


## Tutorial Introduction to Graph Transformation

### A software engineering perspective

Luciano Baresi  
Politecnico di Milano

Reiko Heckel  
University of Paderborn



Politecnico di Milano

Universität Paderborn

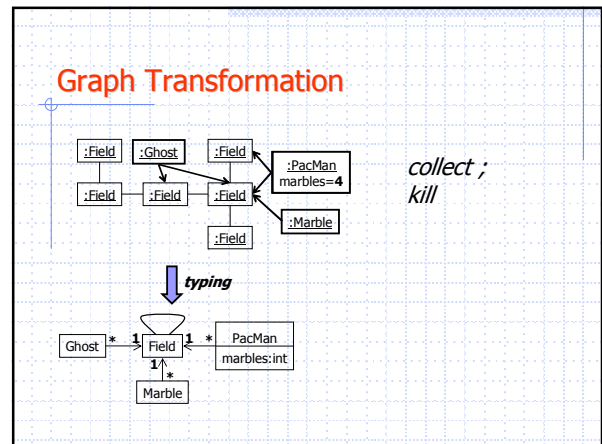
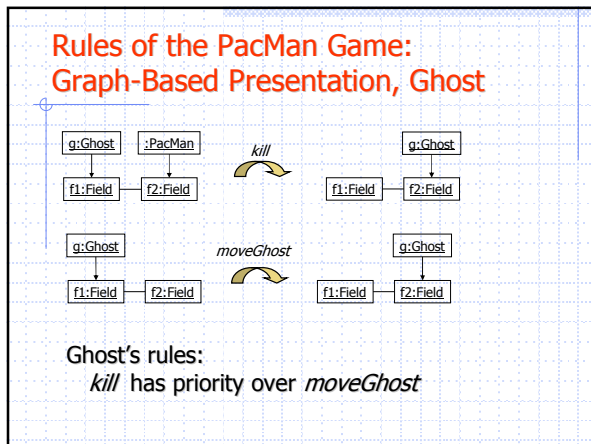
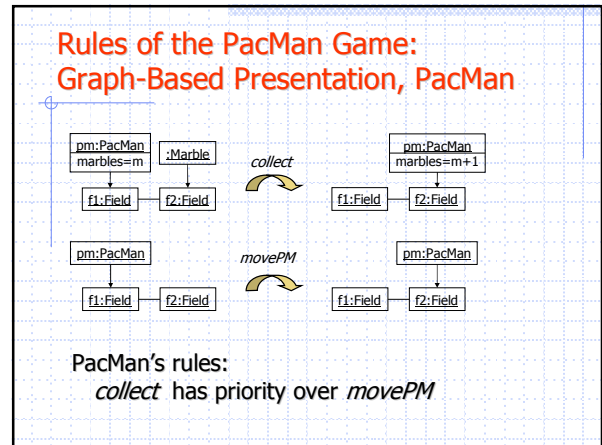
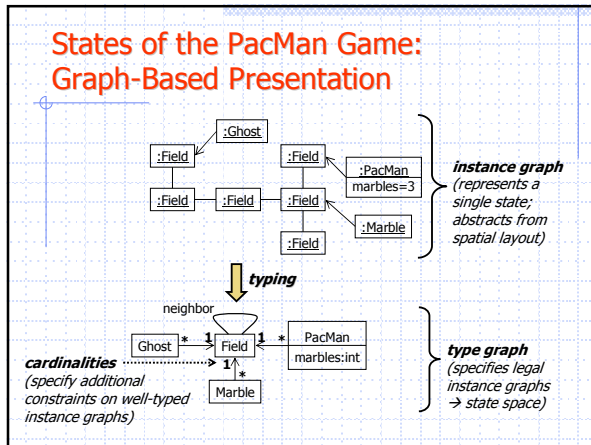
### Why it is fun: Programming By Example

StageCast ([www.stagecast.com](http://www.stagecast.com)): a visual programming environment for kids (from 8 years on), based on

- behavioral rules associated to graphical objects
- visual pattern matching
- simple control structures (priorities, sequence, choice, ...)
- external keyboard control

→ intuitive rule-based behavior modelling

**Next:** abstract from concrete visual presentation



## Foundations of Graph Transformation

How it works.

## Outline

- ✖ A Basic Formalism
  - Light-weight presentation of a categorical approach.
- ✖ Variations and Extensions
  - Syntactic and semantic alternatives, and advanced features.
- ✖ Where it Comes From
  - Roots and inspiration

## How it works: Typed Graphs

Directed graphs as algebraic structures  $G = (V, E, src, tar)$  with  $src, tar: E \rightarrow V$

Graph homomorphism as pair of mappings  $h = (h_V, h_E): G_1 \rightarrow G_2$  with

- $h_V: V_1 \rightarrow V_2$
- $h_E: E_1 \rightarrow E_2$

preserving  $src$  and  $tar$

Typed graphs given by

- fixed type graph  $TG$
- instance graphs  $G$  typed over  $TG$  by homomorphism  $g: G \rightarrow TG$

## Rules

$p: L \rightarrow R$  with  $L \cap R$  well-defined, in different presentations

- like above (cf. PacMan example)
- with  $L \cap R$  explicit [DPO]:  $L \leftarrow K \rightarrow R$

## Rules

$p: L \rightarrow R$  with  $L \cap R$  well-defined, in different presentations

- like above (cf. PacMan example)
- with  $L \cap R$  explicit [DPO]:  $L \leftarrow K \rightarrow R$
- with  $L, R$  integrated [UML]:  $L \cup R$  and marking
  - $L - R$  as {destroyed}
  - $R - L$  as {new}

## Transformation Step: Operational

1. select rule  $p: L \rightarrow R$ ; occurrence  $o_i: L \rightarrow G$
2. remove from  $G$  the occurrence of  $L \setminus R$
3. add to result a copy of  $R \setminus L$

### Semantic Questions: Dangling Edges

- \* conservative solution: application is forbidden
  - invertible transformations, no side-effects
- \* radical solution: *delete dangling edges*
  - more complex behavior, requires explicit control

### Semantic Questions: Conflicts

- \* conservative solution: application is forbidden
  - invertible transformations, no side-effects
- \* radical solution: *give priority to deletion*
  - more complex behavior, requires explicit control

### Transformation Step: Declaratively

Set-theoretic: Assume  $G$  and  $H$  with  $G \cap H$  well-defined.  
 Then,  $G \Rightarrow_{p(\alpha_L)} H$  iff there exists a homomorphism  $\alpha: L \cup R \rightarrow G \cup H$  such that

- $\alpha(L) \subseteq G$  and  $\alpha(R) \subseteq H$
- $\alpha(L \setminus R) = G \setminus H$  and  $\alpha(R \setminus L) = H \setminus G$  (on the underlying sets and functions)

Category-theoretic:  $G \Rightarrow_{p(\alpha_L)} H$  iff (1) and (2) are pushouts

→ conservative solution (DPO, Ehrig et al 73)

### Advanced Features

#### Dealing with unknown context

- set-nodes (multi-objects): match all nodes with the required connections
- explicit (negative) context conditions

(turns f1 into a trap by reversing all outgoing edges to Field vertices, but only if there is no Ghost)

#### Control

- priorities: *movePM* only if *collect* is not possible
- programmed transformation: IF NOT *collect* THEN *movePm*;

### Where it comes from ...

**Graph Transformation and Graph Grammars**

### Chomsky Grammars: Rewriting of Strings

Production  $A \rightarrow aAb$  as (context-free: one vertex or edge in  $L$ ) graphical production rule

- \* Theory of *graph grammars* as formal language theory for graphs
  - hierarchies of language classes and grammars
  - decidability and complexity results
  - parsing algorithms

### Petri Nets: Rewriting of Multisets

A PT net transition as graph transformation rule

- ✗ Theory of concurrency for graph transformation
  - independence, causality, and conflicts
  - processes, unfoldings
  - analysis techniques

### Term Rewriting: Rewriting of Trees or DAGs

TR Rule  $f(x) \rightarrow g(x, f(x))$  as DAG rewrite rule

- ✗ Theory of term graph rewriting
  - soundness and completeness of TGR w.r.t. TR
  - termination, critical pairs, and confluence

### Applications of Graph Transformation

What it is all good for (except video games).

### Where it comes from and what it is good for

### Applications of Graph Transformation

**Behaviour modelling:** conflicts and dependencies in functional requirements

**Model of computation:** the rules of service-oriented architectures

**Diagram languages:** the "complete" definition of visual languages

### Software Development as Integration of Views

1. Aspects of requirements models
2. Conflicts between functional requirements
3. From conflicts to (in)consistency

### 1. Aspects of Requirements Models

**Model A**

- Domain model: Agree on vocabulary first !  
→ class and object diagrams
- Business process model: Which actions are performed in which order ?  
→ use case description in natural language  
→ activity diagrams

**Model B**

### Structure: Class and Object Diagrams

✓ formal, e.g., attributed graphs at the type and instance level

✓ established techniques for view integration

### Behaviour: Use Case Description by Structured Text

✓ based on vocabulary of integrated domain model

✗ no way to tell if views are consistent

### Behaviour: Activity Diagrams

✓ identifies actions and their ordering

✗ no formal integration with structural model

✗ still no indication as to whether the views are consistent

### 1. Aspects of Requirements Models

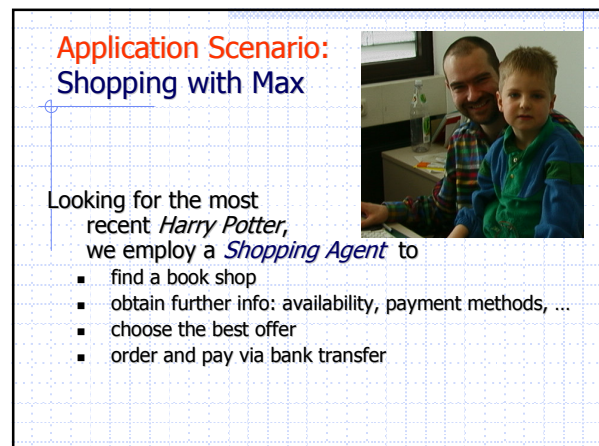
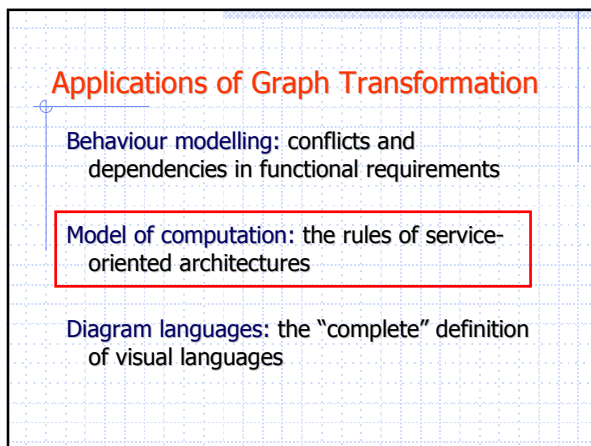
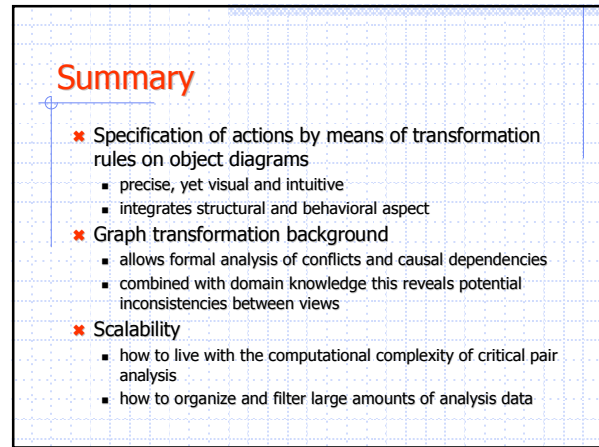
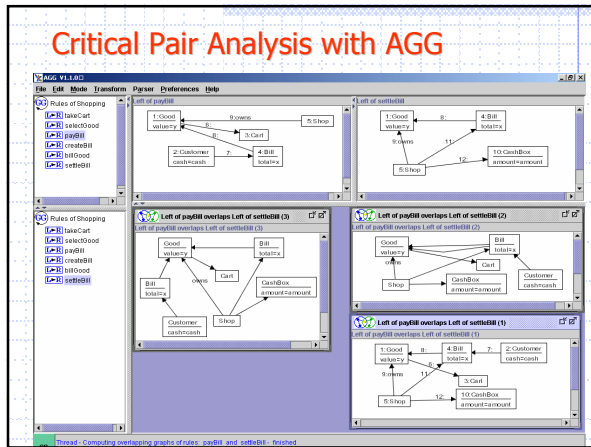
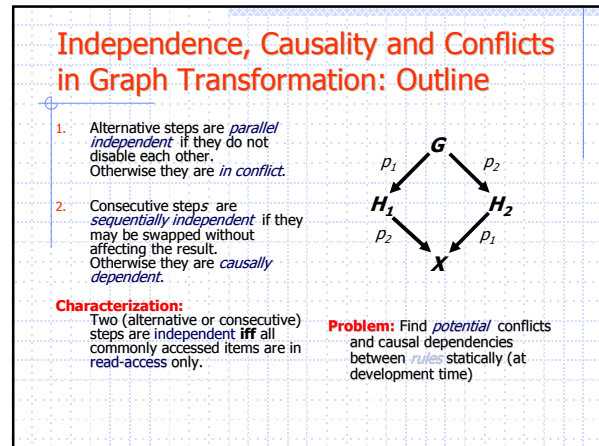
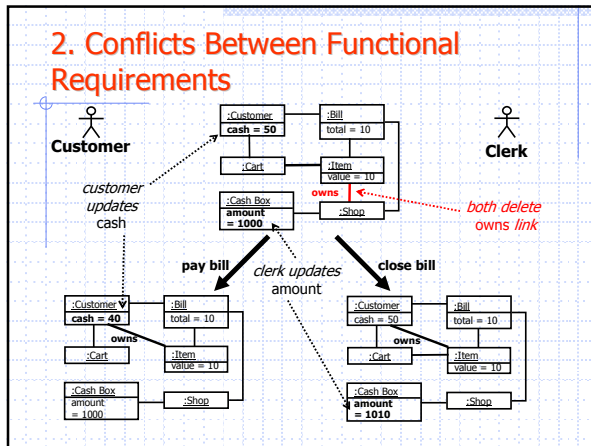
**Model A**

- Domain model: Agree on vocabulary first !  
→ class and object diagrams
- Business process model: Which actions are performed in which order ?  
→ use case description in natural language  
→ activity diagrams
- Functional model: What happens if an action is performed ?  
→ pre-/post conditions as logic constraints  
→ transformation rules on object diagrams  
(Fusion, Catalysis, Fajaba, formally: graph transformations)

**Model B**

### Function: Transformation Rules on Object Diagrams

**conflicting actions**



### As component-based system (CBS)

- 1: get product info
- 2: place order
- 3: pay bill
- 4: request transfer

But, ...

- how does *myAgent* know *Amazon.com*?
- why does *myAgent* not ask *Buecher.de*?

### Service-Oriented Architectures (SOA)

A Web service is a *component* deployed on a *Web accessible platform* provided by a *service provider* to be *discovered* and *invoked* over the Web by a *service requestor*

### SOA Scenario

- 1.1: publish(a)
- 1.2: {a} = find(r)
- 1.3: get product info

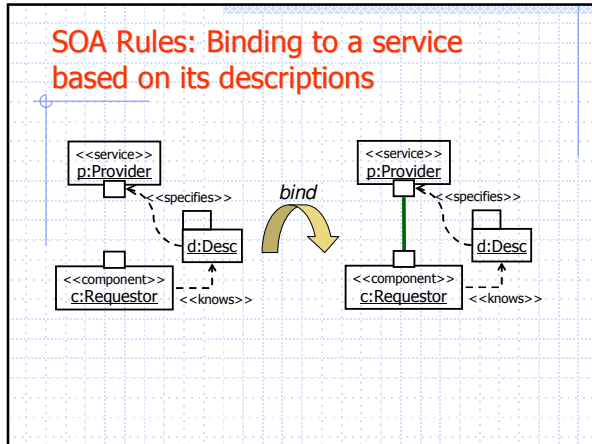
- how does *myAgent* know *Amazon.com*?
- why does *myAgent* not ask *Buecher.de*?

### Service-Oriented Architectures (SOA)

- What are the rules of the game?
- What can we do to reach configuration where X can talk to Y?

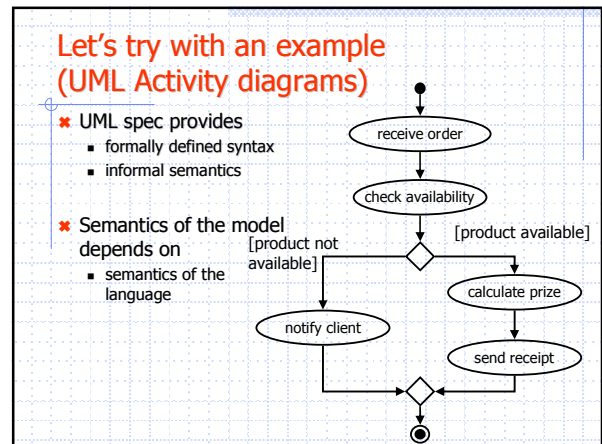
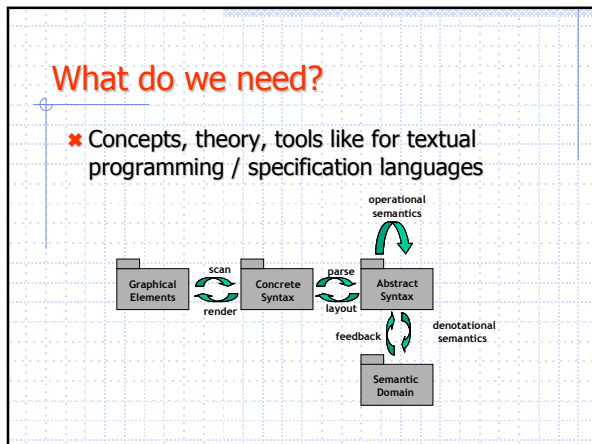
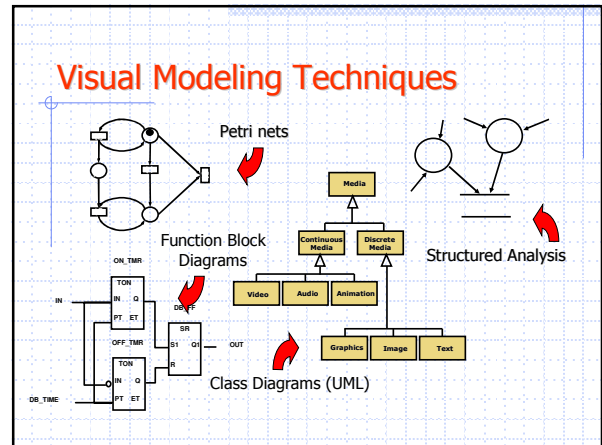
### SOA Rules: Publishing a service description at a registry

### SOA Rules: Discovering service descriptions satisfying requirements



- ### Service-Oriented Architectures (SOA)
- ✘ What are the rules of the game?
    - full spec of about 25 rules
    - based on a meta model structured into 3 packages
  - ✘ What can we do to reach configuration where X can talk to Y?
    - a reachability property ...

- ### Applications of Graph Transformation
- Behaviour modelling: conflicts and dependencies in functional requirements
  - Model of computation: the rules of service-oriented architectures
  - Diagram languages: the "complete" definition of visual languages



### Graphical elements (SVG – Scalable Vector Graphics)

```
<?xml version="1.0" encoding="utf-8" ?>
<ellipse cx="65" cy="24" rx="10.0" ry="10.0" style="fill:#000000; stroke:#000000; stroke-width:1"/>
<rect x="23" y="63" width="88" height="23" rx="16" ry="16" style="fill:none; stroke:#000000; stroke-width:1"/>
<text x="36" y="77" style="font-family:Dialog; font-size:10;">
receive order</text>
<rect x="15" y="111" width="99" height="23" rx="16" ry="16" style="fill:none; stroke:#000000; stroke-width:1"/>
<text x="23" y="125" style="font-family:Dialog; font-size:10;">
check availability</text>
<polyline style="fill:none; stroke:#000000; stroke-width:1" points="66,33 66,63"/>
<line x1="59" y1="50" x2="66" y2="62" style="fill:#000000; stroke:#000000; stroke-width:1"/>
<line x1="73" y1="50" x2="66" y2="62" style="fill:#000000; stroke:#000000; stroke-width:1"/>
<polyline style="fill:none; stroke:#000000; stroke-width:1" points="62,86 62,111"/>
...
</svg>
```

Graphical Elements

### Concrete syntax (Spatial Relationship Graph)

SVG

### How can we create this?

Typed graph (metamodel)

- ✳ Textual attributes
- ✳ Pairs of rules
- ✳ Shared metamodels

Concrete Syntax

### An example rule

```
<1: element> := <2: element>
<3: polyline>
<4: line>
<5: line>
<6: element>
```

Attributes

```
1.x1 = 2.x1
1.y1 = 2.y1
1.x2 = 6.x2
1.y2 = 6.y2
```

Conditions

```
3.points = 2.x2, 2.y2 6.x1, 6.y1
4.x1 = 6.x1-7 4.y1 = 6.y1-12 4.x2 = 6.x1 4.y2 = 6.y1
5.x1 = 6.x1+7 5.y1 = 6.y1-12 5.x2 = 6.x1 5.y2 = 6.y1
```

Scanning

Rendering

### Abstract syntax

Metamodel

Abstract Syntax

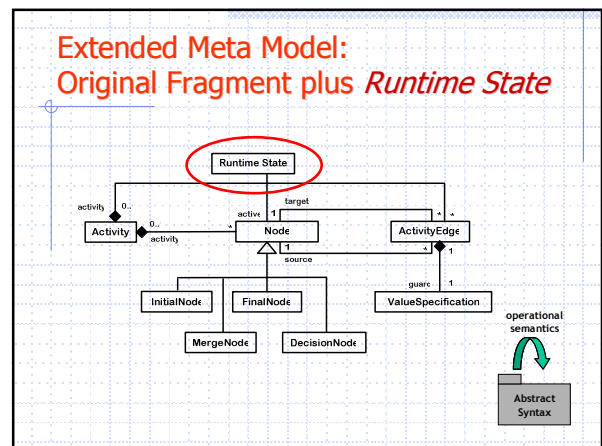
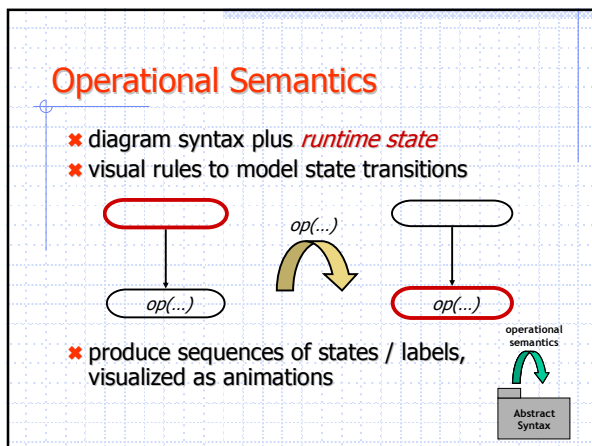
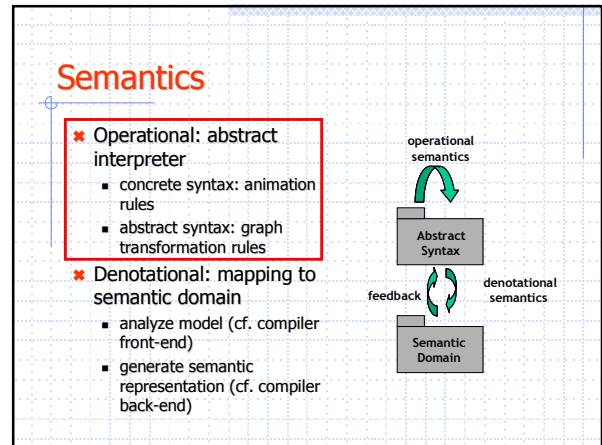
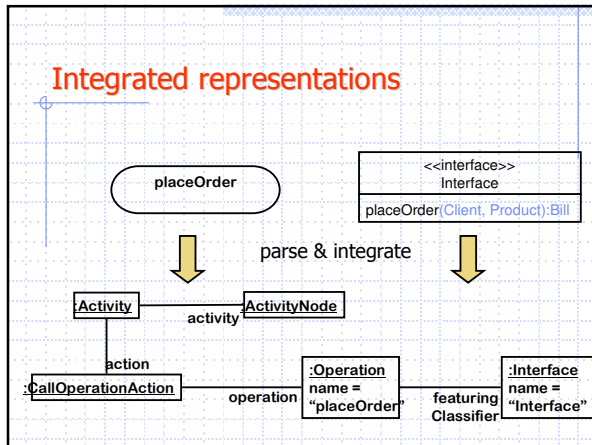
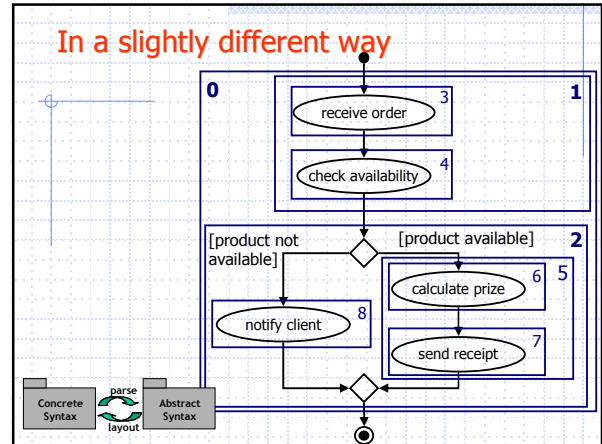
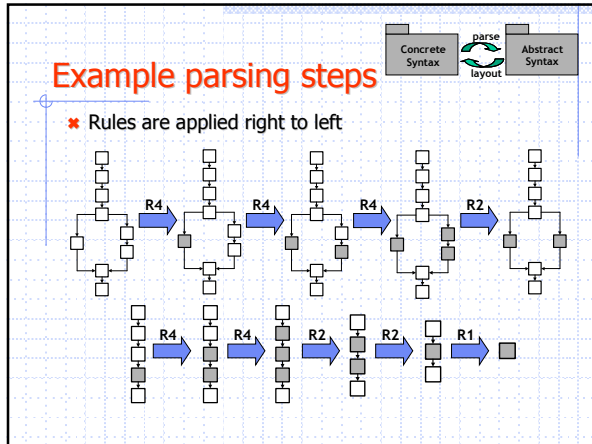
Abstract Syntax graph

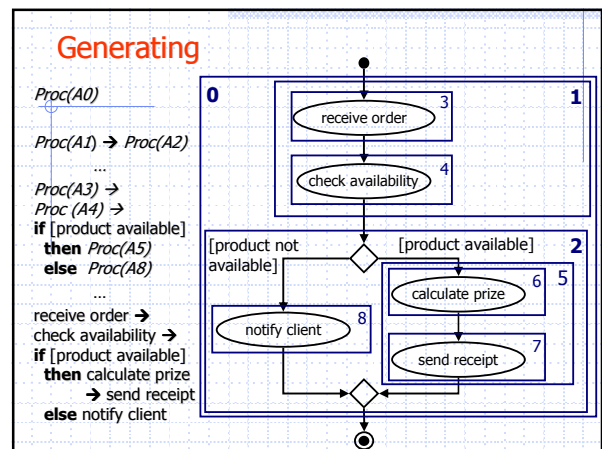
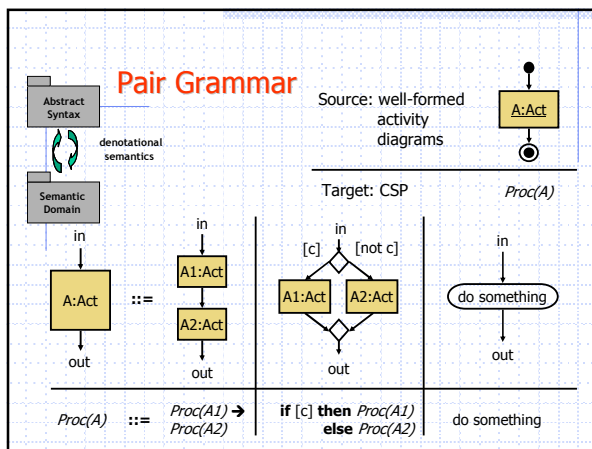
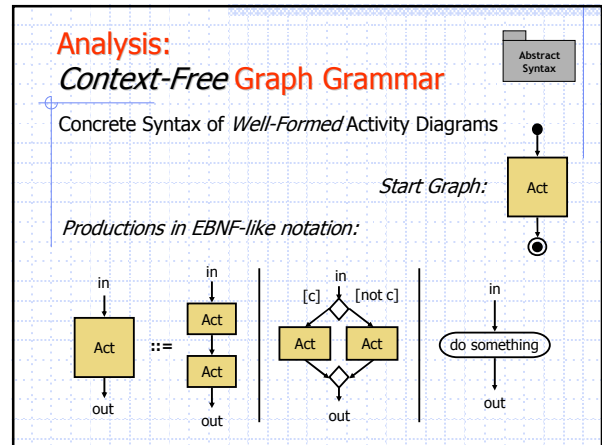
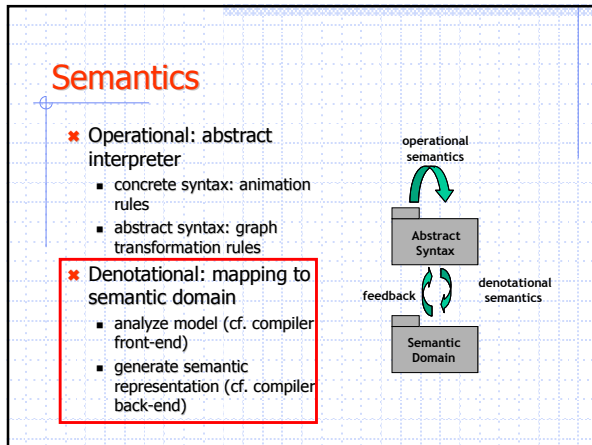
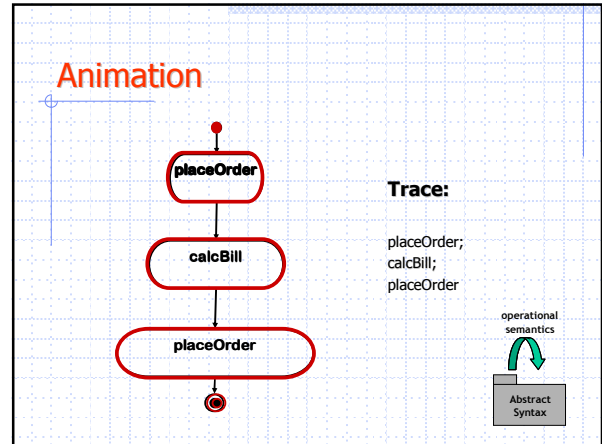
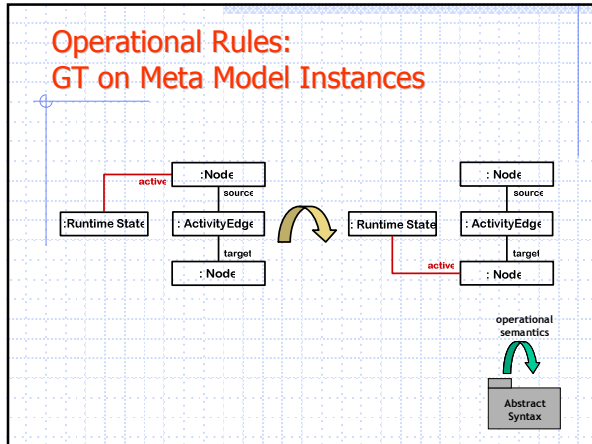
### Layout and parsing

Concrete Syntax

Abstract Syntax

This is only for building the abstract syntax representation





### Feedback

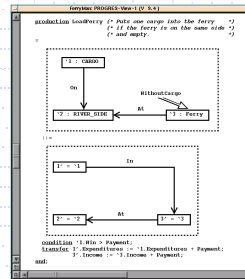
- ✦ We need to keep track of the mapping created to establish some reverse transformation.
- ✦ This means triple graph grammars (see slides at the School, [www.segravis.org/school](http://www.segravis.org/school)).

### Tool support

### Outline

- ✦ Two main groups:
  - General purpose modeling environments
    - PROGRES, AGG, Fujaba, ...
  - Environments for specifying visual notations
    - DIAGEN, GENGEed, MetaEnv, ConWork, ...
- ✦ Good prototype tools developed in academia

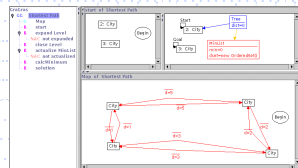
### PROGRES (PROgrammed Graph Rewriting Systems)



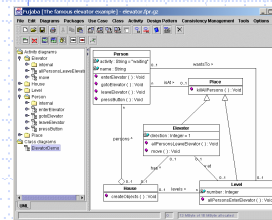
- ✦ Graphical/textual language to specify graph transformations
- ✦ Graph rewrite rules with complex and negative conditions
- ✦ Cross compilation in Modula 2, C and Java

### AGG (The Attributed Graph Grammar System)

- ✦ Algebraic approach to graph transformation
- ✦ Annotations are in Java
- ✦ Efficient graph parsing
  - Parse grammar
  - Critical pair analysis
- ✦ Easy integration with Java code

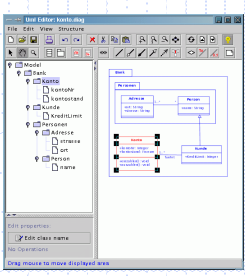


### Fujaba (From UML to Java and BACK)



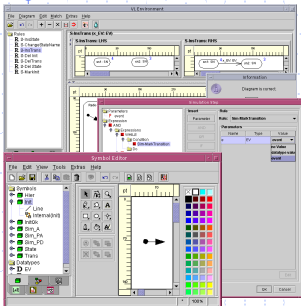
- ✦ Round trip engineering with UML, Java, and design patterns
- ✦ Class, collaboration and activity diagrams for story diagrams
  - Dynamic behavior
  - Automatic generation
- ✦ Reverse engineering

### DiaGen (The Diagram Editor Generator)



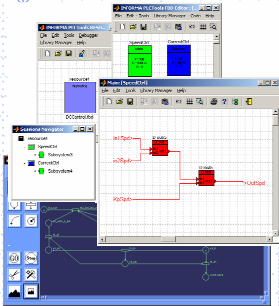
- \* Notations are specified through hypergraphs
- \* Framework of Java classes
  - to provide basic functionality
- \* Generator program
  - to produce Java source code

### GenGED (Generation of Graphical Env.s for Design)



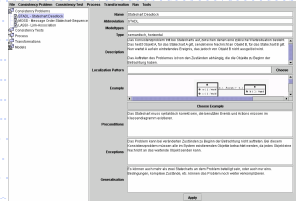
- \* Graphical editors and simulation environments
  - Syntax grammar
    - Actual syntax
  - Parse grammar
    - Free-hand editing
  - Simulation grammar
    - To simulate models
- \* AGG and graphical constraint solving techniques

### MetaEnv



- \* Customizable engine to map diagram notations onto high-level timed Petri nets
- \* Rules are pairs of graph grammars
- \* Results are mapped back onto the diagram model

### ConWork (Consistency Workbench)



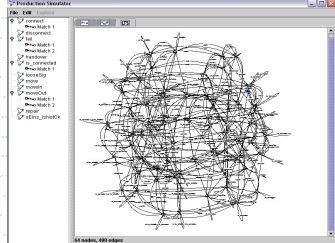
- \* GT to translate models into CSP
- \* rule-based generation of constraints
- \* visual definition of analysis process
- \* catalog of consistency problems
- \* Contact:
  - Jochen Kuester  
U. of Paderborn  
[jkuester@upb.de](mailto:jkuester@upb.de)

### Analysis

- \* CheckVML
  - Encodes graph transformation systems into SPIN to reason on the reachability of specific configurations by means of sequences of rules
- \* Groove
  - Verifies model transformation and dynamic semantics through an (automatic) analysis of the resulting graph transformation systems using model checking

### GRaphs for Object-Oriented VERification (GROOVE)

- \* generation of LTS from GT systems
  - edge-labelled graphs
  - application conditions
  - priorities



<http://wwwhome.cs.utwente.nl/~groove/groove-index>

## Conclusions

## Main results

- ✗ The tutorial has
  - Motivated the use of graph transformation in software engineering
  - Introduced the foundations of graph transformation
  - Shown example applications of graph transformation
    - GT as semantic domain for behavior modeling
    - GT as meta language for visual modeling techniques
  - Presented available tools
- ✗ Now, attendees are likely to be able to
  - Better understand the different proposals
  - Better evaluate if and how they can exploit it in their work

## Future work (Applications)

- ✗ GT should become more "usable" by non experts:
  - It should be better disseminated (This tutorial)
  - More examples and case studies to "convince" skeptical users
  - Further co-operations between GT experts and domain experts
  - More friendly tools (even if they are much better than a few years ago)

## Future work (Foundations)

- ✗ analysis and verification techniques
- ✗ refinement and modularity
- ✗ relation with other areas
  - process calculi (Milner, Montanari)
  - DNA computing (Rozenberg)
  - XML, Meta data, Semantic Web (Rising)

## Research Training Network *SegraVis*\* [10/02 – 9/06]

You want to learn more?

- ✗ Apply for a grant with one of 12 European partners in Belgium, Germany, Italy, NL, and UK (only citizens of EU and associated)
- ✗ Participate in our network events

For details, see [www.segravis.org](http://www.segravis.org) or contact Reiko Heckel

\* *Syntactic and Semantic Integration of Visual Modeling Techniques*


## A few basic references

- ✗ Handbook of Graph Grammars and Computing by Graph Transformation
  1. Foundations
  2. Applications, Languages and Tools
  3. Concurrency, Parallelism, and Distribution
- ✗ Graph Transformation for Specification and Programming
  - Andries, Engels, Habel, Hoffmann, Kreowski, Kuske, Plump, Schürr, Taentzer; Science of Computer Programming, Vol. 34, No. 1, April 1999, pp.1-54
- ✗ Tutorial Introduction to Graph Transformation: A Software Engineering Perspective
  - Baresi, Heckel; Proc. 1st Intl. Conference on Graph Transformation (ICGT 02), Barcelona, Spain, Springer LNCS 2505

### Web sites

- \* Home of the ICGT steering committee
  - www.gratra.org
- \* SegraVis home page
  - www.segravis.org
- \* Graph Grammar Bibliography
  - www.informatik.uni-bremen.de/theorie/appli-graph/bibliography.html
- \* AGG home page
  - tfs.cs.tu-berlin.de/agg/
- \* PROGRES home page
  - www-i3.informatik.rwth-aachen.de/research/projects/progres/
- \* DiaGen home page
  - www2.informatik.uni-erlangen.de/DiaGen/
- \* GenGED home page
  - tfs.cs.tu-berlin.de/~genged/

### Open discussion




### Our Addresses

- \* Luciano Baresi
  - Politecnico di Milano  
Dipartimento di Elettronica e Informazione  
Piazza L. da Vinci, 32 – I20133 Milano (Italy)  
baresil@elet.polimi.it
- \* Reiko Heckel
  - University of Paderborn  
Faculty of Computer Science Electrical Engineering and Mathematics  
D-33095 Paderborn (Germany)  
reiko@upb.de

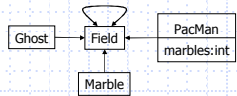
### Exercise 1: Be a (slightly) more clever player!

Extend the *movePM* rule so that *Pacman* does not move next to a *Ghost*.



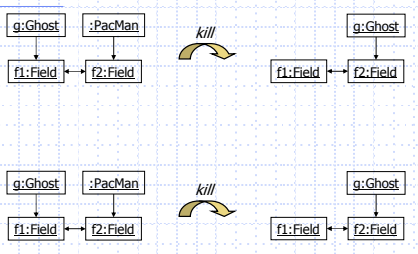
### Exercise 2: Give Pacman another chance

Let *Pacman* have a counter for his lives.



Refine the rule *kill* to remove *Pacman* only if he has run out of lives. Otherwise decrease the counter and remove the *Ghost*.

### Refine rule *kill*



**Solution 1:**  
**Be a (slightly) more clever player!**

Extend the *movePM* rule so that *Pacman* does not move next to a *Ghost*.

Solution: a negative application condition.

**Solution 2:**  
**Give Pacman another chance**

Let *Pacman* have a counter for his lives.

Solution: add an attribute.

Refine the rule *kill* to remove *Pacman* only if he has run out of lives. Otherwise decrease the counter and remove the *Ghost*.

**Refine rule *kill***

Solution: match attribute value.

Solution: an attribute application condition.