A Matching Algorithm and AGG Overview

Marc Provost
McGill University marc.provost@mail.mcgill.ca

March 29, 2004

Abstract

This presentation go over the basic features of agg for graph rewriting.

Structure of the talk

- Introduction
- AGG Features
- Petrinet Graph Grammar Simulator (SPO)
- Petrinet simulator implementation

AGG [1]: Introduction

- AGG [3] [1] is a tool implementing the single pushout approach (SPO) for graph rewriting.
- In the SPO approach [2], there is a partial homomorphism from the LHS to the RHS. This mapping describes which graph elements are kept, deleted and created (figure 1)
- The SPO is a "relaxed" version of the DPO, in the sense that it allows dangling edges to be deleted (figure 1) (And it does not have the identification condition in the case of non-injective matches).

[1] http://tfs.cs.tu-berlin.de/agg/

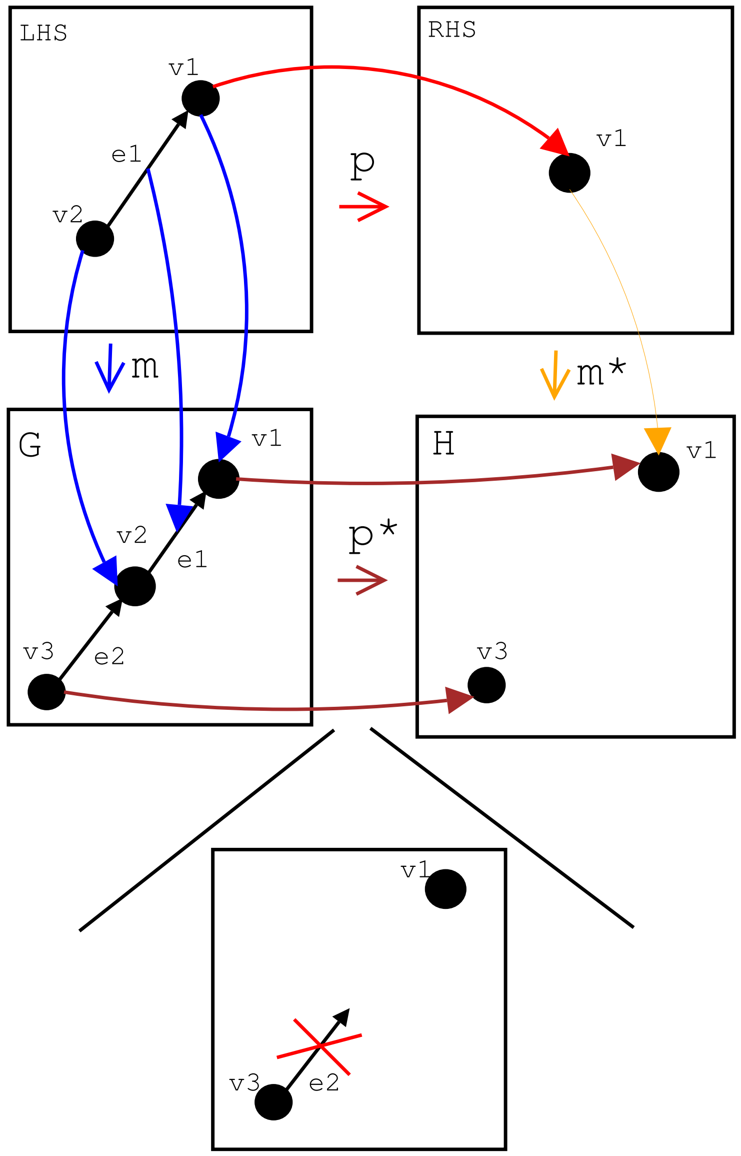

Figure 1: SPO approach

AGG: Basic Features

- At the GUI level, AGG has support for directed *simple graphs*, which have no self-loop.

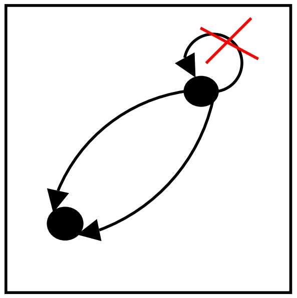

Figure 2: Simple Graph

- Nodes and edges also have attributes, which can be any valid java Object.

- An attribute has a name (unique), a type and a value.
- A graph must be a member of a *graph transformation unit* in order to be transformed
  - Type Graph
  - Start Graph
  - Set of rules
- The first element of a graph transformation unit is a *type graph* (close to AToM^3 meta-model).
- In the type graph, one specify the allowed attributes and the cardinality constraints for each node and arc that will exist in the graph.
- The implicit modelling formalism used in the type graph is entity-relationship. Vertex (Entity) and edge (Relationship) type are created

and connected together.

- The second element of the unit is the start graph (host graph) on which the transformation will be applied.

- Next, a set of rules describe the actual transformation. Each rule has
  - A left-hand side (LHS) graph, specifying a pattern to be transformed
  - A right-hand side (RHS) graph, specifying the pattern after the transformation
  - A vertex mapping from the LHS to the RHS specify the elements that are preserved, deleted and created after the transformation.
  - Potentially, a Negative Application Condition (NAC), which specify a pattern *not* to be located in the LHS. A mapping from the LHS to the NAC, is also needed to relate vertices of the LHS and the NAC.

A Practical example: Petrinets

- To illustrate the basic features of AGG, the Petrinets formalism was meta-modelled, along with a graph-grammar based simulator.

- Note here that we are not talking about place-transition systems, but of original petrinets where places cannot have more than one token.

- The first step: Design the type graph. This is quite simple, as seen on figure 3

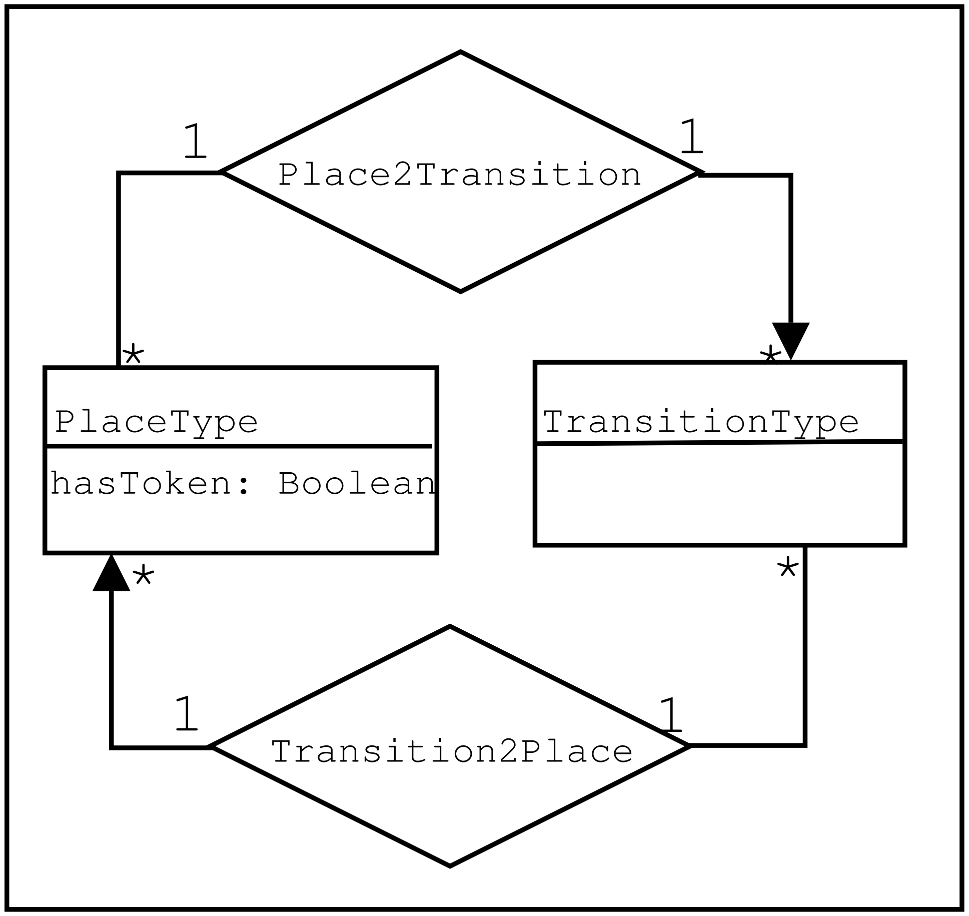

Figure 3: Petrinets Type Graph

- Next step: Design the graph grammar rules. We want to simulate a petrinet. Conditions for a transition to be enabled:
  - All of its input places contain a token.
  - None of its output places contain a token.

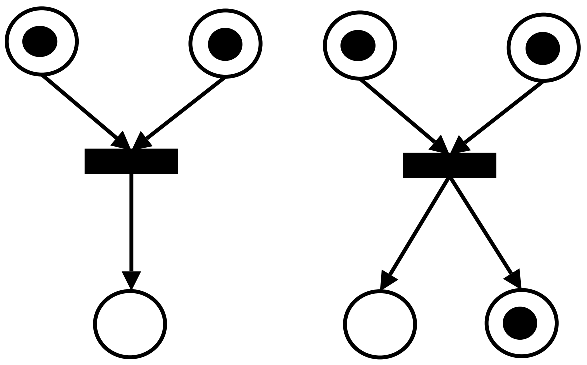

Figure 4: The transition to the left is enabled while the transition to the right is not enabled.

- Once a transition is enabled, it can fire! Firing means removing all the tokens of the input places and adding a token in all of the output places.

- Multiple transitions can be enabled at the same time and fire in parallel if they are independent.

- First of all, we will assume that the transitions are fired sequentially.

- General Idea of a GG rule: Its LHS will match an enabled transition. Its RHS will "fire" the transition. In our case, this means setting the variable hasToken to false in all input places and to true in all output places.

- Start simple: A rule with only one input and one output place:

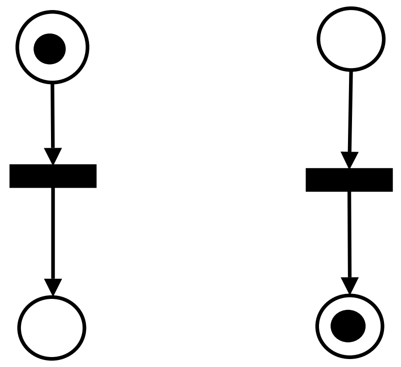

Figure 5: Is this rule complete??

- This rule is NOT complete. We didn't even make sure that the transition was enabled!

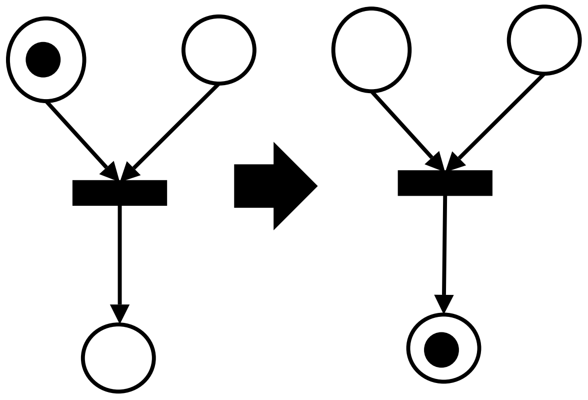

Figure 6: The previous rule would match this!

- To solve this problem, we can make use of AGG Negative Application Conditions:

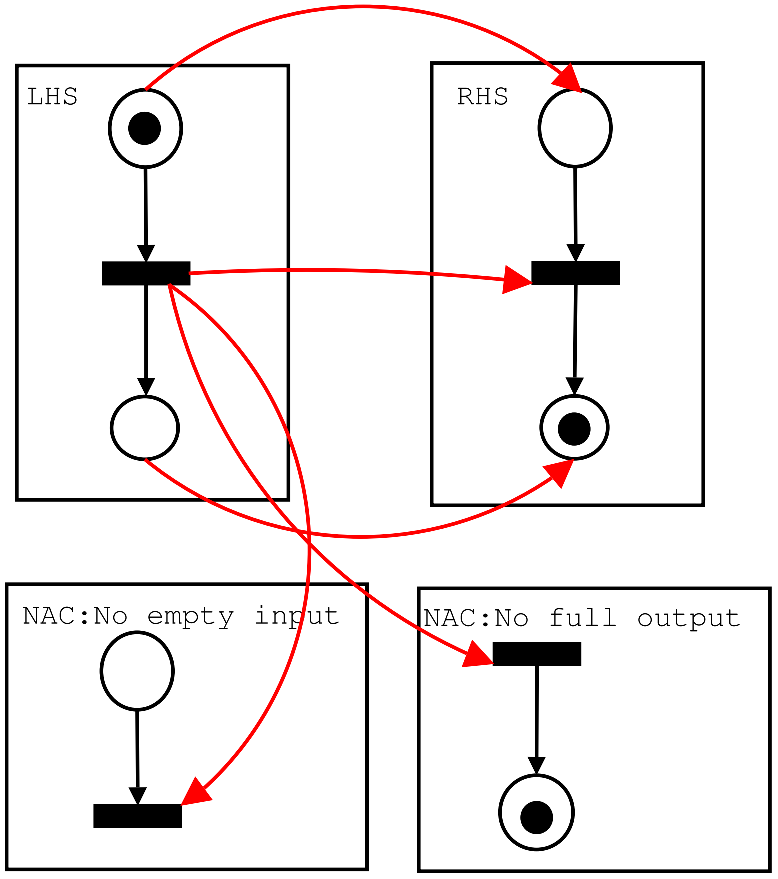

Figure 7: Our first rule of the petrinet simulator

- The previous rule will simulate correctly any transition with only one input/output place. Other petrinets will be simulated incorrectly:

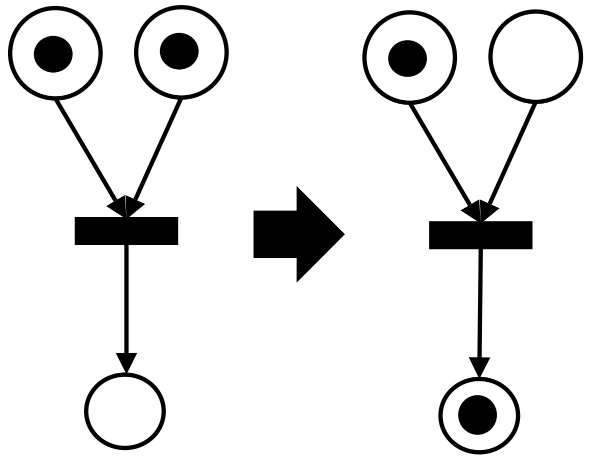

Figure 8: Our first rule does not simulate this correctly

- The graph grammar need to be extended to execute correctly any general petrinet. Does this mean adding a rule for each possible configuration of a transition??

- AGG supports non-injective matches. Could we make use of this feature? If we define a similar rule with n input/output places, and execute the graph grammar with non-injective matches, all the transitions with $x < n$ input/output places will be matched. Will this work?

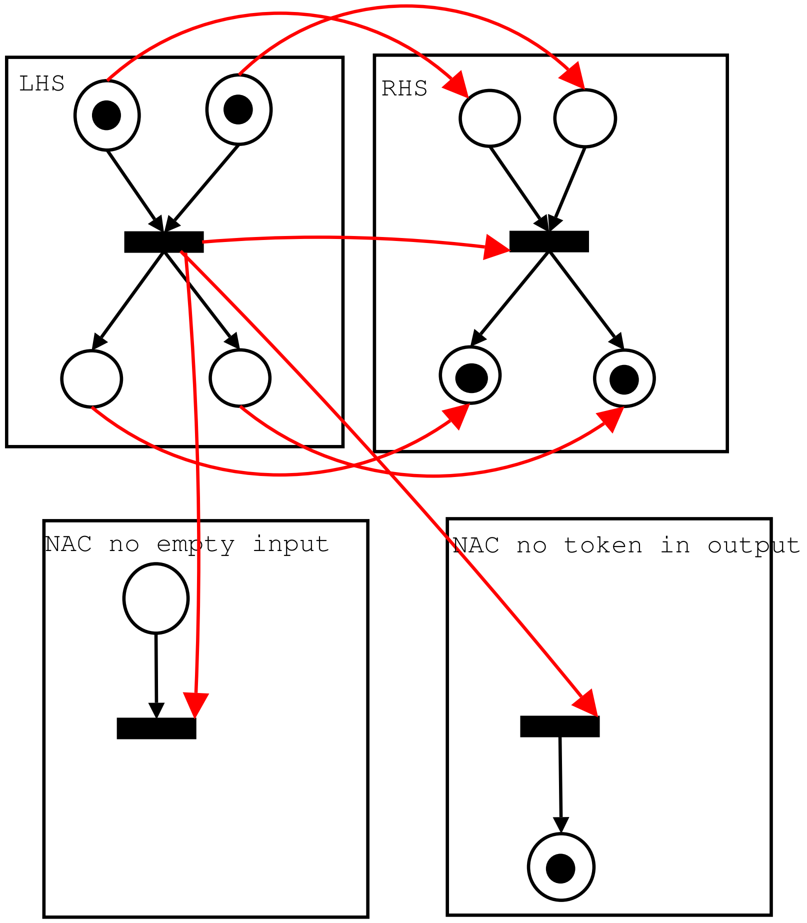

Figure 9: An extension of the first rule to 2 input/output places

- If we use non-injective matches with the previous rule on the following simple petri-net, 3 matches are generated, 1 being legal. We cannot use this solution...

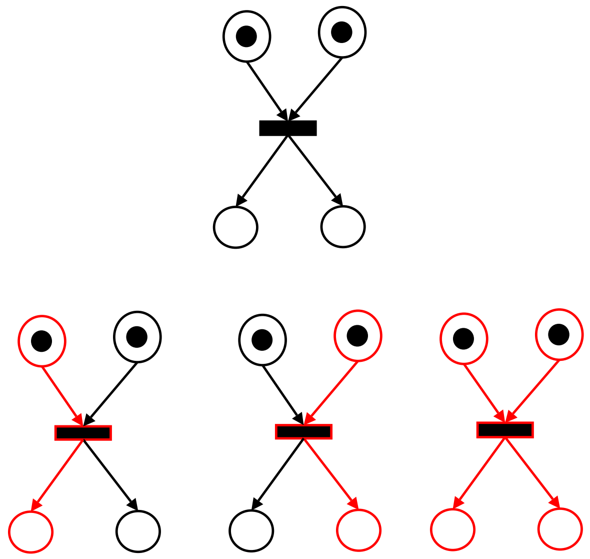

Figure 10: Non injective matches would not work on this graph.

- The lack of expressiveness of the SPO force us to generate all possible patterns. But how can we know which patterns will occur in the host graph in advance?? This is, of course, impossible...

- The easy solution would be to assume that the maximum degree of a given transition is n and generate all the required *firing* patterns having the degre $\leq n$. For instance, with n=3:

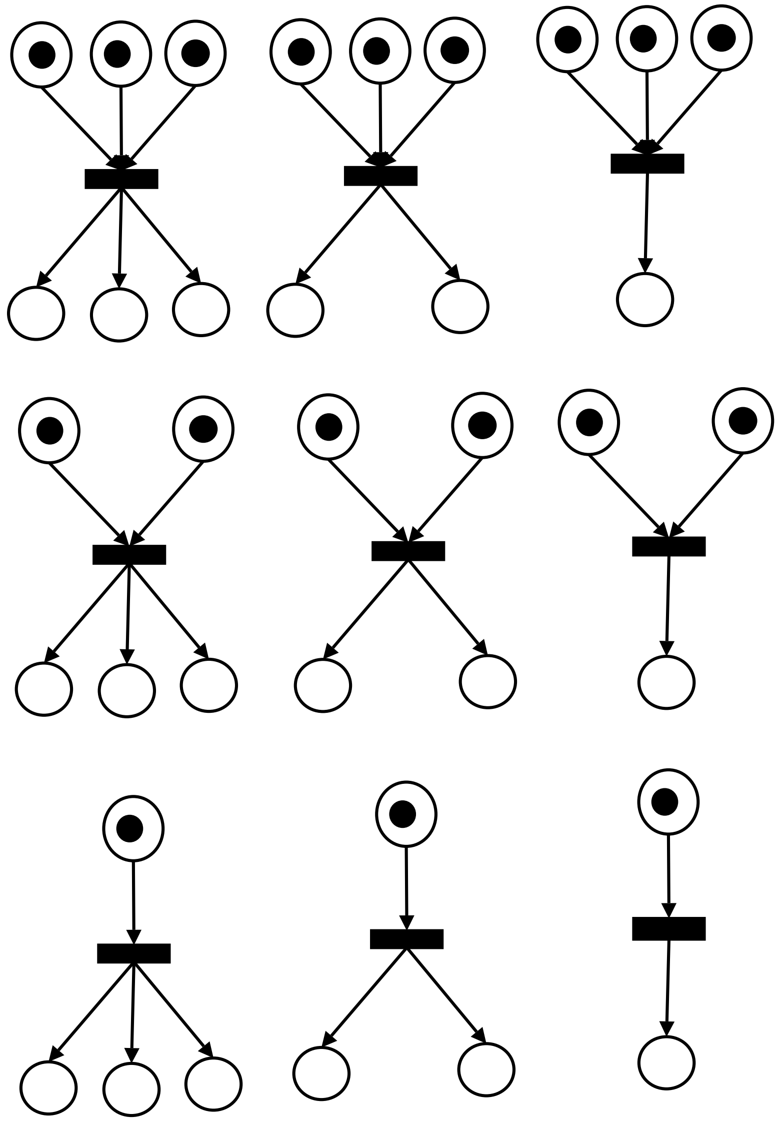

Figure 11: Notice that there are n^2 possible firing patterns for degree n.

- This approach is silly. First of all, we would have to choose a relatively big n to cover all the expected petrinets. And, we would never be sure that one day or another, some petrinet will not be simulated correctly.
- Moreover, most of the patterns will not occur, n^2 can become quite large.
- Another idea: Generate simulators on demand. That is, given an input host graph to be simulated, have another graph grammar generating required simulator rules. Several difficulties occur:
  - There are a lot of rules to generate. However, a graph grammar transform a graph into ONE graph. Since the graph grammar is not a graph itself, but a collection of graphs, we would need $(n^2\ LHS, n^2\ RHS, n^2\ NAC)$ graphs in the worst case. This is the problem with AGG, a graph grammar rule is not a graph by itself, it is a collection of graphs : (LHS, RHS, NAC). A

graph grammar is a collection of rules along with a host graph: $((LHS, RHS, NAC)+, Host)$.

– Suppose we are trying to generate one firing pattern. We start with $((LHS, RHS, NAC)+, Host)$. In this case, $Host$ is the graph that we want to simulate. We have several graph grammar rules expressing how to transform $Host$ into ... what? Well, it could be say, the LHS of the rule. But we also need the RHS. And we have possibly n^2 rules. Thus, we would need n^2 graph grammars...?

– To avoid having n^2 graph grammars, we could generate all the LHS,RHS and the NAC into the same graph as separate components and then, manually retrieve the graphs. But, if we are doing “manual” work, why not simply manually generate the simulator? This would be quite easy if we could simply scan the graphs and generate the rules.

– Lastly, is generating the simulator equivalent to simulating it? We basically need to find all the different firing patterns and generate rules for them. Well, the simulator is exactly doing this. Could the simulator generator by a simulator itself?

- For the reasons discussed before, I decided not to implement a simulator generator based on graph grammars. Instead, the host graph will be manually traversed to determine the maximal in and out degrees. This information will be used to generate the required rules. This approach has the following disadvantages:

  - The simulator is not completely based on a graph grammar
  - MANY rules will not be used during the simulation. This is because we are using SPO: "programming without iteration".
  - Becomes very VERY slow as the maximal degree increases. We are creating quadratic number of rules at EACH step...

  And the following advantages:

  - The core of the simulator is a graph grammar.
  - Any given petrinet model will be simulated correctly.

- There is still a problem with the graph grammar petrinet simulator. It is not working at all! Why? This is related to the order in which the rules are applied, if some rules are executed before others the operational semantic will be incorrect:

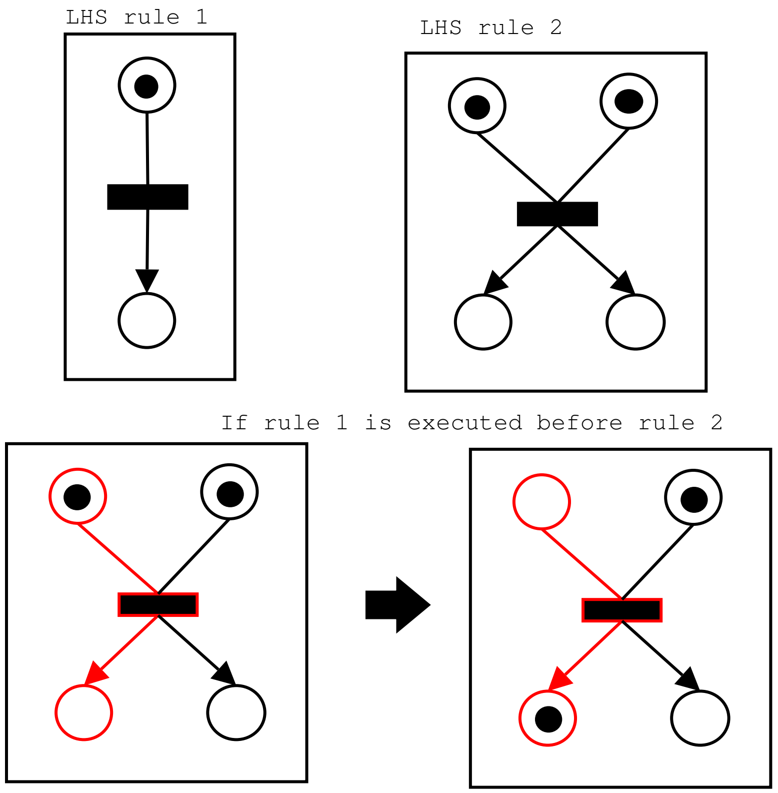

Figure 12: Rule order problem.

- We could use a *layered* graph grammar, that is giving priorities to the rules. But this is incorrect. Why?

- The solution, is to add two NACs to each rule, specifying that it cannot be applied if there is another input place with a token or another output place without a token:

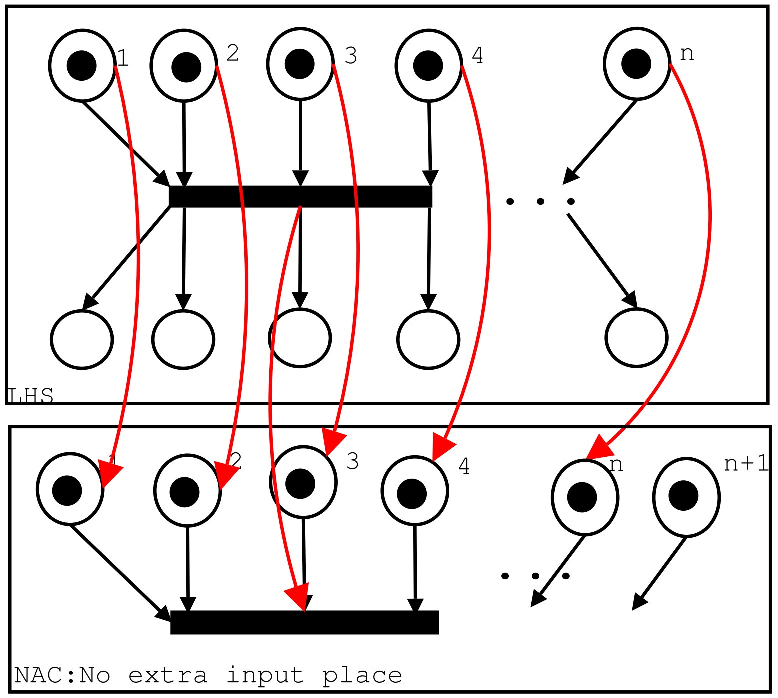

Figure 13: We add a NAC to prevent the rule from applying if there is an extra input place. We add a similar NAC for the output places.

Petrinets implementation

- The non-graphical part of AGG (basically, the graph transformation kernel) was linked to a petrinet graphical user interface used in CS623 (thanks to Clark Verbrugge)
- First step, create the petrinet type graph
- If we are in optimize mode, generate the rules for a preset maximal in/out transition degree.
- Then, wait for the message “doStep” from the GUI.
- When receive the “doStep” message:

- Convert the GUI petrinet representation into a valid AGG graph with petrinet type. This will become the host graph of our graph grammar.
- If we are not in optimize mode, generate the required rules, nacs.
- Apply one step of the graph grammar. This will randomly select a rule, until one can apply. Some enabled transition will be fired.
- Reconvert from the AGG graph to the GUI representation.
- Return the transformed graph to the GUI.

References

[1] T. S. Claudia Ermel. The agg environment: A short manual. Technical report.

[2] A. Corradini, H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, and A. Wagner. Algebraic approaches to graph transformation - part

I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*, chapter 4, pages 247–312. World Scientific, 1997.

[3] M. Rudolf and G. Taenzter. Introduction to the language concepts of agg. Technical report, 1999.