# Model Transformations

## *Transformation of Simplified UML Model Into Simplified Rdbms Model Using AToM³ Meta-modeling Tool*

Michal Antkiewicz
mantkiew@swen.uwaterloo.ca

## Abstract

Model to model transformations are the part of the Model Driven Architecture, an initiative of Object Management Group. Many model transformation tools have been created so far and the transformation of the UML model to the Relational Database model is one of the most representative examples used for validating the tools. In this paper I present details and results of the model transformation project realized using AToM³ Meta-modeling tool. I also present the classification of the tool's features according to the model transformation tools taxonomy proposed by Czarnecki and Helsen in [CH03].

## 1 Introduction

The paper is organized as follows. Section 2 presents the AToM³ tool. Section 3 contains the requirements of the project. Section 4 describes the Entity-Relationship formalism used for specifying meta-models. Section 5 describes the common UML and Rdbms meta-model. Section 6 describes model transformations and a solution to the project – Uml2Rdbms transformation. Section 7 presents transformation execution modes. Section 8 presents the results of execution of Uml2Rdbms on an example model. Section 9 is a classification of a tool. And the last section 10 presents missing features and drawback of the tool encountered during the project.

In Appendix A, I describe how to install and run transformation in AToM³.

## 2 AToM³ Meta-modeling Tool

The name AToM³ is a shorthand of "A Tool for Multi-formalism and Meta-Modelling". The tool was developed at the Modeling, Simulation and Design Lab in the School of Computer Science of McGill University. The two main tasks of AToM³ are meta-modeling and model transforming [ATOM3].

Meta-modeling refers to the process of creating a formal description of some modeling language. AToM³ uses Entity-Relationship formalism for describing meta-models.

Some meta-models available with distribution are:

- Entity-Relationship
- Deterministic and nondeterministic FSA
- Petri Nets

- Data Flow Diagrams
- Structure Charts.

While constructing a meta-model, AToM³ allows the modeler creating of custom graphical representations of entities and relationships, as well as specification of constraints and cardinalities.

Model transforming refers to the process of translating a model conforming to some meta-model to another model, possibly conforming to different meta-model, using transformation rules.

As models and meta-models in AToM³ are represented as graphs, transformations are realized by pattern matching and graph rewriting.

AToM³ is written entirely in Python and Python is the language used for specification of any conditions and actions in models and transformations.

# 3 Project: Uml to Rdbms Transformation

The task of the project is to implement a model transformation from simplified UML model into simplified Relational DB model.

UML model has classes, attributes and associations. Rdbms model has tables, columns, primary and foreign keys. The UML and Rdbms meta-model is described in detail in Section 5.

The Uml2Rdbms transformation has to fulfill the following requirements.

Each class with meta-attribute *kind* equal to *persistent* should be mapped into a table. The new table's name is a concatenation of prefix “t_” and class' name.

Attributes of primitive type are mapped into columns.

Attributes of type class are not mapped into columns. Instead, columns are added for each primitive type attribute of a mapped attribute's class recursively. The new column's name is a concatenation of the attribute's class name, “_”(underscore) and primitive attribute's name.

Attributes with meta-attribute *kind = primary* are collected in primary key of the table.

Instead of attributes of type class with meta-attribute *kind = primary*, columns created from primitive type attributes of the class are collected in primary key of the table.

Directed association between two persistent classes is mapped into foreign key of the table created from source class of the association.

Foreign key contains columns added for each attribute with meta-attribute *kind = primary* in destination class. The new column's meta-attribute *kind := foreign* and new column's name is a concatenation of association role , “_” and primary attribute's name.

There should be only one foreign key and one primary key.

# 4 Entity-Relationship Formalism in AToM³

In AToM³ meta-models can be build from entities and relationships. The description of both entity and relationship consists of:

- name,
- attributes,
- constraints,
- cardinalities,
- appearance.

Attributes can be primitive types like ATOM3Integer or ATOM3String as well as Lists and Enums.

Constraints are Python expressions that evaluate to true or false. Can be used for example to check if attribute values are correct. If any constraint is violated the entity or relationship cannot be created.

Cardinalities specify possible entity and relationship configurations. Cardinalities are of the form:

```
<name> dir=[Source|Destination], <min>, <max>
```

For entity E, it means that E can be either a source or a destination for *min* to *max* relationships *name*

For relationship R, it means that R can have *min* to *max* entities *name* connected to its source or destination.

Appearance is created using built-in graphical editor.

Entity's shape, colors, position of attributes can be specified. If an entity participates in relationships it needs connectors. The graphical representation may be different according to some constraints. For example visibility or color may depend on value of an attribute.

For relationship the appearance can be specified for first link, first segment, center, second segment and second link.

# 5 Uml and Rdbms Meta-model

Figure 1 presents common UML and Rdbms meta-model. The model contains four entities: Class, Attribute, Table, Column and five relationships: contains, type, association, traceability, consistsOf.

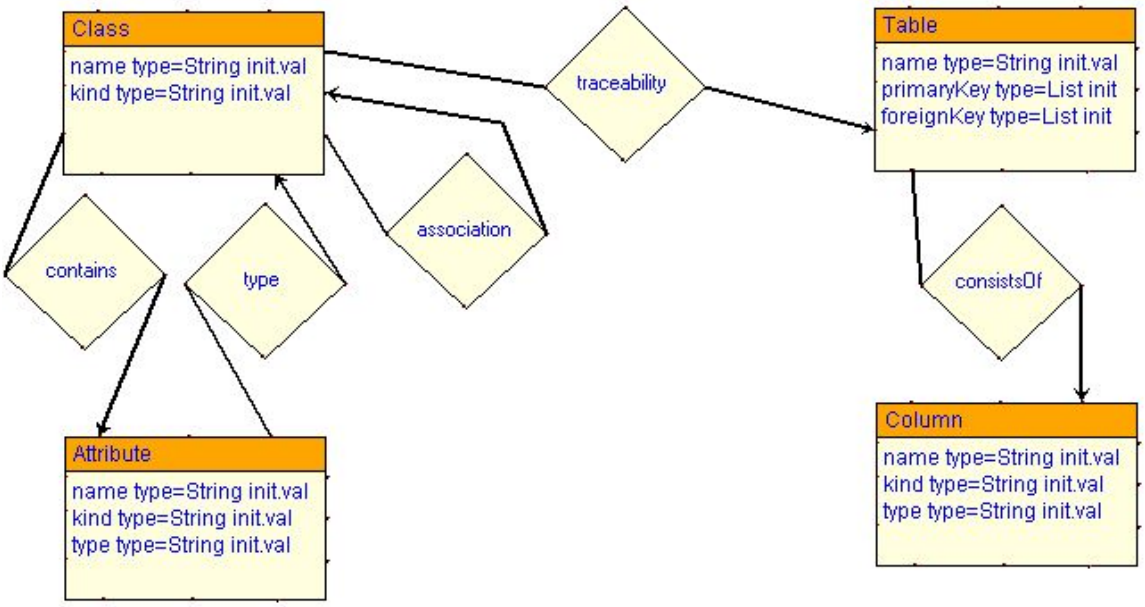


*Figure 1 Common Uml and Rdbms Meta-model*

## *5.1 Entity Class*

The entity Class has two attributes *name* and *kind* of type ATOM3String. *Kind* can be either “”(empty string) or “primary”.

Class has following cardinalities:

– contains dir=Source, 0..N,

- traceability dir=Source, 0..1,
- type dir=Destination, 0..N,
- association dir=Source, 0..N,
- association dir=Destination, 0..N.

There are no constraints.

The appearance of a class is presented on Figure 2. Labels <*name*> and <*kind*> indicate placement of values of attributes *name* and *kind* respectively. Red dots depict connectors.

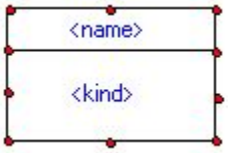


*Figure 2 Appearance of a Class*

## 5.2 Entity Attribute

Attribute has three attributes: *name*, *kind* and *type*, all of type ATOM3String.

*Kind* can be either “”(empty string) or “primary”.

*Type* can be either “Integer”, “String” or “Class”.

Attribute has following cardinalities:

- contains dir=Destination, 1..1,
- type dir=Source, 0..1,

There are no constraints.

The appearance of an attribute is presented on Figure 3.


*Figure 3 Appearance of an Attribute*

## 5.3 Entity Table

Table has three attributes: *name* of type ATOM3String and *primaryKey, foreignKey* both are lists of ATOM3Strings. *PrimaryKey* and *foreignKey* contain names of columns.

Table has following cardinalities:

- contsistsOf dir=Source, 0..N,
- traceability dir=Destination, 0..1,

There are no constraints.

The appearance of a table is presented on Figure 4.

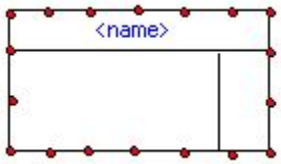


*Figure 4 Appearance of a Table*

Unfortunately, AToM$^3$ does not display list attributes.

## 5.4 Entity Column

Column has three attributes: *name*, *kind* and *type*, all of type ATOM3String.

*Kind* can be either “”(empty string) or “primary”.

*Type* can be either “NUMBER” for “Integer” or “VARCHAR” for “String”.

Column has following cardinalities:

– consistsOf dir=Destination, 1..1.

There are no constraints.

The appearance of a column is presented on Figure 5.


*Figure 5 Appearance of a Column*

## 5.5 Relationship contains

Contains has no attributes and no constraints.

Cardinalities are:

- Class, dir=Destination , 1..1,
- Attribute, dir=Source, 1..1.

The appearance of a contains is presented on Figure 6.

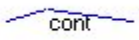


*Figure 6 Contains*

## 5.6 Relationship type

Type has no attributes and no constraints.

Cardinalities are:

- Attribute, dir=Destination , 1..1,
- Class, dir=Source, 1..1.

The appearance of a type is presented on Figure 7.

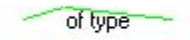


*Figure 7 Type*

## 5.7 Relationship association

Association has two attributes *name* and *role* of the association end.

Cardinalities are:

- Class, dir=Source, 1..1,
- Class, dir=Destination, 1..1.

There are no constraints.

The appearance of an association is presented on Figure 8.


*Figure 8 Association*

## 5.8 Relationship traceability

Traceability has no attributes and no constraints.

Cardinalities are:

- Class, dir=Destination , 1..1,
- Table, dir=Source, 1..1.

The appearance of a traceability is presented on Figure 9.



*Figure 9 Traceability*

## 5.9 Relationship consistsOf

ConsistsOf has no attributes and no constraints.

Cardinalities are:

- Table, dir=Destination , 1..1,
- Column, dir=Source, 1..1.

The appearance of a consistsOf is presented on Figure 10.


*Figure 10 ConsistsOf*

I don't know why cardinalities in relationships have opposite directions than on the diagram. AToM³ does not allow changing that. For example in consistsOf relationship on the diagram Table is the source and in cardinality Table is the destination. One possible explanation is that entities and associations are connected

by connections. Indeed, the source of the relationship is connected to the destination of the connection from the entity.

Now, when Uml2Rdbms meta-model is ready, AToM$^3$ generates (by Model->Generate Code command) the meta-model file, that can be opened later (by File->Open meta-model command) when creating models conforming to that meta-model.

The meta-model is created in Entity-Relationship (ER) formalism, hence it is saved as (by convention):

- UmlRdbms_ER_mdl.py

AToM$^3$ generates:

- Uml2Rdbms.py – main meta-model file
- ASG_UmlRdbms.py
- UmlRdbms_MM.py

For each entity and relationship <*name*>:

- *name*.py – description
- graph_*name*.py – appearance

When opening a meta-model (by File->Open meta-model) these files are simply imported to the AToM$^3$ application code because they also contain Python code.

# 6 Model Transformations in AToM$^3$

A model transformation in AToM$^3$ consists of:

- Initial Action,
- Final Action,
- Set of rules.

All actions, conditions have to be written in Python.

Because models in AToM$^3$ are represented as graphs, each rule defines how to graph-rewrite left hand side (LHS) to right hand side (RHS).

- LHS is a pattern which is matched against model being transformed,
- RHS is a graph that is inserted into the model instead of a matched subgraph.

The complete definition of a rule consists of: Name, Order, LHS, RHS, Condition and Action.

During transformation, rules are executed according to their order. Lower order rules are tried before higher order rules. If none of the rules can be applied (executed) transformation ends.

When rule is tried, first the condition is evaluated. If it evaluates to true (returns true) then LHS pattern is matched against model. If LHS was matched, the part of the model is substituted by RHS and finally action is executed.

If an entity or relationship was in LHS and is not in RHS, it will be deleted, similarly if it was not in LHS and it is in RHS a new element will be created.

Following the project requirements I have created the transformation Uml2Rdbms (Figure 11). It has six rules:

- Class2Table_create – creates a new table for persistent class and traceability link between them,
- Class2Table_extend – adds columns to the table from primitive type attributes in the class,

- ClassAttr2Table_create – if the type of the attribute is Class, this rule adds a traceability link between that class and a table, this allows recursion,
- ClassAttr2Table_extend – recursively add columns for primitive type attributes of a class,
- Assoc2FKey – creates foreign key in table corresponding to source class,
- CleanTraceability – removes all traceability links after main transformation is finished.

Figure 11 presents a dialog box used for editing Uml2Rdbms transformation. On the right side of each rule the number indicates order of the rule.

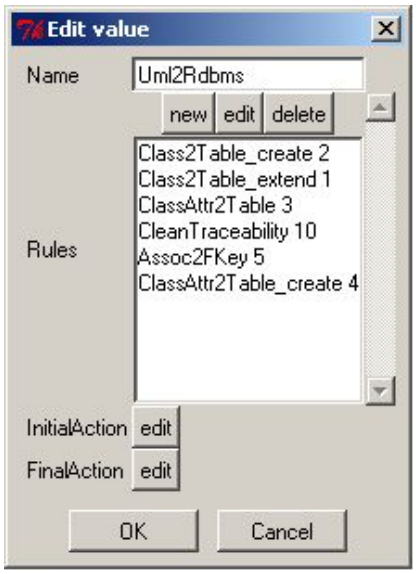


*Figure 11 Uml2Rdbms Transformation*

## 6.1 The idea of the Uml2Rdbms transformation

First, for each persistent class C a corresponding table T has to be created. Then primitive type attributes of C are mapped into columns – rules Class2Table_create and Class2Table_extend.

Then class CT type attribute A has to be processed. Attributes of class CT are mapped into columns of table T instead of attribute A. Because a traceability link between CT and T is added, class type attributes can be processed recursively – rules ClassAttr2Table_create and ClassAttr2Table_extend.

Then, when all classes and all attributes have been already processed associations can be mapped into foreign keys – rule Assoc2FKey. The order of Assoc2FKey is lower then previous rules, so it can be executed only if none of them can be executed.

Then after all the work has been done, the traceability link can be removed – rule CleanTraceability.

The following subsections describe the Uml2Rdbms transformation in detail.

## 6.2 Initial Action

In AToM$^3$ there is no scheduling in a sense “execute each rule on a given matching only once”. There is no “NOT matching” either.

The only possibility to ensure, that there will be only one table and traceability link added is to record somehow, that the given class has already been processed.

Also information about each attribute that has been mapped to a column needs to be recorded, to ensure that only one column for each attribute is created.

The same way, columns that have been copied to create foreign key need to be recorded.

To do that, in Initial Action of the Uml2Rdbms transformation contains:

- declaration of lists of processed elements

```
self.rewritingSystem.procClasses = ATOM3List([1, 1, 1, 0], ATOM3String)
self.rewritingSystem.procAttrs = ATOM3List([1, 1, 1, 0], ATOM3String)
self.rewritingSystem.procForeignCols = ATOM3List([1, 1, 1, 0], ATOM3String)
```

- processed element's state checkers

```
self.rewritingSystem.isProcClass =
  lambda cName: [] != filter (
    lambda cn: cn.toString() == cName,
    []+self.rewritingSystem.procClasses.getValue()
  )
```

Functions *isProcAttr* and *isProcForeignCol* are similar.

- add element to processed list functions

```
self.rewritingSystem.addProcClass =
  lambda cName: self.rewritingSystem.procClasses.setValue(
    [ATOM3String(cName)]+self.rewritingSystem.procClasses.getValue()
  )
```

Functions *addProcAttr* and *addProcForeignCol* are similar.

The only advantage of this solution is simplicity. It has many disadvantages:

- attribute names have to be unique,
- class names have to be unique,
- a class cannot be the target for more than one associations, because foreignKey in a source class is created only once – column names are added to *procForeignCols* list.

However this simple implementation is good enough to evaluate the AToM$^3$ tool and for this project.

### 6.2.1 Usage

These functions will be accessible in rule's Python code by following qualifier:

```
self.graphRewritingSystem.
```

Why? In AToM$^3$ there's no global namespace for creating global variables and functions that can be used throughout the transformation. Instead, they have to be added to the GraphRewritingSystem object which is visible everywhere.

Unfortunately from the level of Initial and Final action it is accessible by transformation's attribute

```
rewritingSystem
```

And from the level of the rule's actions and conditions by rule's attribute

```
graphRewritingSystem
```

This is certainly because AToM$^3$ was designed as a prototype or as a proof of concept tool.

## 6.3 LHS and RHS notation

LHS and RHS look like normal model with some differences:

- small, light gray numbers over an entity or relationship are GGLabels. GGLabels are unique and are used to refer to a particular element,
- in the LHS we specify the conditions on attributes. There are three possibilities:
  - <ANY> - matches to any value
  - *some_value* - matches only to "some_value"
  - specified – matches when specified constraint is satisfied.
- In the RHS we specify the new values of attributes. There are also three possibilities:
  - <COPIED> – value is the same as in LHS,
  - *some_value* – value is "some_value",
  - specified – value is returned by AttrSpecify method.

In the description of rules I use the notation

```
<ename>(n).<aname>.AttrSpecify
```

which means, that the value of the attribute *aname* of the entity *ename* with GGLabel *n* is specified by return value of the AttrSpecify function with given code.

## 6.4 Programming idioms

One of heavily used programming idiom is retrieving a model object that was matched with a node in LHS.

To get the node with GGLabel *n*:

```
N = self.LHS.nodeWithLabel(n)
```

To get the model element which matched to node N in LHS:

```
E = self.getMatched(graphID, N)
```

To get the value of a string attribute *attr* of element *E*:

```
E.attr.toString()
```

## 6.5 Class2Table_create Rule

This rule creates a new table for the persistent class.

The condition ensures, that the class has not yet been processed (corresponding table has not been yet created).

- Condition:

```
C = self.getMatched(graphID, self.LHS.nodeWithLabel(1)).name.toString()
return not self.graphRewritingSystem.isProcClass(C)
```

- LHS and RHS (Figure 12)
  - Table(3).name.AttrSpecify

```
return "t_" +
  self.getMatched(graphID, self.LHS.nodeWithLabel(1)).name.toString()
```

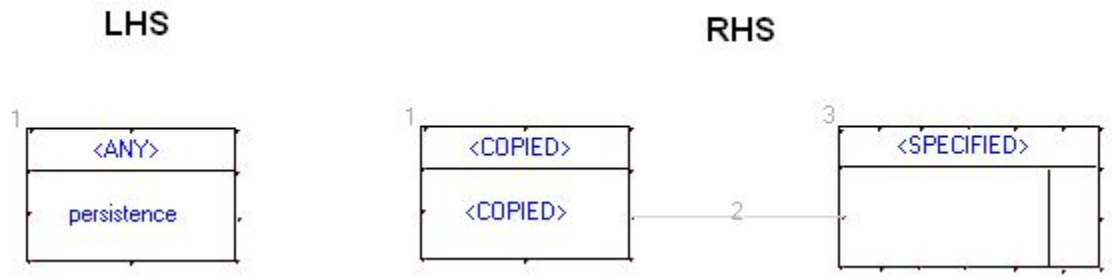


*Figure 12 Class2Table_create Rule*

- Action:

```
C = self.getMatched(graphID, self.LHS.nodeWithLabel(1)).name.toString()
self.graphRewritingSystem.addProcClass(C)
```

After the rule was executed, action adds the name of the class to the processed classes list.

## 6.6 Class2Table_extend Rule

This rule adds columns to the table from primitive type attributes in the class.
The condition ensures that matched attribute is not of type class and has not yet been processed.

- Condition:

```
A = self.getMatched(graphID, self.LHS.nodeWithLabel(3))
if A.type.toString() == 'Class':
  return 0
else:
  return not self.graphRewritingSystem.isProcAttr(A.name.toString())
```

- LHS and RHS (Figure 13)

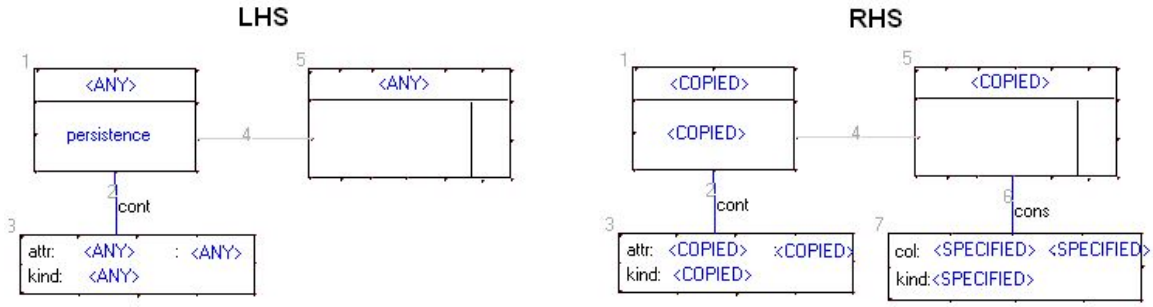


*Figure 13 Class2Table_extend Rule*

– Column(7).name.AttrSpecify

```
return self.getMatched(graphID, self.LHS.nodeWithLabel(3)).name.toString()
```

– Column(7).kind.AttrSpecify

```
return self.getMatched(graphID, self.LHS.nodeWithLabel(3)).kind.toString()
```

– Column(7).type.AttrSpecify

```
n = self.getMatched( graphID, self.LHS.nodeWithLabel(3)).type.toString()
if n == 'String':
```

```
  return 'VARCHAR'
elif n == 'Integer':
  return 'NUMBER'
else:
  return 'unknown'
```

– Table(5).primaryKey.AttrSpecify

```
A = self.getMatched( graphID, self.LHS.nodeWithLabel(3))
T = self.getMatched( graphID, self.LHS.nodeWithLabel(5))
if A.kind.toString() == 'primary':
  return [A.name] + T.primaryKey.getValue()
else:
  return [] + T.primaryKey.getValue()
```

- Action:

```
A = self.getMatched( graphID, self.LHS.nodeWithLabel(3)).name.toString()
self.graphRewritingSystem.addProcAttr(A)
```

## 6.7 ClassAttr2Table_create Rule

If the type of the attribute is class this rule adds a traceability link between that class and a table. This allows recursion

- Condition:

```
C = self.getMatched(graphID, self.LHS.nodeWithLabel(5)).name.toString()
return not self.graphRewritingSystem.isProcClass(C)
```

- LHS and RHS (Figure 14)

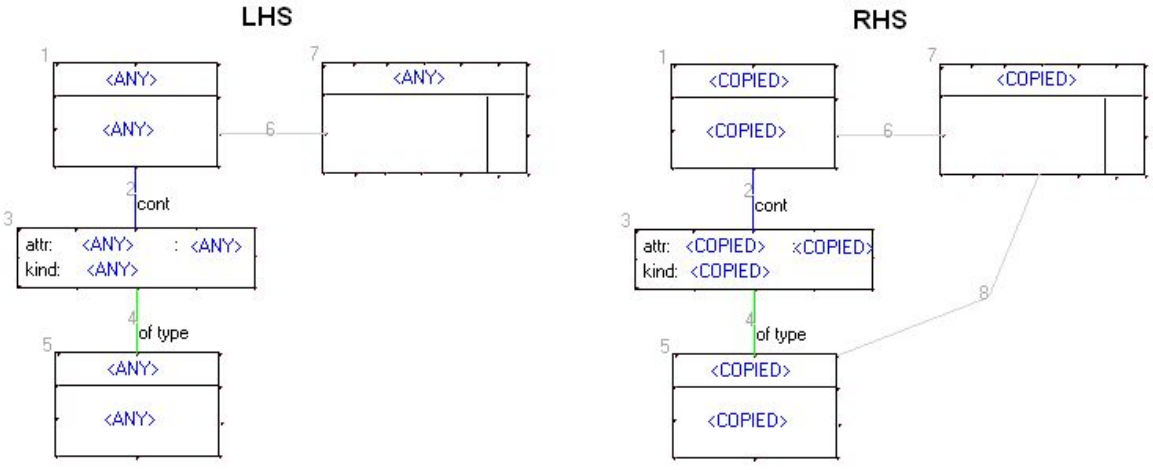


*Figure 14 ClassAttr2Table_create Rule*

- Action:

```
C = self.getMatched(graphID, self.LHS.nodeWithLabel(5)).name.toString()
self.graphRewritingSystem.addProcClass(C)
```

## 6.8 ClassAttr2Table_extend Rule

This rule adds columns for primitive type attributes of a class linked with a table by traceability link.

- Condition:

```
A = self.getMatched( graphID, self.LHS.nodeWithLabel(5))
if A.type.toString() == 'Class':
  return 0
else:
  return not self.graphRewritingSystem.isProcAttr(A.name.toString())
```

- LHS and RHS (Figure 15)

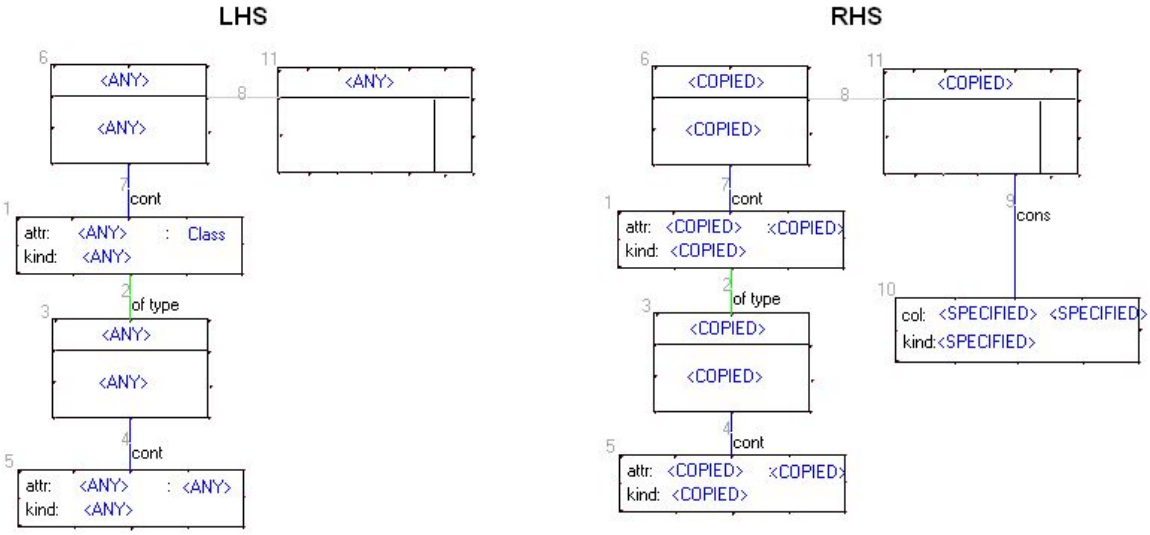


*Figure 15 ClassAttr2Table_extend Rule*

The new column's attributes are set the same way as in Class2Table_extend rule (instead of node 3, node 5 is taken).

- Action:

```
A = self.getMatched( graphID, self.LHS.nodeWithLabel(1))
A2 = self.getMatched( graphID, self.LHS.nodeWithLabel(5))
if A.kind.toString() == 'primary':
  T = self.getMatched( graphID, self.LHS.nodeWithLabel(11))
  newName = A.name.toString() + '_' + A2.name.toString()
  T.primaryKey.setValue([ATOM3String(newName)] + T.primaryKey.getValue())
self.graphRewritingSystem.addProcAttr(A2.name.toString())
```

## 6.9 Assoc2FKey Rule

This rule creates foreign key in table corresponding to source class.

- Condition:

```
C = self.getMatched( graphID, self.LHS.nodeWithLabel(5))
if C.kind.toString() != 'primary':
```

```
    return 0
  else:
    return not
      self.graphRewritingSystem.isProcForeignCol(C.name.toString())
```

- LHS and RHS (Figure 16)

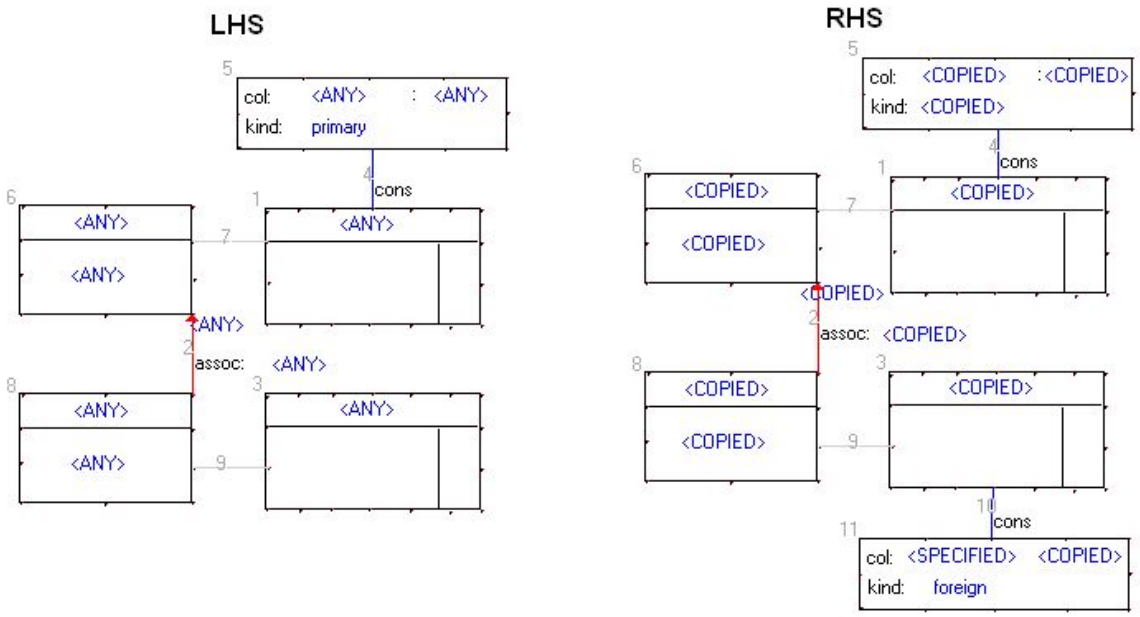


*Figure 16 Assoc2FKey Rule*

  - Column(11).name.AttrSpecify

```
AT = self.getMatched( graphID, self.LHS.nodeWithLabel(2)).role.toString()
C = self.getMatched( graphID, self.LHS.nodeWithLabel(5)).name.toString()
return AT + '_' + C
```

- Action:

```
CName = self.getMatched( graphID, self.LHS.nodeWithLabel(5)).name.toString()
self.graphRewritingSystem.addProcForeignCol(CName)
ATRole = self.getMatched( graphID, self.LHS.nodeWithLabel(2)).role.toString()
T = self.getMatched( graphID, self.LHS.nodeWithLabel(3))
newName = ATRole + '_' + CName
T.foreignKey.setValue([ATOM3String(newName)] + T.foreignKey.getValue())
```

## 6.10 CleanTraceability Rule

This rule removes all traceability links after transformation finished

- Condition: none
- LHS and RHS (Figure 17)
- Action: none

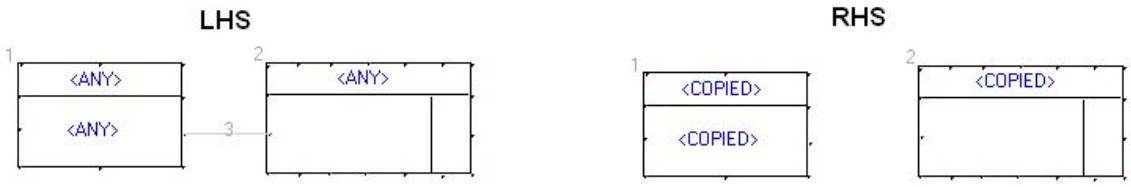


*Figure 17 CleanTraceability Rule*

## 6.11 Final Action

Displays transformation summary

```
print 'Transformation execution finished'
print '---'
print 'processed classes:'
for c in self.rewritingSystem.procClasses.getValue():
  print c.toString()

print '---'
print 'processed attributes:'
for a in self.rewritingSystem.procAttrs.getValue():
  print a.toString()

print '---'
print 'processed foreign key attributes:'
for c in self.rewritingSystem.procForeignCols.getValue():
  print c.toString()
```

# 7 Running a Transformation

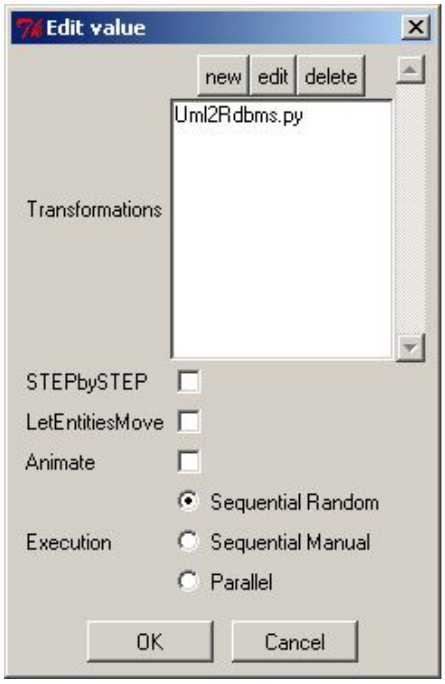


*Figure 18 Run Transformation Dialog*

AToM3 provides several transformation execution modes.

- STEPbySTEP – means that user confirms every step of the execution,

- Execution modes:
  - Sequential Random – a rule is executed against randomly chosen matched subgraph,
  - Sequential Manual – a rule is executed against subgraph chosen by user,
  - Parallel – a rule is executed against all separate matched subgraphs in one step.

# 8 Example Transformation Results

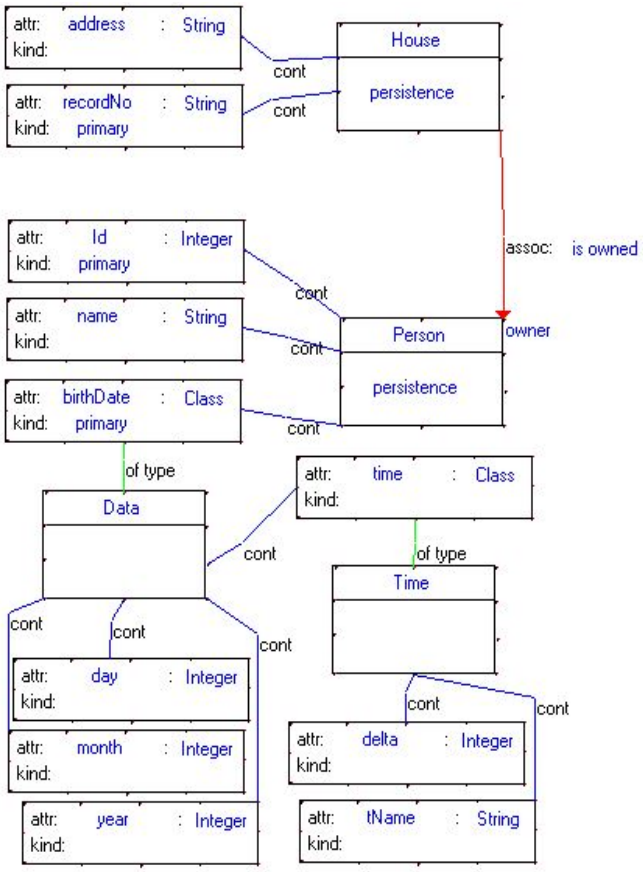


*Figure 19 An Example UML Model*

Expected results:

- t_House.PrimaryKey: {recordNo}
- t_Person.PrimaryKey: {Id, birthDate_day, birthDate_month, birthDate_year}
- t_House.ForeignKey:

  {owner_Id, owner_birthDate_day, owner_birthDate_month, owner_birthDate_year}

Figure 20 presents the model after transformation.

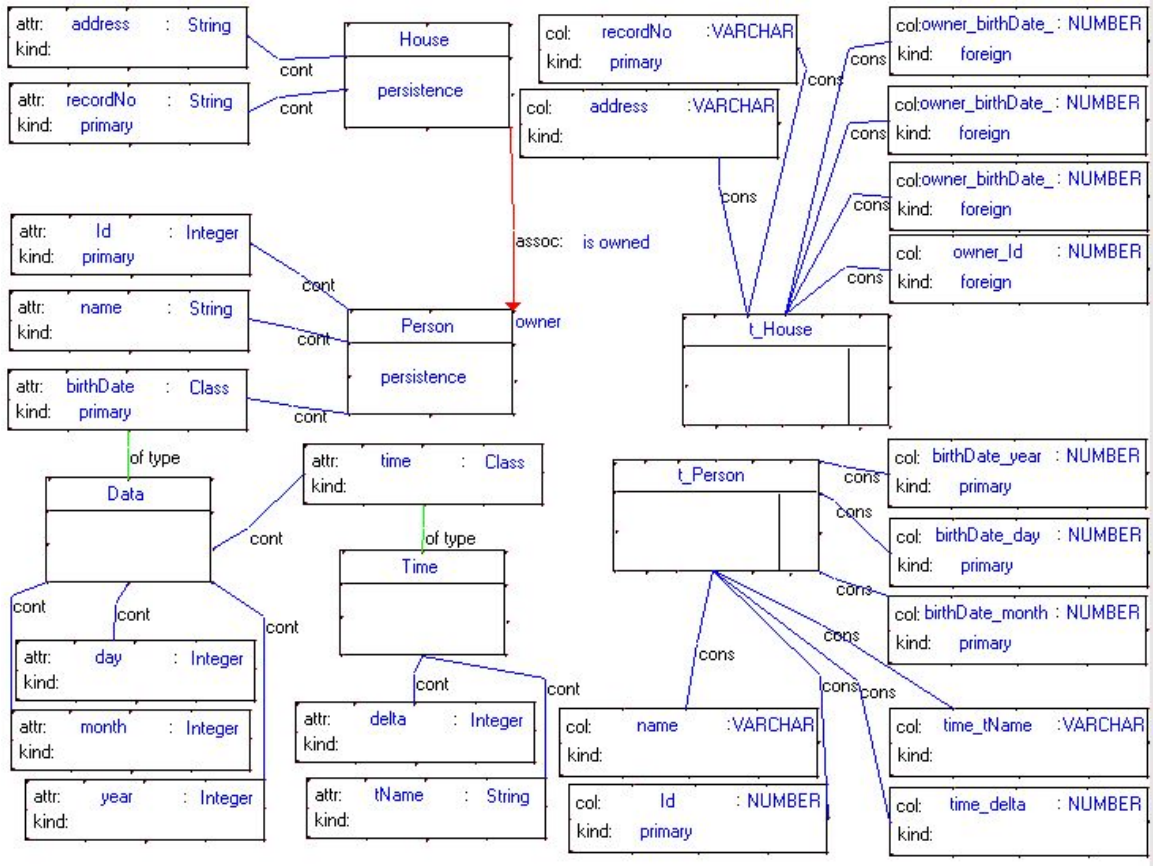


*Figure 20 Example Model Transformation Result*

# 9 Classification of AToM³ features

This classification was done according to [CH03].

- Transformation Rules:
  - Variables – model elements are not contained in variables,
  - Patterns – graphs, concrete graphical and semantically typed,
  - Logic – non-executable (constrains on attributes); executable imperative(actions) and executable declarative (LHS, RHS graphs),
  - LHS/RHS Syntactic Separation,
  - Unidirectional,
  - Parametrized,
  - No Intermediate Structures.
- Rule Application Scoping
  - No Scoping (rules can be applied to the whole model).
- Source-Target Relationship
  - Target model is the same as source model. Modifications are in-place and update is destructive.

- Rule Application Strategy
  - Non-deterministic (both concurrent and one-point (sequential)),
  - Interactive.
- Rule Scheduling
  - Explicit, Internal (rule order),
  - Rule Selection – Explicit Condition (higher order before lower order rules),
  - Fixed-point rule iteration (until none of the rules can be applied),
  - No Phasing.
- Rule organization
  - No modularity mechanisms,
  - No reuse mechanisms,
  - Organizational Structure – rules are independent.
- Tracing
  - No Dedicated Support.
- Directionality
  - Unidirectional.

# 10 Missing Features and Drawbacks

In this section I present the summary of missing features. They can be divided into two types – tool related and conceptual.

## *10.1 Tool Related Drawbacks*

- Multi-formalism models

First I wanted to create two meta-models UML and Rdbms separately. Unfortunately, saving models and transformations in more than one formalism does not work. When reopening the model or transformation only model parts from one meta-model stay, other are lost.

- Parallel meta-model and transformation development

I was working in small iterations. Extending a bit meta-model, then extending a bit transformation. Problems started when I needed to change the meta-model – the transformation becomes invalid. It is good in case of deleting something from meta-model. But it happens also when adding newattribute to entity or relationship.

Changes to meta-model should be in that case incorporated into transformations.

My solution to the problem was to remove the element that is going to be changed from the whole transformation, then change the meta-model and finally add the changed element back to the same places.

- Results of transformation are not in separate model

There's no possibility of separation the source model from the result. It is especially important in case of bigger models. One possible solution to this problem is simply remove the parts from the source model and save the rest in the next file.

- Actions and constraints

The use of Python as a programming language is not so bad, because Python is easy to learn and use, but also powerful.

The problem is that the internal graph data structure of AToM$^3$ is not hidden from the user. The knowledge of it is necessary especially when debugging – error messages from Python go deeply into the internal structure of AToM$^3$.

Another big problem is the lack of transformation global name-space, for creation transformation variables and functions.

One of strange problems I encountered was also that ATOM3String type is not visible in AttrSpecify. Instead of specifying the new value of foreignKey in AttrSpecify function I had to move that code to final action (Assoc2FKey rule).

- User interface is not suitable for larger projects

There is no modularity and explicit sub-transformation execution It would be good to have a sub-transformation Attribute2Column that would transform a primitive attribute into a column. It could be used by Class2Table_extend and ClassAttr2Table_extend rules and I could avoid duplication of code.

- Debugging support

This is the problem in case of a user not very familiar with Python – there is no syntactic analysis of Python code that is entered in conditions and actions.

Debugging can be done only during transformation run-time, errors going deeply into the internal structure of AToM3. This makes work very hard because to test the transformation it has to be imported for execution. Then in case of an error, even if transformation is fixed in another instance of AToM$^3$, reimporting the transformation doesn't work. You have to restart the tool, reload the model, import the transformation and run it.

## *10.2 Conceptual Drawbacks*

- Scheduling

When some more advanced scheduling is necessary, the user has to program it himself. In case of my project the problem was the lack of "only once" matching for given matching or some elements in a matching.

- Pattern matching

How to specify that some parts in LHS should not appear? The problem was the lack of "not" matching. I solved the problem by remembering the lists of processed elements so that the rule condition could check if an element has already been processed.

# Appendix A

The general advice is to restart AToM$^3$ when switching to new meta-model, reloading transformation or after code generation. AToM$^3$ is still a prototype.

1. To install the project unzip *Uml2Rdbms.zip* file into the installation directory of AToM$^3$.
2. Run AToM$^3$ and choose File->Options. Add *UmlRdbms* to Path Directories and type it in Dir.for code generation input.
3. To open UmlRdbms meta-model Entity-Relationship model choose File->Open Model and select *UmlRdbms\UmlRdbms_ER_mdl.py*.
4. To open sample UML model restart the tool and choose File->Open Model and select *UmlRdbms\exampleFull.py*. Note that AToM$^3$ automatically loaded correct meta-model.

5. To run the transformation choose Transformation->Execute transformation. Click New button, then Browse. Select *UmlRdbms\Uml2Rdbms.py*. Press OK to run the transformation.
6. To edit/browse the transformation choose Transformation->Load transformation. Select *UmlRdbms\Uml2Rdbms_mdl.py*. To edit choose Transformation->Edit transformation. To edit the rule select it in the Rules list and press Edit button.

# 11 References

[ATOM3] http://atom3.cs.mcgill.ca/

[CH03] K. Czarnecki, S. Helsen, "Classification of Model Transformation Approaches", OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003