0 to N
TP_RHS_T_Next
Attributes:
- label :: String
Multiplicities:
- To TP_RHS_T_Place: 0 to N
- From TP_RHS_T_Place: 0 to N
TP_RHS_T_Car
Attributes:
Multiplicities:
- To TP_RHS_T_DrivesOn: 0 to 1
TP_LHS_T_ForkJoin
Attributes:
Multiplicities:
- From TP_LHS_T_Join: 0 to 1
TP_LHS_T_Place
<<Abstract>>
Attributes:
Multiplicities:
- From TP_LHS_T_Split: 0 to 1
- To TP_LHS_T_Join: 0 to 1
- To TP_LHS_T_Next: 0 to 1
- From TP_LHS_T_Next: 0 to 1
- From TP_LHS_T_DrivesOn: 0 to 1
TP_LHS_T_DrivesOn
Attributes:
- label :: String
Multiplicities:
- To TP_LHS_T_Place: 0 to N
- From TP_LHS_T_Car: 0 to N
TP_RHS_T_ForkJoin
Attributes:
Multiplicities:
- From TP_RHS_T_Join: 0 to 1
TP_RHS_T_Place
<<Abstract>>
Attributes:
Multiplicities:
- To TP_RHS_T_Next: 0 to 1
- From TP_RHS_T_Next: 0 to 1
- From TP_RHS_T_DrivesOn: 0 to 1
- To TP_RHS_T_Join: 0 to 1
- From TP_RHS_T_Split: 0 to 1
TP_RHS_T_DrivesOn
Attributes:
- label :: String
Multiplicities:
- To TP_RHS_T_Place: 0 to N
- From TP_RHS_T_Car: 0 to N
TP_LHS_T_ForkSplit
Attributes:
Multiplicities:
- To TP_LHS_T_Split: 0 to 1
TP_LHS_T_Split
Attributes:
- label :: String
Multiplicities:
- To TP_LHS_T_Place: 0 to N
- From TP_LHS_T_ForkSplit: 0 to N
TP_LHS_T_RoadSegment
Attributes:
TP_RHS_T_ForkSplit
Attributes:
Multiplicities:
- To TP_RHS_T_Split: 0 to 1
TP_RHS_T_Split
Attributes:
- label :: String
Multiplicities:
- To TP_RHS_T_Place: 0 to N
- From TP_RHS_T_ForkSplit: 0 to N
TP_RHS_T_RoadSegment
Attributes:

Figure 4: The traffic - Petri net transformation model

3.2.2. Augmentation

To be used in a pattern specification language both our source and target meta-model need to be augmented with extra features that are required for the transformation. We added the prefixes *TP_LHS* and *TP_RHS* to traffic and Petri net entities to avoid type collisions. Another reason for this is that entities can have different attributes and multiplicities depending on being a LHS or a RHS entity. One extra motivation for adding extra info to the type names is that it will easily allow for conformance checking since a valid rule never should have a *TP_RHS* entity in its LHS and vice versa.

To clearly separate LHS from RHS entities we added a *TP_Separator*. Apart from being useful in conformance checking, adding a separator will also give us a clear visual distinction between what is LHS and what is RHS.

All entities (apart from the separator) inherit from the abstract entity *TP_Element* which has a label attribute. The label will be used by the transformation engine for identity matching on LHS and RHS entities.

Finally we added a *TP_GenericNode* to our meta-model to make it easier to formulate transformation rules. However we did not get this concept completely to work in AToM^3 and therefore had to make a switch to the GenericGraph meta-model included in the kernel of AToM^3.

3.2.3. Modification

We also did some modifications on the LHS and RHS entity attributes. It is important to understand that, when used in a LHS context, attributes depict the role of constraints. We therefore chose to append the suffix *_constraint* to the names of the attributes and change all the types to *String*. Changing the types to *String* makes it possible to define constrains such as *<ANY>* which will match any attribute value.

3.3. Graph Grammar Meta-model

A graph grammar, also called a graph rewriting system, consists out of a collection of rules. It describes in what order those rules should be applied on a source model to result in a target model. To construct such a graph grammar model we created a graph grammar meta-model. It consists out of one entity *TP_Rule* which accepts a *name* attribute and can be connected to itself. The graph grammar meta-model is shown in figure 5.

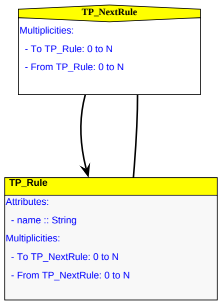


Figure 5: The graph grammar meta-model

Equivalent with all other meta-models we also implemented a graph grammar compiler. In listing 2 we show how a graph grammar can be obtained. On lines 3-15 we try to find a start entity for our grammar. A start entity has no incoming connections and is of type *TP_Rule*. If no start entities are found we select all *TP_Rule* entities in our graph grammar. To implement fairness we then randomly select a start entity out of the obtained set. In the *__buildGG* function we then recursively continue to build the grammar beginning from our start entity. We stop if we encounter a *TP_Rule* entity with no more outgoing connections. The result of the *getGG* function is then a list of *TP_Rule* entities defining the order in which rules must be applied to the source model.

3.4. The Traffic Model

The next step is to create a traffic model using the traffic meta-model we created in section 3.1. We intentionally kept the model simple to maintain clarity. As can be seen in figure 6, a car is driving on one of the six road segments and there is one split and one join segment. After creating

```python
def get_GG( self ):
  # Start nodes without in connection
  startNodes = []
  for s in self.__model.get_nodes():
    if s.get_parameters()['type'] == 'TP_Rule':
      inConnectionCount = 0
      for inConnection in s.get_inConnections():
        inConnectionCount = inConnectionCount + 1
      if inConnectionCount == 0:
        startNodes.append(s)
  # If no start nodes found, use all nodes in model.
  if len(startNodes) == 0:
    for s in self.__model.get_nodes():
      if s.get_parameters()['type'] == 'TP_Rule':
        startNodes.append(s)
  gg = random.sample(startNodes, 1)
  return self.__buildGG(gg)

def __buildGG( self , gg ):
  lastIndex = len(gg)-1
  if lastIndex >= 0:
    if gg[lastIndex].get_parameters()['type'] == 'TP_Rule'
            and len(gg[lastIndex].get_outConnections()) > 0:
      next = random.sample(gg[lastIndex].get_outConnections(), 1)
      if next[0].get_parameters()['type'] == 'TP_NextRule'
                  and len(next[0].get_outConnections()) > 0:
        # a nextrule can only be connectect to one rule
        if next[0].get_outConnections()[0].get_parameters()['type']
                    == 'TP_Rule':
          gg.append(next[0].get_outConnections()[0])
          return self.__buildGG(gg)
  return gg
```

Listing 2: Graph grammar functions

the model we compile it to a Python class by clicking the corresponding button. The compiled class will be written to the same directory the model is saved to.

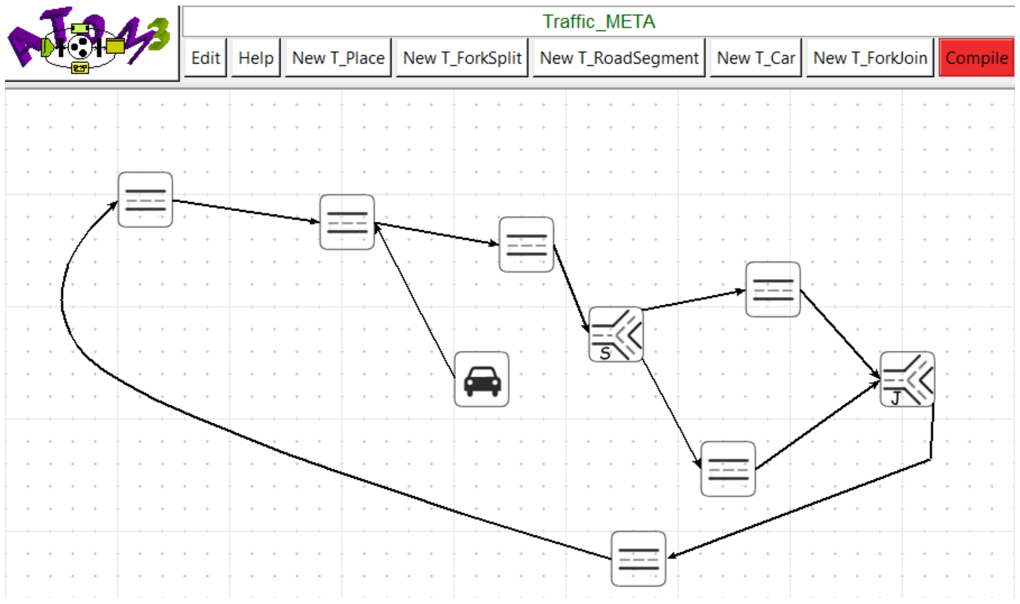


Figure 6: The traffic model

3.5. Transforming Traffic to Petri Net Models

The next task is the actual transformation of the Traffic model. However before being able to transform a model we will first need to create a graph grammar consisting of a prioritized list of rules to be matched on the source model. Next we implement a sub-graph matching algorithm which will try and find LHS matches in the source model. After we located such a match we will need to transform the model by replacing the match with the RHS.

3.5.1. Composing Rules

To compose these rules we use the traffic - Petri net transformation meta-model we created in subsection 3.2. We will not dive into every rule but instead discuss one example rule for transforming traffic join entities to their Petri net equivalence.

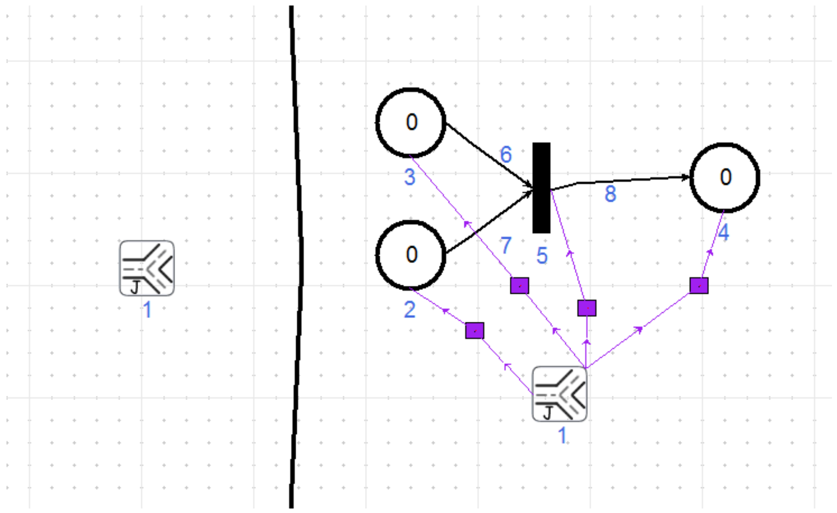
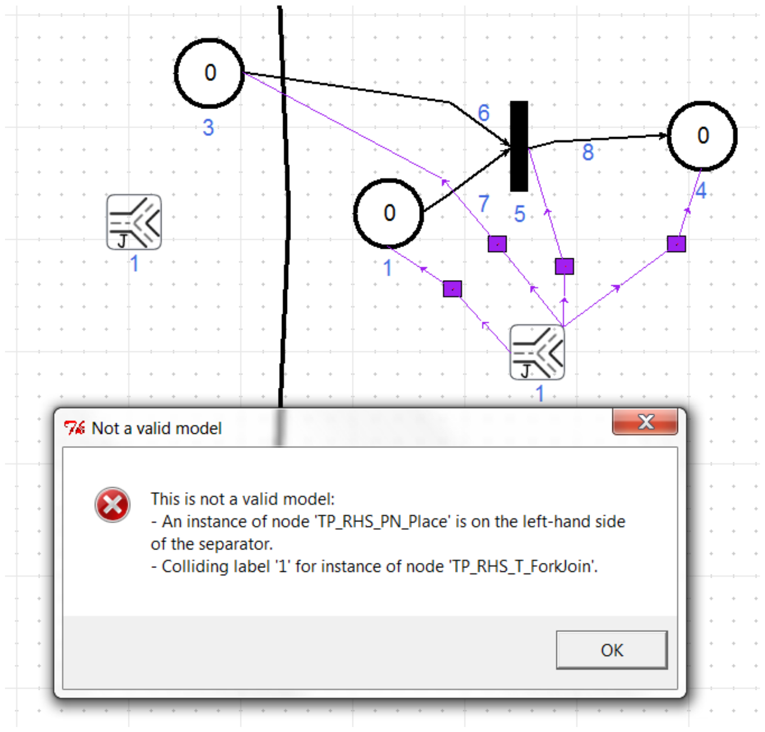


Figure 7: A rule for transforming join entities to Petri nets

The rule model shown in figure 7 describes the behavior of a *TP_Join* entity. It clearly separates the LHS from the RHS by using the *TP_Separator.* To help with the transformation process we connected the Petri net entities trough generic links with the join segment. Labels are given to all entities except to generic links and nodes. We are well aware that for optimal behavior both generic links and nodes should be labeled too. However since we used the GenericGraph meta-model included with AToM^3 this wasn't an option.

Checking if a rule is valid can be done by clicking on the validate button of the traffic - Petri net transformation formalism. The user will then immediately be presented with a pop-up message as can be seen in figure 8.



This is not a valid model:
- An instance of node 'TP_RHS_PN_Place' is on the left-hand side of the separator.
- Colliding label '1' for instance of node 'TP_RHS_T_ForkJoin'.

Figure 8: An invalid rule

For LHS attribute constraints wild-cards can be used. Constraints can be given the value $<ANY>$ to match any attribute value (figure 9). For RHS entities we could also allow attributes to have a $<COPIED>$ value to indicate that a value must be copied from the corresponding matched sub-graph entity. However this remains future work and is not implemented yet.

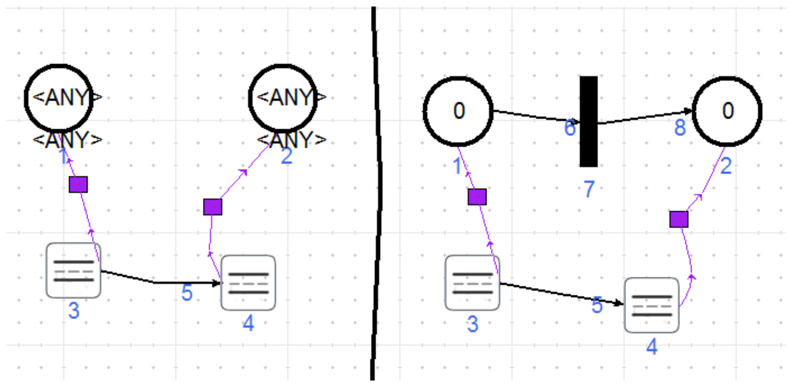


Figure 9: Wild-card attribute values for LHS entities

3.5.2. Constructing a Graph Grammar

By using the meta-model we defined in 3.3 we now construct our graph grammar model. The order in which rules are executed is specified by this model. In figure 10 we show our traffic - Petri net graph grammar which consists out of 8 rules.

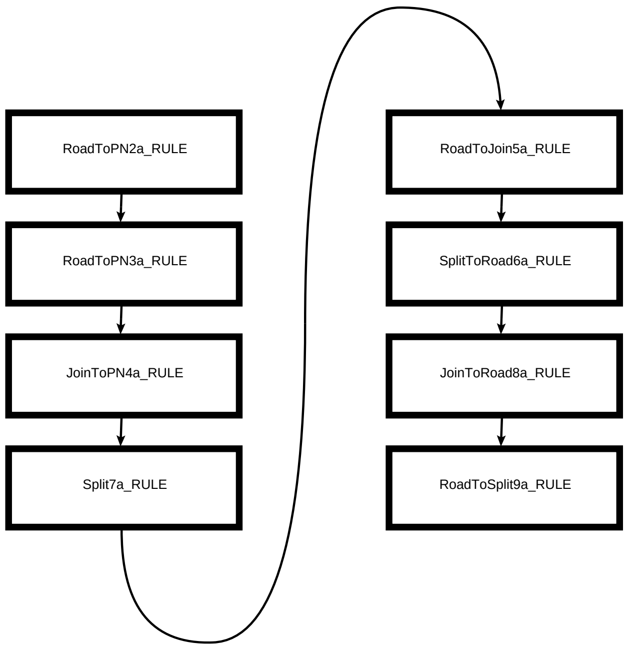


Figure 10: Traffic - Petri net graph grammar

3.5.3. Sub-Graph Matching and Model Transformation

A transformation engine will try to match the LHS graph of a rule. If match is found it will try to replace the matched sub-graph with the RHS of the matched rule. For the traffic - Petri net example we implemented a generic matching and transformation algorithm. A lot of study has gone into exact sub-graph matching which is in fact a NP-complete problem [3]. In our traffic - Petri net example we used a generic matching algorithm which will visit all sub-graph entities of the source model and try to recursively match them with the LHS graph. If the algorithm comes to a dead end it will back-off and try to match another candidate sub-graph. An advantage of using a generic algorithm is that we can reuse it in all of our rules. Every rule then has a match function which can be called to apply its matching algorithm on a given model.

```
def __matchNode(self, ruleNode, node):
  attributesMatched = True
  for attribute in ruleNode.get_parameters():
    # Ignore these
    if attribute == 'label'
            or ruleNode.get_parameters()[attribute] == '<ANY>'
            or ruleNode.get_parameters()[attribute] == '':
      continue
    nodeAttribute = attribute
    if attribute.endswith('_constraint'):
      nodeAttribute = attribute.split('_constraint')[0]
    if node.get_parameters().has_key(nodeAttribute):
      if str(node.get_parameters()[nodeAttribute])
                  != str(ruleNode.get_parameters()[attribute]):
        attributesMatched = False
        break
  if attributesMatched:
    # The rule's node and the models node match
    node.set_matched(True)
    ruleNode.set_matched(True)
    # Match the in_connections
    for inConnection1 in ruleNode.get_inConnections():
      if not inConnection1.get_matched():
        match = False
        for inConnection2 in node.get_inConnections():
          if not inConnection2.get_matched():
            if self.__matchNode(inConnection1, inConnection2):
              match = True
              break
        if match:
          continue
        node.set_matched(False)
        ruleNode.set_matched(False)
        return False
    # Match the out_connections
    for outConnection1 in ruleNode.get_outConnections():
      if not outConnection1.get_matched():
        match = False
        for outConnection2 in node.get_outConnections():
          if not outConnection2.get_matched():
            if self.__matchNode(outConnection1, outConnection2):
              match = True
              break
        if match:
          continue
        node.set_matched(False)
        ruleNode.set_matched(False)
        return False
    # Set the matching label for this node
    if ruleNode.get_parameters().has_key('label'):
      node.get_parameters()['label'] = ruleNode.get_parameters()['label']
    return True
  else:
    return False
```

Listing 3: Recursive matching function

Our recursive matching function is shown in listing 3. It takes as input parameters a *ruleNode* and a *node*. The first part of the matching is done on the attributes. On line 5-7 we make sure we do not match on the label attribute. We also ignore the attribute matching for wild-card values, which are in our case the empty string and the value <*ANY*>. From lines 5 till 16 we perform the actual matching of the attributes. Line 19-20 set flags on both the rule entity and the model entity. Continuing with line 21-48 we recursively match all incoming and outgoing connections. Finally on line 51 we set the label on the model entity.

```python
def __transform(self, model):
  # First make the newModelPart by using the rhs model
  newModelPart = copy.deepcopy(self.__rhsModel)
  # Make sure all connections are ok
  modelNodes = copy.copy(model.get_nodes())
  for newNode in newModelPart.get_nodes():
    if newNode.get_parameters().has_key('label'):
      for node in modelNodes:
        if node.get_parameters().has_key('label'):
          if newNode.get_parameters()['label']
              == node.get_parameters()['label']:
            # Start replacing connections
            inConnections = copy.copy(node.get_inConnections())
            for inConnection in inConnections:
              if not inConnection.get_matched():
                newNode.get_inConnections().append(inConnection)
                inConnection.get_outConnections().remove(node)
                inConnection.get_outConnections().append(newNode)
            outConnections = copy.copy(node.get_outConnections())
            for outConnection in outConnections:
              if not outConnection.get_matched():
                newNode.get_outConnections().append(outConnection)
                outConnection.get_inConnections().remove(node)
                outConnection.get_inConnections().append(newNode)
            model.get_nodes().remove(node)
  # Insert newModelPart in the model
  model.get_nodes().extend(newModelPart.get_nodes())
  # Remove all remaining old parts from the model
  # (labels that are in lhs but not in rhs)
  modelNodes = copy.copy(model.get_nodes())
  for node in modelNodes:
    if node.get_matched():
      model.get_nodes().remove(node)
    elif node.get_parameters().has_key('label'):
      del node.get_parameters()['label']
    inConnections = copy.copy(node.get_inConnections())
    for inConnection in inConnections:
      if inConnection.get_matched():
        node.get_inConnections().remove(inConnection)
    outConnections = copy.copy(node.get_outConnections())
    for outConnection in outConnections:
      if outConnection.get_matched():
        node.get_outConnections().remove(outConnection)
```

Listing 4: Transformation function

In listing 4 we show our transformation function. On line 3 we instantiate the new model part by making a copy of the RHS graph. On lines 5-25 we set the connections to our newly created model part. We also remove old entities of the model who have matching labels with RHS entities. On line 27 we continue with inserting the new sub-graph in the model. Finally on lines 30-43 we further clean up the new model by removing and resetting temporary attributes which were needed in the transformation process.

```
def match(self, model):
  nac = False
  for lhsRuleNode in self.__lhsModel.get_nodes():
    for node in model.get_model().get_nodes():
      nacMatched = False
      # Try to match the NAC
      for rhsRuleNode in self.__rhsModel.get_nodes():
        if self.__matchNode(rhsRuleNode, node):
          nacMatched = True
          nac = True
        for n in model.get_model().get_nodes():
          n.set_matched(False)
          if n.get_parameters().has_key('label'):
            del n.get_parameters()['label']
        for n in self.__rhsModel.get_nodes():
          n.set_matched(False)
        if nacMatched:
          break
      if nacMatched:
        continue
      # Match the LHS
      if self.__matchNode(lhsRuleNode, node):
        for n in self.__lhsModel.get_nodes():
          n.set_matched(False)
        print 'LHS rule matched'
        self.__transform(model.get_model())
        print 'Model transformed'
        return True
      for n in self.__lhsModel.get_nodes():
        n.set_matched(False)
  if nac:
    print 'NAC matched'
  else:
    print 'No match for LHS rule'
  return False
```

Listing 5: Main matching function

The recursive matching function is used in the matching on NAC and LHS graphs. In listing 5 our main matching function is shown. It takes a model as input parameter and will transform the model if the LHS was matched.

```
def run(self):
  atleastOneRuleMatched = True
  while atleastOneRuleMatched:
    gg = self.__gg.get_GG()
    atleastOneRuleMatched = False
    for ruleNode in gg:
      exec "from " + ruleNode.get_parameters()['name'] + " import *"
      rule = eval(ruleNode.get_parameters()['name'] + '()')
      print '--- Matching rule ---'
      print rule
      if rule.match(self.__model):
        atleastOneRuleMatched = True
        break
```

Listing 6: Main function

The final step in the transformation process is to apply our graph grammar on the model. We iterate over the rules of the graph grammar and try to apply each rule. If a rule is matched we restart the process until no more rules match. The function performing this logic is shown in listing 6.

```
>python Transform.py
-- Matching rule -- <RoadToPN2a_RULE.RoadToPN2a_RULE object at 0x01FBBED0>
LHS rule matched
Model transformed
-- Matching rule -- <RoadToPN2a_RULE.RoadToPN2a_RULE object at 0x01FBBF30>
LHS rule matched
Model transformed
-- Matching rule -- <RoadToPN2a_RULE.RoadToPN2a_RULE object at 0x01FBBFD0>
LHS rule matched
Model transformed
...
-- Matching rule -- <RoadToPN2a_RULE.RoadToPN2a_RULE object at 0x02024BD0>
NAC matched
...
-- Transformation done --
```

Listing 7: Transformation output

If we now execute our Transformation.py file which binds everything together we get the output in listing 7.

3.6. The Result

The result of the transformation process is shown in figure 11. It shows the traffic model we defined in 3.4 along with its Petri net equivalent. Note that the result is still a mix of both Petri net, traffic and generic graph entities. Traffic and generic graph entities can however easily be removed by declaring additional rules.

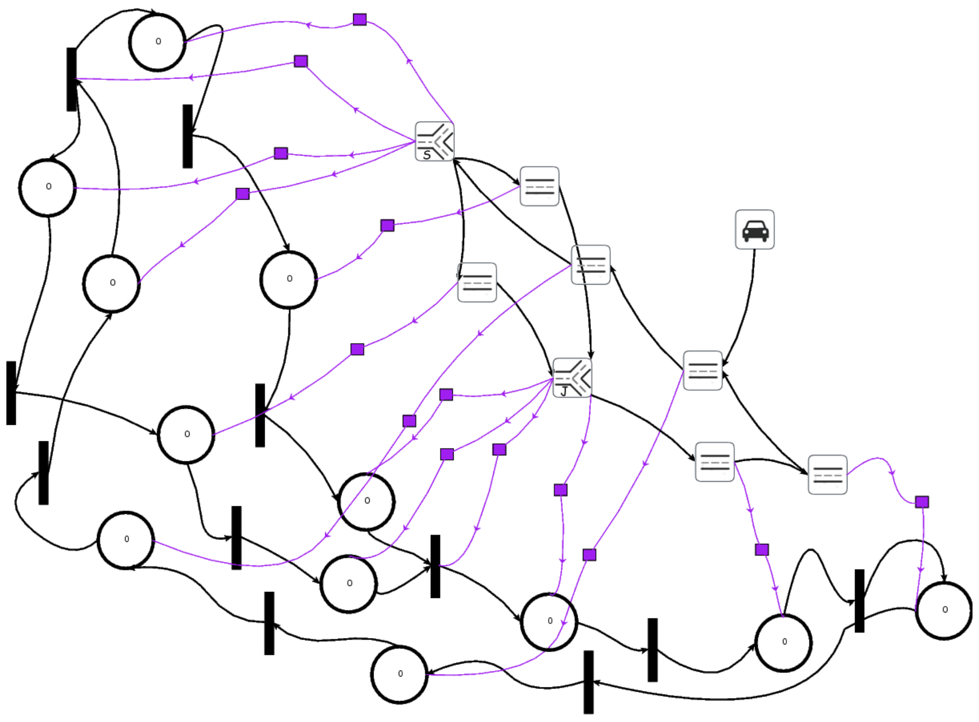


Figure 11: The transformed traffic model

4. Future Work

In this study we do not yet fully match on attribute values. To support this each RHS attribute should have a way to specify if its value needs to be <*COPIED*> from the matched sub-graph or if its value is <*SPECIFIED*> in the RHS attribute itself.

In some cases it can be necessary to have labeled generic entities. Therefore another point of focus for the future is to enable the use of our self-modeled *TP_ GenericNode* and *TP_ GenericLink* instead of the GenericGraph meta-model included with the AToM^3 kernel.

5. Conclusion

We implemented explicit transformation modeling for traffic to Petri net transformations. In our study we worked in close conjunction with AToM^3. The results are however not bound to this specific system. We created a transformation formalism by applying relaxation, augmentation and modification on both target and source models. We then used this formalism to compose transformation rules for traffic to Petri net models. We continued with creating a graph grammar which defined the order in which rules must be applied. Finally we implemented our matching and transformation algorithms which are the workhorses in the transformation process.

6. Related Work

In [4] a proposal is done to explicitly model transformation definitions. Based on the components of relaxation, augmentation, and modification a cost-effective customized pattern specification language can be created. Due to its modularity explicitly modeling transformations allows for easy addition of new behavior. Finally the concept of higher order transformations is explained as ample motivation for the explicit modeling of transformations.

[1] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[2] J. De Lara, H. Vangheluwe, and M. Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in atom 3. *Software and Systems Modeling*, 3(3):194–209, 2004.

[3] M.R. Garey and D.S. Johnson. *Computers and intractability*, volume 174. Freeman San Francisco, CA, 1979.

[4] T. Kuhne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer. Explicit transformation modeling. *Models in Software Engineering*, pages 240–255, 2010.

[5] J. Lara and H. Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, pages 174–188. Springer-Verlag, 2002.

[6] H. Vangheluwe and J. de Lara. Computer automated multi-paradigm modelling for analysis and design of traffic networks. In *Proceedings of the 36th conference on Winter simulation*, pages 249–258. Winter Simulation Conference, 2004.