$ATo M^3$ meta-modelling research project

Marc Provost
School of Computer Science, McGill University
Montréal, Québec, Canada

August 17, 2002

Thanks!

This project could not have come to life without the help of my "mentors" Hans Vangheluwe and Juan de Lara. I have learned quite a lot in only 4 months! Specials thanks also to Ernesto Posse and Simon Lacoste for their very useful comments on my work!

Preface

This report describes the results of a complete summer of work in the **Modelling, Simulation and Design lab (MSDL)** with Prof. Hans Vangheluwe. Although the primary objective of the project was to introduce me to the world of research, some concrete results were obtained and may be used for the next generation of $AToM^3$ (I hope!!). [1] The results are divided into 2 parts:

Chapter 1 discuss the details of a UML class diagram modelling environmeng I meta-modeled with the Entity-Relationship formalism. Many aspects of its implementation are discussed in details including the problems that were encountered. The features that could not be implemented are also justified, possible solutions are suggested in some cases.

Chapter 2 explains how a sketch of a UML class diagram meta-meta-model containing inheritance was built in $AToM^3$. Everything is discussed in details so that it will be possible to improve the implementation.

[1] More information about $AToM^3$ can be reached on the web at **http://moncs.cs.mcgill.ca/MSDL/research/projects/AToM3/**

Contents

1 UML class diagram modelling environment in AToM3

1.1 Introduction

One of the first objective of my project was to get used to $AToM^3$, and the best way to learn a designing tool is actually to design with it! So we decided to **meta-model** UML class diagram using entity-relationship (ER) as a first step toward creating a new **meta-meta model** based on UML class diagram to replace ER. We wanted to create a meta-model as close as possible to the meta-model described in the OMG [2] specifications document. Of course, since the latter meta-model is constructed using UML class diagrams formalism, it would have been quite a task to reproduce it exactly with entity-relationship. The goal was not to built a perfect equivalent formalism, but was to select important features from the OMG specifications document and try to meta-model them in $AToM^3$ (A document containing the chosen features was provided). [3] Hence, we wanted to prove that it was possible to implement a usable UML class diagram modelling environment in $AToM^3$. Afterwards, $AToM^3$ devellopers could polish my implementation by adding the missing features and try to implement solutions to the problems I wasn't able to solve.

[2] http://www.omg.org

[3] http://moncs.cs.mcgill.ca/people/mprovost/projects/designV1

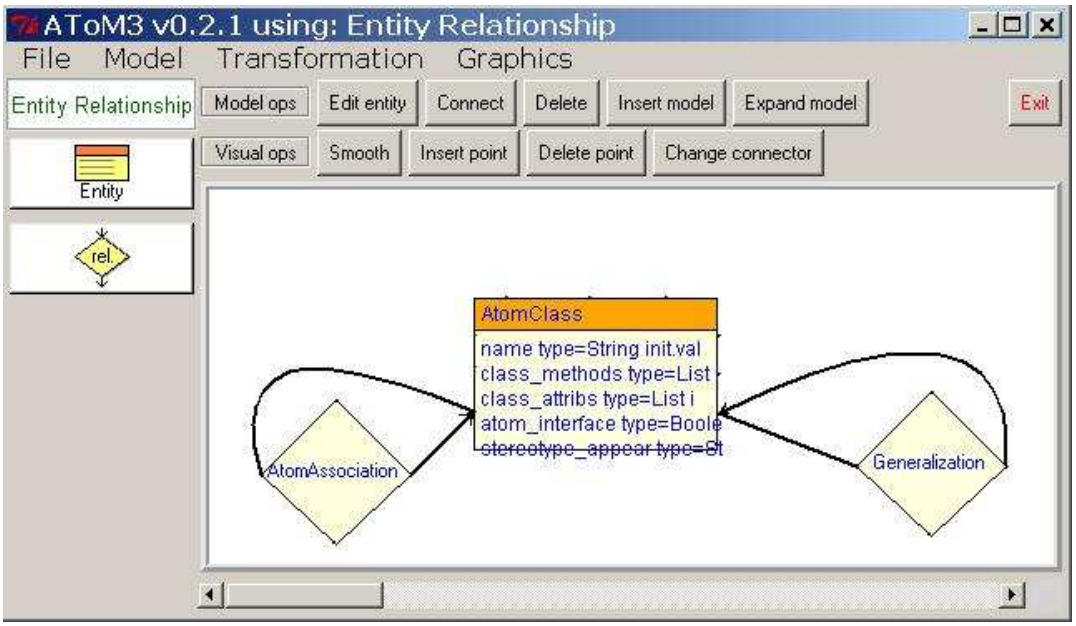

Figure 1: **UML class diagram meta-model**

Filename: **ClassDiagram_MM_mdl.py**

The UML class diagrams meta-model seems quite simple. It has only one entity *AtomClass* related with itself via two relationships *AtomAssociation* and *AtomInheritance*. This was expected since, in the UML class diagrams specifications, two non-stereotyped class can only be connected via inheritance or associations.

1.2 AtomClass entity

Usual classes attributes were given to the AtomClass entity such as a *name*, a list of *methods* and a list of *attributes*. Other attributes were added for the purpose of the UML editor, *attrib_appear, method_appear*, *stereotype_appear* and *abstract_appear*. Those unusual attributes are used for the appearance of the class in the modelling environment. They should be considered as being *private*, but the user can still edit them since *private* attributes are not supported yet in $ATOM^3$. Also several constraints were created in the

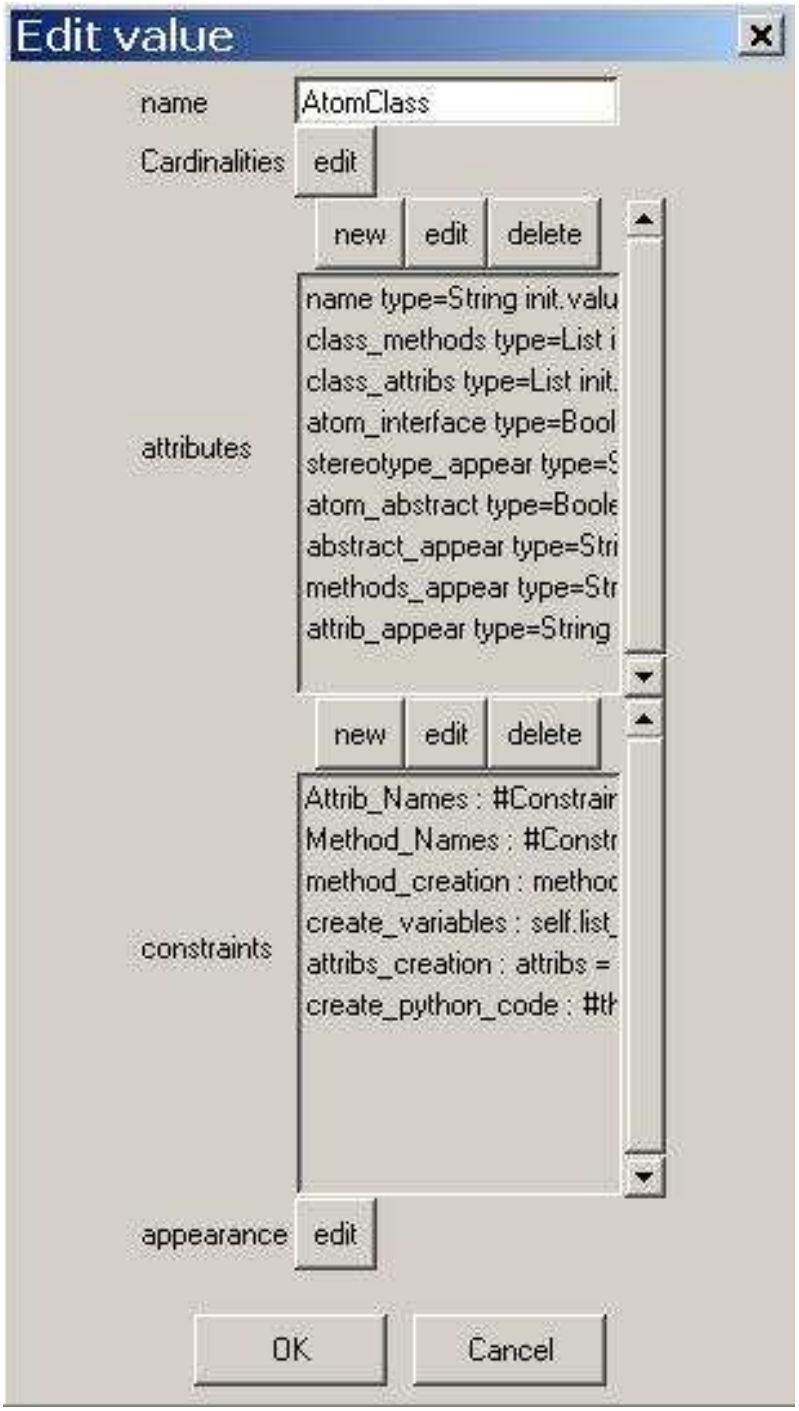

Figure 2: **AtomClass entity details**

AtomClass entity. The constraints will be explained in detail later, but they are used mostly for *python code generation, method/attribute parsing* and, of course, to verify the correctness of user-defined models.

1.2.1 Appearance

An expected rectangular box was defined as the apprearance (figure 3) for the class in the modelling environment. The *name* of the class, the *stereotype* and the *abstract* tag are drawn in the top sub-rectangle. Without any surprises, class *attributes* and *methods* are located respectively in the middle and botton sub-rectangle.

Some constraints were defined (table 1) to update the appearance of the class as the user modify the internal variables. Indeed, the constraints *attrib_constr, method_constr, abstract* and *interface* update the variables attrib_appear, method_appear, stereotype_appear and abstract_appear each time the user edit an AtomClass in a model.

```
#Code for interface
#set up the appearance depending if abstract property is on
inter = self.semanticObject.atom_interface.getValue() #interface value
self.gf12.setVisible(inter[1])
#Code for method_constr
list = self.semanticObject.class_methods.getValue()
tmpString = ""
for a in list:
    tmpString = tmpString + a.getValue() + '\n'
self.semanticObject.methods_appear.setValue(tmpString)
```

table 1

But, why do we need to create such constraints for the appearance of the attributes/methods? Why don't we just put the *list_of_methods* and *list_of_attrib* instead of creating a formatted string on the fly? Because most of the ATOMTypes are not well formatted when they are displayed on the appearance of an entity or a relationship at the *model* level. Indeed, they were designed to be well formatted when we create meta-models and, since $ATOM^3$ uses the same interface for creating both models and meta-models,

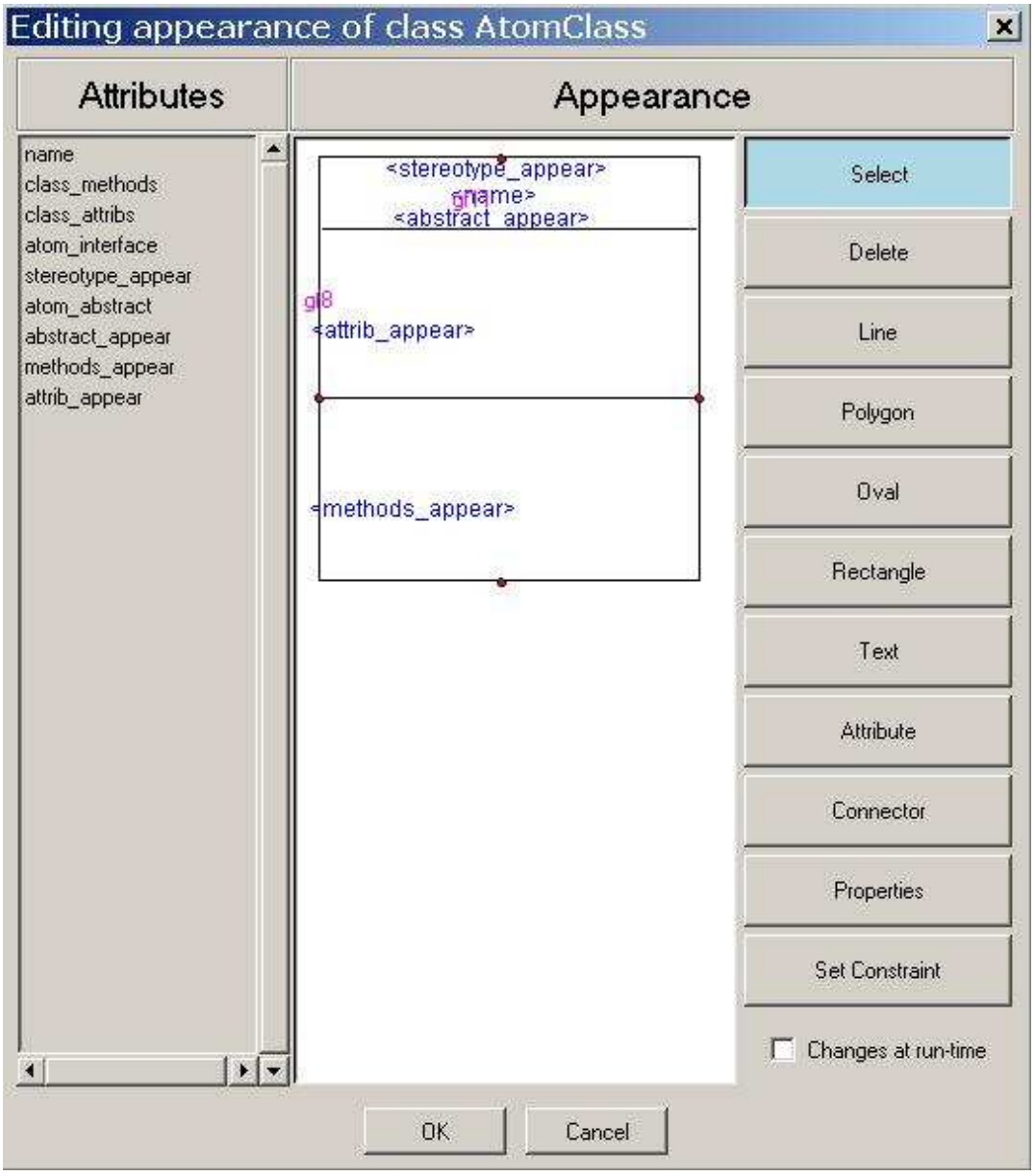

Figure 3: **AtomClass appearance**

$ATOM^3$ cannot satisfy (for now) two modelling environments that have different appearance needs. This is why I had to create manually a formatting for the *modelling* level. A solution to this problem in the next generation of $ATOM^3$'s GUI could be that we pass an indicator of the environment we are working on to the type, so that the "toString()" function returns a nicely formatted string. It would even be better if $ATOM^3$ could automatically detect that environment.

1.2.2 Constraints

The constraints section of the AtomClass entity is the most complicated part of the meta-model. We will go over them one by one.

A constraint, *create_variables*, was added to initialize hidden variables in an AtomClass at creation time (table 2). Unfortunately, this trick to hide variables did not work completely since the saving process of an $ATOM^3$ model does not include variables created on the fly. Indeed, saved variables must be included in a particular list that is also used to generate the AtomClass user interface! So, even if we added the variables to that list, they would be editable in an instance of an AtomClass! So, to solve the saving problem, I could have made all the variables "public". I did not implement that solution because the class diagram modelling environment with hided variables is closer to a real editor such as dia. Not to mention that it is more impressive and simple to demonstrate a formalism with 8 hided variables, than to show all the implementation in every single example. Also, I am confident that we will, in the near future, implement a way to add hidden variables in a meta-model since the implementation of a formalism must be abstracted from the user in order to simplify model creation. At that point in time, it will be easy to port my class diagram formalism in $ATOM^3$.

```
self.list_of_methods = ATOM3List([1,1,1,0], None)
self.list_of_attribs = ATOM3List([1,1,1,0], None)
#will contains methods code, index correspond to list_of_methods
self.method_code = ATOM3List([1,1,1,0], None)
self.attrib_code = ATOM3List([1,1,1,0], None)
self.default_code = "\t\tpass\n"
self.user_code = {} #dictionnary method signature mapped to code part
self.code = ATOM3String() #String that contains the global class code
self.code.setValue("Class My_class\n\tpass")
```

table 2

There are two constraints that update *list_of_methods* and *list_of_attribs* each time the the user edit *class_methods* and *class_attribs*. *Method_creation* and *attribs_creation* are called after an instance of AtomClass are edited. Some checks (not all) were implemented to warn the user if he use the wrong input syntax for the methods. The parsing was done (figure 3) by extracting the types, names and signatures and store them as individual strings in the two lists, thus creating a lists of lists.

class_methods	list_of_methods
"double meth(int a, int b)"	((double, meth, (int, a), (int, b)))
class_attribs	list_of_attribs
"int a"	((int, a))

table 3

Also, two contraints were created in the AtomClass entity to verify the correctness of models. The first one, as you can see on table 4, *Attrib_Names* checks that attribute's names are unique. The name of an attribute is simply compared to the names in the list of attributes and the list of methods. If we find out that the user wrongly assigned equal names when the model is saved, $ATOM^3$ pop-up a warning message in the modelling environment.

```
# Constraint that check that attribute's names are unique

attrib_len = len(self.list_of_attribs.getValue()) # length of attrib list
attrib = self.list_of_attribs.getValue() # get the attributes list
methods = self.list_of_methods.getValue() # get methods list

attrib.extend(methods) # combining the 2 lists
# We only compare attrib–attrib and attrib–methods
for a in range(0, attrib_len):
  for b in range(0, len(attrib)):
    if(a != b):
      if(strip((attrib[a].getValue())[1].toString()) ==
# strip removes blank spaces
        strip((attrib[b].getValue())[1].toString())):
          return ("Attribute/methods names must be unique in "
            + self.name.getValue(), attrib[a].getValue()[1].toString())
```

table 4

Similar code is used in *Method_Names* to ensure that method's signatures are unique. We check if the following conditions hold before issuing a warning to the user:

- equality in the method names
- equality in the number of arguments
- equality in the corresponding argument types

Finally, one last constraint is responsible for generating python code. How does it work? First, it checks if the current class is a children of one or multiple parent. Then it creates the appropriate class definition, such as def class Child(Parent): , etc. After that, the attributes are being scanned in the *list_of_attribs* and then written correctly. Finally, the methods are being written with their appropriate code part specified or not by the user. Where does the code part come from? I have added an "Edit Code" button to the buttons model of my meta-model. It allow the user to add code part to the automatically generated code for the methods. The signature of each method is mapped its code part in a dictionnary (yes, the hidden one we talked about). If the signature is not found in the dictionnary, a defaut one

is written ("pass"). Methods or attributes added in by the user that were not specified in the *class_methods* or *class_attribs* are erazed from the code part automatically. The code for each class is stored in an independent variable, thus python files can be easily created.

1.3 AtomAssociation

The AtomAssociation relationship contains anticipated attributes, a name, a type, a multiplicity integer for the source and a multiply integer for the target. The type of the AtomAssociation can be Aggregation, Composition or plain association.

1.3.1 Appearance

I wanted to adapt the appearance of the relationship depending on its type, but I couldn't due to some $ATOM^3$ bug. However, the multiplicity and the name were added on the link.

1.3.2 Constraints

Only one constraint was created to parse the multiplicity string entered by the user into some hidden variables. Again, This should be modified in order to save models. This section was not completed, but the multiplicities were going to be used in the python code generation to add automatically created attributes such as its_friend[x] (for a x..x multiplicity).

Syntax: "5..10"

table 5

1.4 AtomGeneralization

This relationship was not given any attributes, and no fancy constraint or appearance were created. Very simple since my project did not need anything more complex.

1.5 Model examples

Filename for the modelling environment: **ClassDiagram.py**

The following figures are models using my UML class diagram formalism. As you can see in the figures, the button edit code is used, as stated, to edit the python code of the classes.

Figure 4 represent a company hiearchy where the *Manager* and the *Cashier* are more specialized than the plain employee.

As you can see on figure 5, the class *Cashier* inherits from the class *employee*. Also, default methods were created for the variable *item_per_min*.

Moreover, multiple inheritance is also supported by the code generation, as you can see on the figure 6 and 7

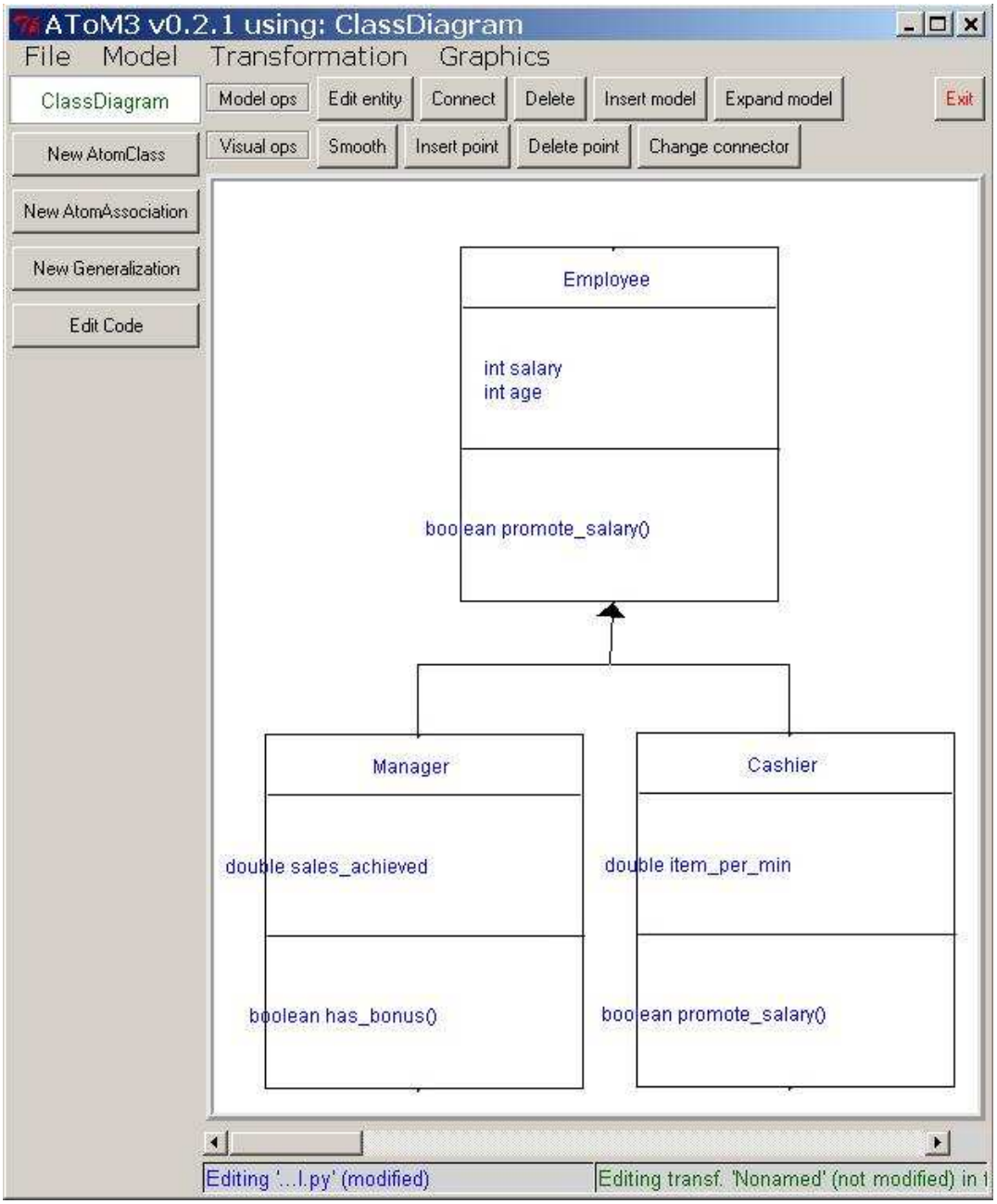

Figure 4: **Company diagram**

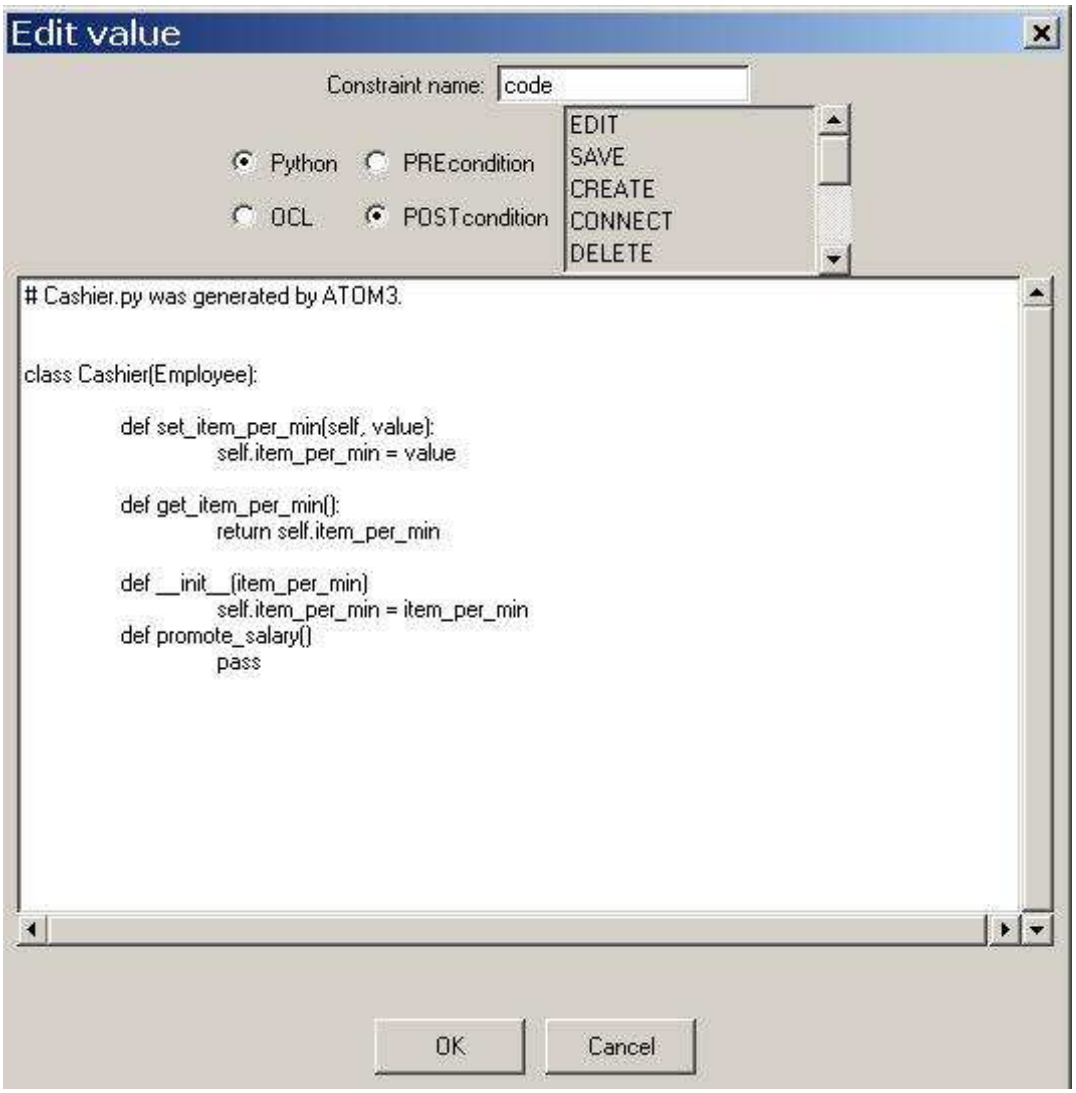

Figure 5: **Edit code for the class cashier**

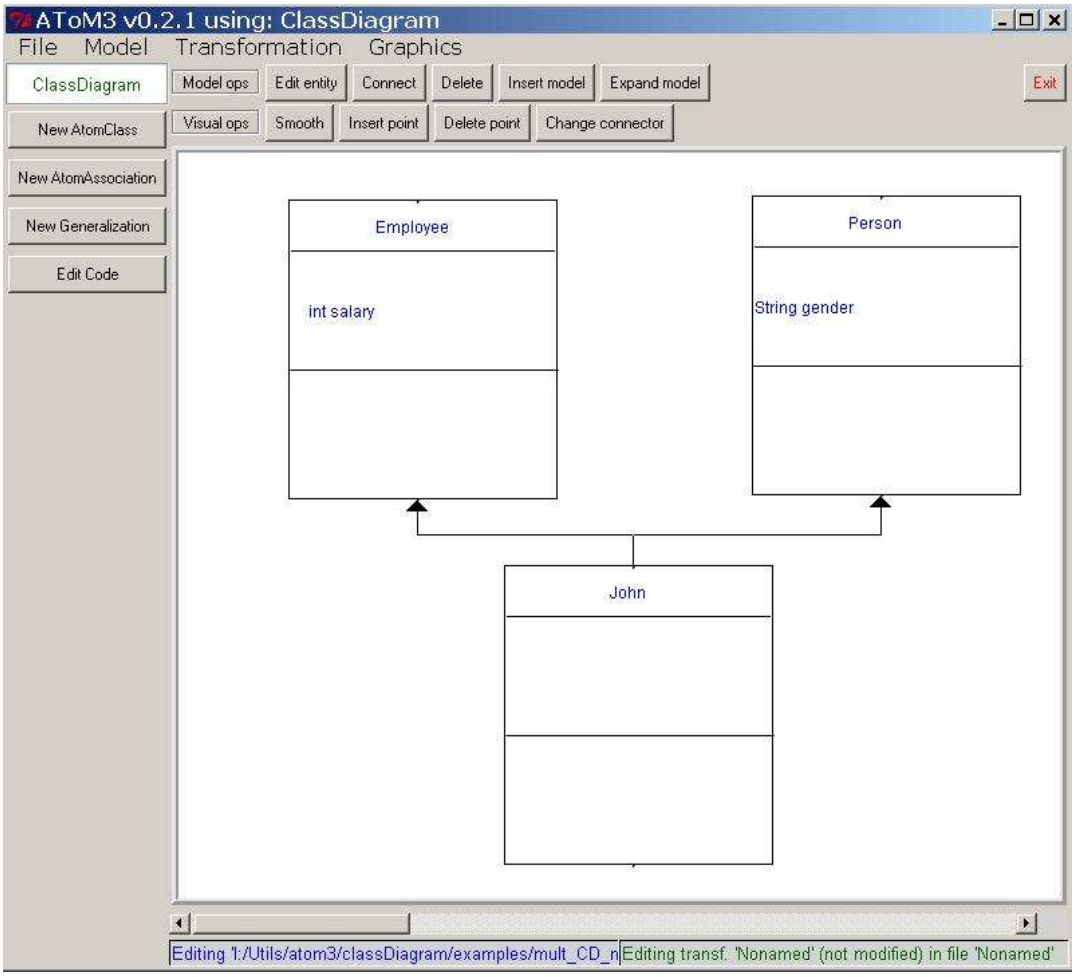

Figure 6: **Multiple inheritance code generation**

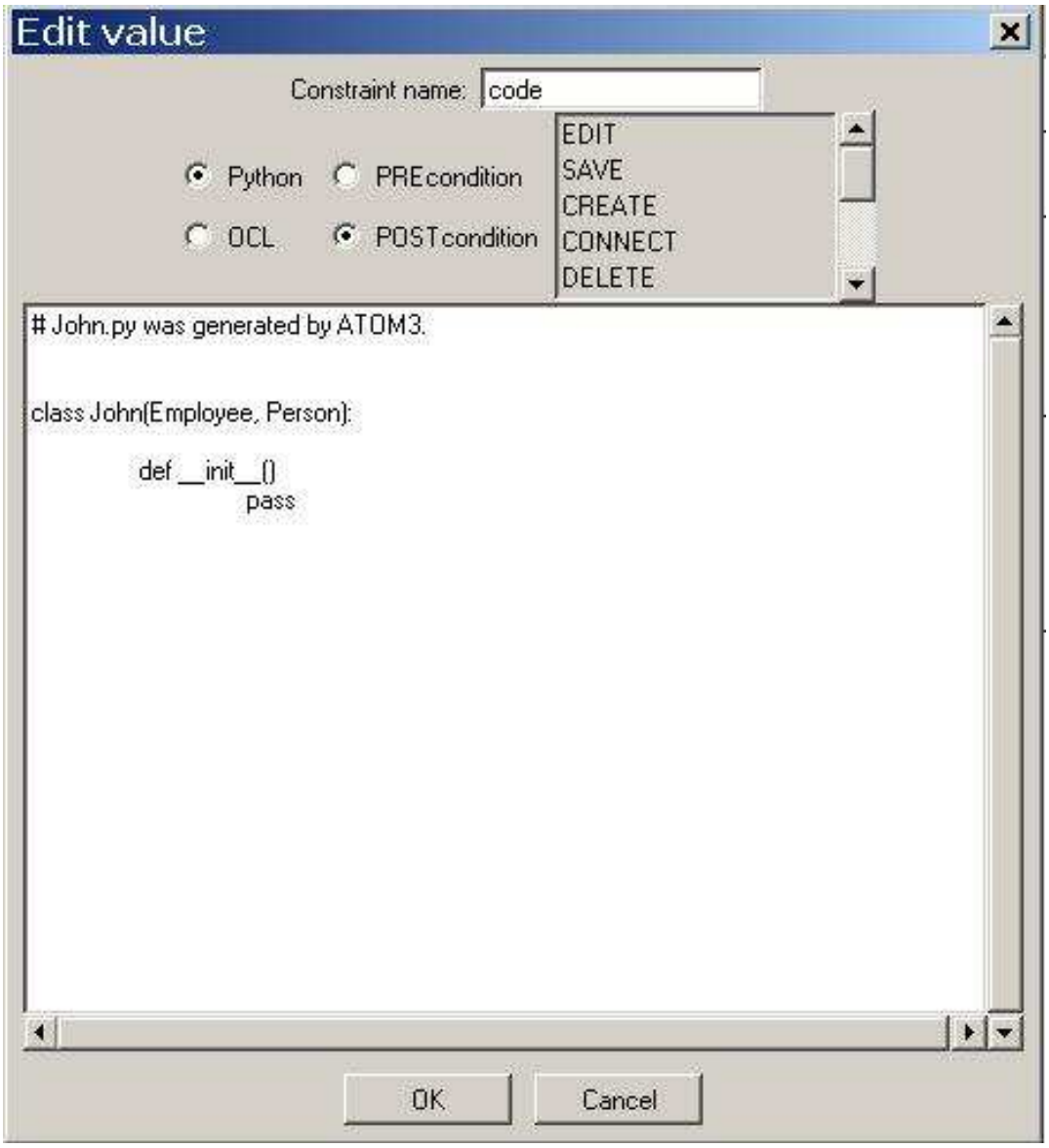

Figure 7: **Multiple inheritance example**

2 Entity-Relationship with inheritance meta-meta model

2.1 Introduction

The second step of my project was to meta-meta-model a new formalism which would include, as its most important feature, inheritance. The plan was to implement a partial UML class diagram formalism including inheritance in $AToM^3$. The first solution we had was to create a graph-grammar that would transform a model of my class diagram formalism (as explained in chapter 1) directly into entity-relationship. Hence, each AtomClass at the model level would be replaced by an entity with a list of attributes corresponding to the attributes that the AtomClass had. If a Generalization was encountered, an algorithm, via the graph grammar, would copy the attributes from the parent to the child and add an appropriate number of relationships to express its semantics. Afterwards, we could easily use ER code-generation for modelling environments. However, this approach had its limits. Since I am using a list of strings to store the class attributes in the ClassDiagram formalism, identifying the type of a given attribute and then create the appropriate AtomType in an entity would have been a nightmare. Not mentionning that the meaning of a method is not clear in a meta-model. I started to think that maybe we could create a simpler meta-meta-model with inheritance but without methods. Then, I learned how the formalism code-generation was working in $AToM^3$ and we develloped another solution that could be called *entity-relationship with inheritance*.

The idea was to meta-meta-model a formalism like entity-relationship with an entity named AtomClass with atrributes such as a name, a list of ATOM3Attributes, a list of ATOM3Constraints, a list of ATOM3Connection and an ATOM3Appearance. But, instead of having only one relationship like in entity-relationship, an inheritance and an association relasionship would be included. The main difference between this approach and the previous one is that every class attribute is an ATOM3Attribute so its they can be generated at the lower meta-level (depending on the type) by the build-in code-generation algorithm used by entity-relationship. Also, the other generative types also generates an appearance, a list of constraints and cardinalities at the lower meta-level. In the next sections, we will go over the implemen-

tation of that meta-meta-model and we will explain the steps required to create a modelling environment from the newly created meta-meta model.

Filename for the meta-meta-model **ClassDiagram_ER_mdl.py**

2.2 AtomClass

One entity, AtomClass, is related with itself via AtomAssociation and AtomInheritance relationships to constitute the meta-meta-model. Key attributes, that are used for the code-generation, are specified for each object in the meta-meta-model. There is also a model key attribute: ModelName.

2.2.1 Attributes and Cardinalities

AtomClass, as stated before, possesses the following attributes: a name (key), a list of ATOM3Attributes, a list of ATOM3Constraints, a list of ATOM3Connections [4] and an ATOM3Appearance. The cardinalities are 0..N as a source and as a destination via AtomAssociation and AtomInheritance.

2.2.2 Constraints

Several constraints were written to create the cardinalities at lower meta-level. At first, the list of ATOM3Connections is empty and the role of the constraints is to update it as soon as AtomAssociations are linked with AtomClasses, in order that the user can edit the cardinalities at the lower meta-level. For now, the connections are considered undirected, but it's easy to modify the code to make them directed. [5] If we go over the constraints one by one :

createCardinality is trigerred to add ATOM3Connections to the current AtomClass object, after a link had been created. In order to do this, the in and out connections of the current AtomClass are scanned and the connected AtomAssociations are added to the list of ATOM3Connections.

[4] for the cardinalities, see section 2.2.2

[5] comments in the code explain what to do

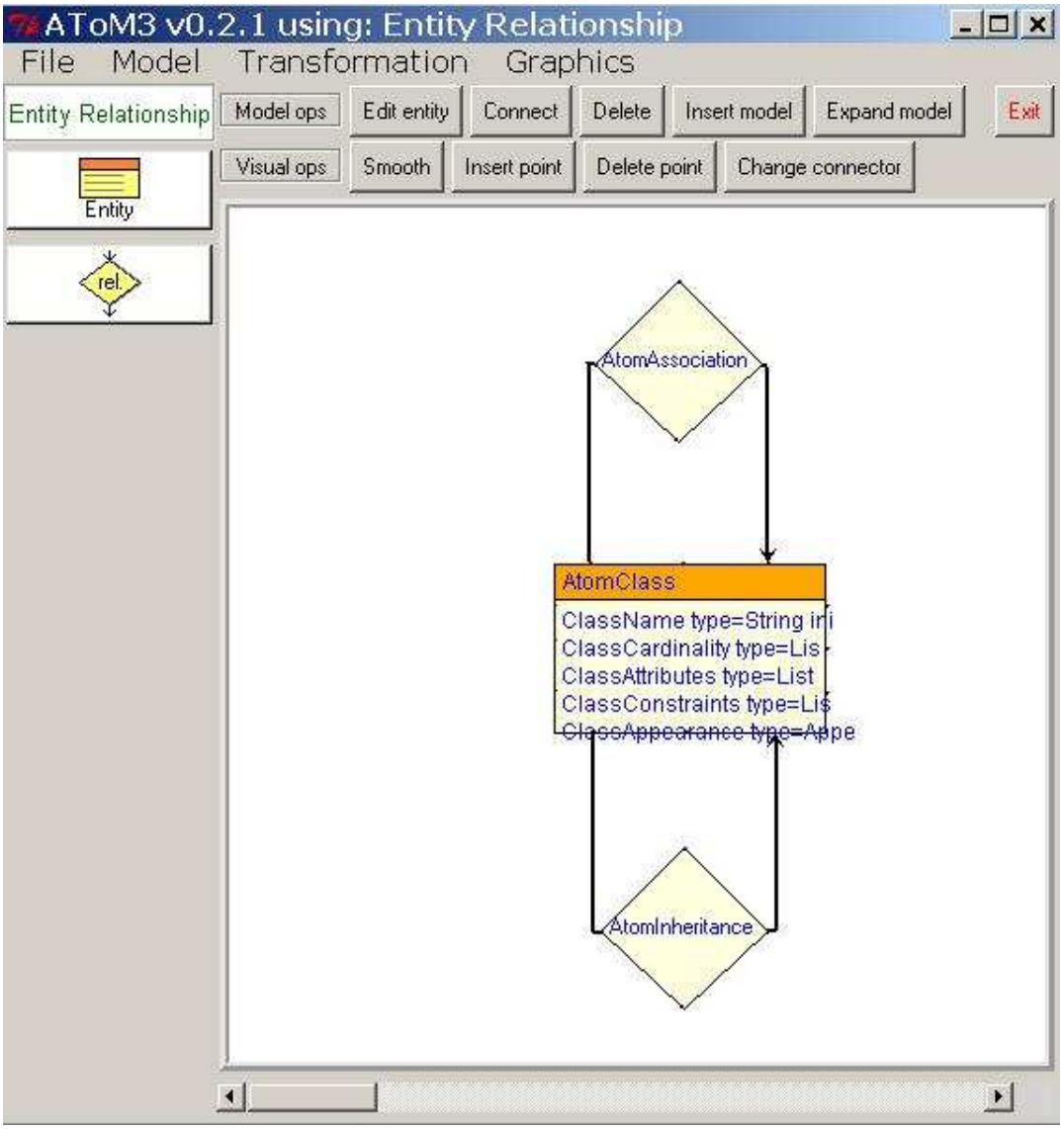

Figure 8: **Entity-Relationship with inheritance meta-meta model**

preDeleteCardinality is called just before a disconnection to remove an AtomAssociation connection object from the AtomClasses linked to it, if the AtomAssociation is deleted completely.

deleteCardinality is called after a disconnection to delete the AtomAssociations that are not connected anymore to the current AtomClass.

preEditName and *postEditName* are called when the name of the current AtomClass changes, to update the names in the connection list of the AtomAssociations connected to the current AtomClass.

2.3 AtomAssociation and AtomInheritance

The simplest form of relationship was implemented for the AtomAssociation and the AtomInheritance with a basic appearance. In AtomAssociation, two constraints were added to generate the ATOM3Connections if two or more AtomClasses are linked. The key attributes are AssociationName and InheritanceName.

2.4 Steps required to generate a modelling environment

2.4.1 Introduction

Now that we have created a meta-meta model, how can we generate a modelling environment from a meta-model? Since the algorithm to generate code from the entity-relationship formalism does not understand the meaning of "AtomInheritance", we must first use a graph grammar that will express its semantics by transferring the parent's attributes to the child and adding AtomAssociations (that are equivalent as the relationships in ER). And then, we will be able to generate the formalism when we execute the code-generation algorithm on any model. However, a special graph grammar for generating buttons at the lower meta-level need to be used by the code-generation alorithm. Lets go over each step in details.

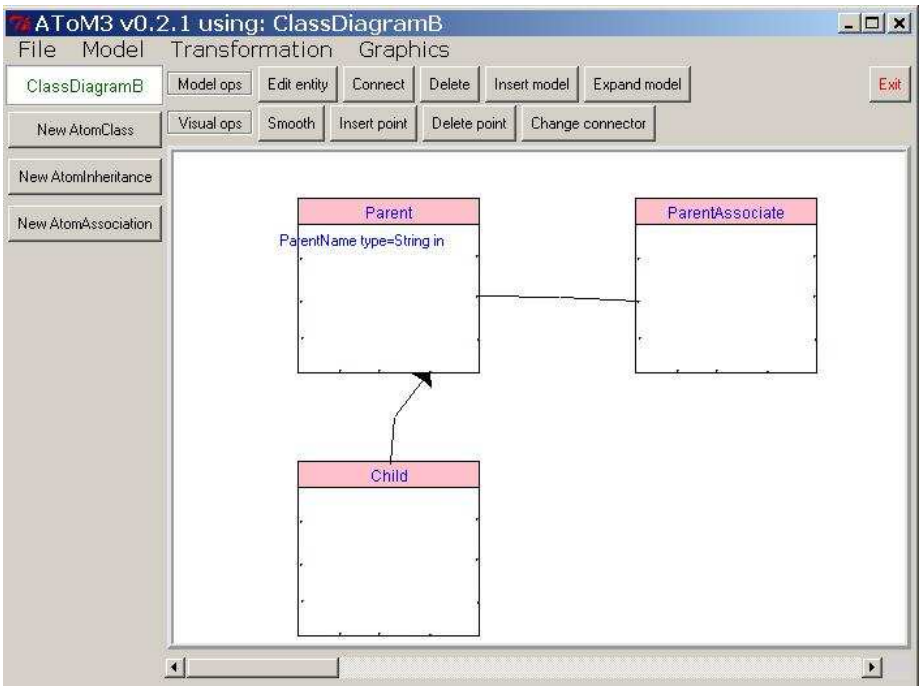

Figure 9: **Simple meta-model using entity-relationship with inheritance**

2.4.2 Creating a model

To create a model, open our formalism [6] as a meta-model in $ATOM^3$. you can see an example of a model on figure 9.

2.4.3 Running the GG to express inheritance semantics

The second step is to execute the graph grammar *CD_to_ER* to transform the current instance of the model into something that $ATOM^3$ will understand. How does the graph grammar works ? As you can see on figure 10 and 11, a simple pattern is matched by the graph grammar algorithm and replaced without inheritance.

In the condition part of the rule, we verify that the model is transformed

[6] Filename: ClassDiagramB.py

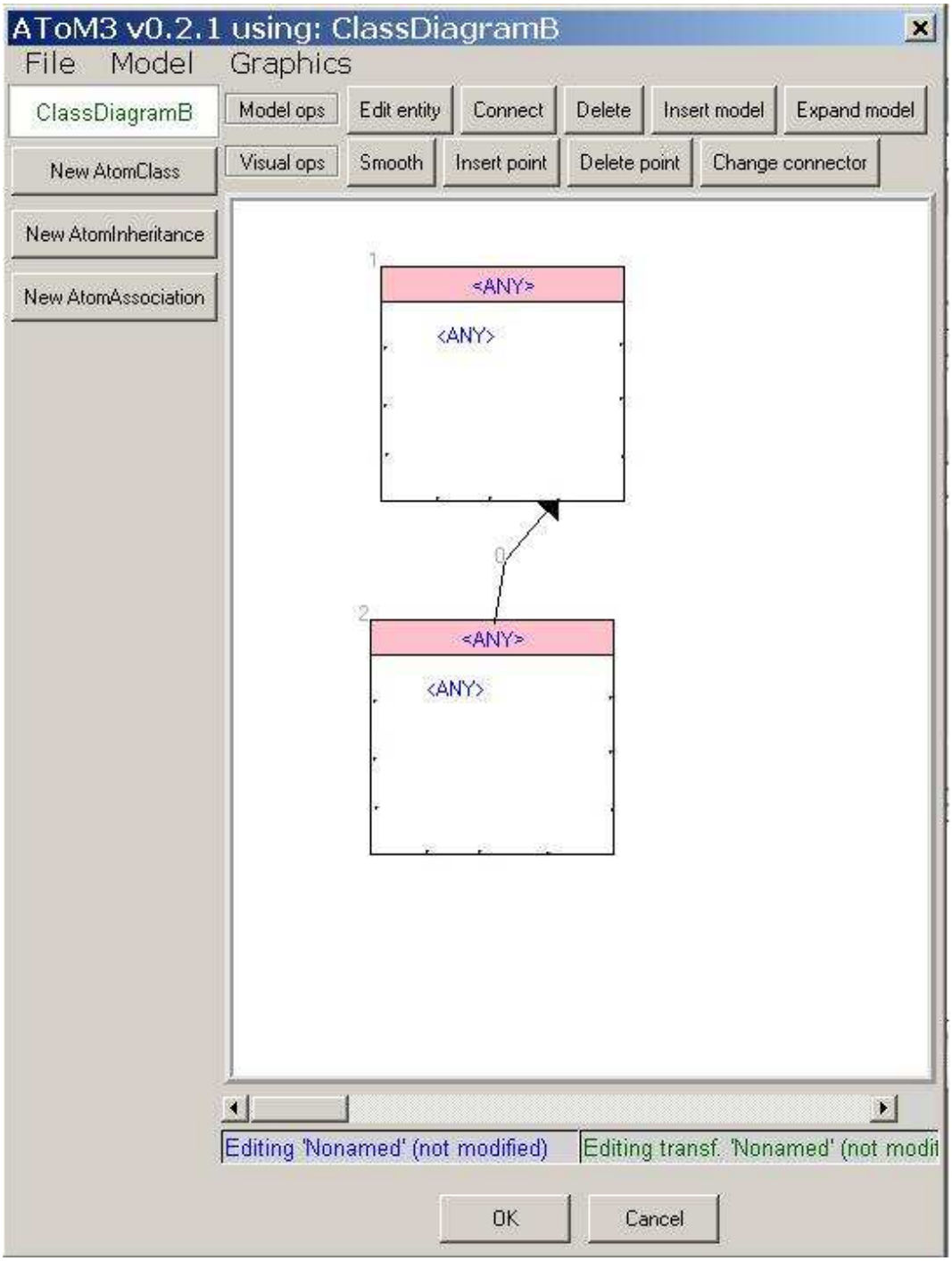

Figure 10: **LHS of the unique rule of *CD_to_ER* graph grammar**

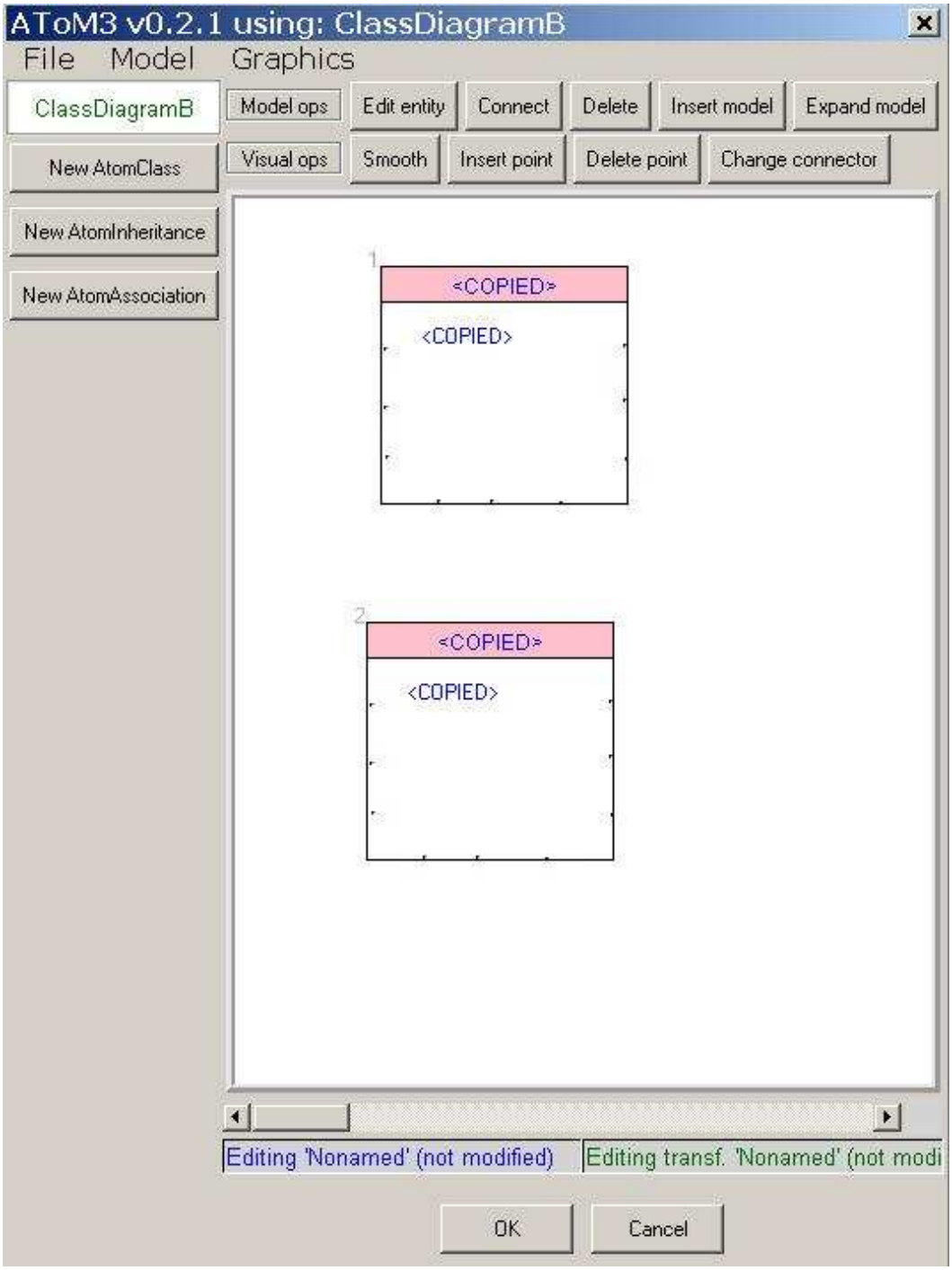

Figure 11: **RHS of the unique rule of *CD_to_ER* graph grammar**

in the right order. Indeed, we must transform the root of the model, at any given point in time, to ensure that every sub-type will inherit from all the parents. Take notice that, since the inheritance relation from the root is removed at each step of the graph grammar, we will always find a root, until the execution is finished.

In the action part of the rule, we copy, from the parent to the child, the attributes and the appearance if they are not already present in the child. Also, we must create the appropriate number of AtomAssociation. Indeed, every sub-types must be associated with the AtomClasses related to the parent-types. In order to to that, we have used the power of hyper-edges as you can see on figure 12 (which is the result of the graph grammar applied on figure 9). However, some constraint must be created to prevent the sub-types from being able to connect to other sub-types via the hyper-edges. [7] Unfortunately, this functionality was not implemented during the summer but I have understood how to do it. It could be implemented during the fall semester, if time permits it.

Finally, the last thing to do is to choose *CD_buttons* as a graph grammar to generate buttons at the lower meta-level. Indeed, $ATOM^3$ uses a graph grammar to generate the user interface of the lower modelling environment. Hence, I created a graph grammar for the class diagram formalism that generates buttons for AtomClass, AtomInheritance and AtomAssociation. You can see the modelling environment in action in figure 13.

[7] see my talk about this issue at http://moncs.cs.mcgill.ca/people/mprovost/present/index.html

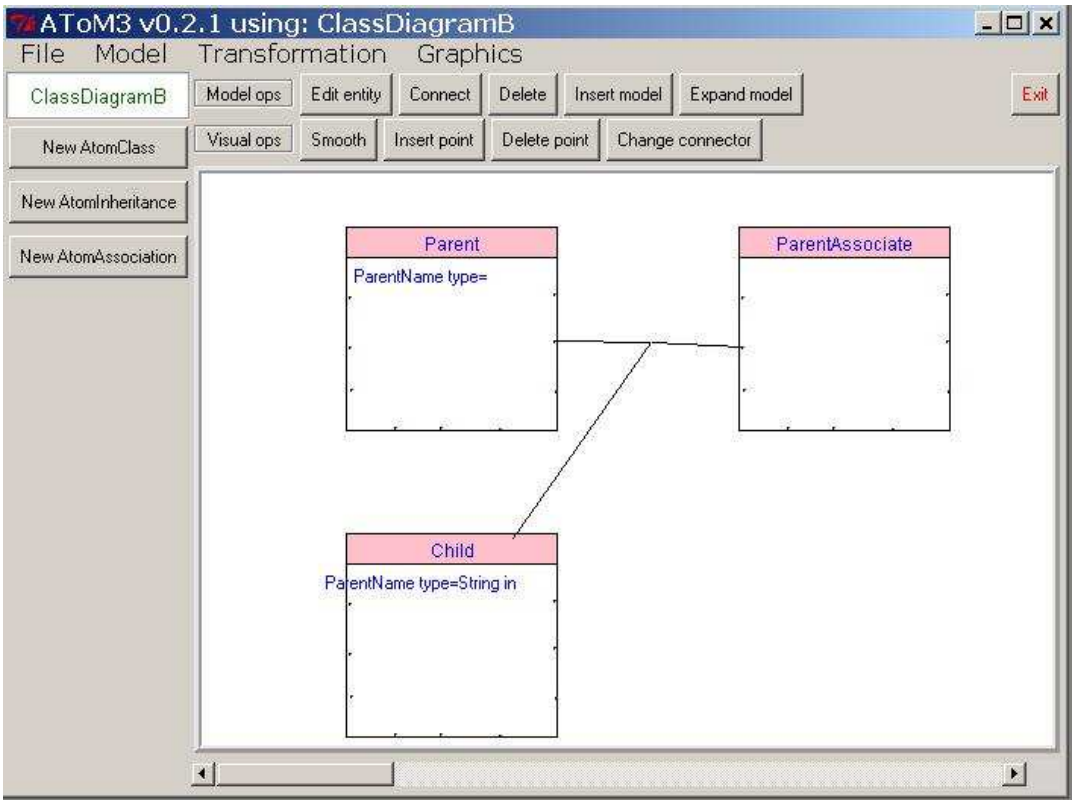

Figure 12: **Result of the graph grammar applied on figure 9**

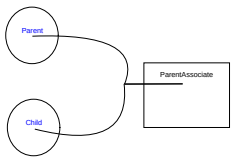

Figure 13: **Parent/Child modelling environment example**

3 Conclusion

I think that the primary objective of the summer was attained: convincing me to pursue a career in research. However, some details need to be improved in order that my UML class diagram modelling environment be near the level of commercial tools. Still, it can be used for "local" modelling, and suggestion of improvement are appreciated! Also, my approach of meta-meta modelling has proven that we must update the graph grammar engine before becoming able to implement inheritance completely in $AToM^3$. But, my approach can still be reproduced to add inheritance in small projects that do not need a powerful graph grammar knowing about the implemented semantics. [8]. Finally, the amount of knowledge I acquired is quite satisfying, and I will continue to learn about meta-modelling in the next several years.

If there are any questions, fell free to email me mprovo1@cs.mcgill.ca

[8] for example, pattern matching of sub-types