Model Completion

Robbe Claessens

Univerity of Antwerp

robbe.claessens@student.uantwerpen.be

Abstract

Model completion is a great tool that enhances the productivity of a user. It checks, and completes, whether the provided model conforms to the meta-model. If not, it gets completed. We will create a model completion proof of concept in AtomPM, based on the Diagram Predicate Framework[1]. Beside the completion rules completing the partial models we need to take a look at transformation rules that form critical pairs.

Keywords: `Model Completion`, AtomPM, Diagram Predicate Framework, Critical Analysis

1. Introduction

We have all worked with IDE's for various programming languages. Every good IDE will have a auto-complete functionality. Autocompletion can vary from the completion of variable names to the keywords defined in the programming language. But this can also be applied in the area of modelling languages. What follows is a motivation of why model completion is useful. In section 2 we will discuss the approach used to implement model completion in AtomPM. In section 3 we will talk about critical analysis. At last, section 5 gives a conclusion of the paper.

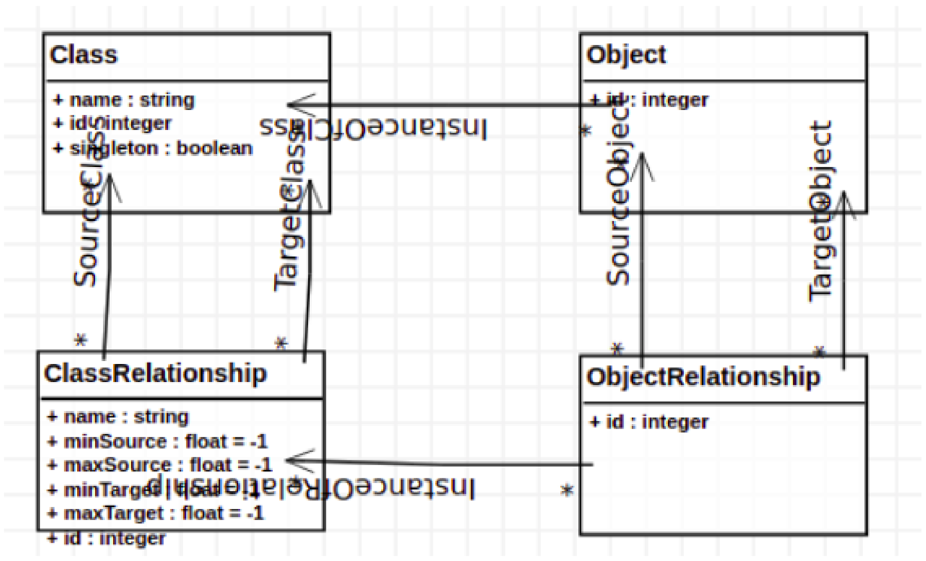
2. Motivation

Autocomplete functionality is a great tool that enhances the productivity of the modeller. But another great use of autocompletion is to check that a user didn't forgot to add something. It makes sure that the user makes a correct model from a metamodel.

3. Approach

3.1. The mini environment

The first thing I did, was to work out a proof of concept with AtomPM, based on the Diagram Predicate Framework and more specifically, the DPF Workbench[2] for model completion. We will first build a general language that can be used to model any metamodel. That meta modelling language looks something likes this:



The next step I did is to add some completion rules in the form of transformation rules (we are working with AtomPM). It is important to have very generic rules, because we are not aware of the model that the user will create with our meta modelling language. Let us take a look at the following generic constraints:

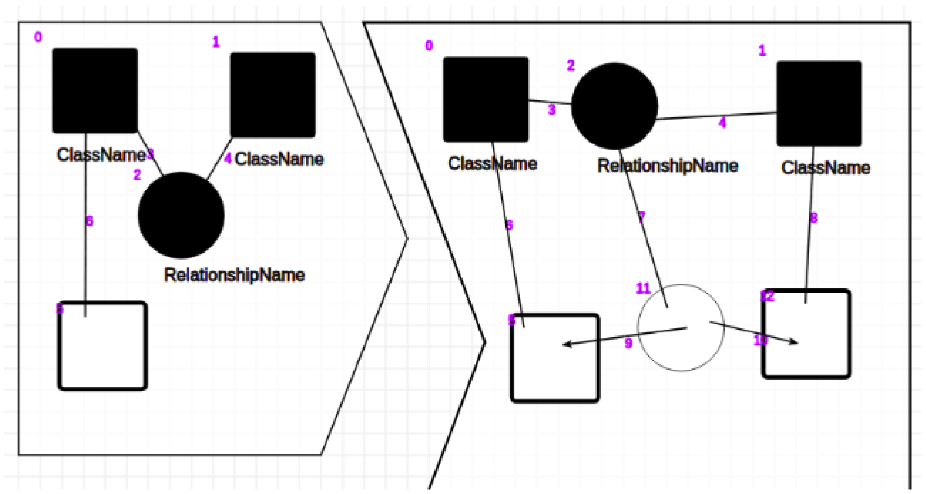
1. Cardinality constraint: when defining a model of our language generic metamodel. We want to be able to represent cardinality constraints. A

great completion rule would be to add modelling elements based on the cardinality constraints. For example, if an object of class A needs at least 1 object of class B, but the user created model does not satisfies that cardinality, the autocomplete functionality can simply add an object of class B and link it to the object of class A.

2. Sometimes, we want to make sure that each object of class A has an equal amount of objects of class B. This would also be a generic rule that we can implement.
3. A singleton class constraint: sometimes we only want 1 instance of a given class A.
4. Conditional constraint: if a relationship ‘a’ exists on an instance of class A, then a relationship ‘b’ must also be present on that instance.

Constraint 1:

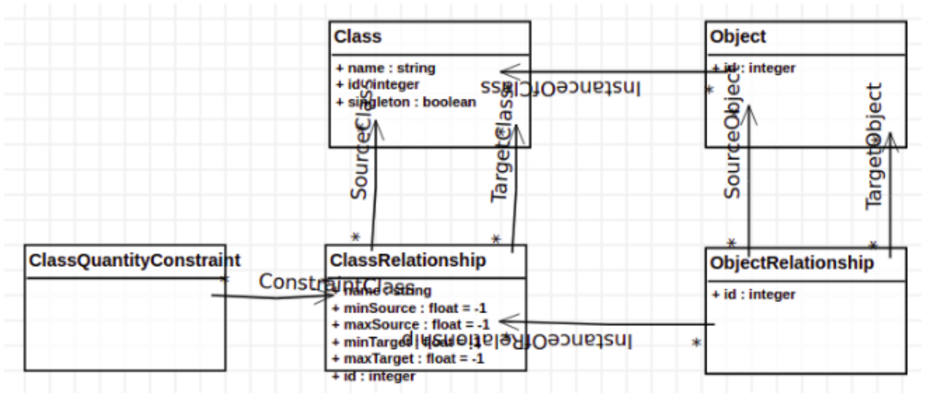
Constraint 1 his completion rule looks as follows:



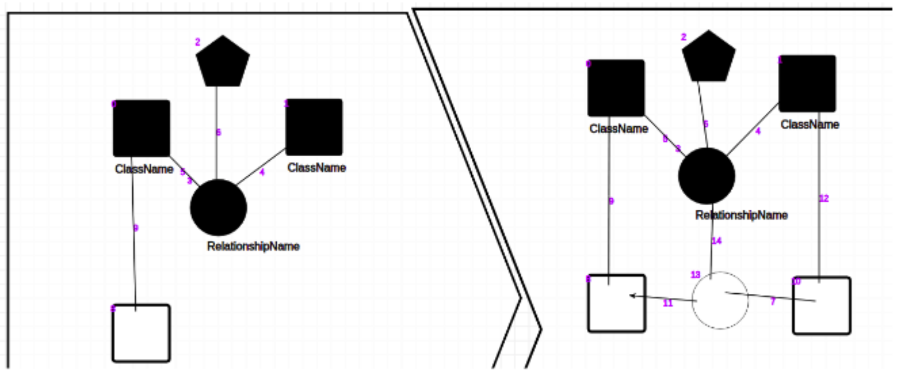
Whenever we find the pattern as described in the LHS and there is been set a minimum number of cardinality (minTarget or minSource) then the rule will add one extra object. It also checks whether there is reached this number already, if so it will not execute the rule. Note that the addition is done one by one. We use a so called A-rule to execute this rule in the schedule.

Constraint 2:

We will implement this constraint with an additional class, to visualize it more clearly. We change our metamodel to the following:



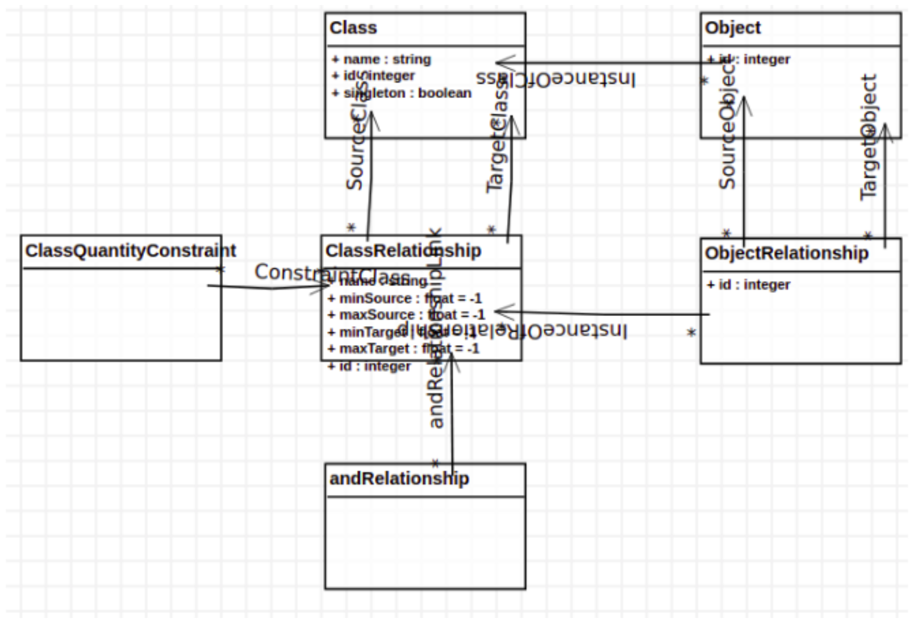
The ClassQuantityConstraints' visual representation is a black pentagon, as can be seen in the rule below.



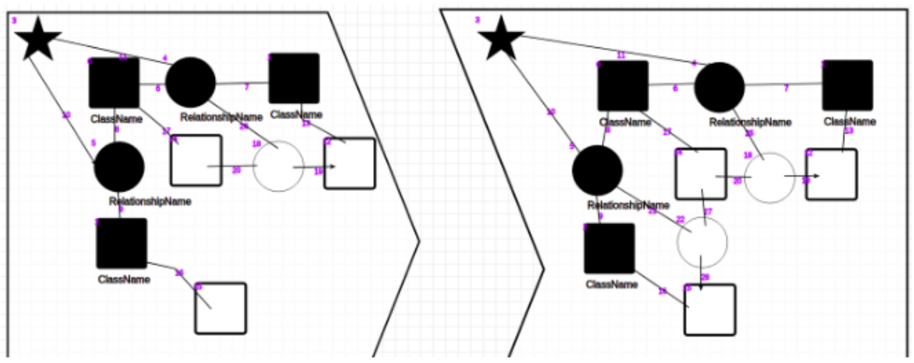
This completion rules makes sure that for each object of class A has the same amount of objects of class B. The rule checks for each found match, that the number of relationships with class B is the highest among all objects A, if this is not the case then the completion rule will add an object. This is also an A-rule as in the previous rule.

Constraint 3: We simply add an attribute to the Class 'Class', when a class is a singleton, its border will change to a red color to indicate the presence of a singleton class.

Constraint 4: As in constraint 2, I choose to add a symbol (a star) to represent the conditional select. Hence, we need to change our metamodel again:

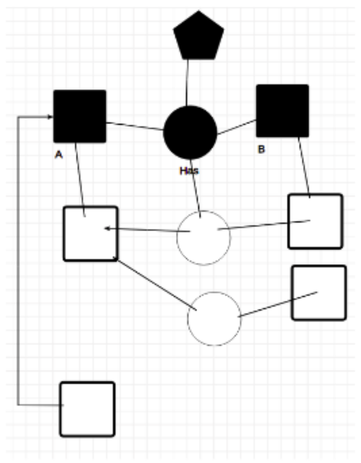


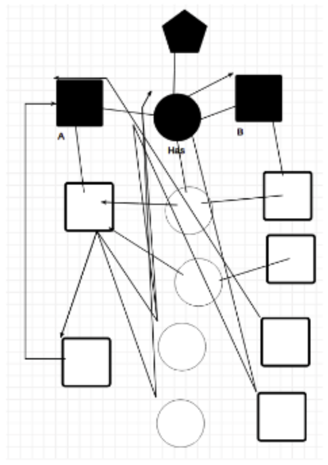
This rule looks as follows:



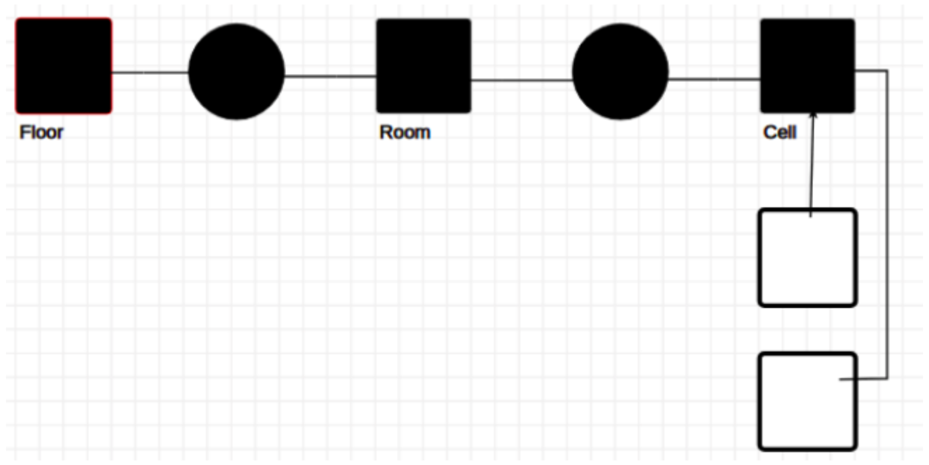
This rule will, without going in depth and exploring the NaC's, check whether a specific relationship is between two objects. If this is the case, and the relationship is noted with a star, another relationship must be present as well.

When we test each individual rule, everything works fine and the models get completed the way we want them to. For example when testing constraint 2:

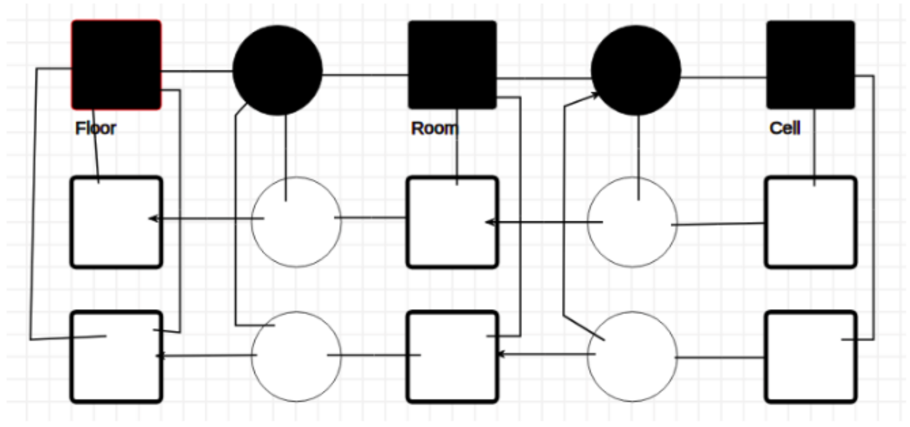




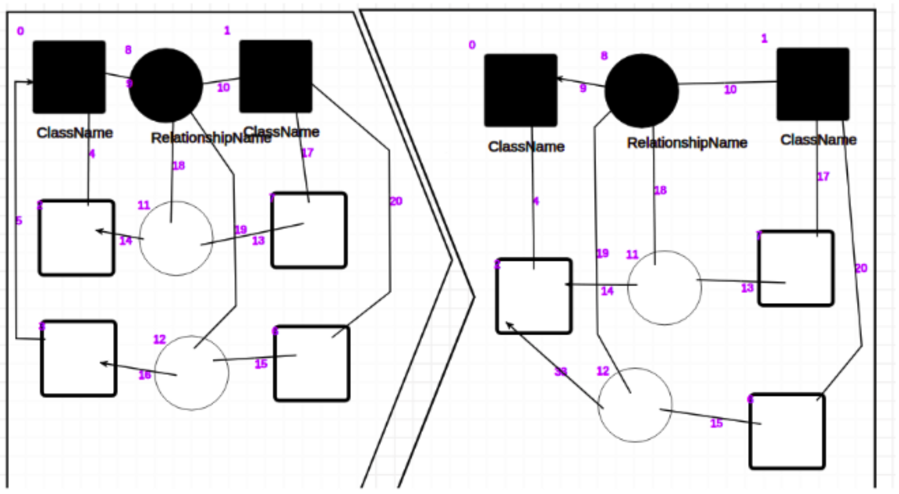
But now comes a rather interesting part, what if we combine everything? We can quickly see that combining two rules will result in a faulty model, namely the singleton constraint and the cardinality constraint. For instance, let us consider a formalism describing a building. A building consist out of a floor, a floor out of rooms and rooms out of cells. Let us define the class Floor as a singleton. Each cell belongs to 1 room and each room belongs to 1 floor. Now, take a look at the following example:



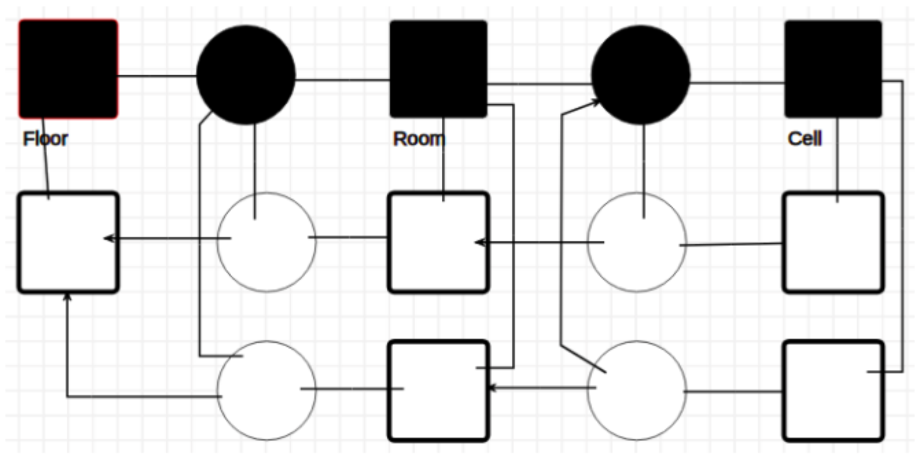
When we let our cardinality rules run, the first run it will create a room for both cells and the run afterwards it will create a floor for each of these rooms. But, we declared the floor as a singleton class. This results in a faulty model:



This takes us to a rather important aspect of model completion, solving potential conflicts. We can easily solve this by making an additional rule that looks as follows (where the class with label 0 is a singleton):

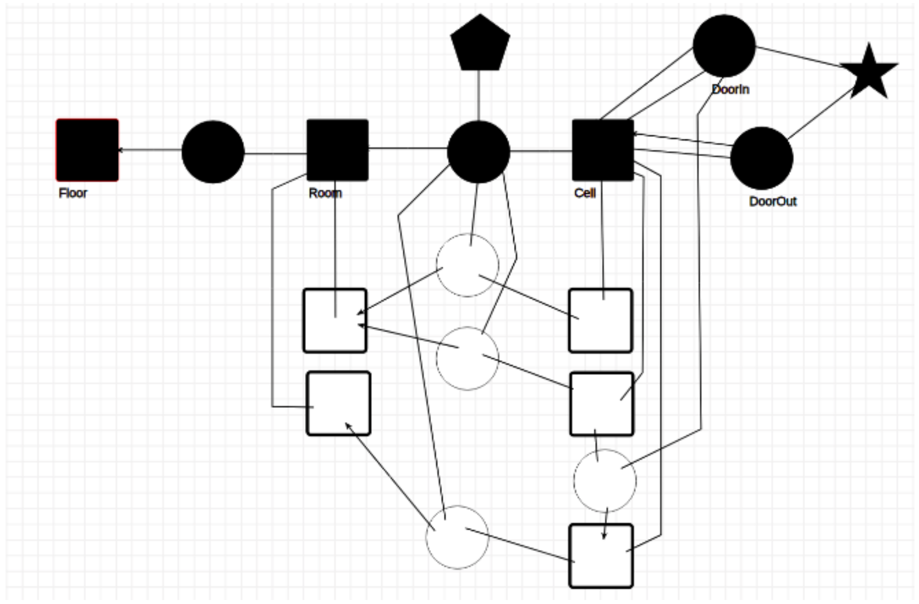


This will fix the conflict we had before:

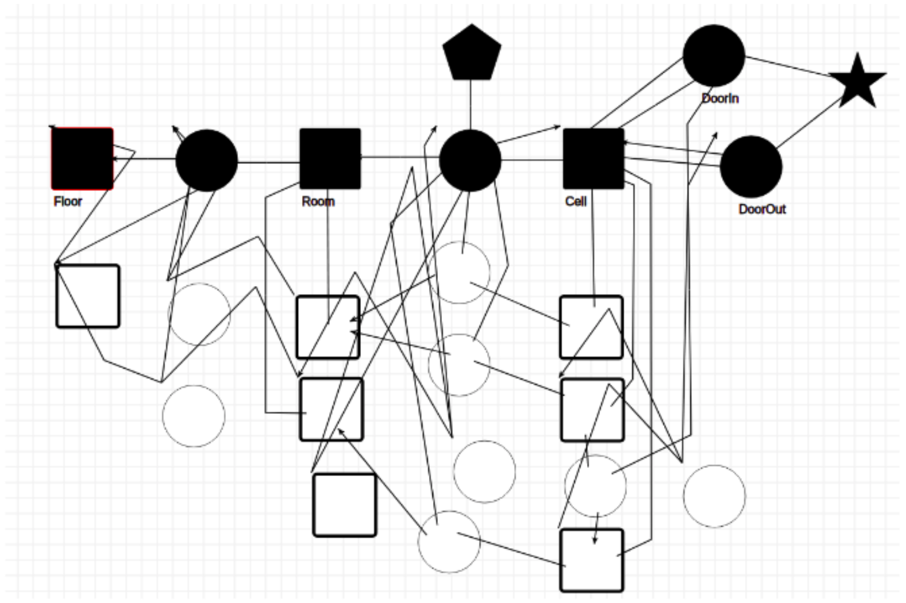


This way a simple solution, but with only a few rules, we already have conflicting ones. When implementing a lot of generic rules, the probability of two or more rules conflicting with each other, becomes high. And the more rules the more complex it gets. There are ways to detect such rules, we will see one in the next section.

Let us take a look at another example which has no conflicting rules. Let us take the following model in consideration:



We see that the conforming model is a partial model, because of the conditional constraint (illustrated with the star), because of the equal number of object constraint (illustrated with a pentagon) and also because of the singleton constraint (on the floor class). When executing our rules we get the following result:



Our partial model has been transformed in a complete model. Now let us take a look at the critical analysis.

4. Critical Analysis

We have seen in the previous section that we already have a few problems with a limited amount of completion rules. Although we could check every conflicting rules manually, it is not a desired way of doing this. A way to detect whether a pair of rules could interfere with each other, is by doing a so called Critical Pair Analysis. This technique check for all the rules if there are critical pairs, meaning that they could interfere. Two rules could be interfering if:

1. One rule application deletes a graph object which is in the match of another rule application.
2. One rule application generates graph objects in a way that a graph structure would occur which is prohibited by a negative application condition (NAC) of another rule application.
3. One rule application changes attributes being in the match of another rule application.

Unfortunately, AtomPM has no standard way of doing a critical pair analysis. There are some external tools that can be used to check whether your rules are conflicting with each other. You can read more about it at the article (http://www.user.tu-berlin.de/o.runge/agg/AGG-ShortManual/node36.html).

5. Conclusion

We worked out a simple proof of concept that illustrates the working of modeling complection in AtomPM based on DPF. We saw that even when implementing a small amount of rules, it can already lead to conflicts. Needless to say, the number of conflicts rises with the number of general rules. It is very important to adopt a strategy to check whether there are any conflicts or maybe build an automated tool that detects possible conflicts among the metamodel and rules or between rules themselves.

6. Further Work

A next stage would be to to implement more generic rules, based on languages like OCL. Even implementing this would be a great challenge. Another very interesting thing to do, is to let the user creates its own constraints based on the metamodel relationships. This is what DPF does. So creating an environment where it possible to work with DPF would surely be a very interesting way to go. A last thing would be to implement the critical analysis to make sure that the user is aware of the critical pairs in his/her rules. This would allow the modeller to be certain that the non-determinism of the tool would not lead to a deadlock or to unexpected transformation results.

7. Acknowledgments

I would like to give special thanks to Claudio Gomes, he helped me with the realisation of this paper.

8. Related Work

References

[1] F. M. W. M. A. R. Yngve Lamo, Xiaoliang Wang, Dpf workbench: A diagrammatic multi-layer domain specific (meta-)modelling environment.

[2] I. C. Y. L. M. K. Fazle Rabbi, Yngve Lamo, A diagrammatic approach to model completion.