On the use of graph transformations for **model refactoring**

Tom Mens

Software Engineering Lab
University of Mons-Hainaut
http://w3.umh.ac.be/genlog

Tutorial outline

- Introduction
  - What is model-driven engineering, model transformation, model refactoring?
  - Where does graph transformation fit in?
- Graph transformation theory
- Graph transformation experiments
  - In *Fujaba*: model refactoring plug-in
  - In *AGG*: critical pair analysis
  - With pencil and paper: behaviour preservation
- Conclusion

Introduction

Model-driven engineering

- Goal: Raise the level of software development from *source code* to *models*
  - models = software artifacts at higher level of abstraction
  - e.g. UML diagrams = design models

- Principle: "Everything is a model"
  - Uniform approach to all kinds of software artifacts
    - source code is a kind of model
    - the syntax of a model is described by a *metamodel*

Model Transformation

- Goal: Apply transformation techniques to modify, refine and evolve models

- Classification of model transformations

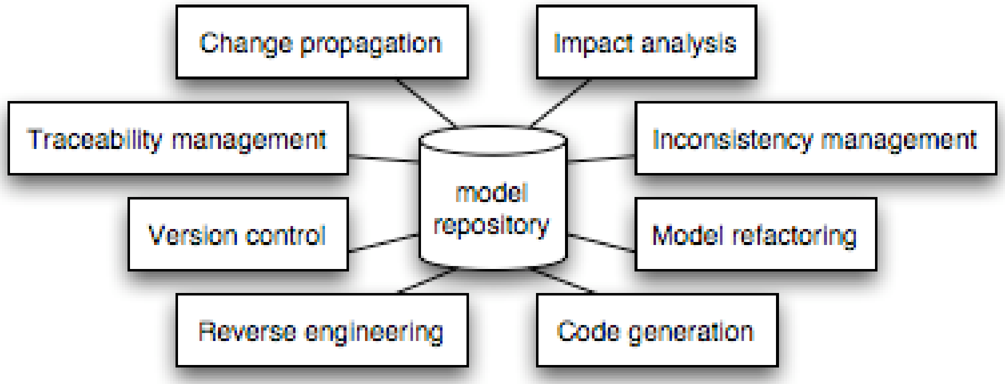
	endogenous	*exogenous*
horizontal	refactoring	language migration, bridging techn. spaces
vertical	formal refinement	code generation

Model Transformation

- Endogenous versus exogenous
  - Endogenous transformations
    - transformations between models expressed within the same metamodel
  - Exogenous transformations
    - transformations between models expressed in different metamodels
- Horizontal versus vertical
  - Horizontal transformation
    - transformation between models residing at the same level of abstraction
  - Vertical transformatation
    - transformation between models at different abstraction levels

Model Evolution

- Goal: Provide support for software evolution at the level of models



- Better tool support needed for all these activities
- Formalisms can be helpful for some of these tools

Model Refactoring

- Goal: special kind of model evolution that improves the structure of the model, while preserving (certain aspects of) its behaviour

- Model refactoring is an example of an endogenous, horizontal model transformation

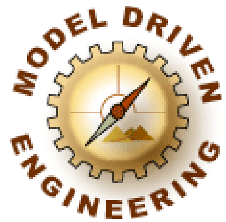
- Model refactoring is based on the idea of program refactoring
  - *"the process of changing a program in such a way that it does not alter the external behavior of the code, yet improves its internal structure"* [Martin Fowler, 1999]

Model Refactoring

- Model refactorings can be applied to different views of a UML model
  - class diagrams
  - sequence diagrams
  - statecharts
  - activity diagrams

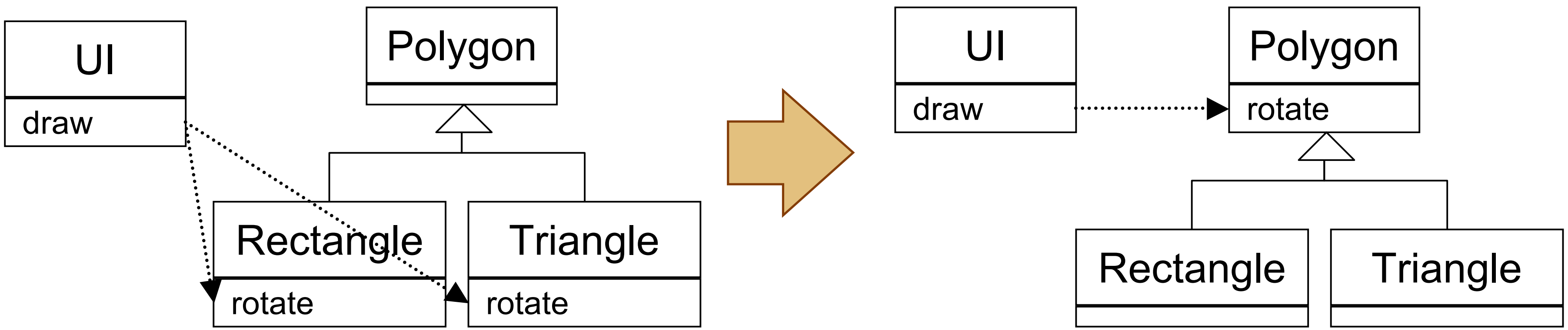
- Some model refactorings have been proposed by Boger et al.

  M. Boger, T. Sturm, P. Fragemann. *Refactoring Browser for UML.* Proc. 3rd Int'l Conf. on eXtreme Programming, pp. 77-81, 2002



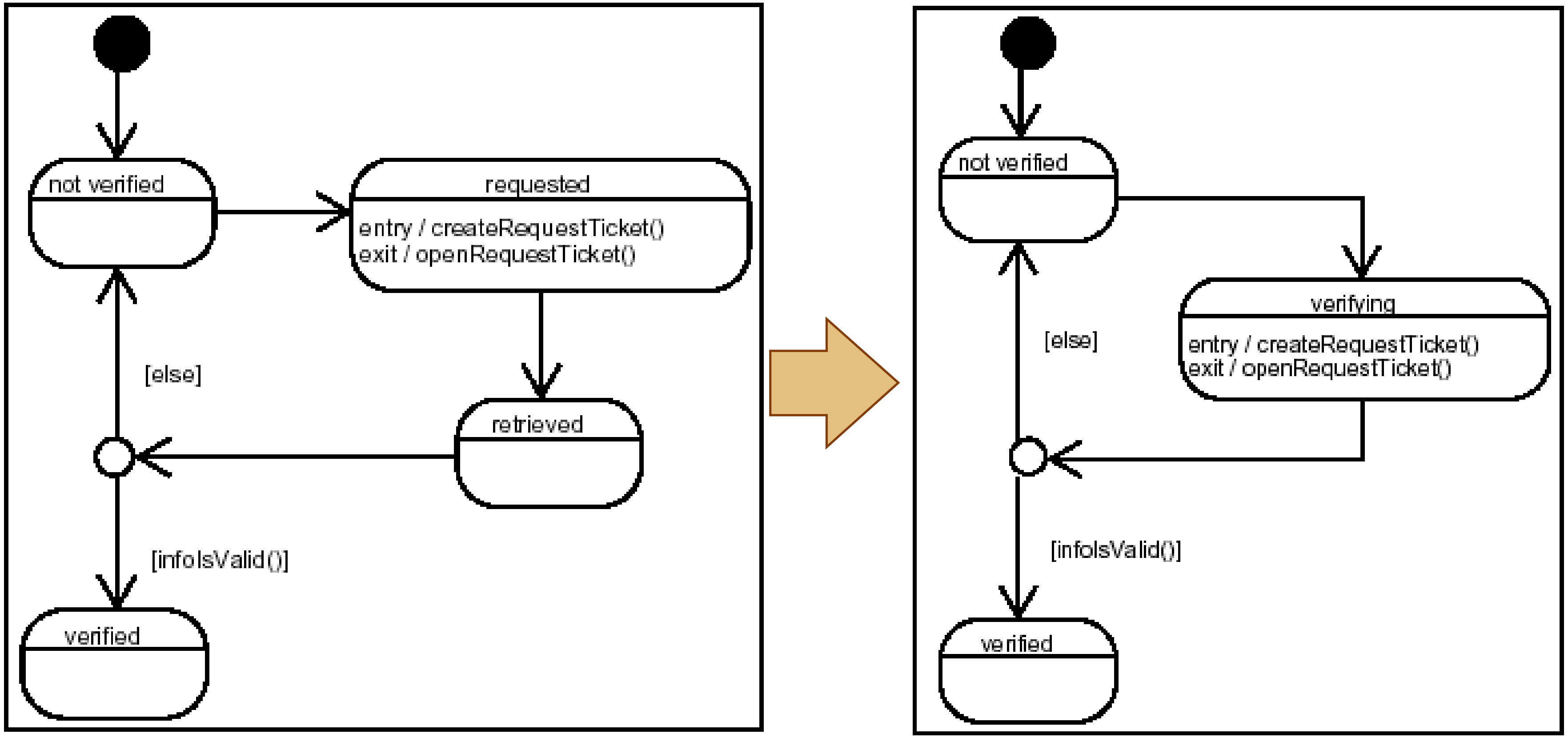
Model Refactoring

- Examples of class diagram refactorings
  - Pull Up Method



Model Refactoring

- Examples of statechart refactorings
  - Merge states

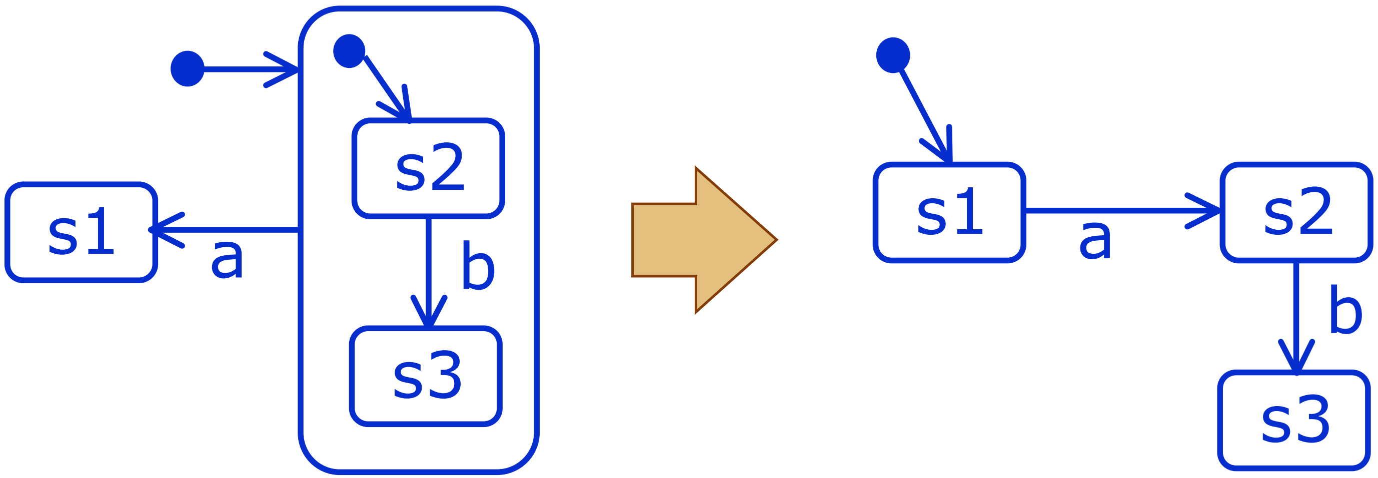


Model Refactoring

- Examples of statechart refactorings
  - Merge states
    - combine a set of states into a single composite state
  - Decompose sequential composite state
    - remove a composite state but keep its internal states
  - Create composite state
    - create a new composite state and move selected states to the interior
  - Sequentialise concurrent state
    - replace a concurrent state by a product automaton
  - Flatten states
    - see next slide

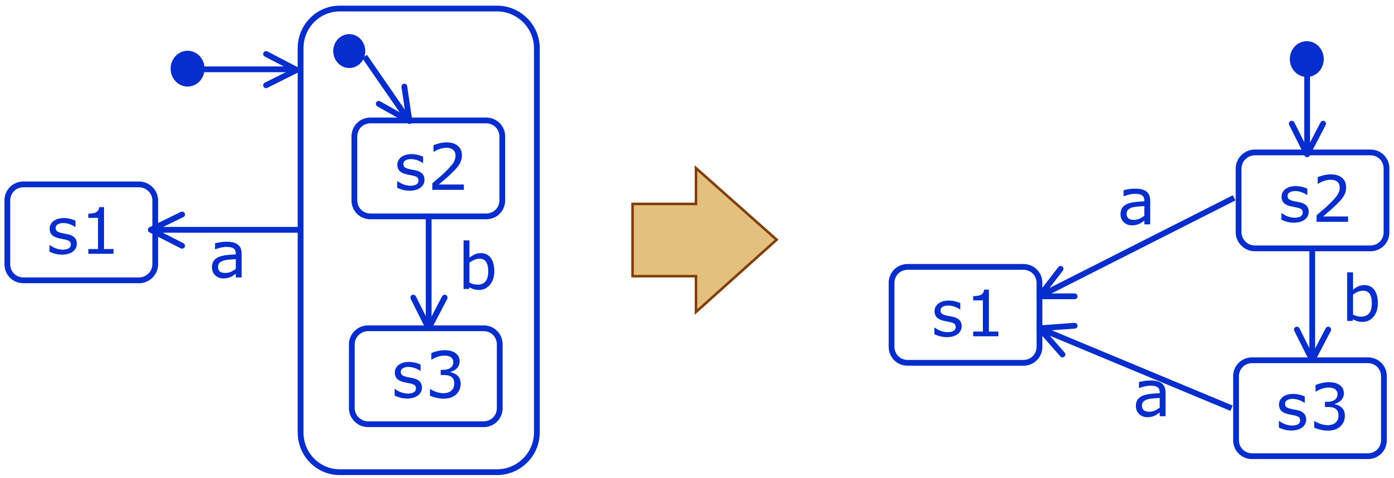
Model Refactoring

- Examples of statechart refactorings
  - Flatten states: Incoming transitions
    - Transition from state *s1* to the boundary of a complex state represents a transition from *s1* to the initial state of the complex state



Model Refactoring

- Examples of statechart refactorings
  - Flatten states: Outgoing transitions
    - Transition from boundary of complex state to state *s1* represents corresponding transitions from all substates to s1

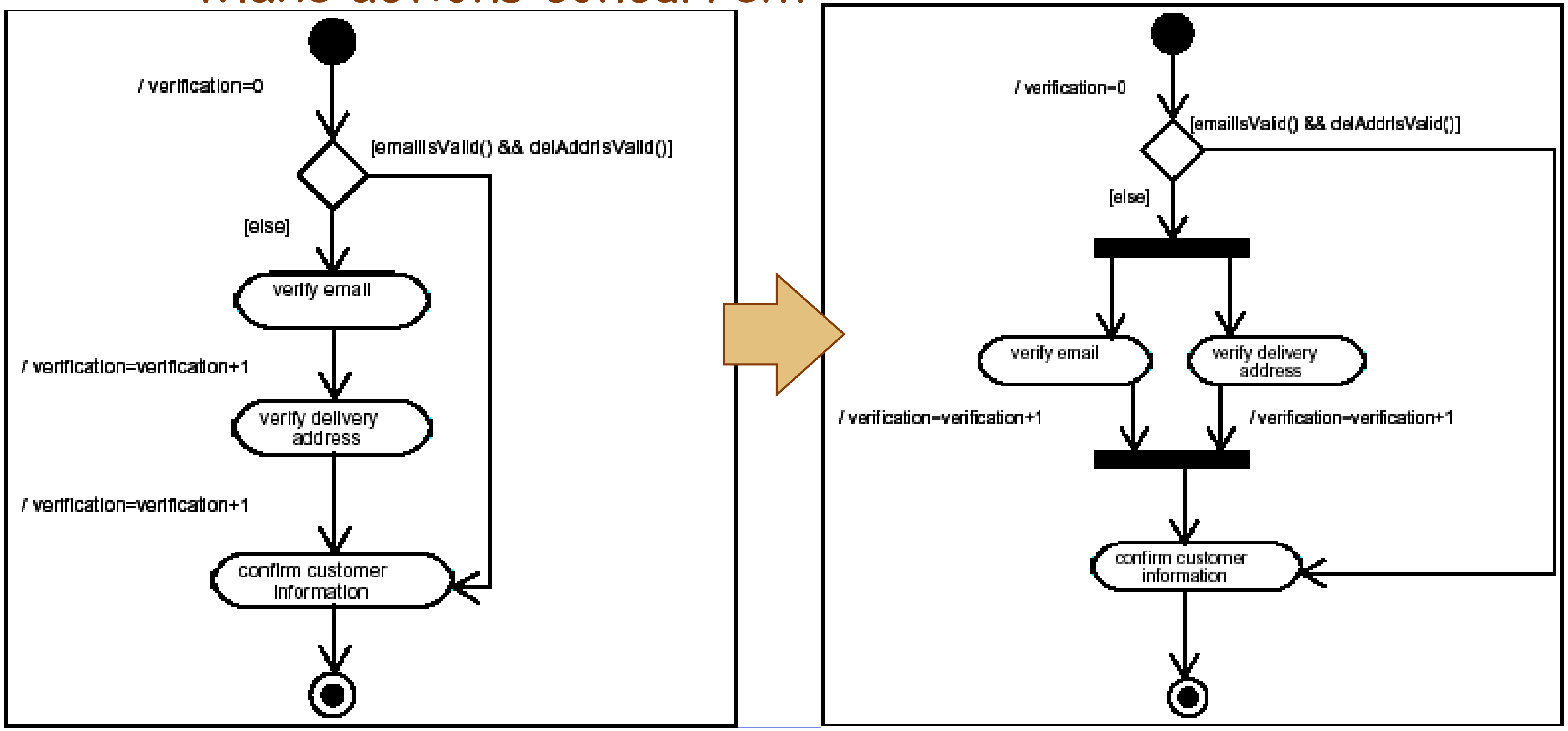


Model Refactoring

- Examples of activity diagram refactorings
  - Make actions concurrent
    - Create a fork and a join pseudostate, and move several sequential groups of actions between them, thus enabling their concurrent execution
  - Sequentialize concurrent actions
    - Removes a pair of fork and join pseudostates, and links the enclosed group of actions to another

Model Refactoring

- Examples of activity diagram refactorings
  - Make actions concurrent



Model Refactoring

- Other related work
  - Sunye *et al.* [UML 2001]
    - statechart refactorings expressed using OCL pre- and postconditions
  - Van Gorp *et al.* [UML 2003]
    - UML extension to support source consistent refactoring
    - integrated as plug-in in Fujaba tool
  - Correa and Werner [UML 2004]
    - UML refactorings in OCL-script, and extension of OCL
  - …

Research Context

- Several ongoing research projects
  - "*A Formal Foundation for Software Refactoring*"
    - Financed by FWO - Flanders, Belgium
    - Duration: January 2003 - December 2006
    - In collaboration with: Serge Demeyer and Dirk Janssens, University of Antwerp
  - "*Research Center on Structural Software Improvement*"
    - Financed by FNRS-FRFC, Belgium
    - Duration: January 2005 - December 2008
    - In collaboration with: Kim Mens, UCL - Roel Wuyts, ULB
    - For more information, see
      http://www.info.ucl.ac.be/ingidocs/people/km/FRFC

Research Context

- Several international research networks
  - ESF Scientific Network RELEASE
    « Research Links to Explore and Advance Software Evolution »
    http://www.esf.org/release/
  - ERCIM Working Group on Software Evolution
    http://w3.umh.ac.be/evol

Tutorial outline

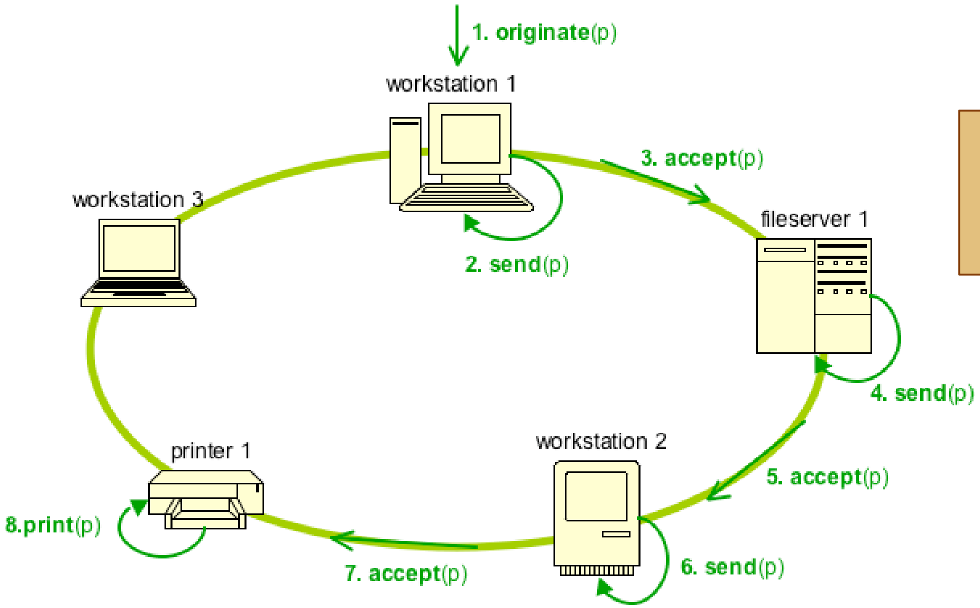
- ✓Introduction
- Graph transformation theory
- Graph transformation experiments
- Conclusion

Graph Transformation Theory

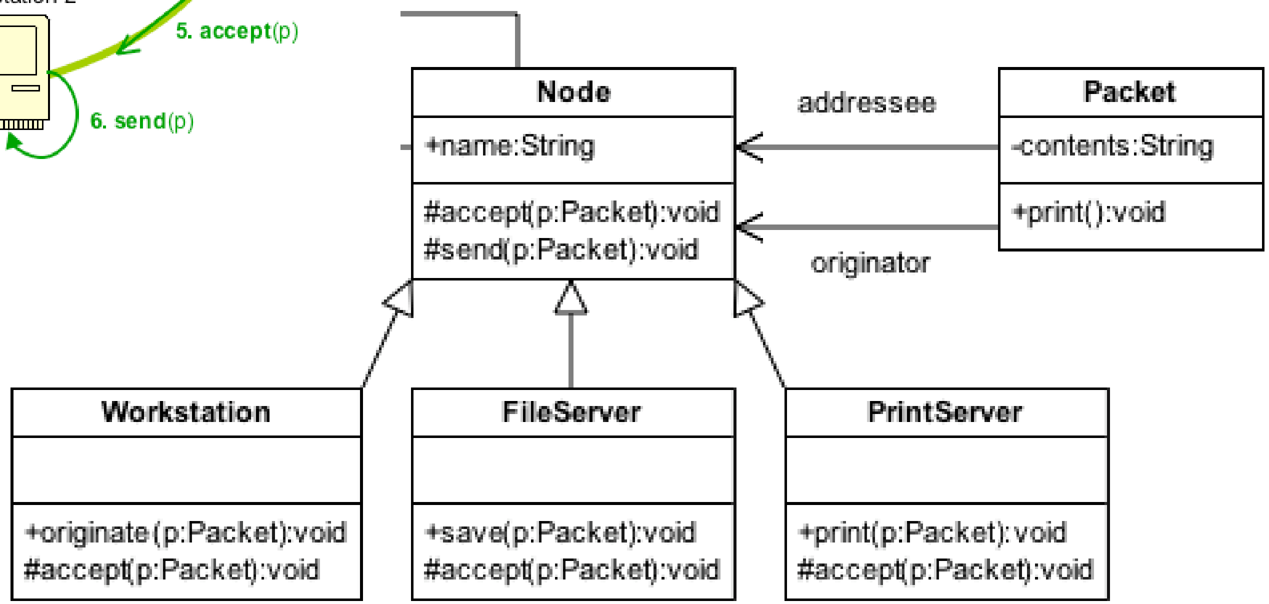
Models are Graphs

- Models can be represented naturally as graphs
  - many diagrams are intrinsically graph-based
    - class diagrams, statecharts, collaboration diagrams, Petri-nets, database schemas

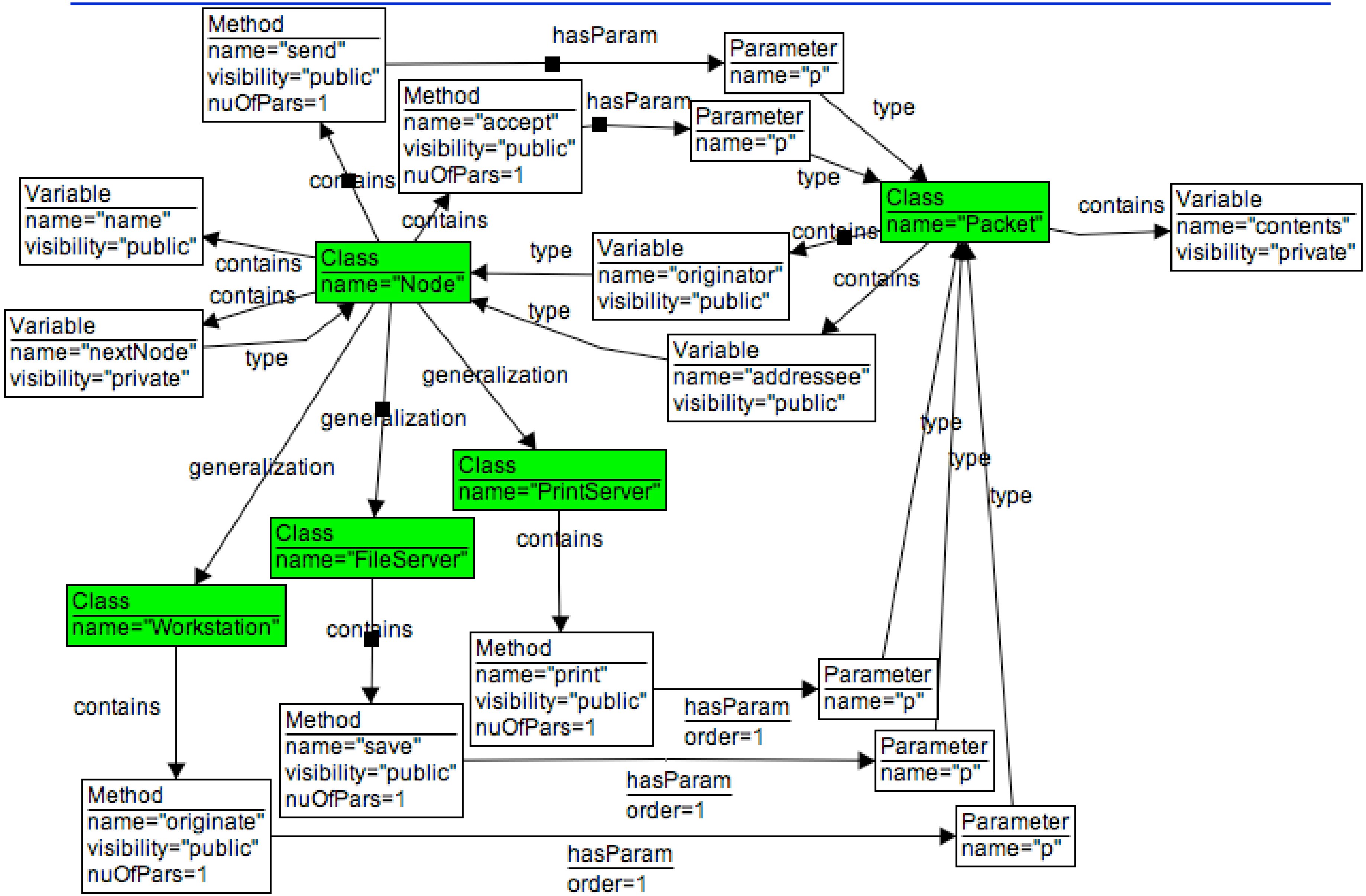
Models are Graphs



Simple example: class diagram of a LAN simulation

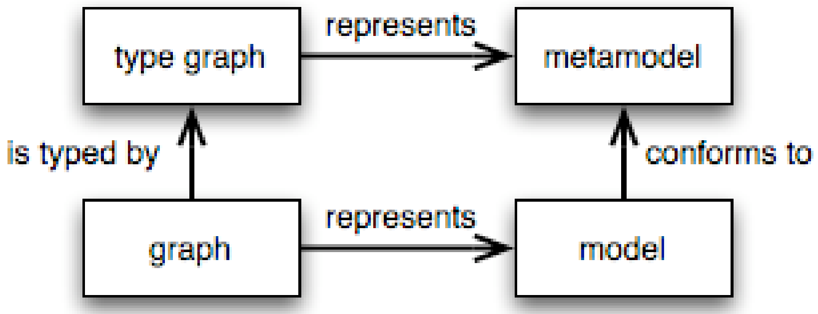


Models are Graphs



Metamodels are type graphs

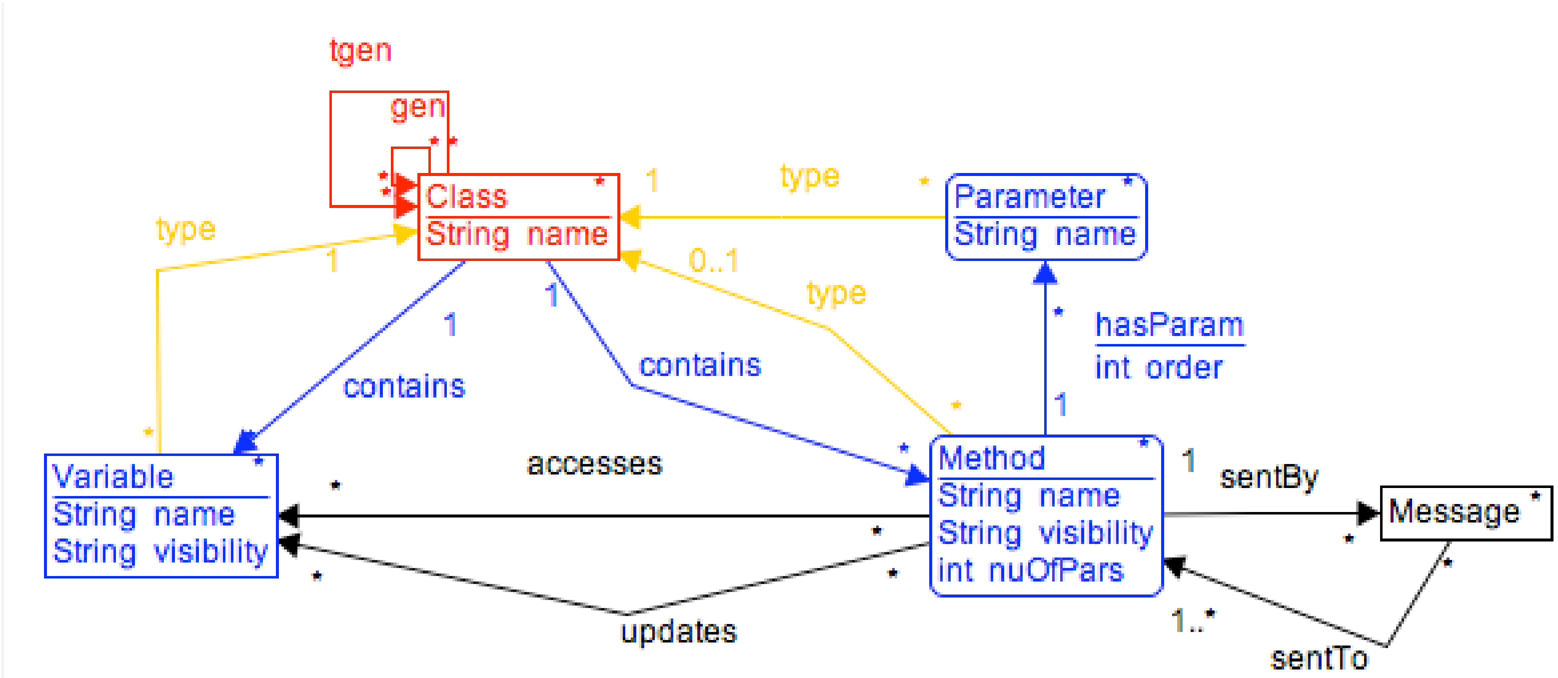
- All models conform to a metamodel that specifies their syntax
- All graphs conform to a type graph that specifies their well-formedness constraints



- Hence, type graphs are the graph-theoretic equivalent of metamodels

Metamodels are type graphs

- Example of a type graph
  - represents a simplified metamodel for UML class diagrams

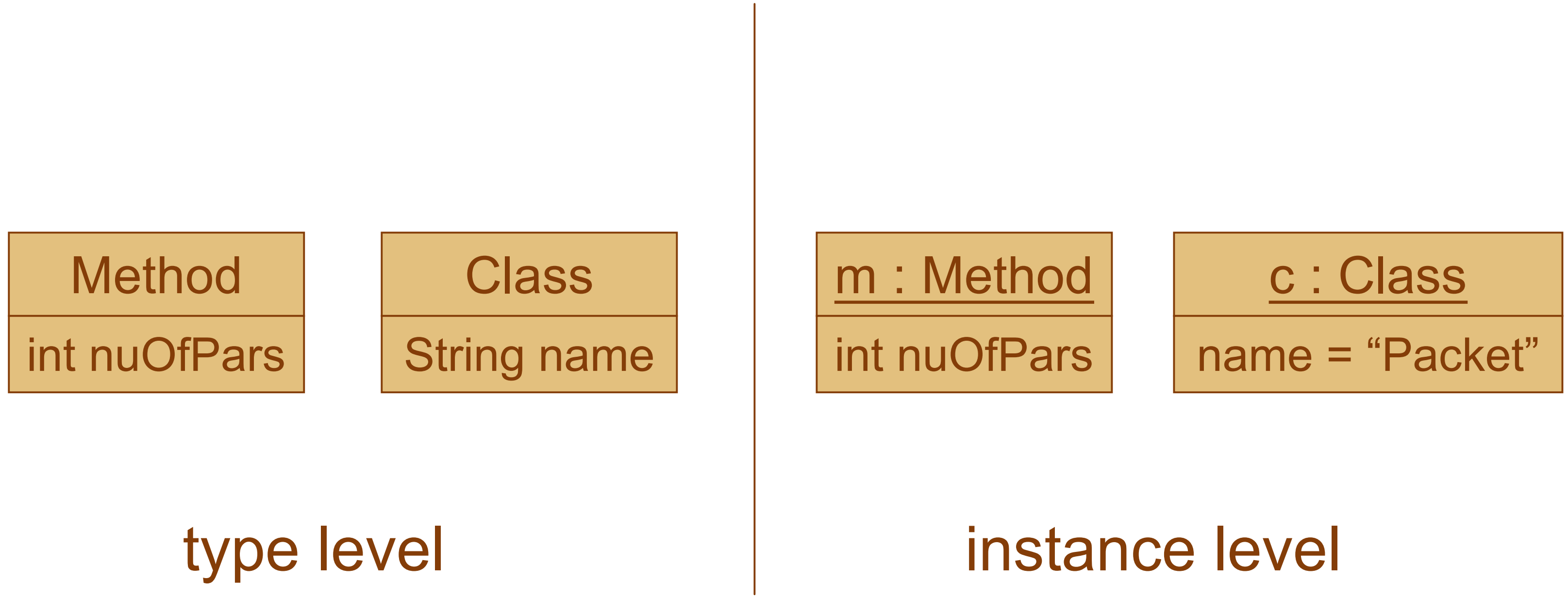


Definitions

- A (directed) graph is an algebraic structure $G = (V, E, s: E \to V, t: E \to V)$
- A graph homomorphism is a mapping $h: G_1 \to G_2$ where $h = (h_V: V_1 \to V_2, h_E: E_1 \to E_2)$ and h_E preserves source and target nodes
- A graph G is typed by a type graph TG if there is a homomorphism $g : G \to TG$
- Direct extension of definitions to labeled graphs
  - where each node and edge may be labeled

Definitions

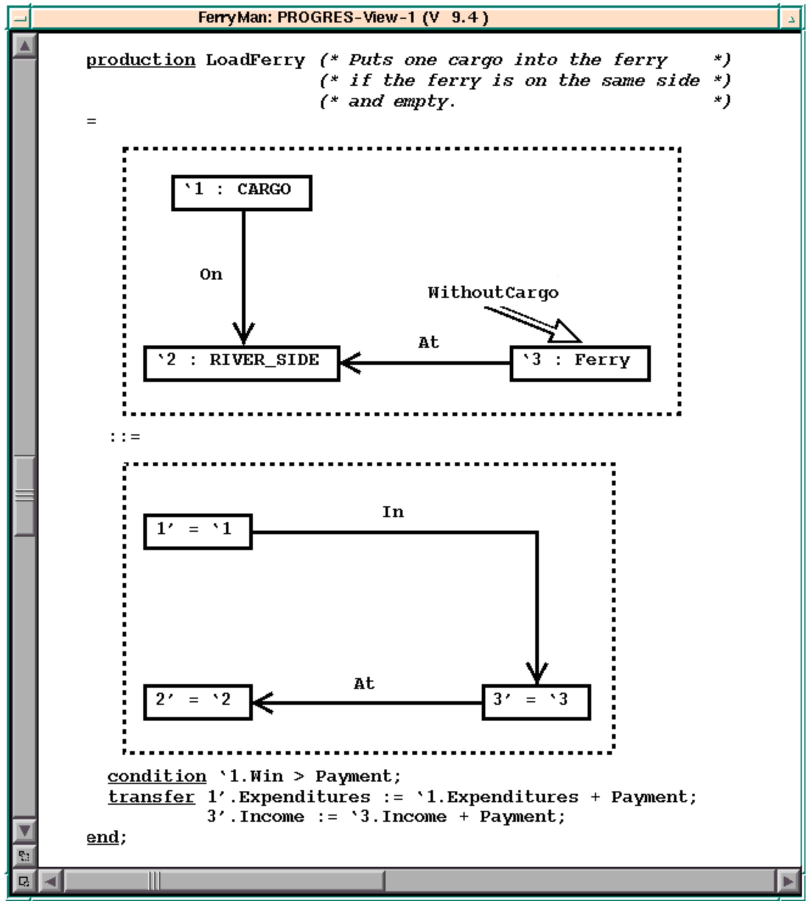
- A *labeled graph* is a graph where each node and edge is labeled over an alphabet L
  - labeling function $l: V \cup E \rightarrow L$
- An *attributed graph* is a graph where nodes an edge are labeled over an abstract data type



Model transformations are Graph transformations

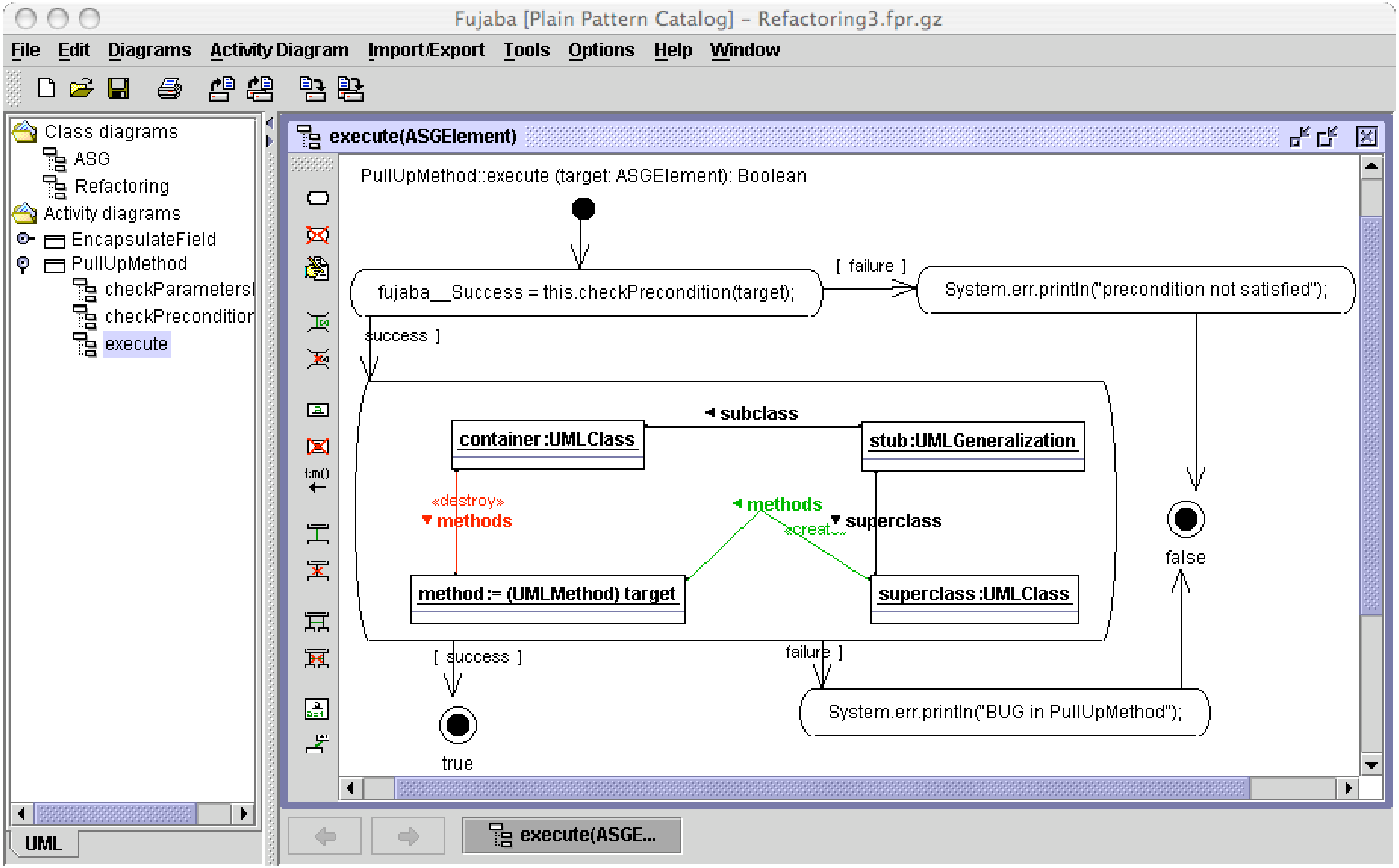
- Model transformations can be naturally represented as graph transformations
- GT theory offers many theoretical results that can help during analysis
  - type graph, negative application conditions, parallel and sequential (in)dependence, confluence, critical pair analysis
- GT tools allow us to perform concrete experiments
  - Fujaba, *AGG*, Progres, ...

GT tools - PROGRES

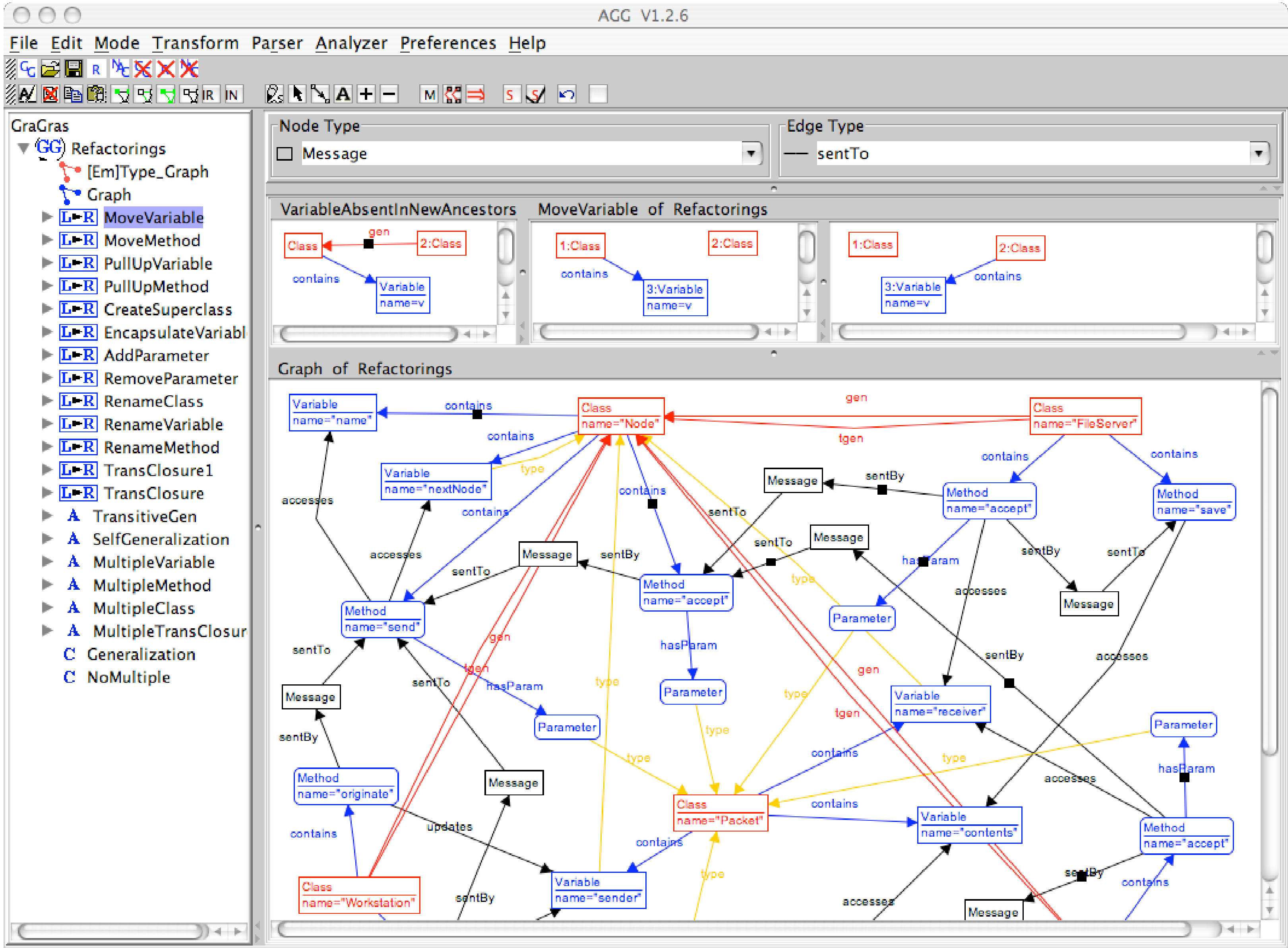


- Graphical/textual language to specify graph transformations
- Graph rewrite rules with complex and negative conditions
- Cross compilation in Modula 2, C and Java

GT tools - Fujaba



GT tools - AGG


AGG V1.2.6
File Edit Mode Transform Parser Analyzer Preferences Help
GraGras
Refactorings
[Em]Type_Graph
Graph
MoveVariable
MoveMethod
PullUpVariable
PullUpMethod
CreateSuperclass
EncapsulateVariabl
AddParameter
RemoveParameter
RenameClass
RenameVariable
RenameMethod
TransClosure1
TransClosure
TransitiveGen
SelfGeneralization
MultipleVariable
MultipleMethod
MultipleClass
MultipleTransClosur
Generalization
NoMultiple
Node Type
Message
Edge Type
sentTo
VariableAbsentInNewAncestors
MoveVariable of Refactorings
Graph of Refactorings

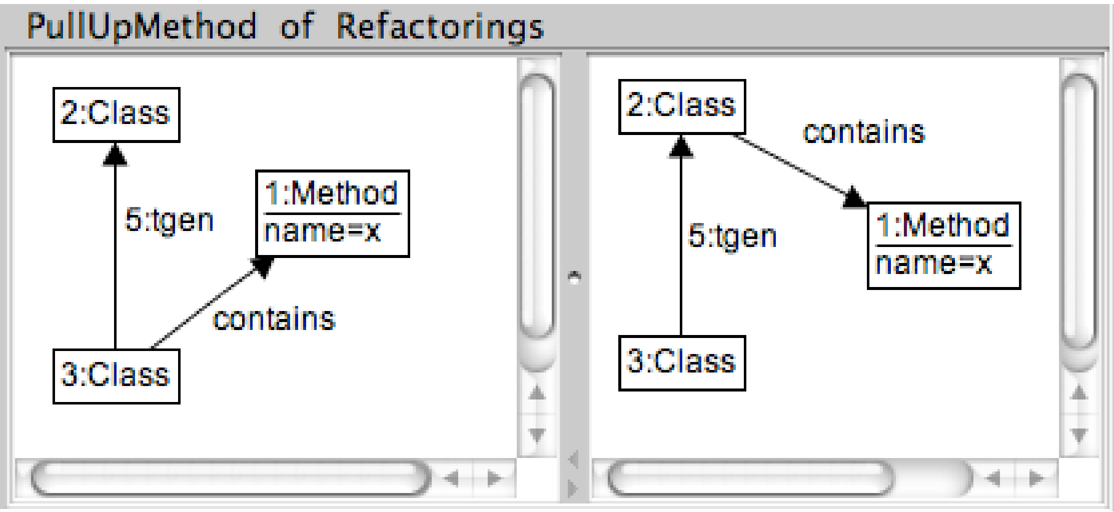
Graph Production

- A graph production p: L→R is a structure-preserving partial mapping between (directed, labeled, typed) graphs
  - Preserves sources and targets of edges
  - Preserves node and edge types
  - Preserves node and edge labels
  - Partial means that nodes or edges may be deleted

Graph Production

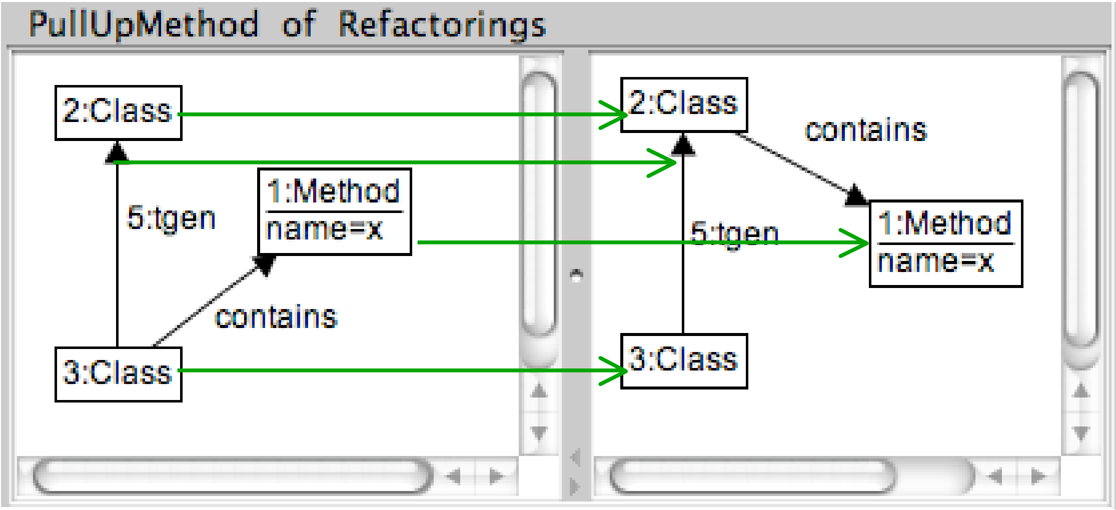
- Exemple: Pull Up Method refactoring

left-hand side L right-hand side R



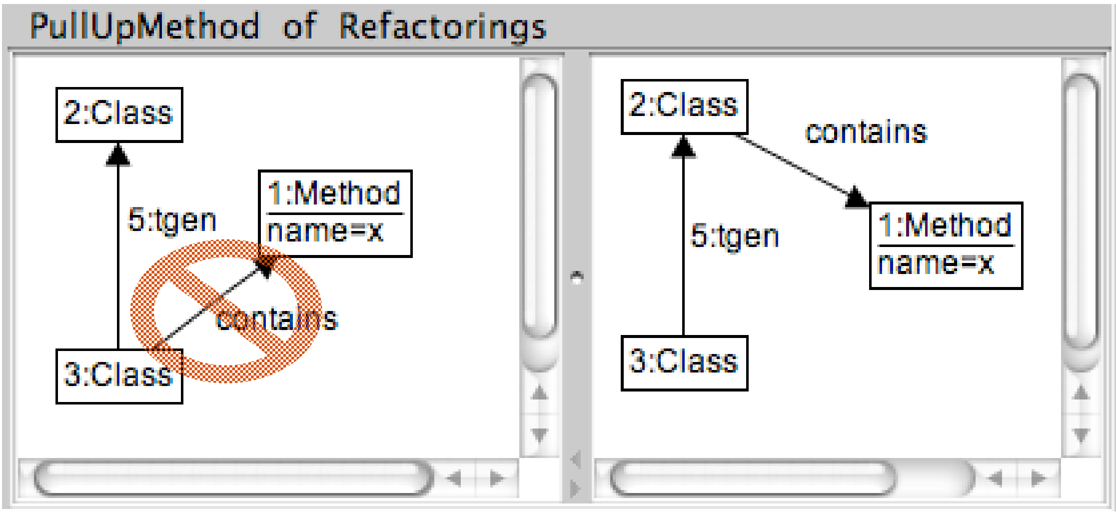
Graph Production

- Exemple: Pull Up Method refactoring
  - Some nodes and edges are preserved



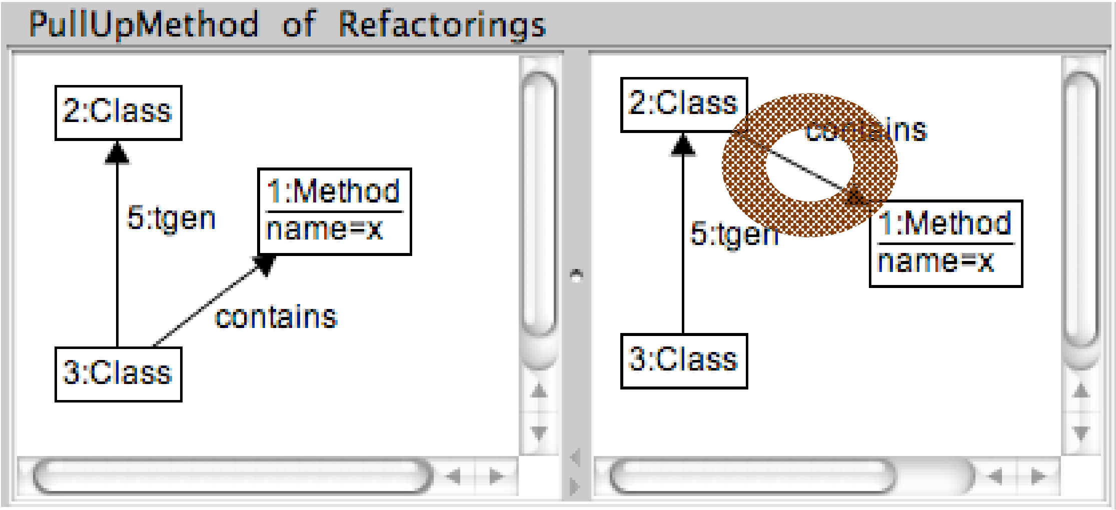
Graph Production

- Exemple: Pull Up Method refactoring
  - Some nodes and edges are deleted



Graph Production

- Exemple: Pull Up Method refactoring
  - Some nodes and edges are added



Graph Transformation

- A graph transformation $t: G \Rightarrow H$ is the application of a graph production $p: L \rightarrow R$ that is matched in the context of a given graph G
  - $t = (p,m)$ where $m: L \rightarrow G$ is an injective graph morphism (match)

$$\begin{tikzcd} L \arrow[r, "p"] \arrow[d, "m"'] & R \\ G & \end{tikzcd}$$

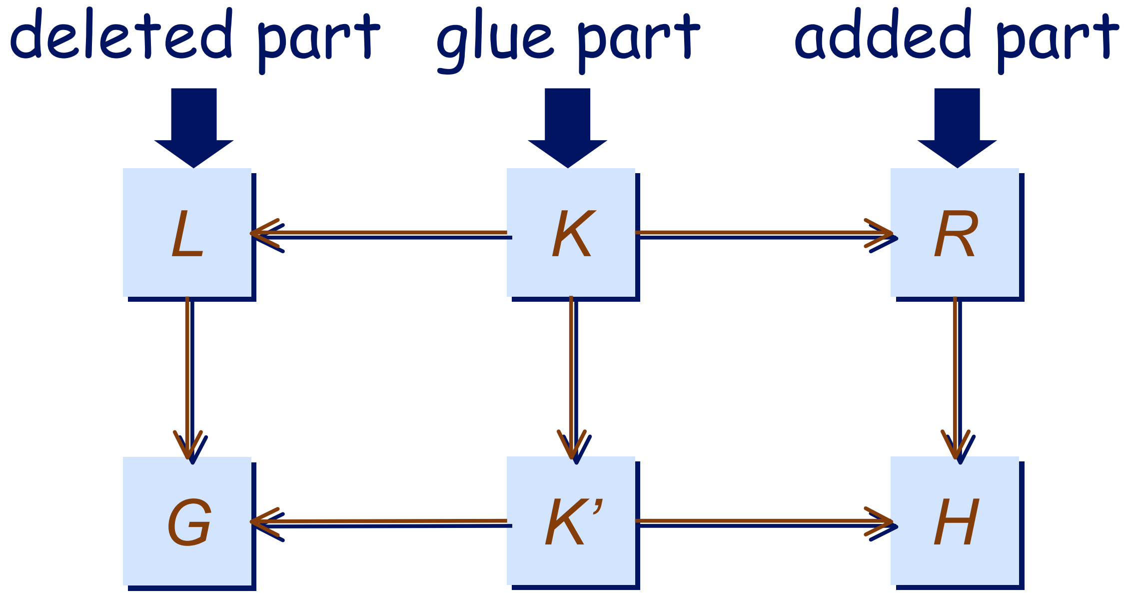
Graph Transformation

- A graph transformation $t: G \Rightarrow H$ is the application of a graph production $p: L \rightarrow R$ that is matched in the context of a given graph G
  - $t = (p,m)$ where $m: L \rightarrow G$ is an injective graph morphism (match)

$$
\begin{tikzcd}
L \arrow[r, "p"] \arrow[d, "m"'] & R \arrow[d] \\
G \arrow[r] & H
\end{tikzcd}
$$

Graph Transformation

- Alternative definition: algebraic double-pushout (DPO) approach
  - Explicit presentation of intersection K = L ∩ R



Graph Transformation Step

- Operational description
  - prepare transformation by
    - selecting rule $p: L \rightarrow R$
    - selecting match $m: L \rightarrow G$
  - create new graph H by
    - removing from G the occurrence of $L \setminus R$
    - adding to result a copy of $R \setminus L$

$$\begin{tikzcd} L \arrow[r, "p"] \arrow[d, "m"'] & R \arrow[d] \\ G \arrow[r] & H \end{tikzcd}$$

Graph Transformation Step

- Declarative description
  - Set-theoretic
    - $G \Rightarrow_{p(m_L)} H$ iff there exists a homomorphism $m\colon L \cup R \rightarrow G \cup H$ such that $m(L \setminus R) = G \setminus H$ and $m(R \setminus L) = H \setminus G$
  - Category-theoretic (DPO):
    - $G \Rightarrow_{p(m_L)} H$ iff (1) and (2) are pushouts

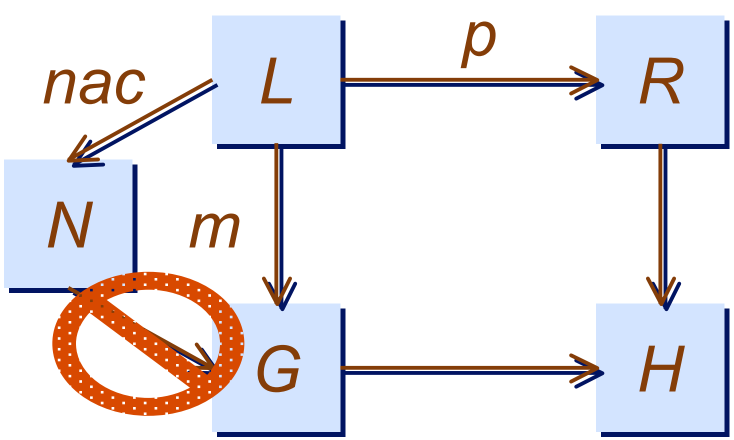
$$\begin{tikzcd}
L \arrow[d, "m_L"] \arrow[dr, phantom, "(1)"] & K := L \cap R \arrow[l] \arrow[r] \arrow[d, "m_K"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "m_R"] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Advanced GT features

- Negative application conditions
- Set nodes (multi objects)
- Attributed graphs and GTs
- Programmed graph transformation
- Graph grammars

Negative application condition

- A negative application condition *nac*: $L \rightarrow N$ of a graph production *p*: $L \rightarrow R$ represents a forbidden context. In a graph transformation *t*: $G \Rightarrow H$, no match $N \rightarrow G$ must be found



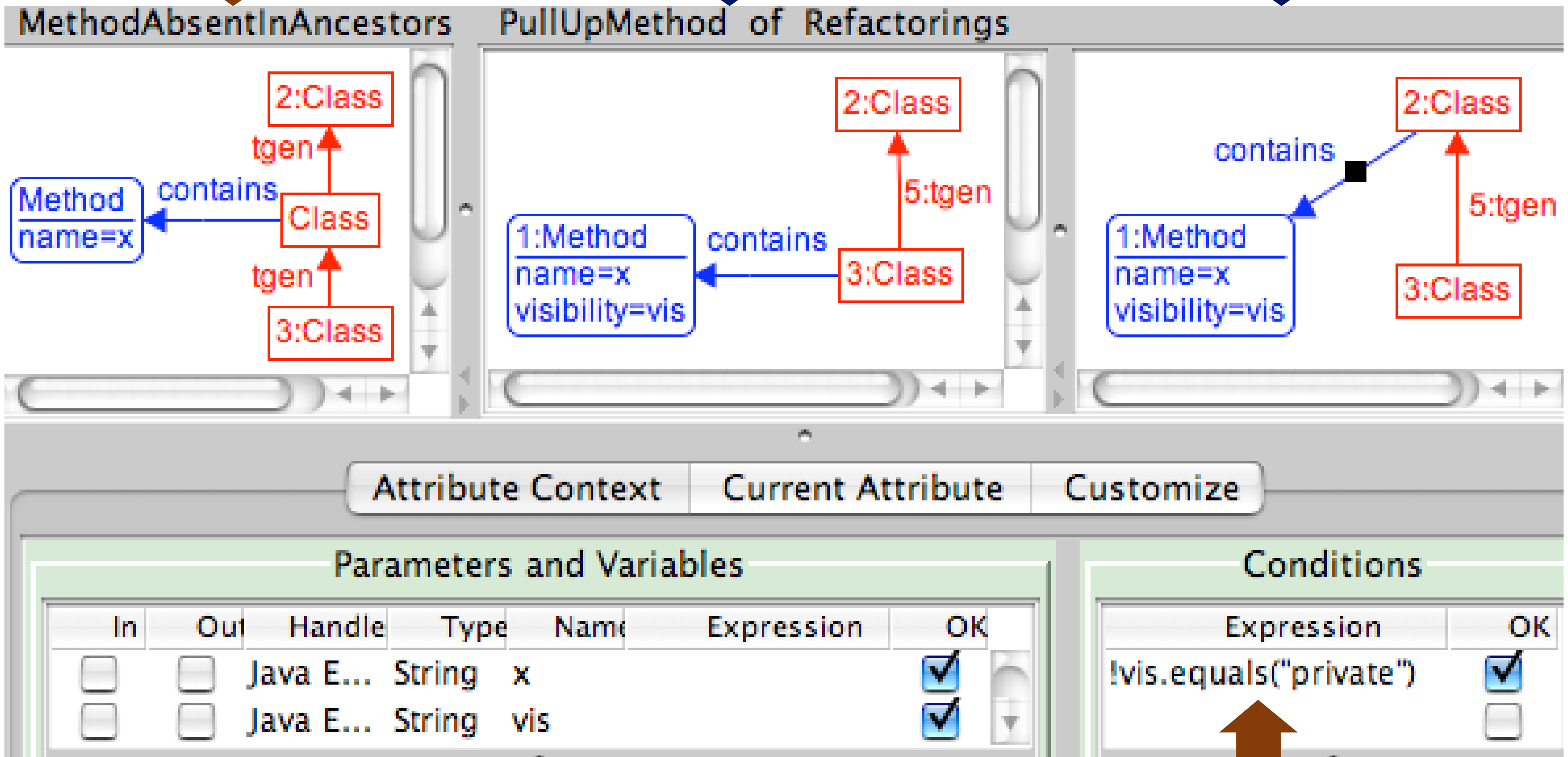
Negative application condition

- Example: Pull Up Method refactoring
  - node attributes needed
    - to express name and visibility of methods
    - to *constrain* (or *modify*) visibility of methods
  - negative application conditions (NAC) needed
    - Method signature of method to be pulled up should be absent in ancestors
    - Method to be pulled up should not be private

Negative application condition

- Example: Pull Up Method refactoring part 1 (P_1)

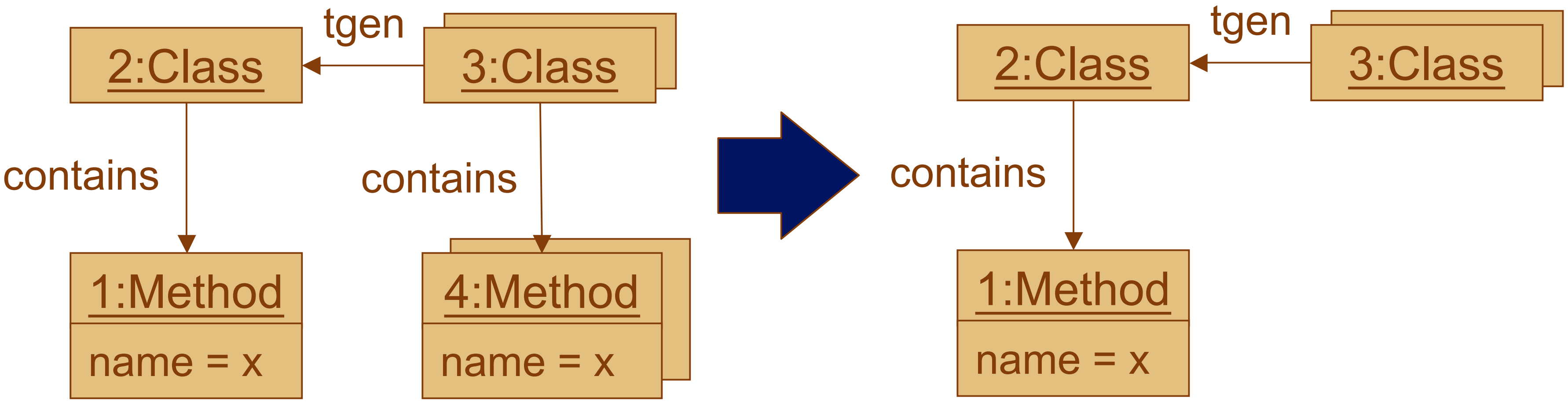
NAC　　left-hand side L　　right-hand side R



NAC

Set nodes

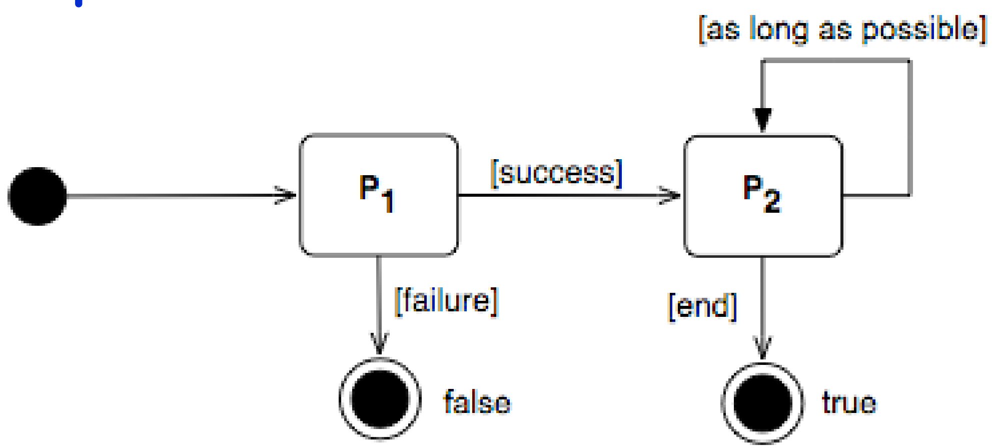
- Example: Pull Up Method refactoring part 2 (P_2)
  - Remove *all* remaining methods with same name in *all* subclasses
    - requires set nodes …



    - … or programmed GT

Programmed GT

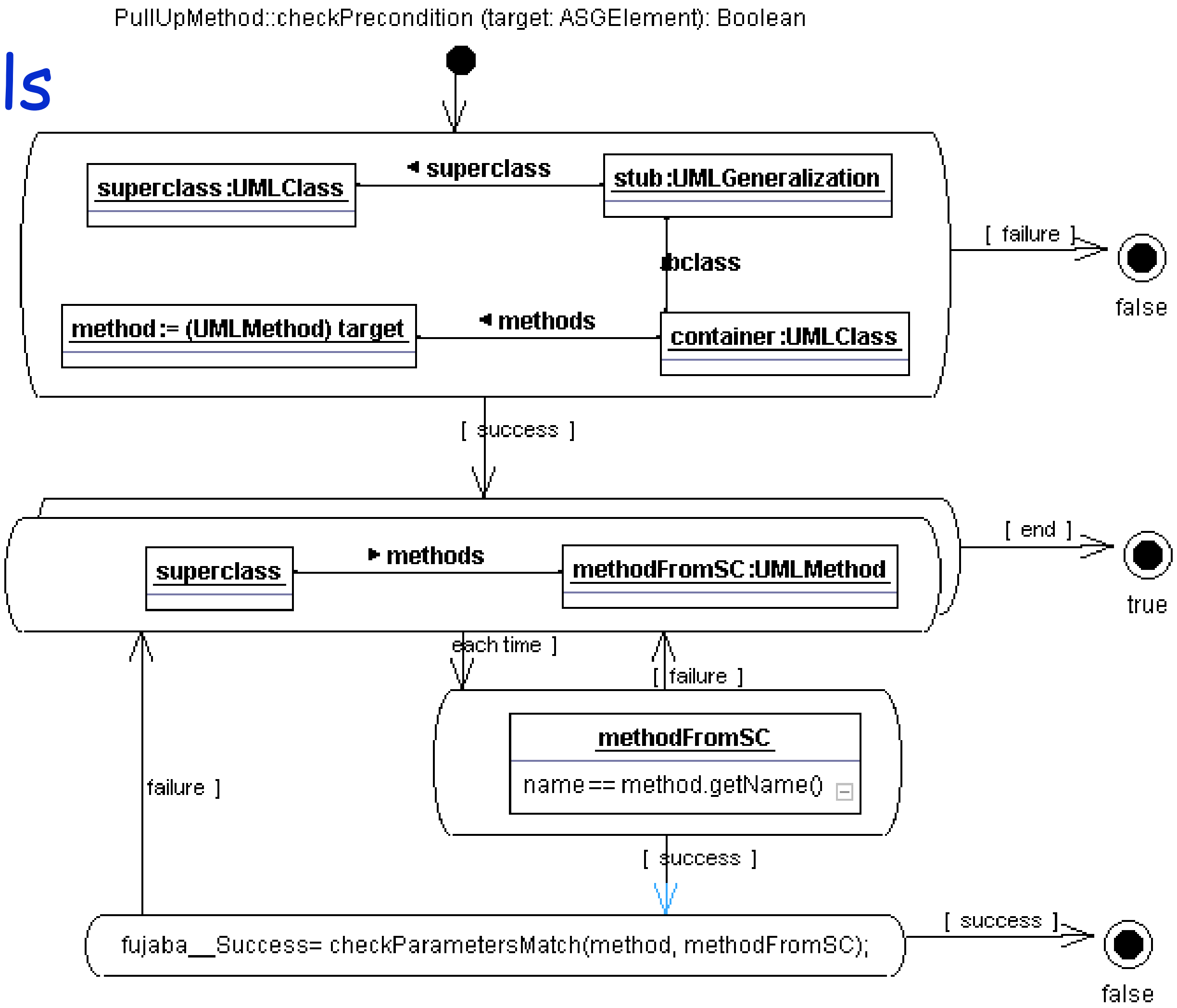
- For composite graph productions, we need to control their order of application by means of
  - sequencing
  - branching
  - looping
- For example



Programmed GT

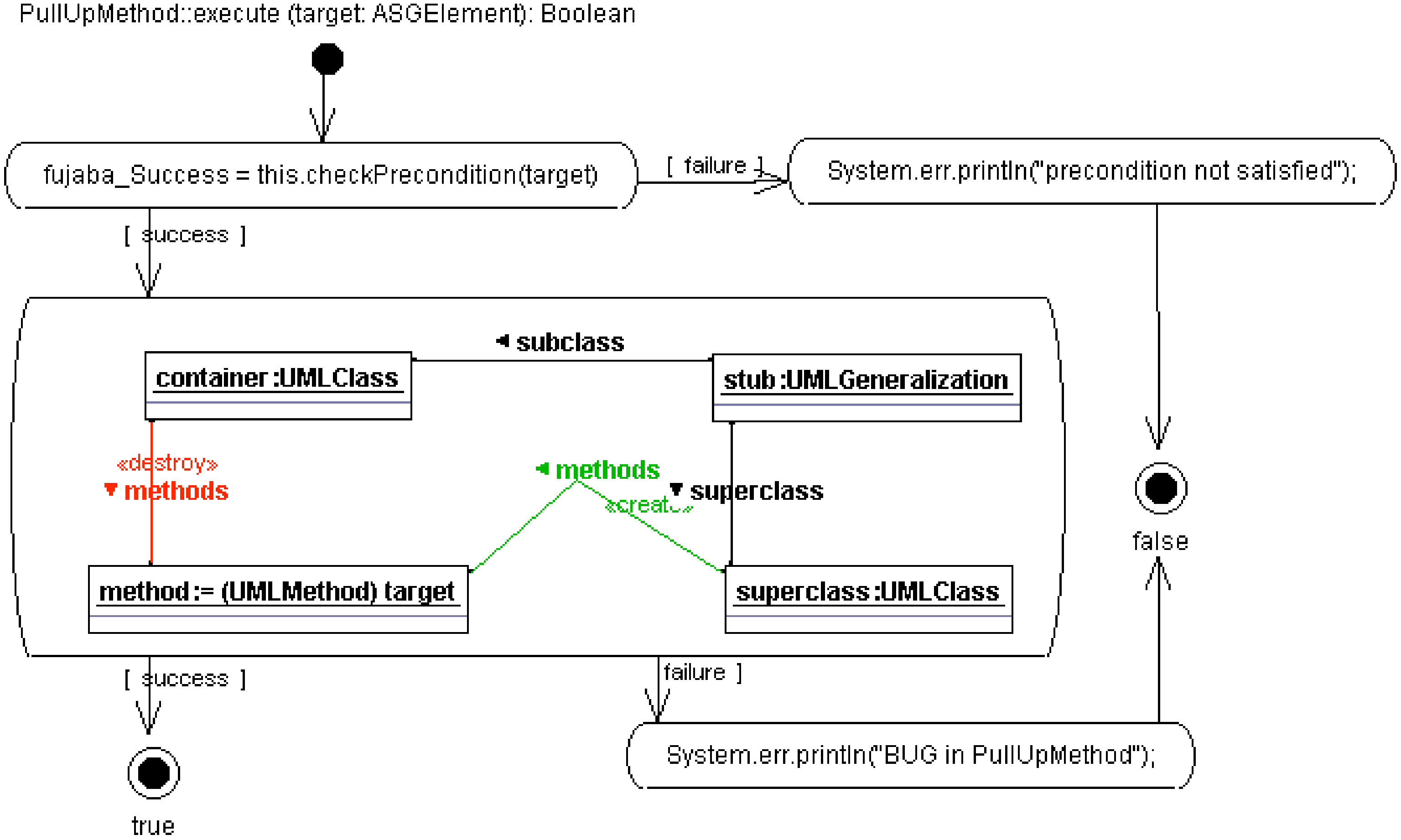
Supported by tools like Fujaba

using the story diagram notation



Programmed GT

- Supported by tools like Fujaba



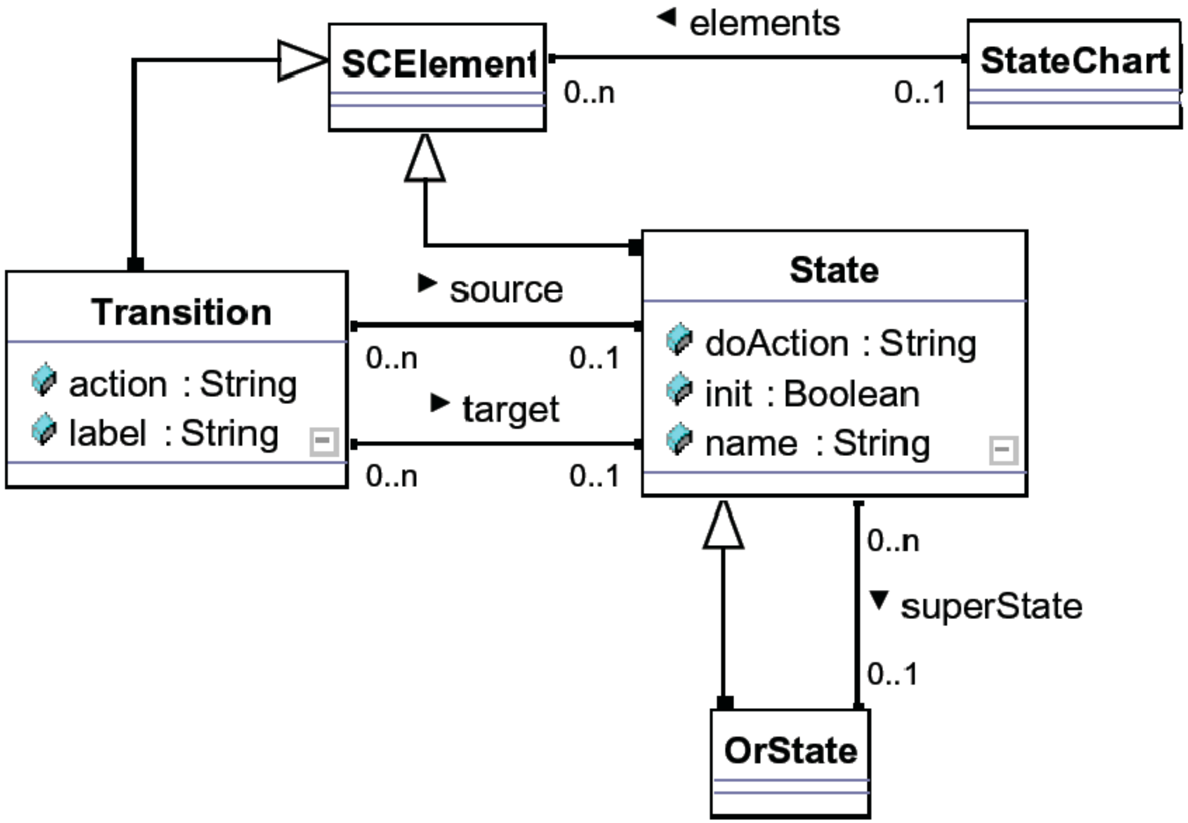
Programmed GT

- Supported by tools like *Fujaba*
  - using the story diagram notation

- Example
  - Statechart flattening revisited

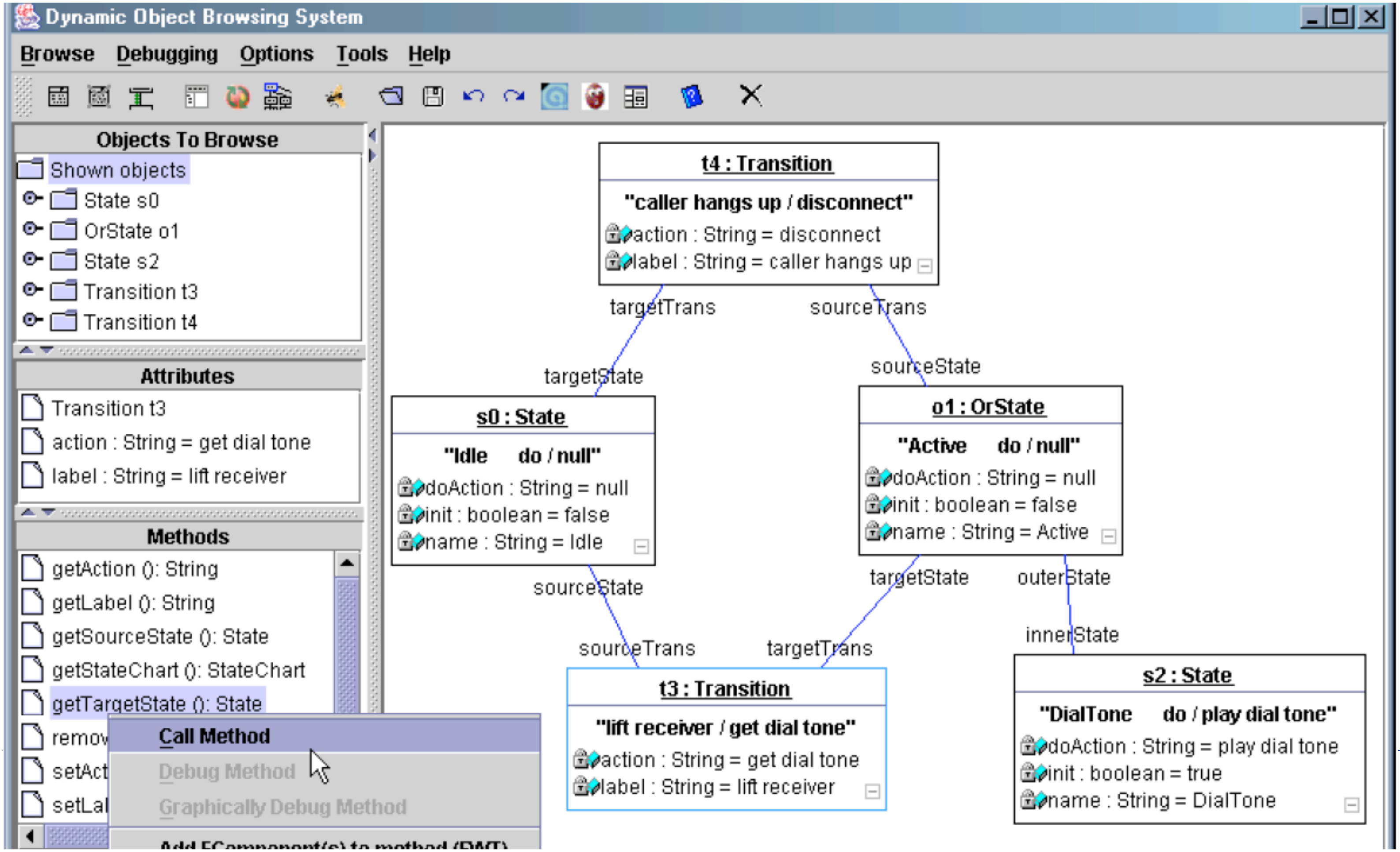
Programmed GT

- Example: Statechart flattening revisited
  - Step 1: type graph for statecharts



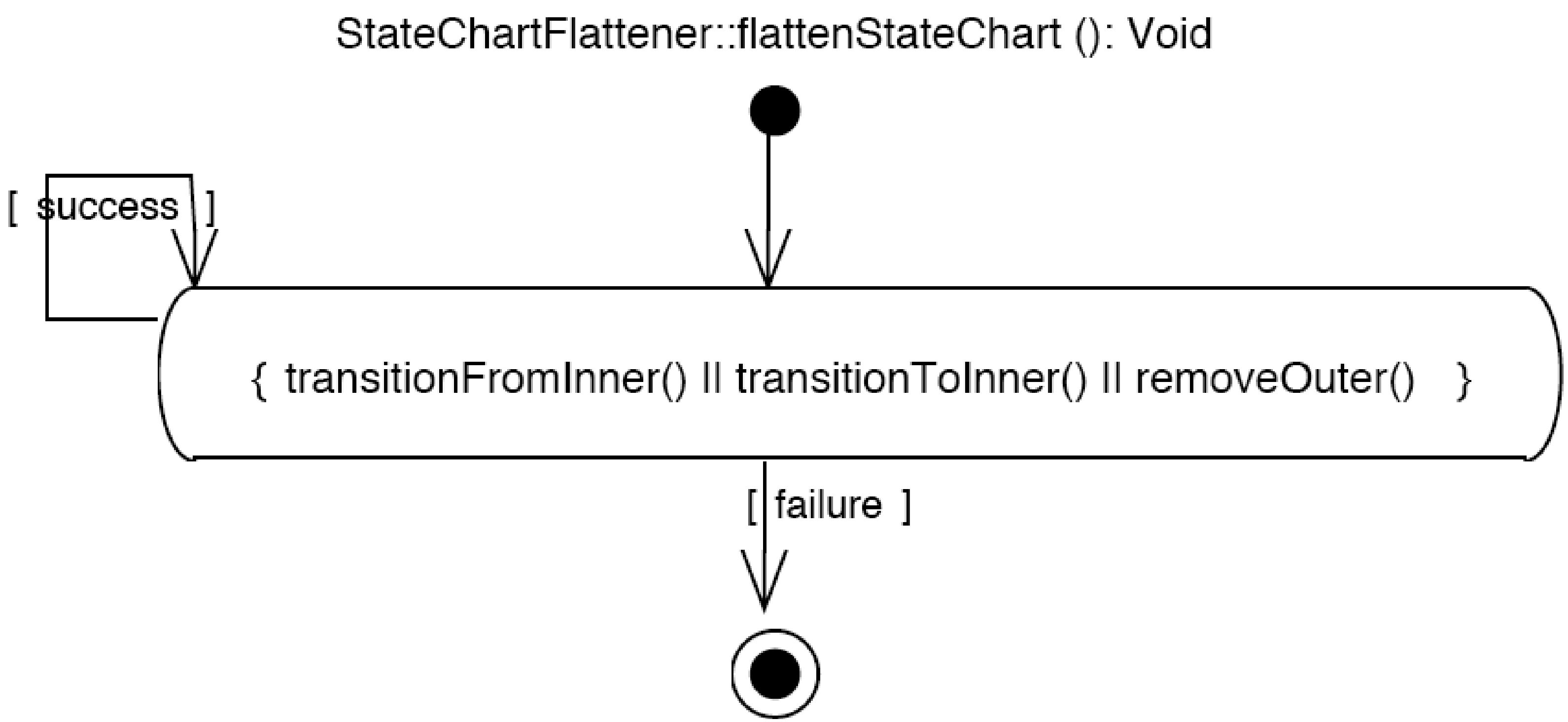
Programmed GT

- Example: Statechart flattening revisited
  - Step 2: statecharts as executable graphs



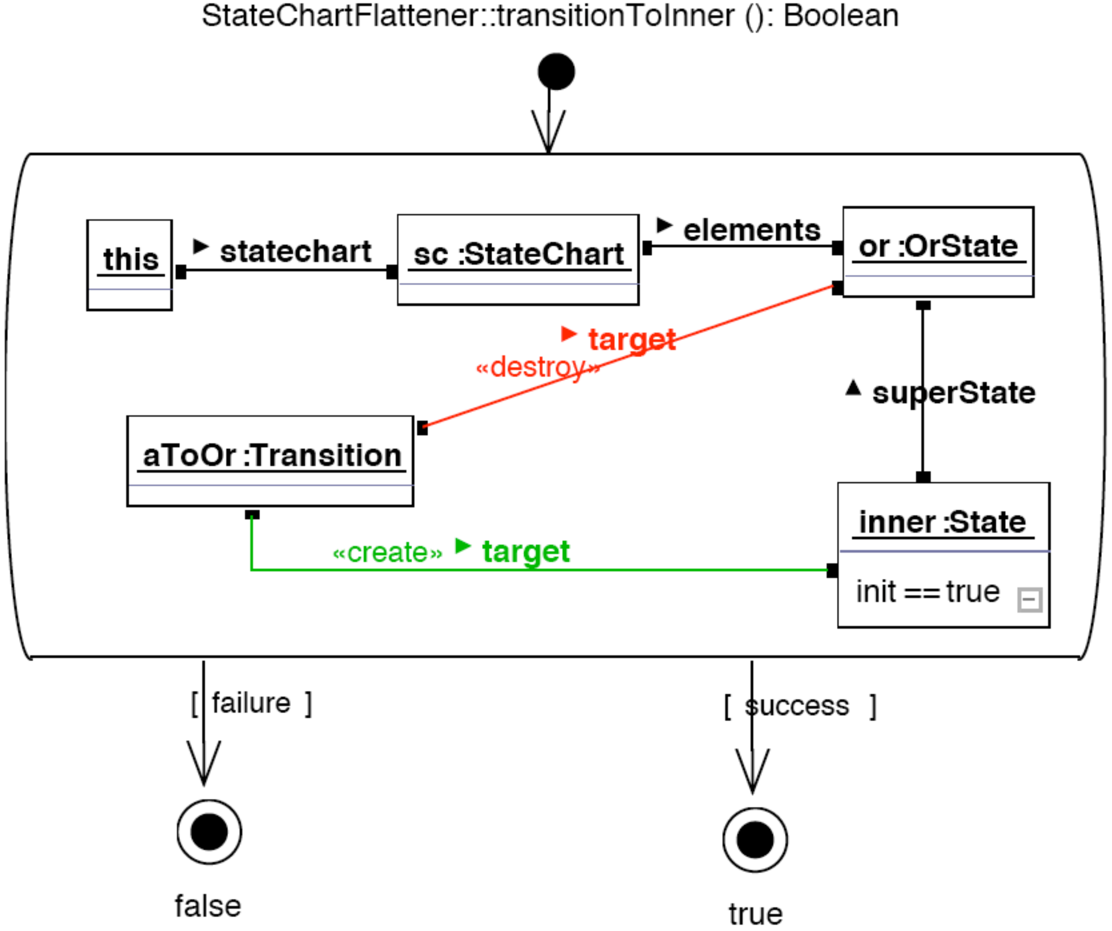
Programmed GT

- Example: Statechart flattening revisited
  - Step 3: specifying the statechart flattening



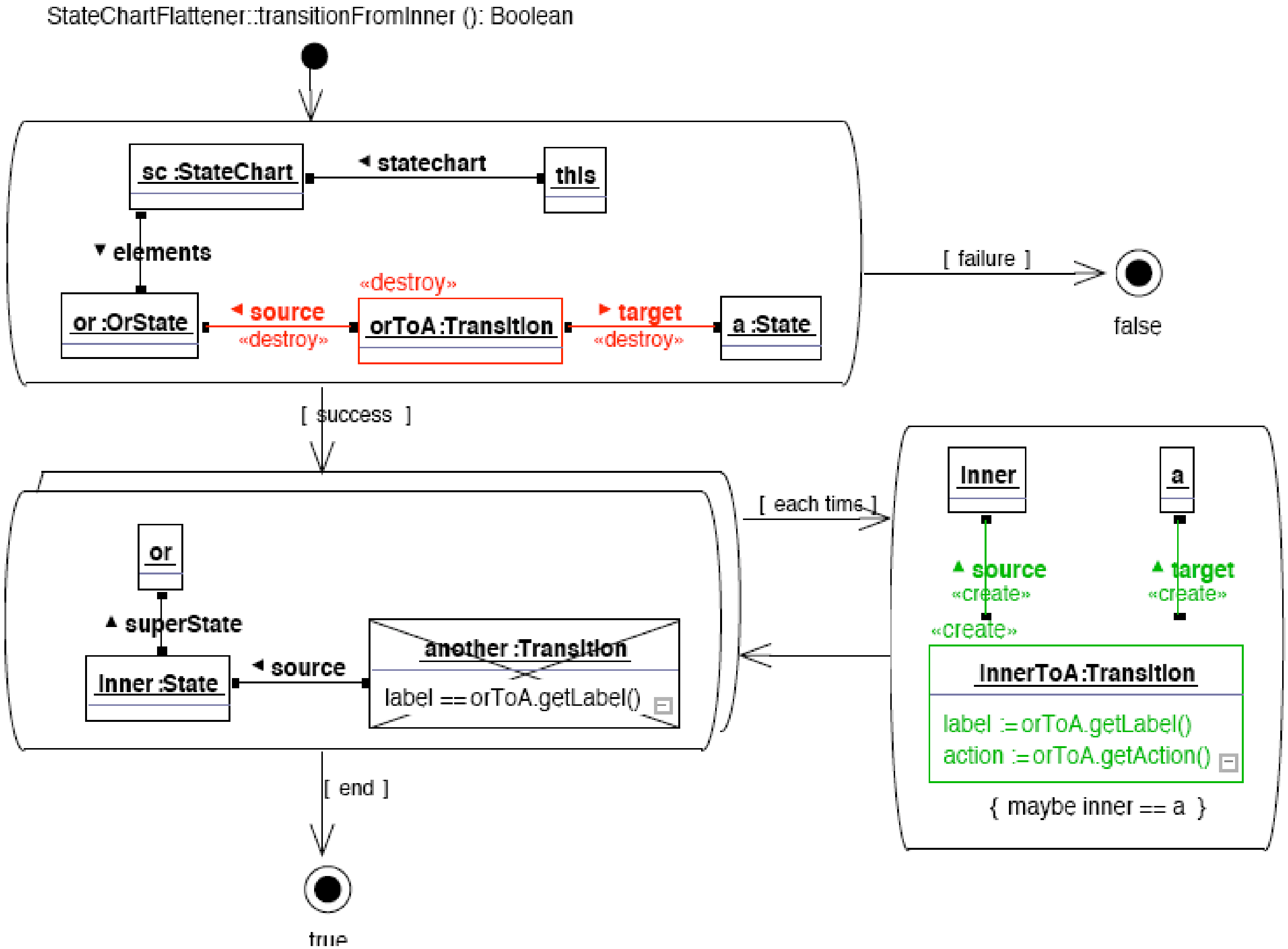
Programmed GT

- Example: Statechart flattening revisited
  - Step 3: specifying the statechart flattening



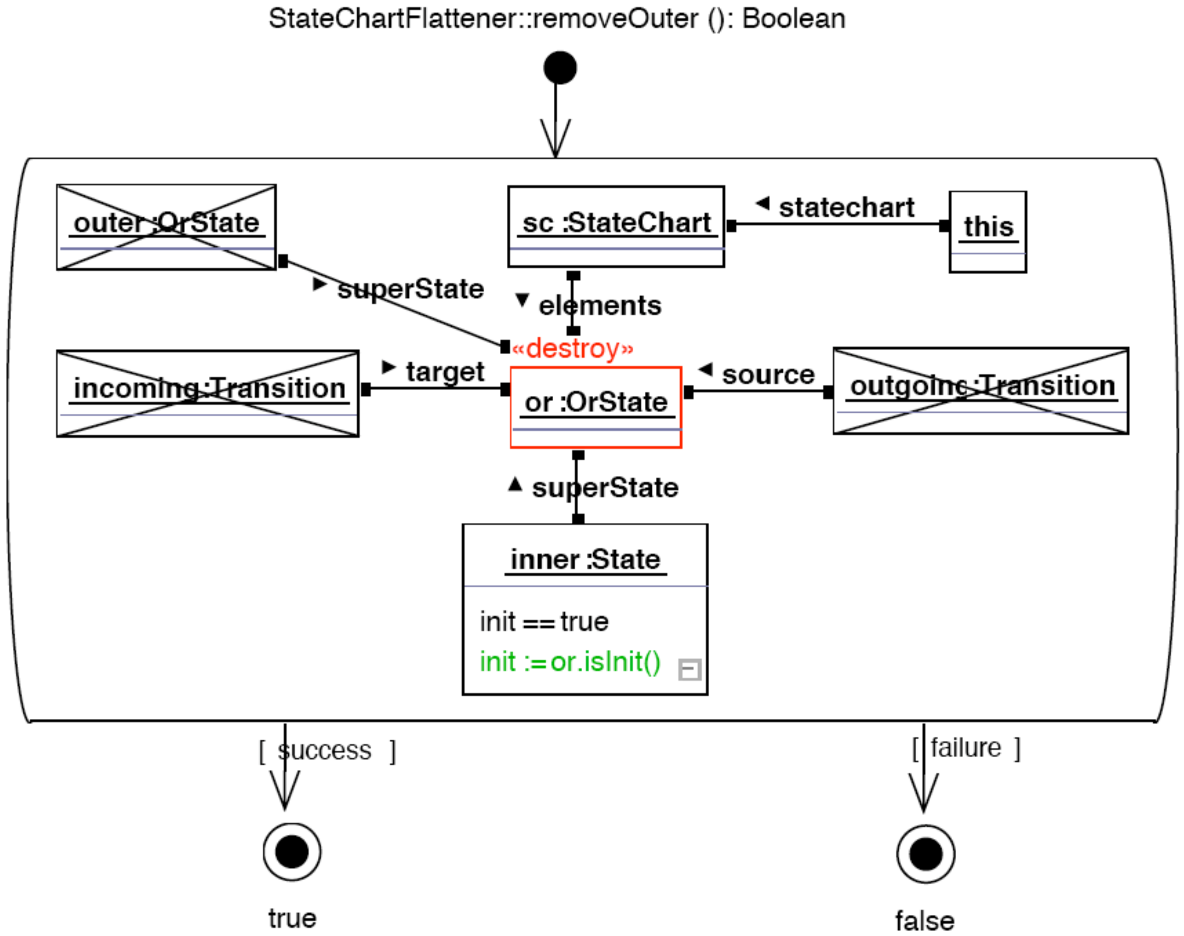
Programmed GT

- Example: Statechart flattening revisited
  - Step 3: specifying the statechart flattening



Programmed GT

- Example: Statechart flattening revisited
  - Step 3: specifying the statechart flattening



Graph Grammars

- *A graph grammar* is a set of graph productions
- No control structure is imposed on the graph productions to be applied
  - Productions are applied at random, whenever a match is found
- Graph grammars are supported by *AGG* tool

Where it comes from ...

Chomsky Grammars

Term Rewriting

Petri Nets

Graph Transformation and Graph Grammars

What it is good for ...

Chomsky Grammars

Term Rewriting

Petri Nets

Graph Transformation and Graph Grammars

Diagram Languages

Models of Computation

Behaviour Modelling

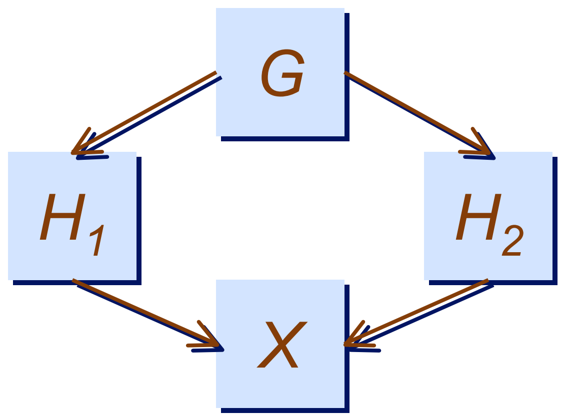
Visual Programming

What it is good for ...

- Behaviour modelling
  - Detecting dependencies and conflicts in functional behaviour
  - Crucial in collaborative/parallel software development

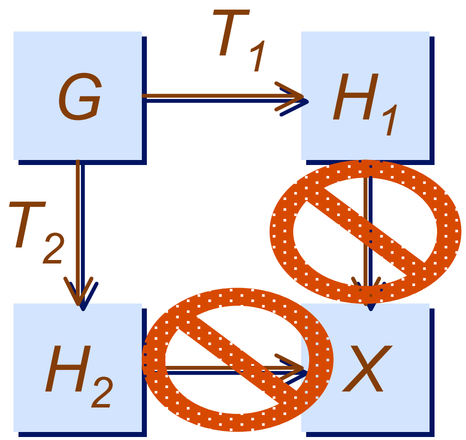
Confluence

- A graph grammar is confluent if it has functional behaviour
  - The end result does not depend on the order in which graph productions are applied



Conflicts

- Two graph transformations T_1 and T_2 are in conflict if
  - it is not possible to apply T_1 after T_2, or T_2 after T_1, or both, via the same match



Parallel independence

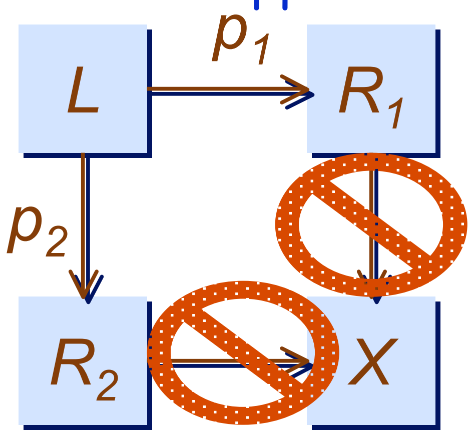
- Two graph transformations T_1 and T_2 are *parallel independent* if
  - they are not in conflict
- They are *parallel dependent* if they are in conflict
  - In that case they may or may not be sequentialisable

Critical pair analysis

- Needed to detect whether a graph grammar has the confluence property
  - p_1 and p_2 form a *critical pair* if

    they can both be applied to the same minimal context graph L

    but applying p_1 prohibits application of p_2 and/or vice versa



Critical pair analysis

- Conflicts between graph transformations can be found by detecting critical pairs of graph productions
- Critical pair lemma
  - For each pair $H_1 \Leftarrow G \Rightarrow H_2$ of graph transformations in conflict, there is a critical pair $R_1 \Leftarrow L \Rightarrow R_2$ expressing the same conflict in a minimal context

$$\begin{tikzcd}
R_1 \arrow[d] & L \arrow[l, "p_1"'] \arrow[r, "p_2"] \arrow[d, "m"'] & R_2 \arrow[d] \\
H_1 & G \arrow[l, "T_1"'] \arrow[r, "T_2"] & H_2
\end{tikzcd}$$

Critical pair analysis

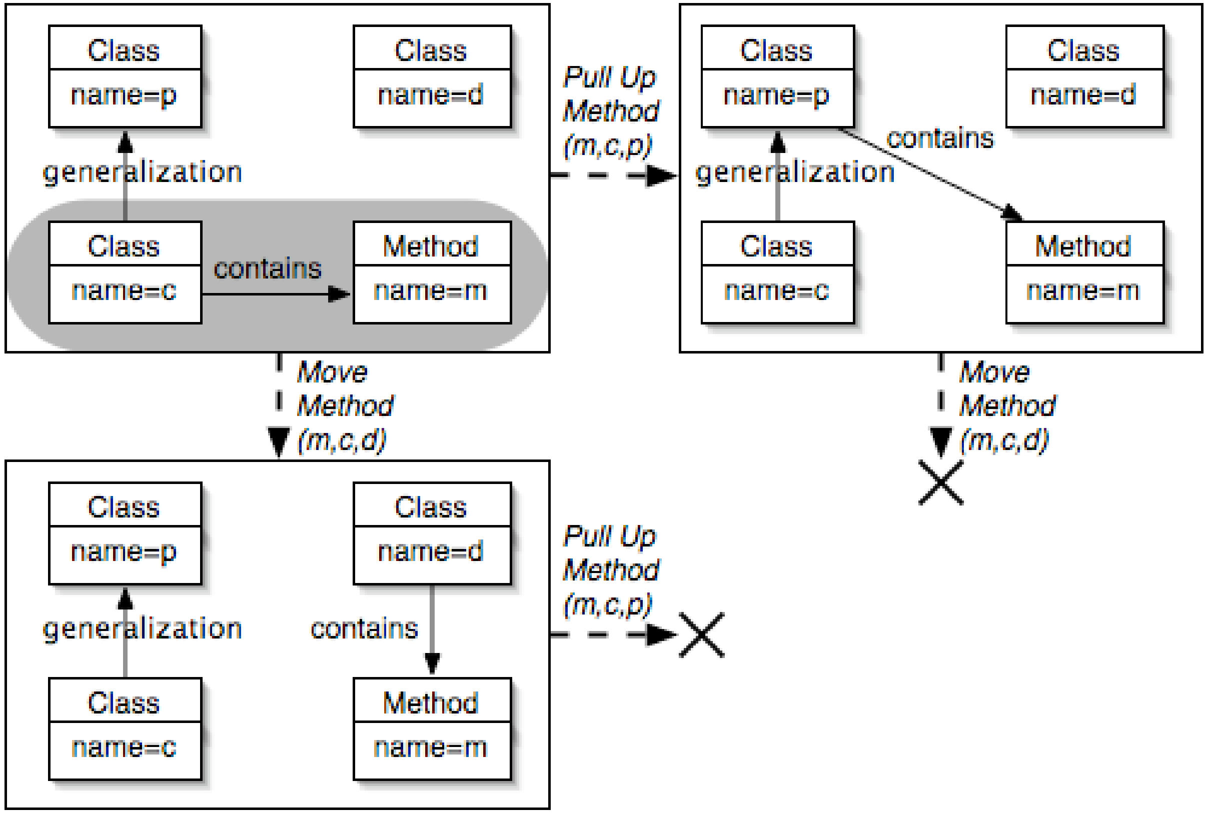
- Supported by *AGG* tool
  - potential conflicts can be detected statically
  - e.g. critical pairs between refactoring productions

Critical Pairs

first \ second	1: Mo...	2: Mo...	3: Pul...	4: Pul...	5: Cr...	6: En...	7: Ad...	8: Re...	9: Re...	10: R...	11: R...
1: MoveVariable	3	0	4	0	0	2	0	0	0	2	0
2: MoveMethod	0	3	0	4	0	2	2	2	0	0	2
3: PullUpVariable	3	0	4	0	0	2	0	0	0	1	0
4: PullUpMethod	0	4	0	3	0	2	3	3	0	0	1
5: CreateSuperclass	0	0	0	0	0	0	0	0	3	0	0
6: EncapsulateVariable	2	2	2	2	0	0	0	0	0	0	1
7: AddParameter	0	0	0	0	0	0	0	2	0	0	0
8: RemoveParameter	0	0	0	0	0	0	2	2	0	0	0
9: RenameClass	0	0	0	0	2	0	0	0	2	0	0
10: RenameVariable	2	0	2	0	0	1	0	0	0	2	0
11: RenameMethod	0	2	0	2	0	1	1	1	0	0	2

Critical pair analysis

- Concrete example of a critical pair
  - *PullUpMethod* versus *MoveMethod*



Sequential dependencies

- Given two graph productions p_1 and p_2 (not necessarily applicable to the same initial graph), are there sequential dependencies between them?

  - Negative dependency
    - p_1 cannot be applied after p_2
    - p_1 violates the truth of p_2's precondition
    - Corresponds to a critical pair
  - Positive dependency
    - p_1 can only be applied after p_2
    - p_1 enables the truth of part of p_2's precondition
    - Not (yet) supported by *AGG* tool