6.2. Advanced Concepts of Graph Transformation

Juan de Lara, Elena Gomez, Esther Guerra
{Juan.deLara, MariaElena.Gomez, Esther.Guerra}@uam.es
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Index

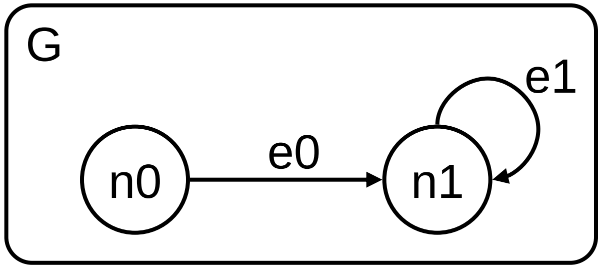
- **Formalizations** 
- Analysis
- Graph Constraints and Application Conditions.
- Triple Graph Grammars.
- Conclusions.

Formalizations

- Rules are not only nice pictures, they have a mathematical underpinning, which allows precise reasoning.
- The two main formalizations of GT are:
  - Double pushout.
  - Single pushout.
- Both formalizations use concepts from category theory. [https://en.wikipedia.org/wiki/Category_theory]

Formalizations.
Graph

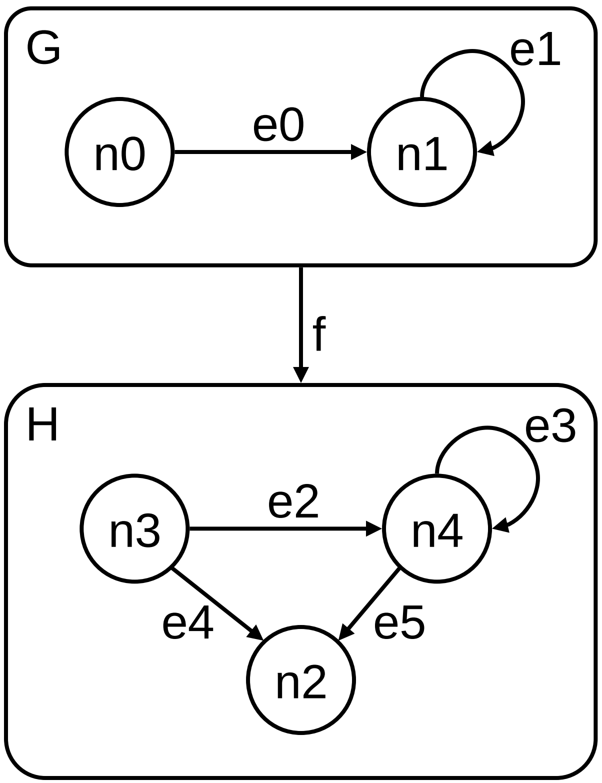
- Graphs can be encoded using sets and functions.
- G = (V, E, src, tgt)
  - V = Set of nodes (vertices).
  - E = Set of edges.
  - src: E→V (gives the source node of an edge).
  - tgt: E→V (gives the target node of an edge).



V={n0, n1}
E={e0, e1}
src={(e0, n0), (e1, n1)}
tgt={(e0, n1), (e1, n1)}

Graph morphisms

- How to identify a smaller graph into a bigger one?
- How to identify elements of the LHS and RHS of a rule?

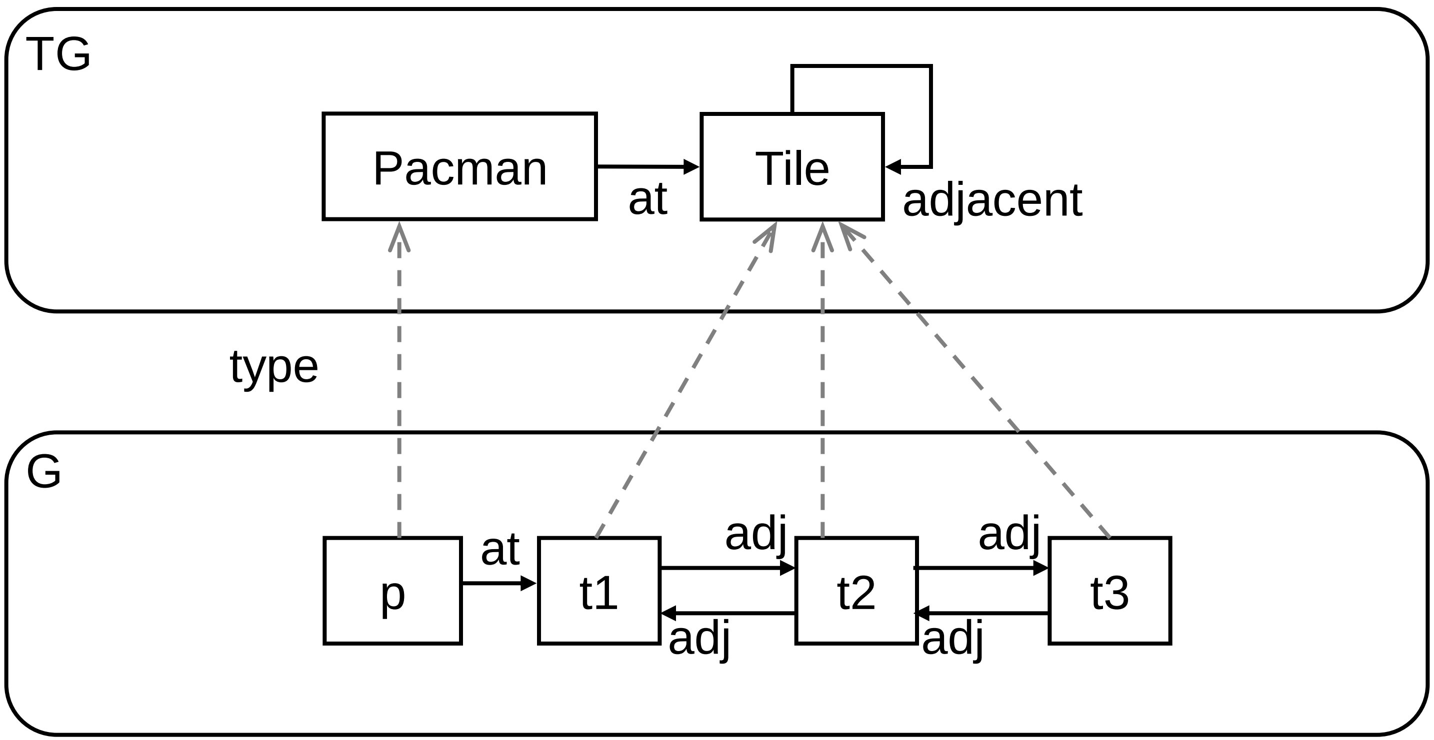


- f maps nodes to nodes and edges to edges.
- $f=(f^V, f^E)$:
  - $f^V: V_G \rightarrow V_H$
  - $f^E: E_G \rightarrow E_H$
- f has to preserve the structure of the graph, so for all edge e:
  - $f^V(src_G(e))=src_H(f^E(e))$
  - $f^V(tgt_G(e))=tgt_H(f^E(e))$

Graph Morphisms

Typing

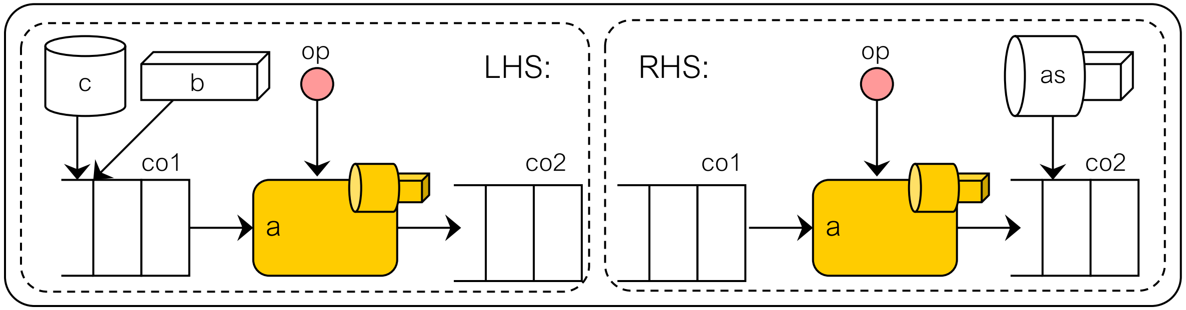
- Morphisms are also used to represent the type-instance relation between models and meta-models.



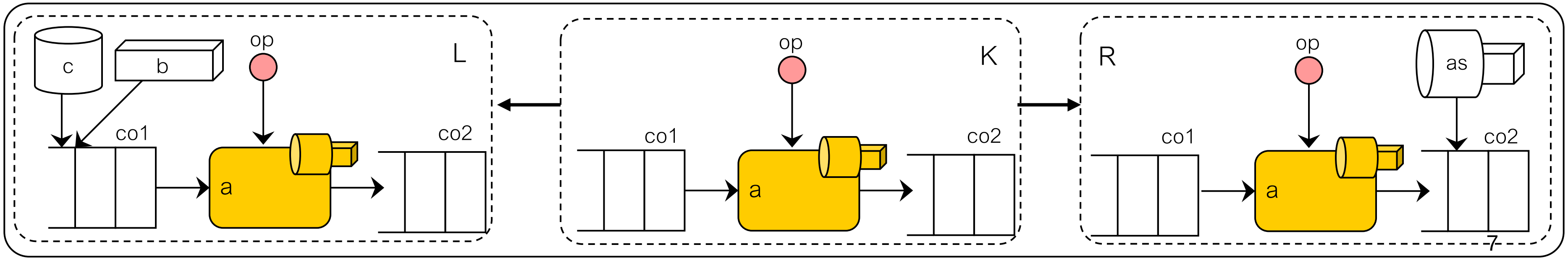
Rules in DPO

- However in general, given a rule we cannot say that its LHS is bigger than its RHS or the other way round.
- How do we represent rules?

assemble

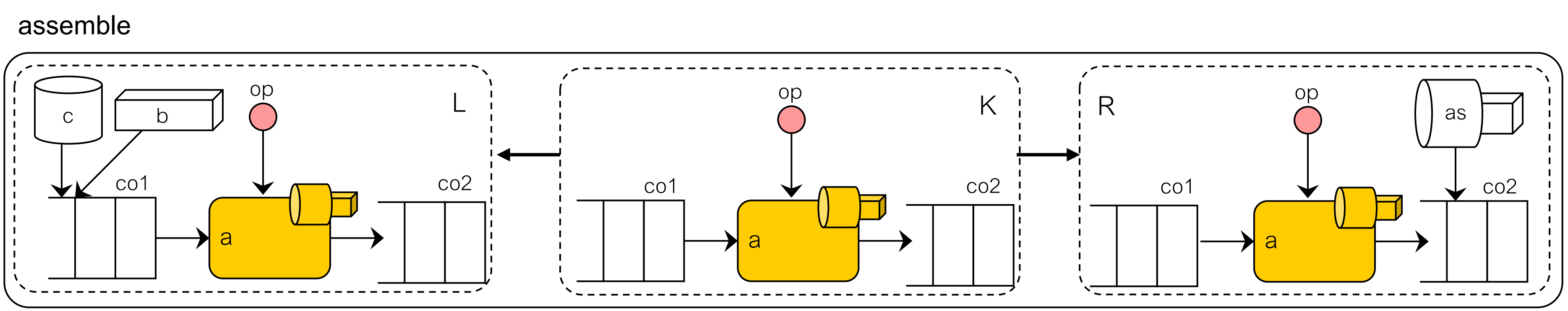


assemble



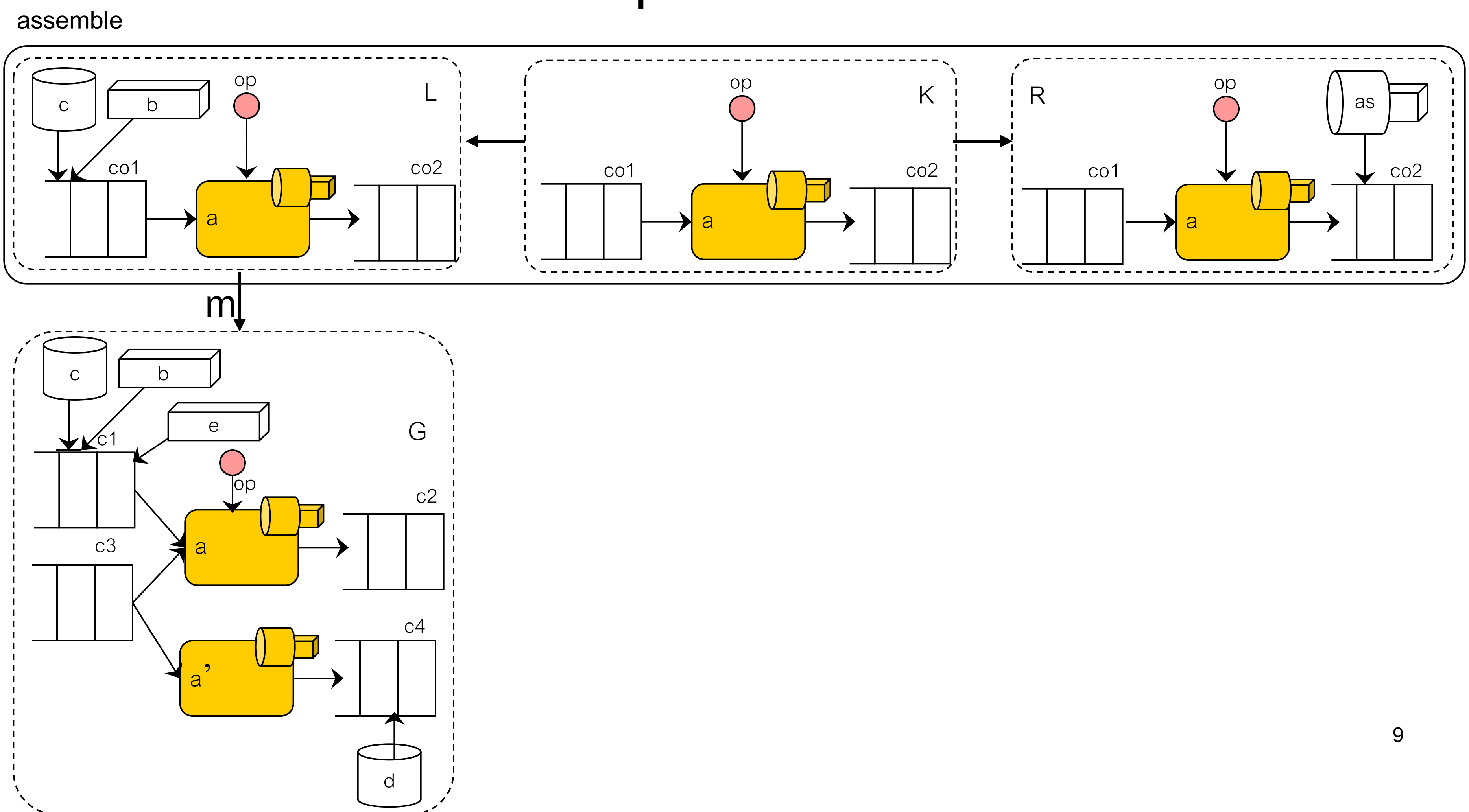
Rules in DPO

- In DPO, rules are divided in 3 graphs:
  - L = LHS
  - R = RHS
  - K = LHS∩RHS
- and two graph morphisms: L ←K→R



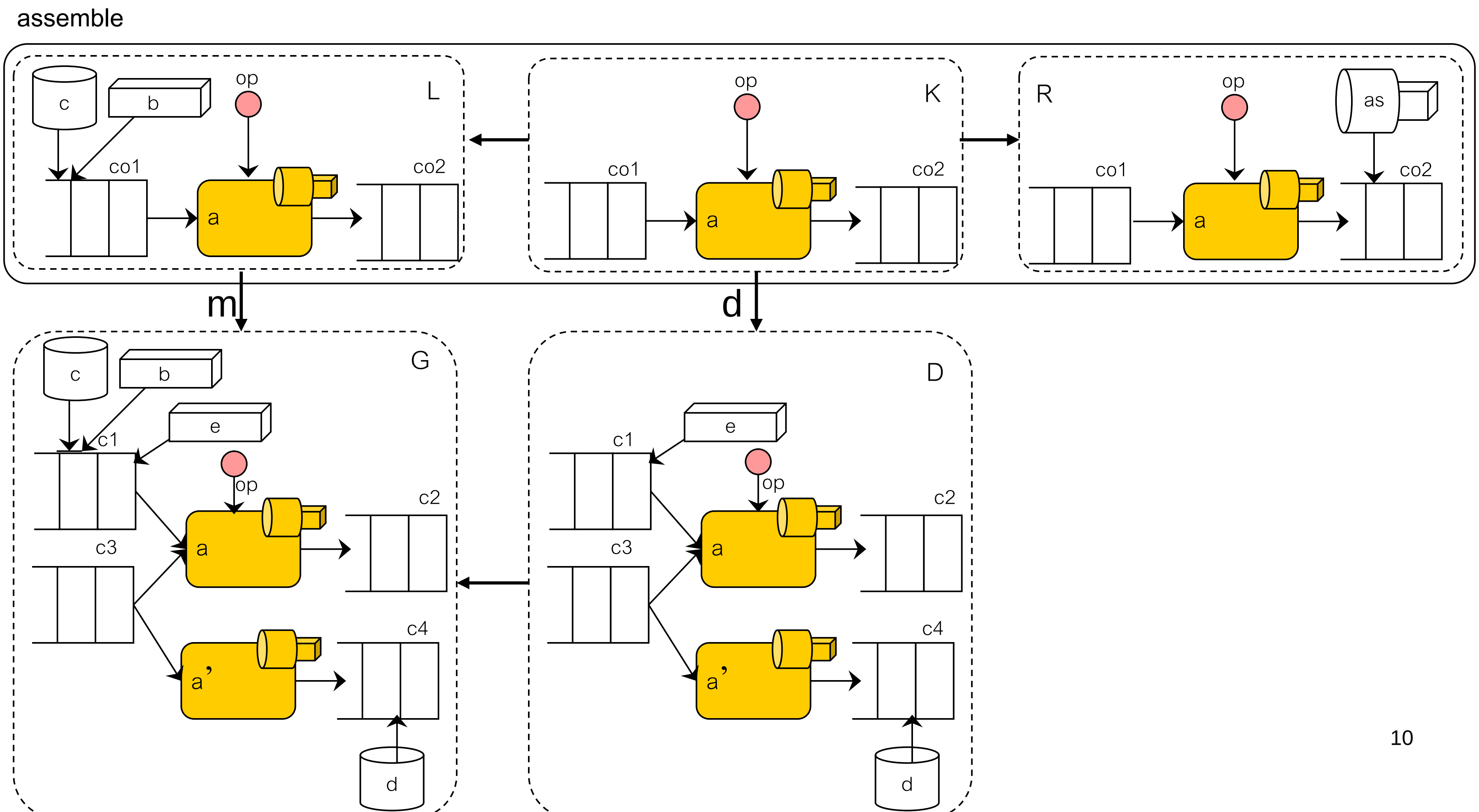
Derivations in DPO

- The match is a morphism m:L→G.



Derivations in DPO

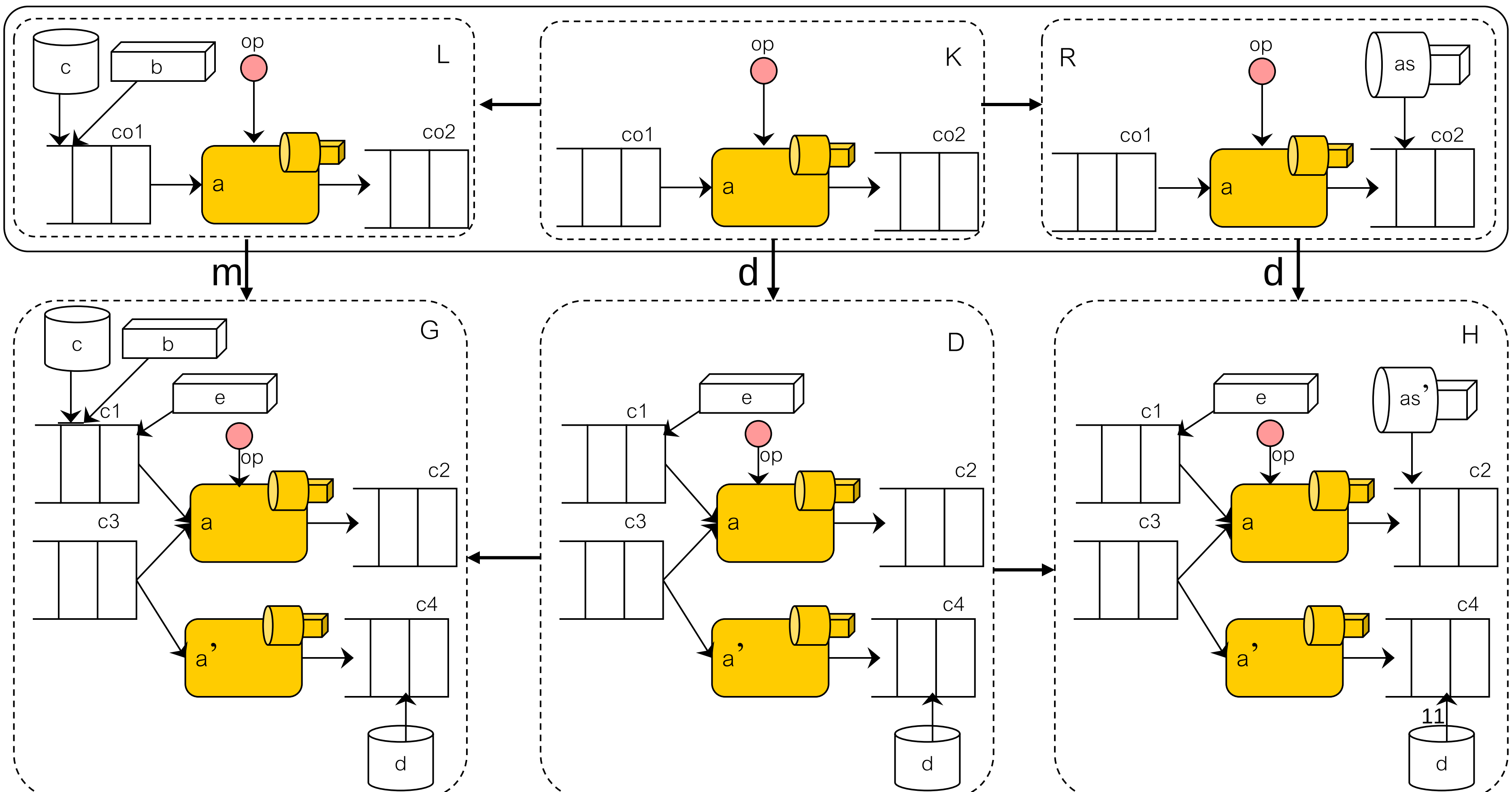
- First step: deletion



Derivations in DPO

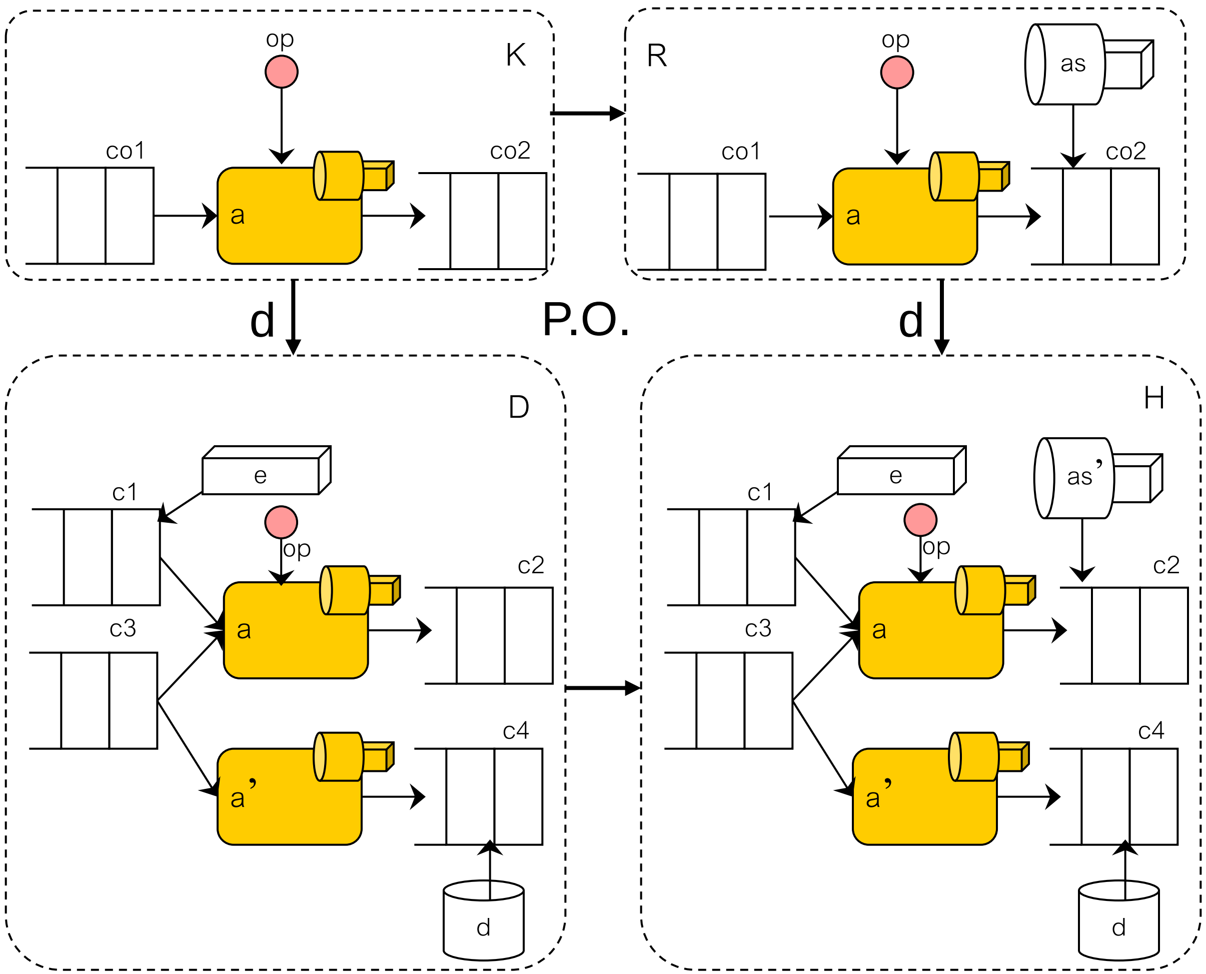
- Second step: creation

assemble



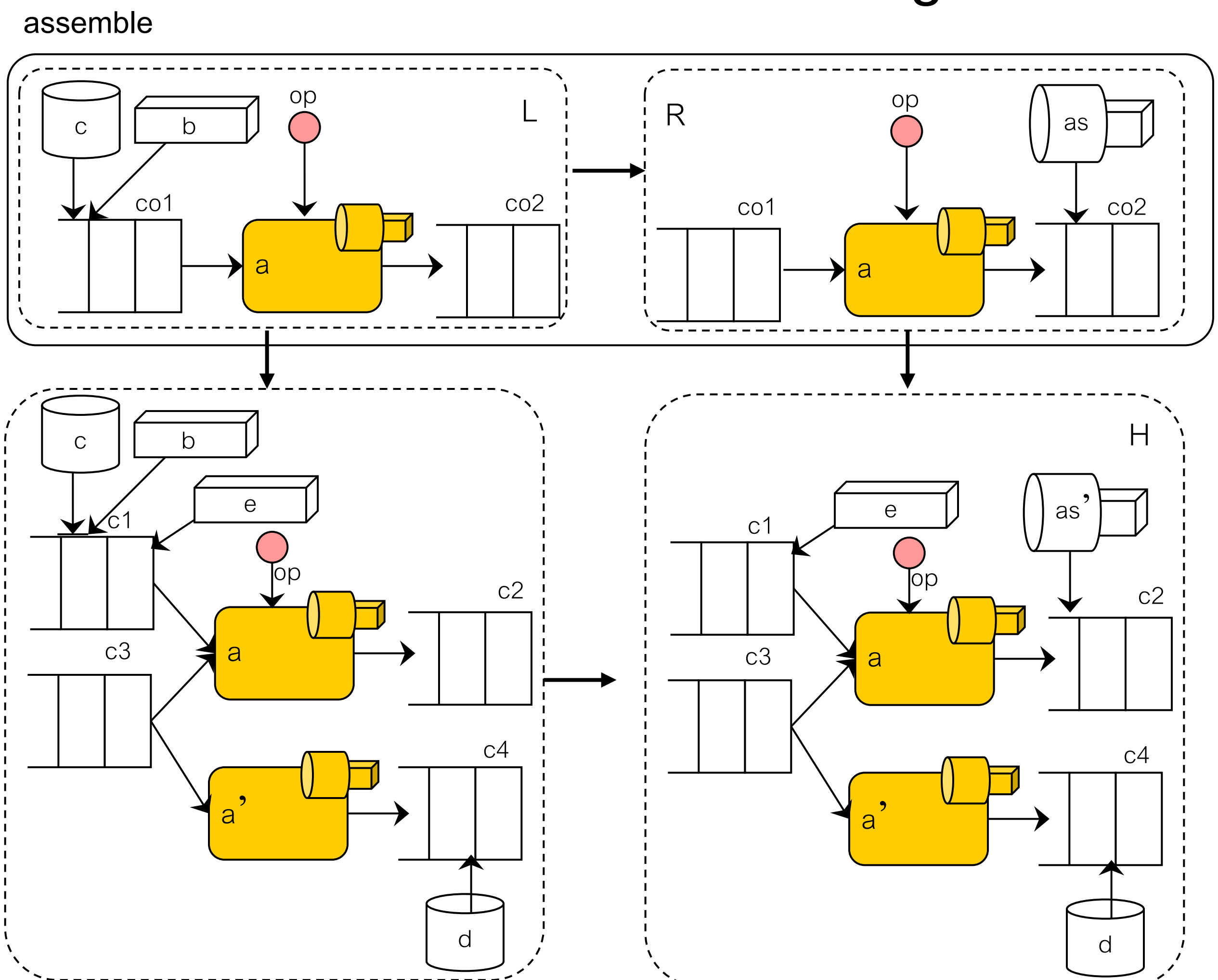
Derivations in DPO

- Steps based on glueing construction: Pushout.



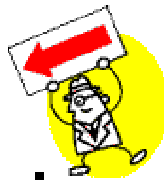
SPO

- SPO uses partial functions.
- Matches should be total functions though.



Index

- Formalizations
- **Analysis**



  - **Sequential and Parallel Independence.**
  - **Church-Rosser and Parallelism Theorems.**
  - **Concurrency Theorem.**
  - **Confluence, local confluence and critical pairs.**
  - **Functional behaviour and termination.**
  - **Other results.**
  - **Other analysis techniques: Model checking.**
- Graph Constraints and Application Conditions.
- Triple Graph Grammars.
- Conclusions.

Parallelism

- Two ways of modelling parallelism:
  - **Explicit**: Two processors that can apply two or more rules at the same time. Parallel independence.
  - **Interleavings**: Parallelism is modelled through all possible interleaving sequences. Sequential independence.

Church-Rosser Theorem

- **Parallel independence**: Two alternative derivations are independent if one does not exclude the other.

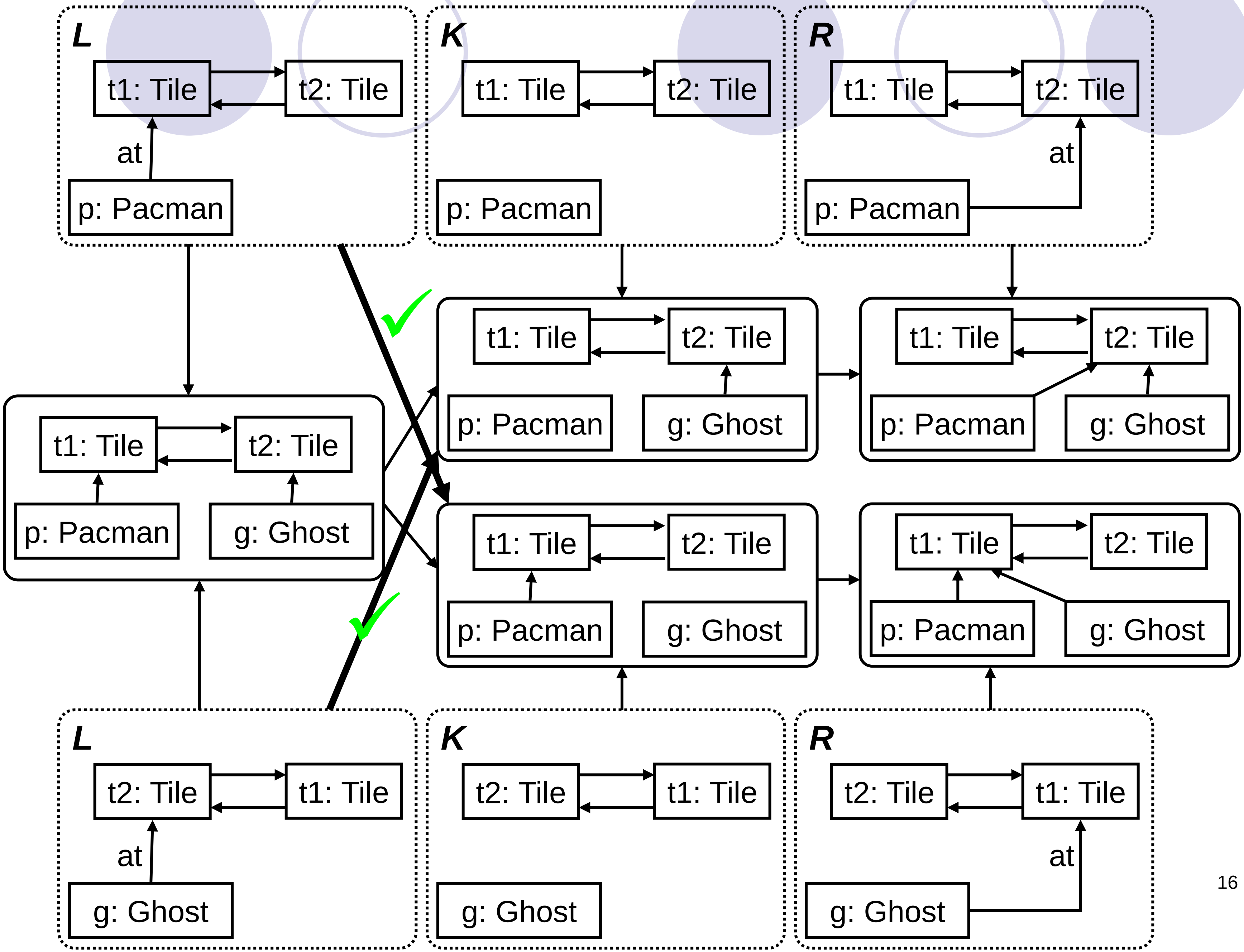
$$
\begin{tikzcd}
 & H \arrow[dr, Rightarrow, "r_2"] & \\
G \arrow[ur, Rightarrow, "r_1"] \arrow[dr, Rightarrow, "r_2"'] & & X \\
 & H' \arrow[ur, Rightarrow, "r_1"'] &
\end{tikzcd}
$$

- **Sequential independence**: Two consecutive derivations are independent if they do not have causal dependencies.

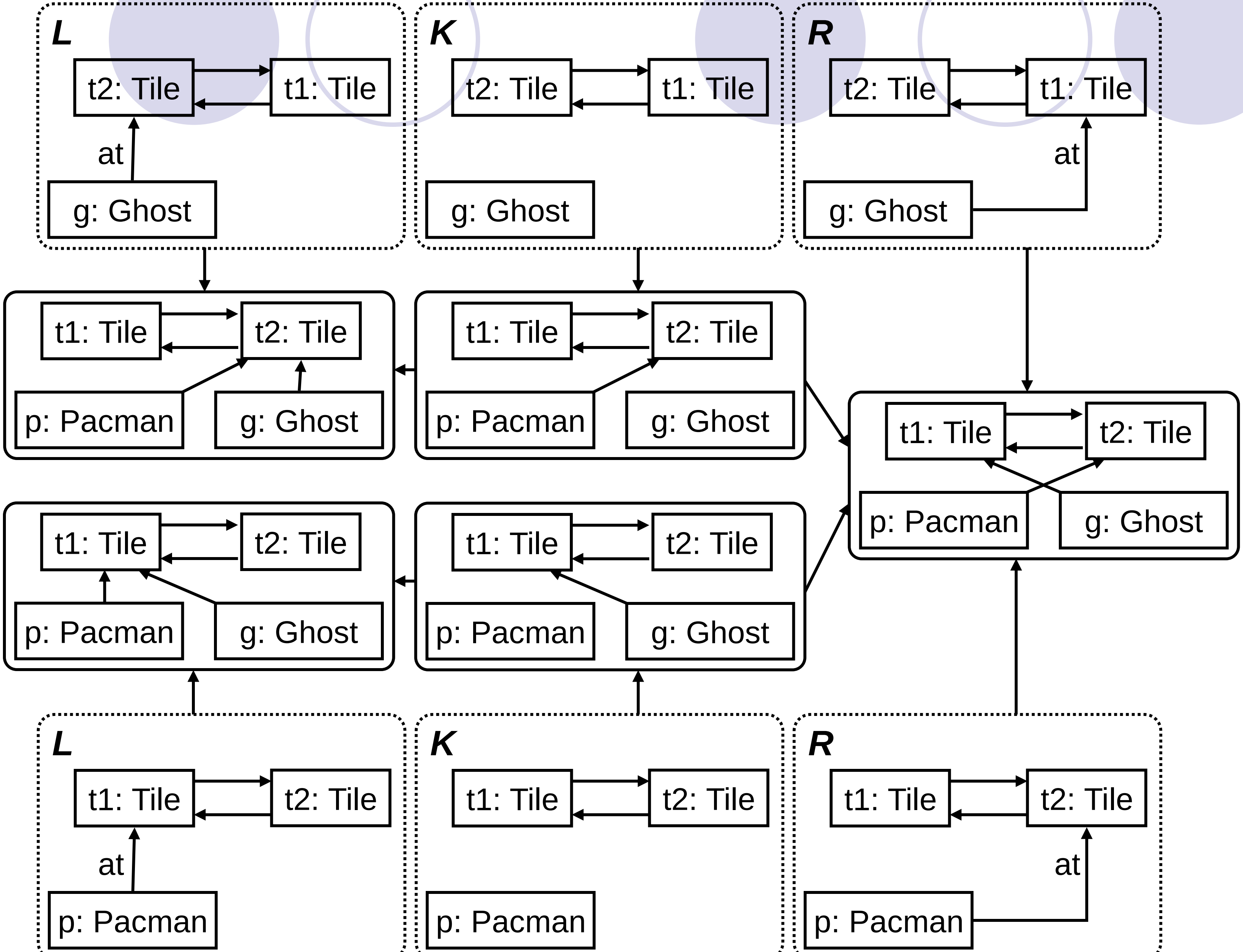
$$G \xRightarrow{r_1} H \xRightarrow{r_2} X$$

$$G \xRightarrow{r_2} H' \xRightarrow{r_1} X$$

Parallel Independence: characterization



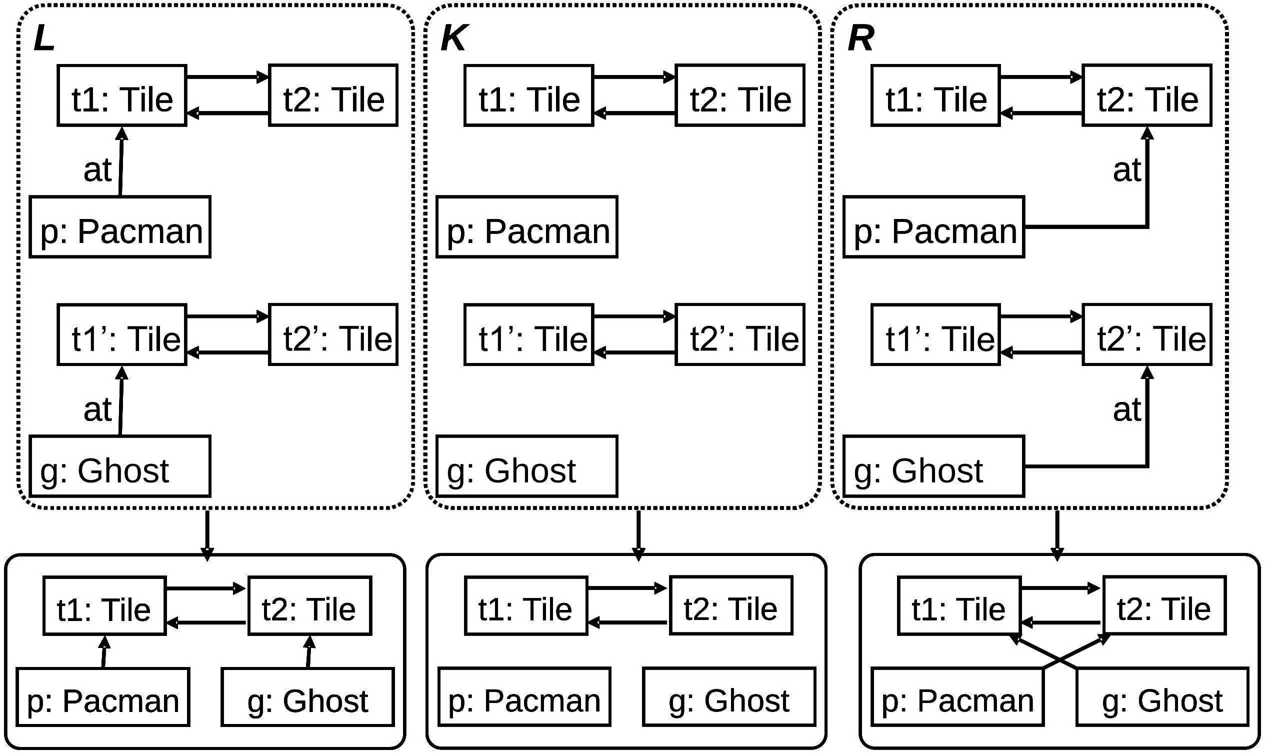
Parallel Independence (2)



Parallel Independence

Parallelism Theorem

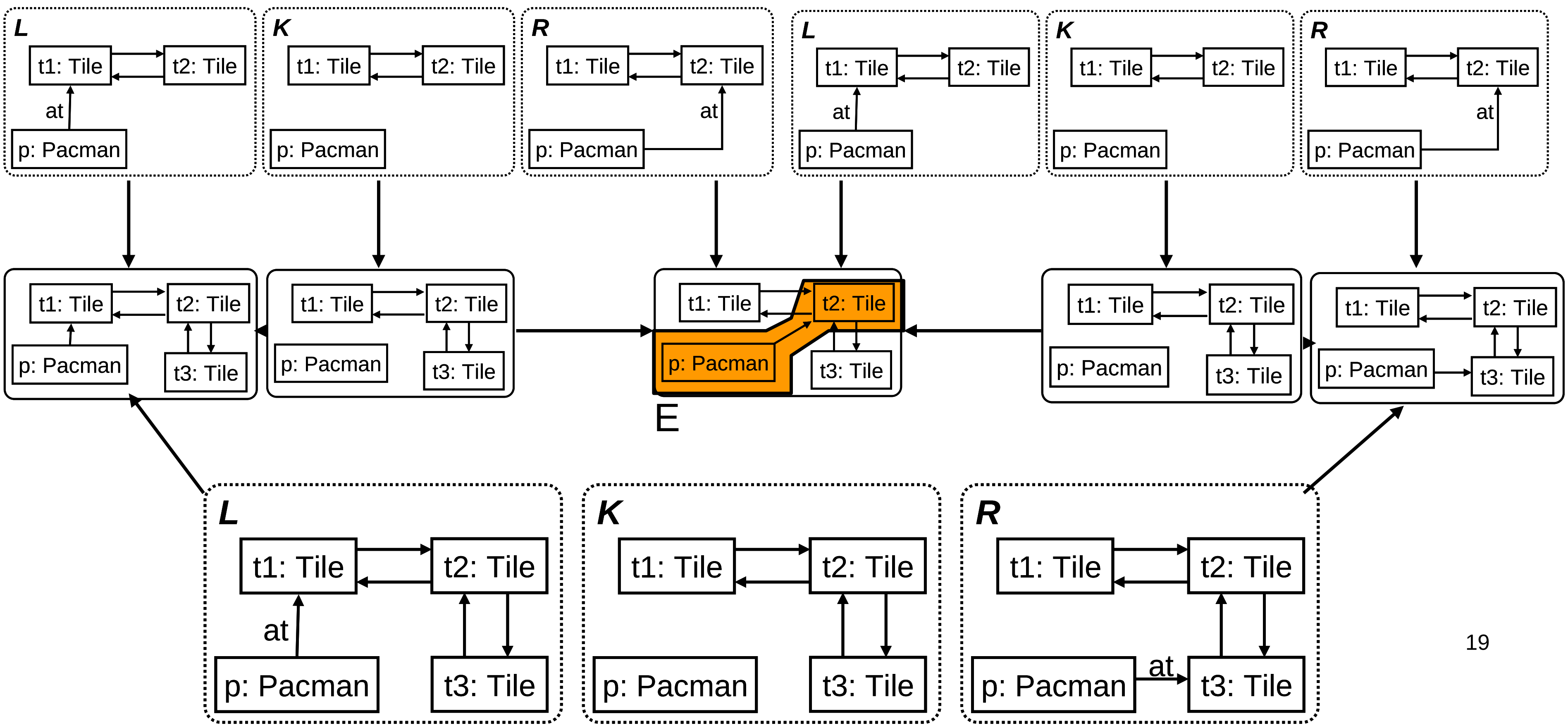
- **Synthesis:** if G⇒H⇒H' is sequential independent, we can go in one step using the ***parallel rule***.



- Trick: use of non-injective matches.
- **Analysis:** the converse decomposition of a parallel rule.

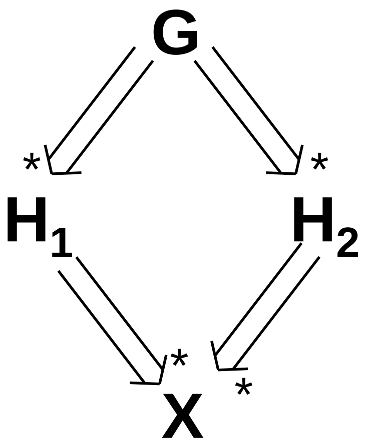
Concurrency Theorem

- Similar to the parallelism theorem, but here rules may have a dependency.
- Concurrent rule built through a dependency graph *E*.



Confluence

- **Global determinism**. Unique result of a graph transformation system.
- A transformation system is *confluent* if for each pair of derivations $G \Rightarrow^* H_1$ and $G \Rightarrow^* H_2$, there is a graph X and derivations $H_1 \Rightarrow^* X$, $H_2 \Rightarrow^* X$.



Local Confluence

- A transformation system is *locally confluent* if for each pair of **direct** derivations $G \Rightarrow H_1$ and $G \Rightarrow H_2$, there is a graph X and derivations $H_1 \Rightarrow^* X$, $H_2 \Rightarrow^* X$.

- A graph transformation system that is terminating and locally confluent is confluent.

- A transformation system is terminating if there is no infinite derivation.

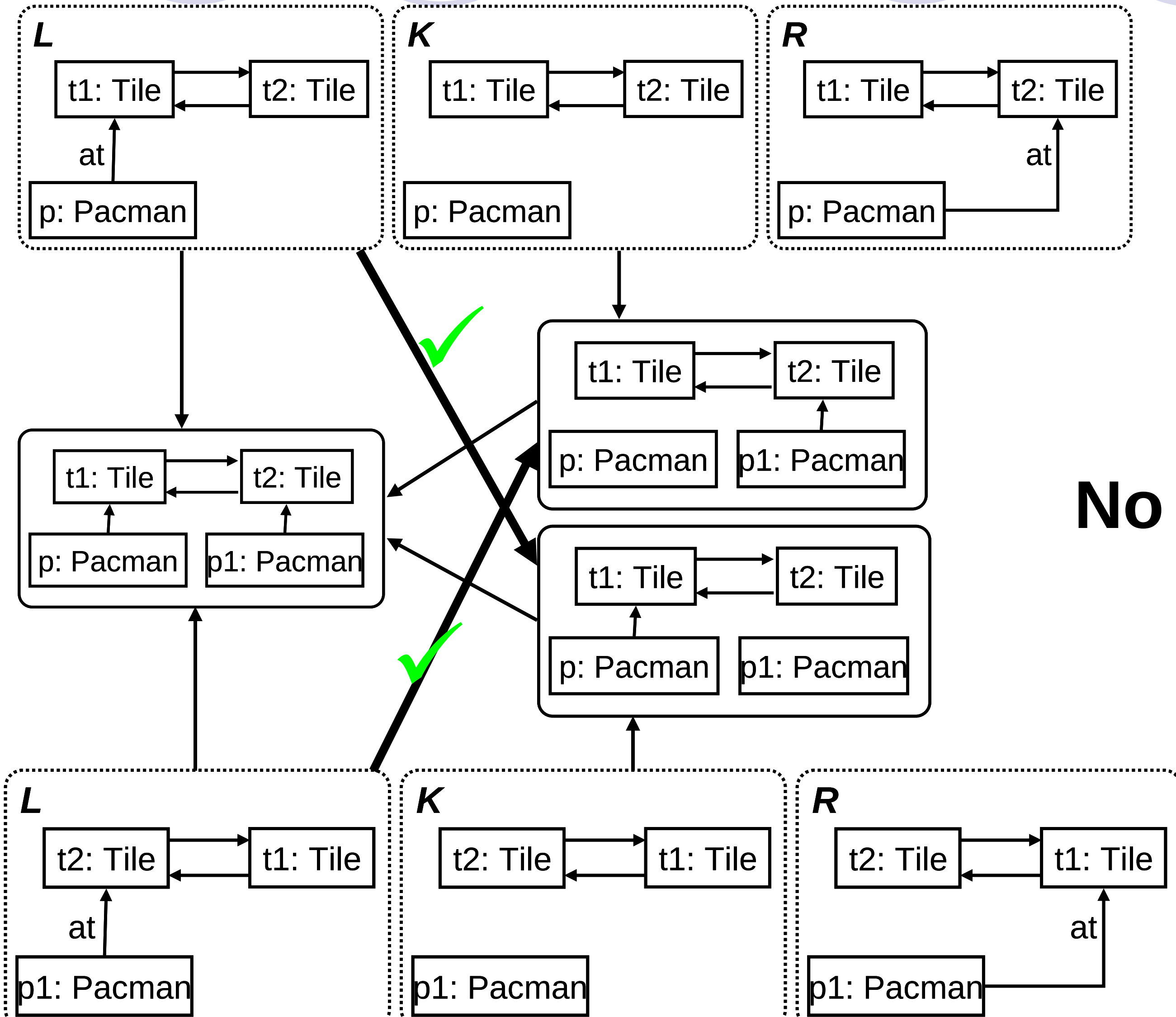
Critical Pairs

- Used to study local confluence.
  1. Take each pair of rules r_1 and r_2 and calculate all "small graphs" $\{S_i\}$.
     1. Both r_1 and r_2 can be applied in S_i.
     2. Every element in S_i is matched by some element of either L_1 or L_2 (***jointly surjective***).

  2. Check if the pairs of derivations from each S_i are parallel independent.

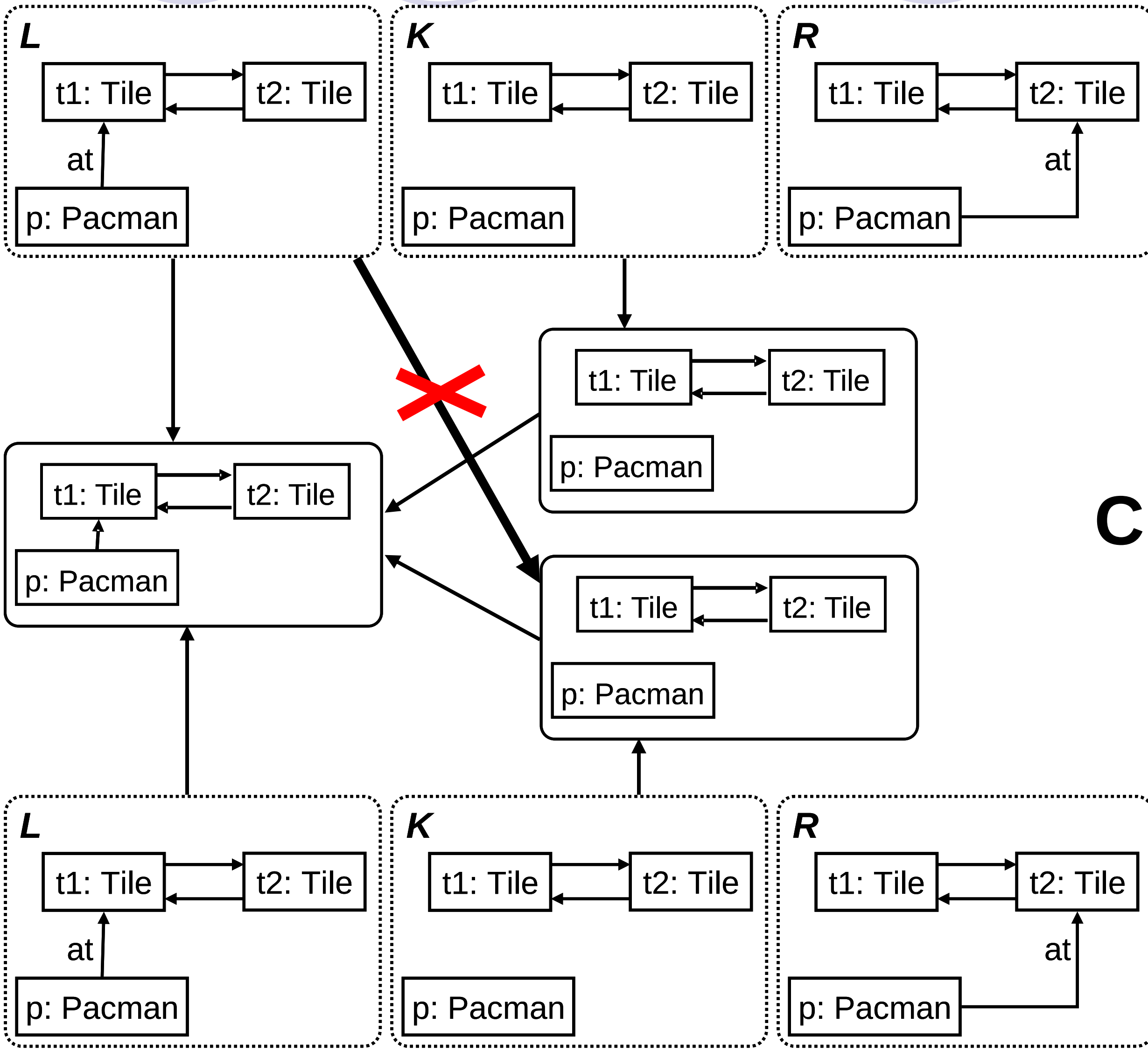
  3. If all derivations, from all $\{S_i\}$, for all pairs of rules are independent, the system is locally confluent.

Critical Pairs



No critical pair

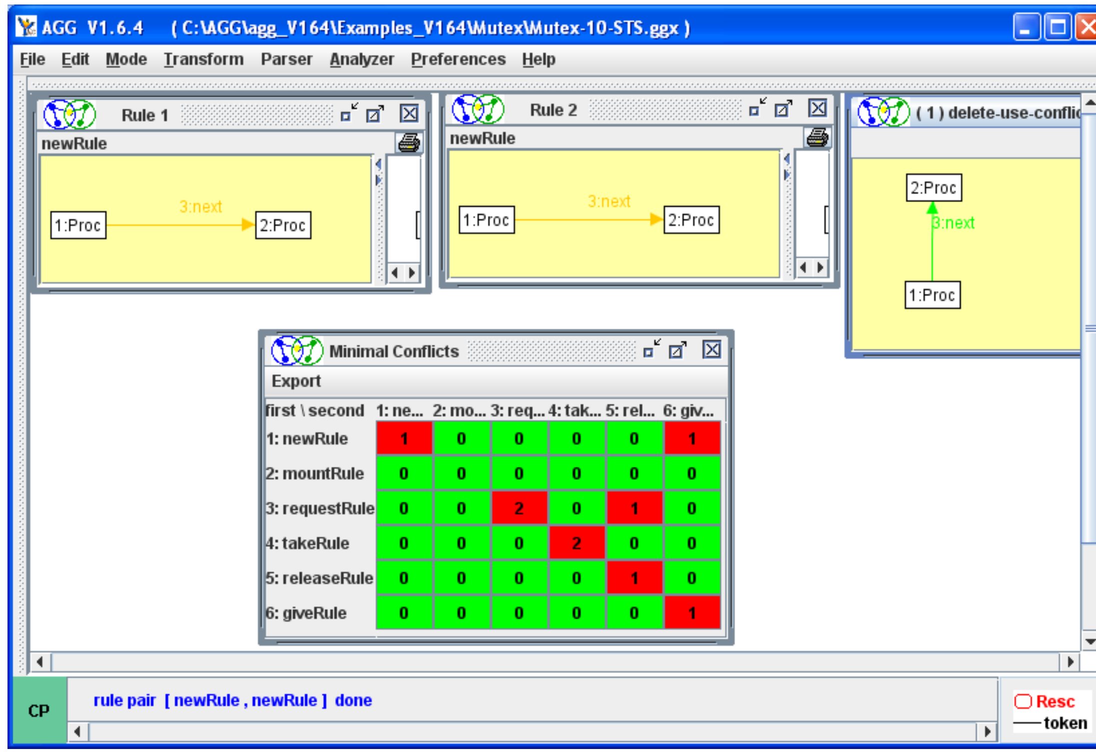
Critical Pairs



Critical pair

Critical Pairs

- Conflicts:
  - Delete-use.
  - Produce-forbid (with NACs).



- The AGG tool supports this analysis.

Functional Behaviour

- Termination+Local confluence.
- Termination of a transformation system is undecidable.
- There are sufficient criteria to ensure the termination of a transformation system with control layers.
  - Deletion layers.
  - Non-deletion layers: NACs ensuring finite application.

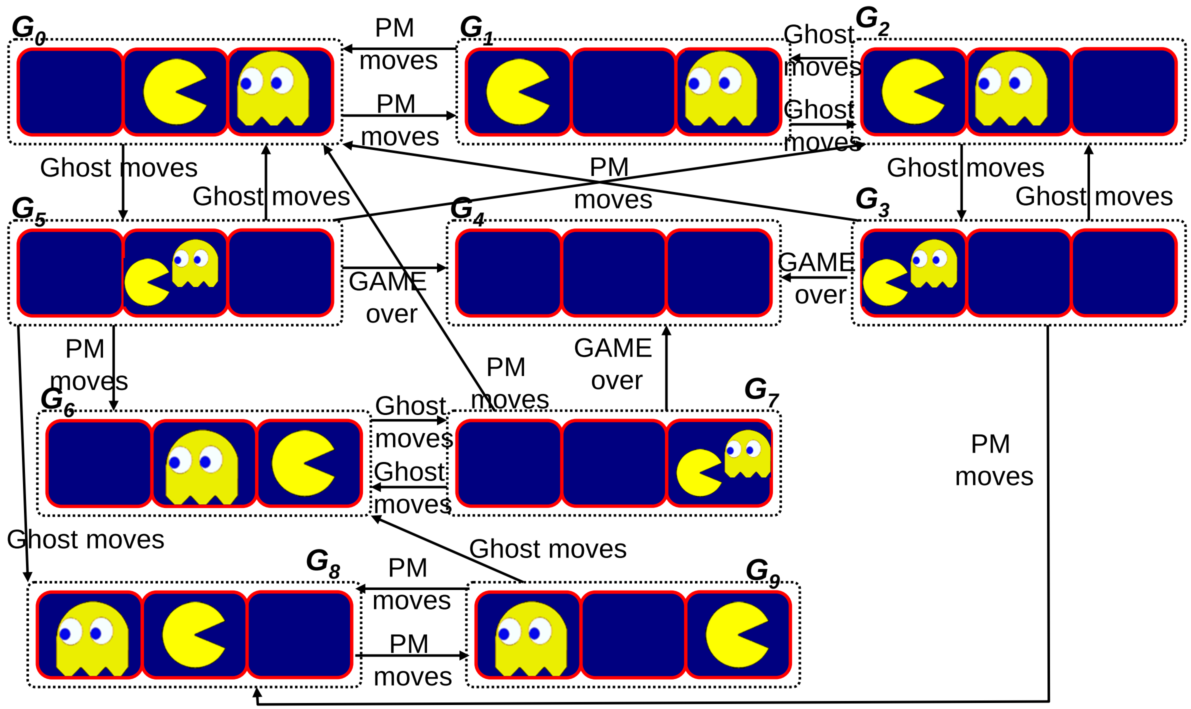
H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, S. Varró-Gyapay: Termination Criteria for Model Transformation. FASE 2005: 49-63

Other Results

- **Embedding and Extension Theorem.**
  - Conditions under which a derivation starting in a graph G_0 can be embedded in a bigger graph G_0'.

Model Checking

- The execution of a graph transformation system on an initial graph spawns a computation tree:
  - The nodes are the reachable states.
  - The transitions are the rules that have been executed.



Model Checking

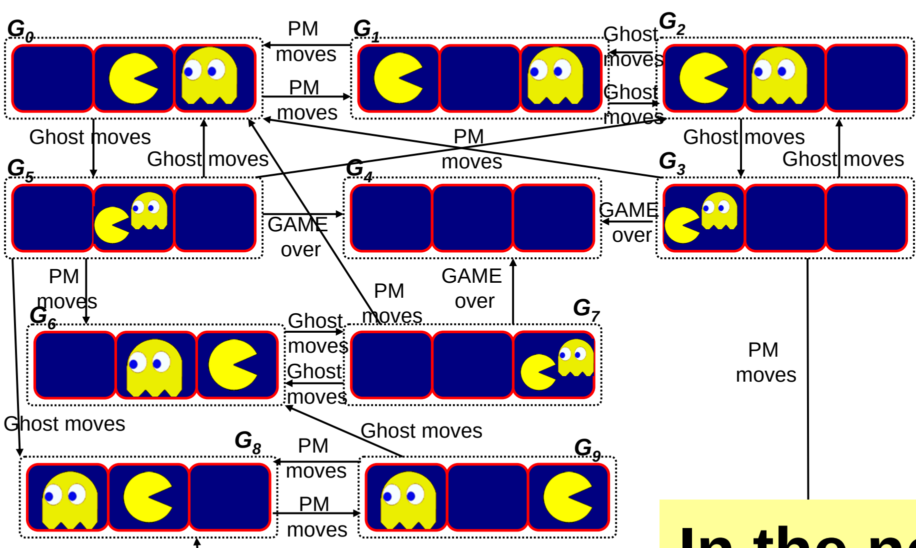
- We can use logics to formulate properties of execution paths (or sub-trees):
  - Linear Temporal Logic.
  - Computational Tree Logic.
- Efficient algorithms check if such property holds or not in all possible executions (i.e., in the execution tree).
- This is called ***model-checking***.
- The GROOVE tool supports this kind of analysis.

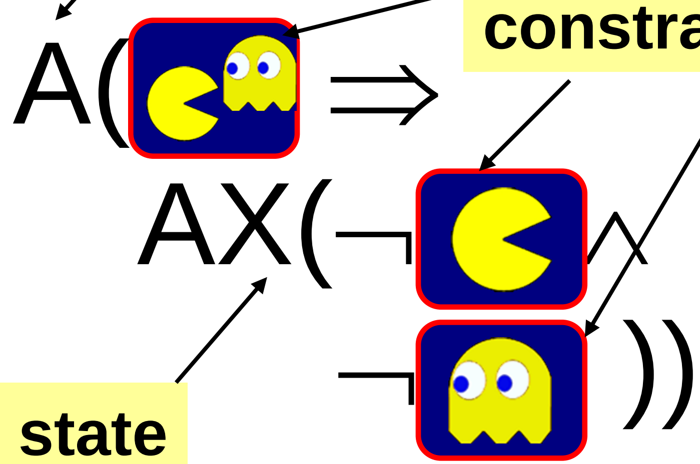
Model Checking

- “In all reachable paths if the pacman and the ghost are in the same tile, then in the next step, there is no pacman or ghost.”
- Expresses a correctness proper
  - we implemented correctly the “gam

for all execution paths

Graph constraints





In the next state

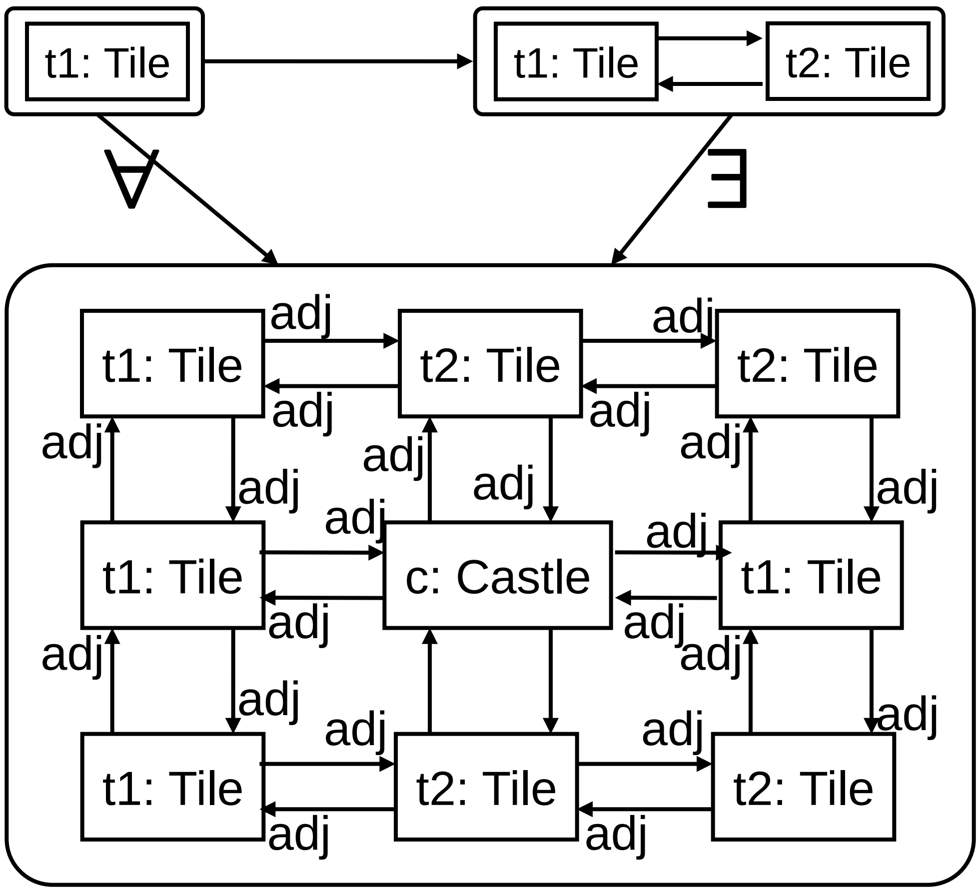
Index

- Formalizations
- Analysis
- **Graph Constraints and Application Conditions.**

- Triple Graph Grammars.
- Conclusions.

Graph Constraints

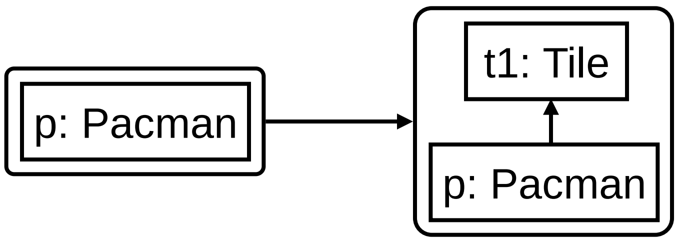
- The GT way of expressing constraints over graphs (i.e., instead of OCL).



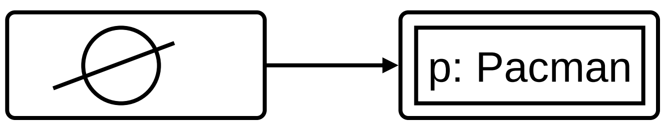
- Each tile should have another adjacent one (i.e., demands a connected model).
- This is called an atomic constraint.
- They can be combined using boolean connectives (and, or, not).

Graph Constraints

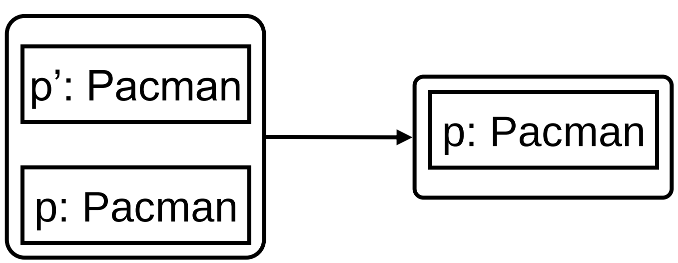
- What do these constraints mean?



Each pacman is on a tile



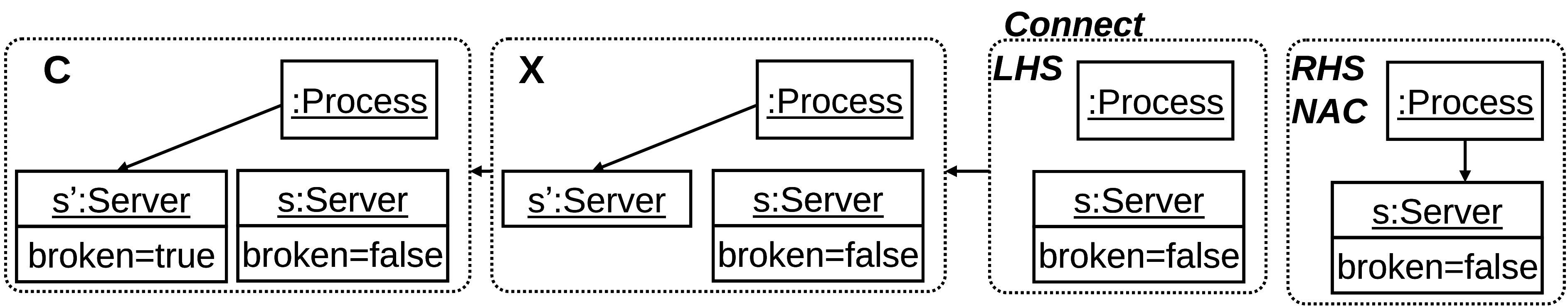
There is at least one pacman



There is at most one pacman

Application conditions.

- We have seen Negative Application Conditions (NACs).
- They are a special case of (left) Application Conditions.



"If the process is connected to another server (X) then such server should be broken (C)."

- A premise (X) and a set of consequences, one of them should be found if X is found.
- If the set of consequences is empty, we have a NAC.

Right Application Conditions

- They are assigned to the RHS.
- These are evaluated once the rule is applied.
- If they are not satisfied, the rule application is “undone”.
- GT has techniques to advance right to left application conditions.
  - No need to undo the rule, we know in advance!

From constraints to application conditions

- Graph constraints express some properties of the graphs we manipulate.
- Given a grammar, do their rules break any such constraint?
- What we can do is convert each graph constraint into local application conditions for the rule, so that:
  - if the application condition is satisfied, the rule does not break the graph constraint.
  - if the application condition is not satisfied, the rule cannot be applied (as it would break the constraint).

Index

- Formalizations
- Analysis
- Graph Constraints and Application Conditions.
- **Triple Graph Grammars.** 
- Conclusions.

Triple Graph Grammars

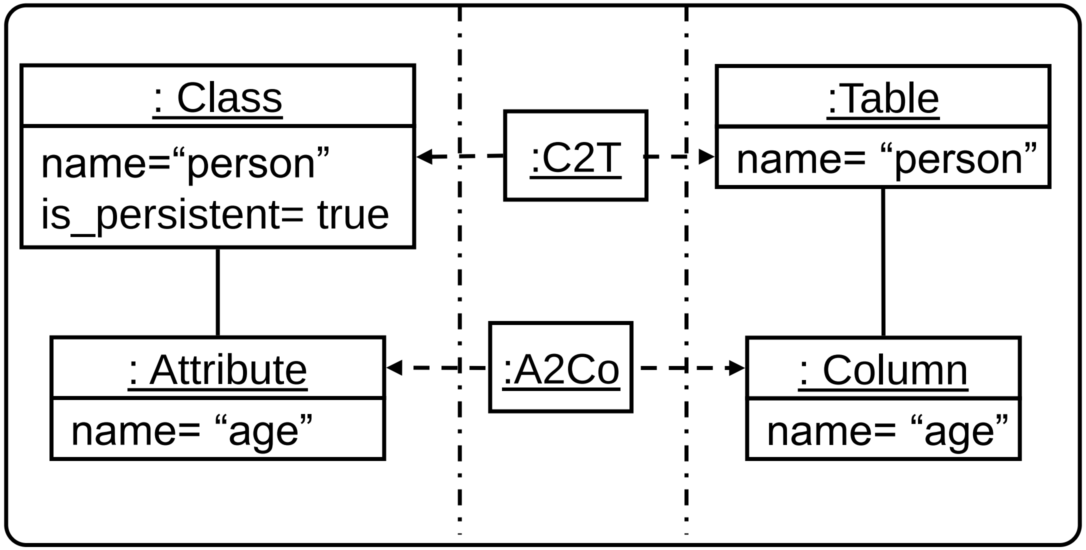
- Useful to describe Model to Model transformations, in a declarative, bidirectional way.

- A **unique** grammar is able to generate operational transformations to go from source to target and target to source.

- Two levels:
  - specification by a declarative grammar and
  - operational (also by rules), to perform some scenario (i.e. forward or backwards transformation).
  - Automatic generation of operational rules.

Triple Graph Grammars

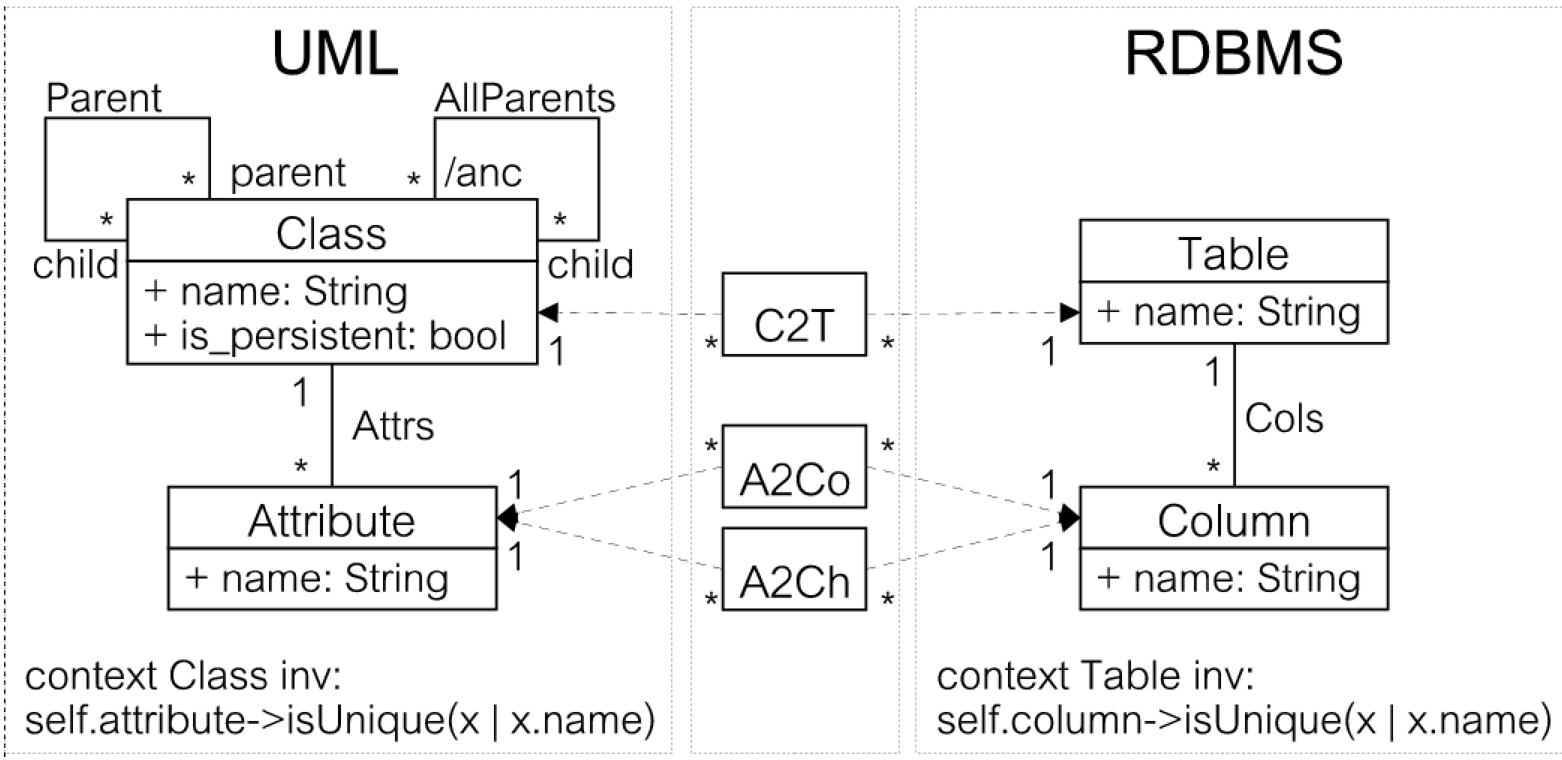
- A TGG describes a language of triple graphs.
- **Triple graph**. Two models related through a mapping model.



Triple Graph Grammars

Example: UML-RDBMS

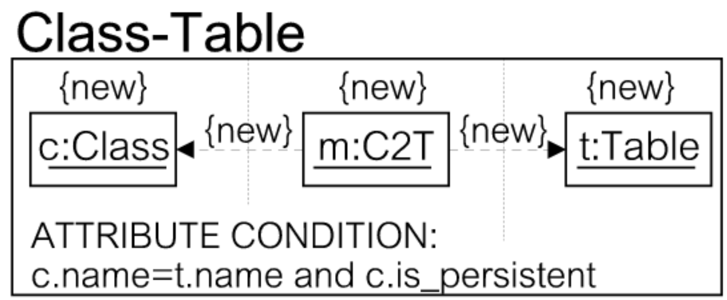
Meta-Model Triple

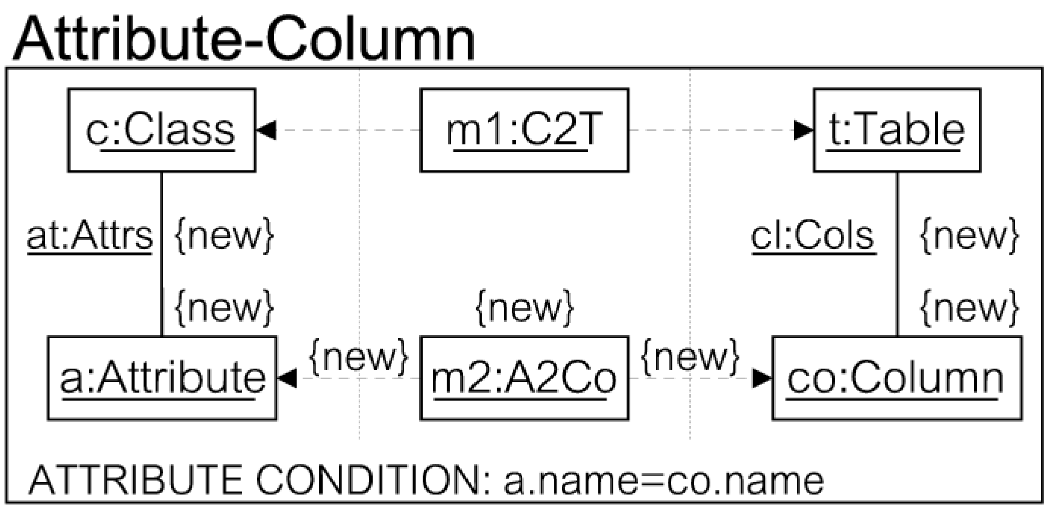


Triple Graph Grammars

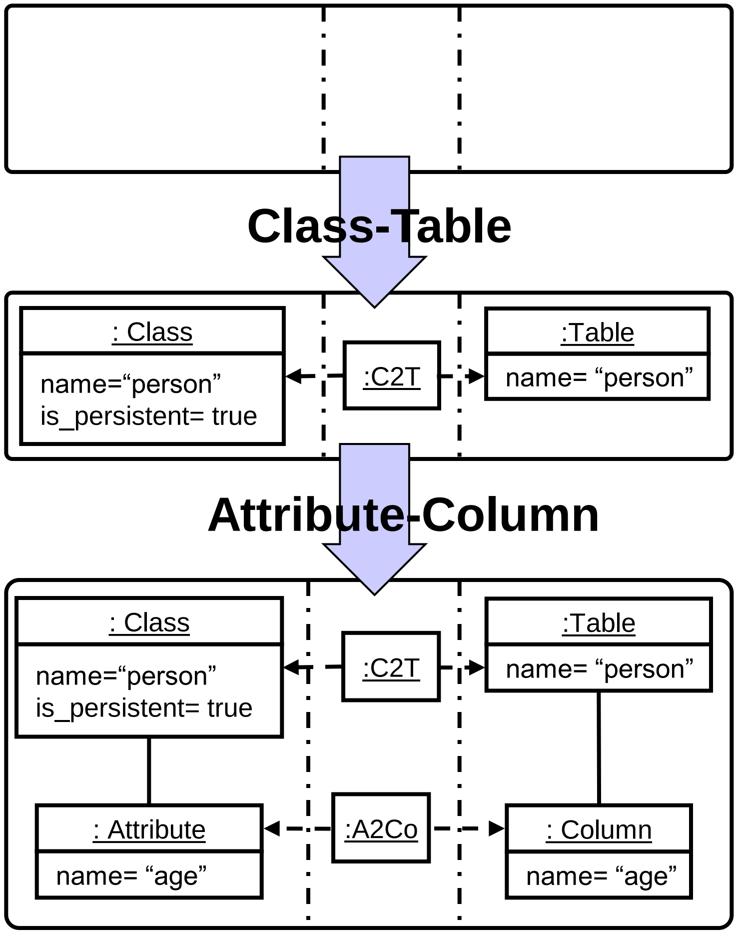
Example: UML-RDBMS

Some declarative rules:





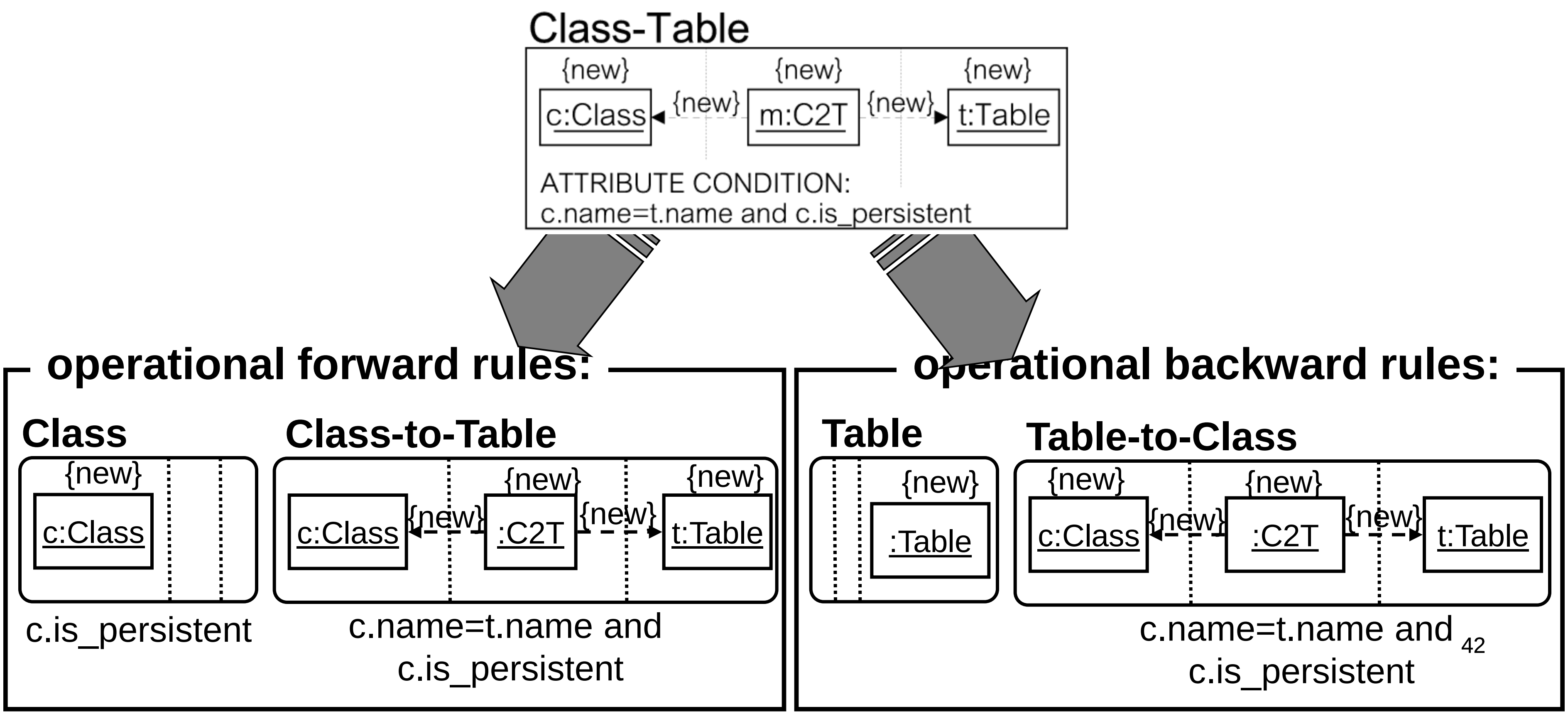
Derivation Example:



Triple Graph Grammars

Operational Rules

- Choose the "*job to be done*" with the specification, e.g. source-to-target or target-to-source transformations.
- Generation of operational rules:



Index

- Formalizations
- Analysis
- Graph Constraints and Application Conditions.
- Triple Graph Grammars.
- **Conclusions.** 

Conclusions

- The formalizations of GT allows analysis of the rules:
  - independence (parallelism).
  - confluence (unique result).
  - guides for termination.
- Graph constraints allow formulating conditions to be satisfied by graphs.
  - the grammars can be added application conditions so that the constraints are preserved.
- Triple Graph Grammars to perform M2M transformations.
  - A unique specification is useful to perform forward and backwards transformations.

Bibliography.

- Handbook of Graph Grammars and Computing by Graph Transformation. 3 Vols. 1997. World Scientific.

- Ehrig, H., Ehrig, K., Prange, U., Taentzer, G. 2006. “Fundamentals of Algebraic Graph Transformation”.Springer.