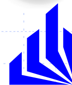
Foundations and Applications of Graph Transformation

An introduction from a software engineering perspective

Luciano Baresi
Politecnico di Milano

Reiko Heckel
University of Paderborn

Politecnico di Milano



Universität Paderborn

Motivation: Programming By Example

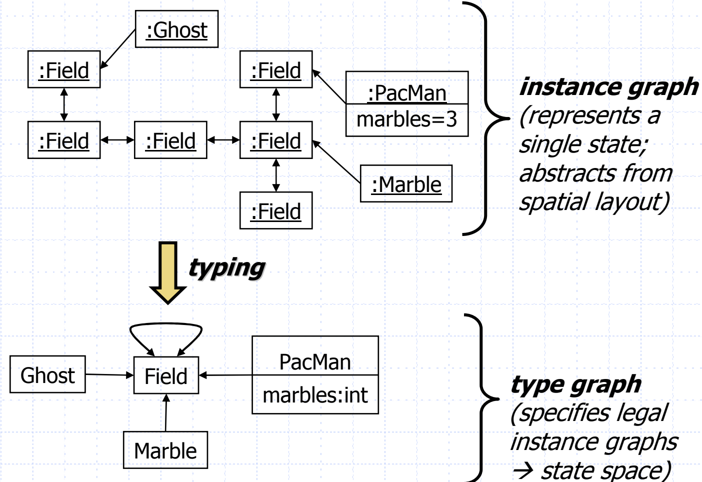
StageCast (www.stagecast.com): a visual programming environment for kids (from 8 years on), based on

- behavioural rules associated to graphical objects
- visual pattern matching
- simple internal control structures (priority, sequence, non-determinism, ...)
- external keybord control

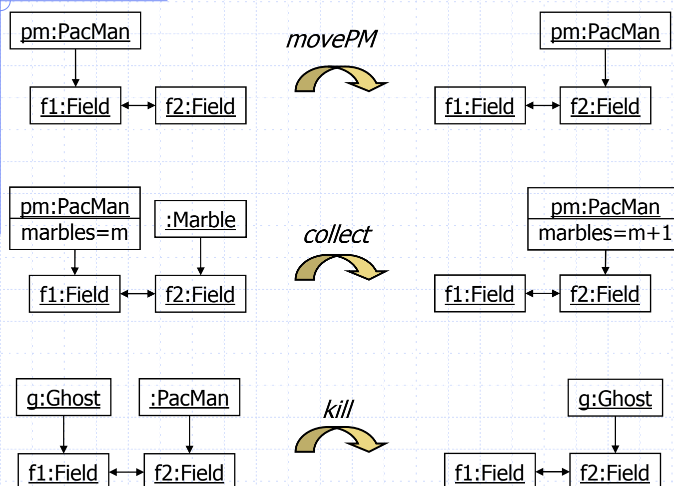
➔ Rule-based behaviour modelling is a natural and intuitive paradigm!

Example: A simple PacMan game; concrete (StageCast) vs. abstract (graph-based) presentation.

States of the PacMan Game: Graph-Based Presentation



Rules of the PacMan Game: Graph-Based Presentation



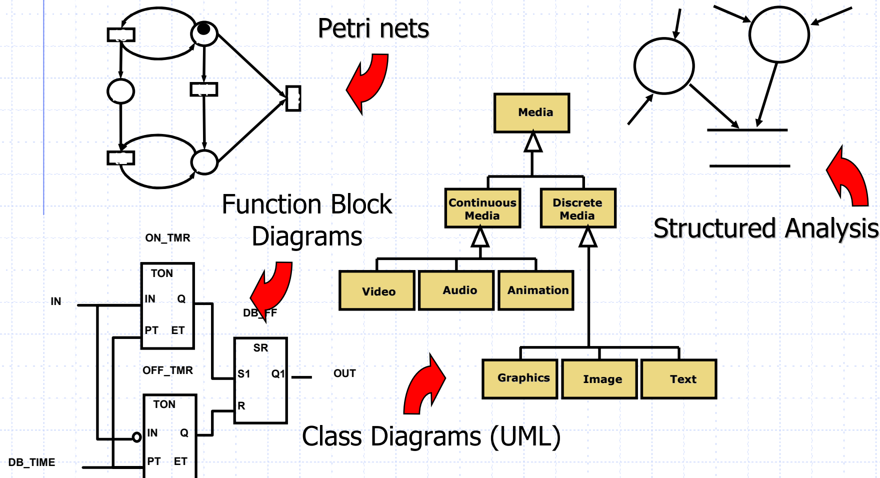
Outline

- Motivations
- Foundations
- Sample applications
- Tool support
- Conclusions

1- Motivations

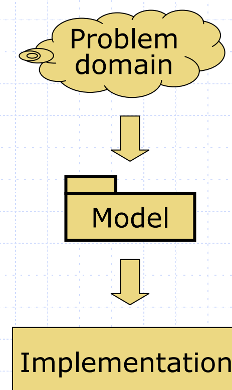
Why you are here.

Visual Modeling Techniques

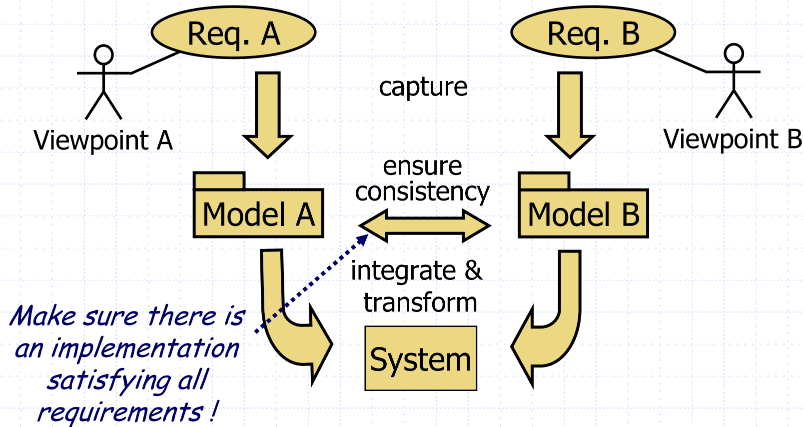


Separation of Concerns

- Complexity requires abstractions. Models allow to focus separately on:
  - System parts
    - class, component, subsystem
  - aspects
    - data, function, distribution, security
  - user views
    - clerk, customer, system administrator
  - abstraction levels
    - requirements, design, ...
    - Black- vs. white-box
- Development processes
  - human-oriented (→ visual)
  - incomplete and redundant

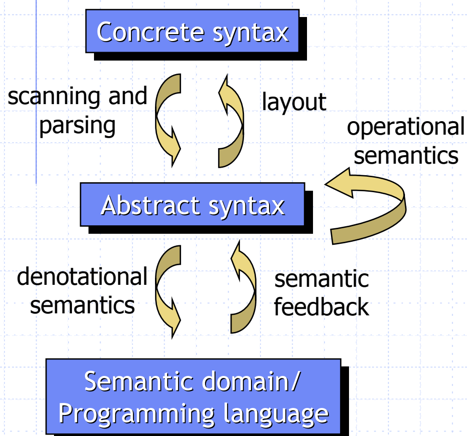


Integration and Consistency



Make sure there is an implementation satisfying all requirements !

What do we need?



- Concepts, theory and tools like for textual programming languages
  - syntax
  - semantics
    - denotational
    - operationa
  - transformation / refinement
  - verification
  - ...
- GT at two levels
  - rule-based behaviour modelling
  - meta modelling

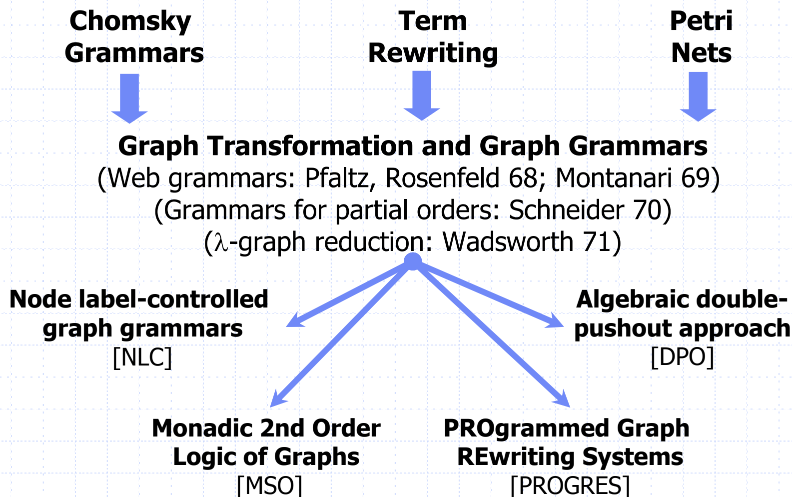
2- Foundations of Graph Transformation

How it works.

Outline

- Roots and Sources
  - Where it all came from and who invented it.
- A Basic Formalism
  - Light-weight presentation of a categorical approach.
- Variations and Extensions
  - Syntactic and semantic alternatives, and advanced features.
- Relation with Classic Rewriting Techniques
  - Inspiration for application and theory.

Roots and Sources



A Basic Approach: Typed Graphs

- Graphs as algebraic structures

  $G = (V, E, src, tar)$ with $src, tar: E \to V$

- Graph homomorphism as pair of mappings

  $h = (h_V : V_1 \to V_1, h_E : E_1 \to E_2) : G_1 \to G_2$
  preserving the graph structure

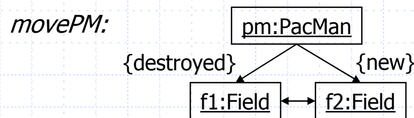
- Typed graphs (cf. PacMan example)
  - fixed type graph TG
  - instance graphs $(G, g : G \to TG)$ typed over TG
  - UML-like notation $x : t$ for x in G with $g(x) = t$

Rules

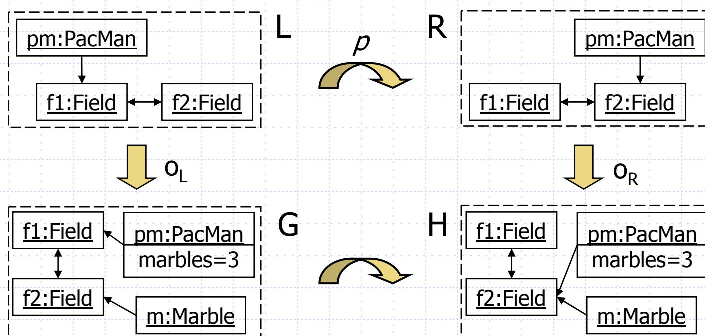
p: L → *R* with *L* ∩ *R* well-defined, in different presentations

- like above (cf. PacMan example)
- with *L* ∩ *R* explicit [DPO]: *L* ← *K* → *R*
- with *L*, *R* integrated [Fujaba]: *L* ∪ *R* and marking
  - *L* - *R* {destroyed}
  - *R* - *L* {new}

movePM:



Transformation: Operational Intuition



1. select rule $p : L \to R$; occurrence $o_L : L \to G$
2. remove from G the occurrence of $L \setminus R$
3. add to result an occurrence of $R \setminus L$

Transformation: Declarative Formalization

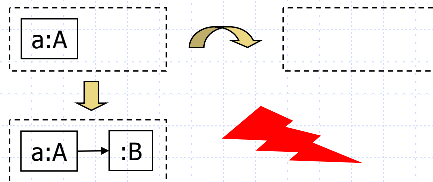
Transformation $G \rightarrow_{p(o)} H$ with $p: L \rightarrow R$

- occurrence $o: L \cup R \rightarrow G \cup H$
- $o(L \setminus R) = G \setminus H$ and $o(R \setminus L) = H \setminus G$

That is, a conservative approach:

- don't delete if this causes „dangling edges"
- → invertible transformations, no side-effects

E.g.: violation of dangling edge condition [DPO]



Variants and Extensions: Graphs

Graphs as relational structures: $G = (V, E)$ with $E \subseteq V \times V$

- no parallel edges; special case of algebraic variant

Undirected graphs

- directed graphs with symmetric edges

Hyper graphs: edges have lists of source (and target) vertices

- encoding as bipartite graphs

Labelled graphs: Vertices and edges labelled over an alphabet L:

- $G = (V, E, lv)$ with $E \subseteq V \times L \times V$; $lv : V \rightarrow L$ resp.
- $G = (V, E, src, tar, lv, le)$ with ... ; $lv : V \rightarrow L$; $le : E \rightarrow L$

Attributed grahps: labelled over an abstract data type, e.g.

type level:

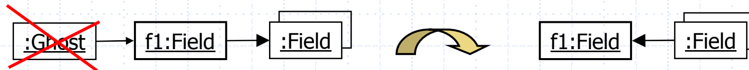
PacMan
marbles : int

instance level:

pm:PacMan
marbles = 3

Variants and Extensions: Rules and Transformations

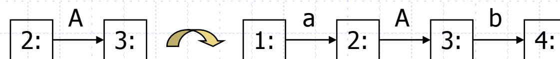
- context-free rules: one vertice or edge in *L*
- dealing with unknown context
  - node-label controlled embedding and set-nodes [NLC, PROGRES]
  - explicit (negative) context conditions



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

Chomsky Grammars: Rewriting of Strings

Production **A → aAb** as (context-free) graph grammar production

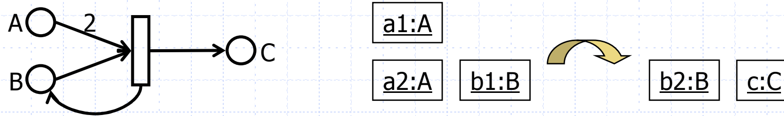


Theory of graph grammars as formal language theory for more-dimensional structures

- hierarchies of language classes and grammars
- decidability and complexity results
- parsing algorithms
- L-system-like parallel graph grammars

Petri Nets: Rewriting of Multisets

A PT net transition as graph transformation rule

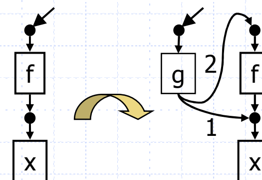


Theory of concurrency of graph transformation

- independence, causality, and conflicts
- concurrent *shift* - equivalence of transformations sequences
- processes and unfoldings
- event structure semantics

Term Rewriting: Rewriting of Trees or DAGs

TR Rule **f(x) → g(x, f(x))**
as DAG rewrite rule



Theory of term graph rewriting

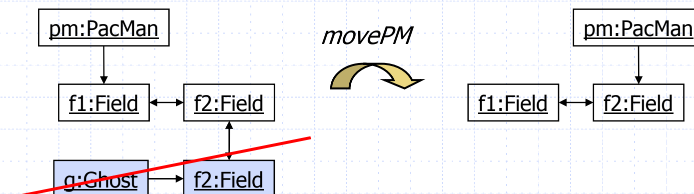
- soundness and completeness w.r.t. TR
- termination, confluence and critical pairs
- implementation of functional languages
- semantics of process (e.g., pi-, ambient-) calculi

Exercise 1

Improving Pacman.

Be a (slightly) more clever player!

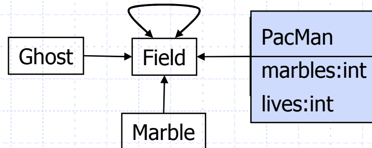
Extend the *movePM* rule so that *Pacman* does not move next to a *Ghost.*



Solution: a negative application condition.

Give *Pacman* another chance

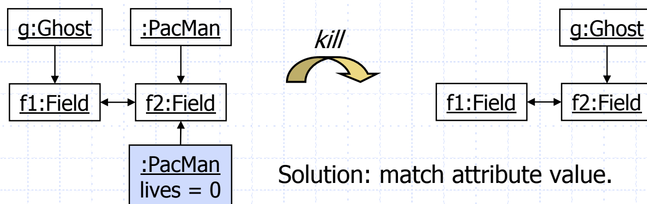
Let *Pacman* have a counter for his lives.



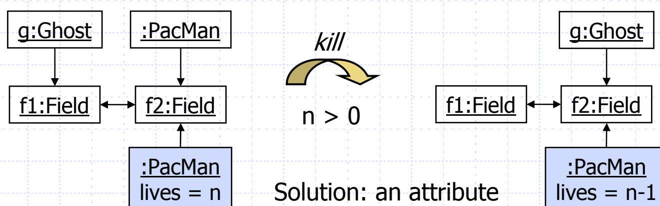
Solution:
add an attribute.

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.

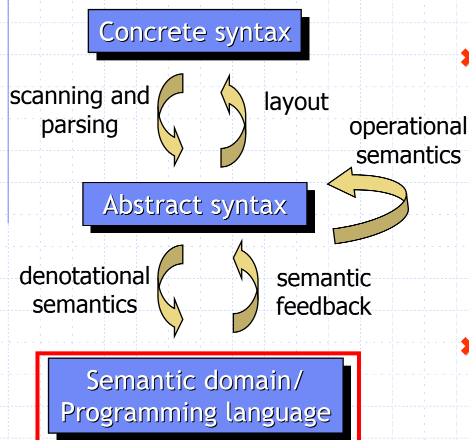


Solution: an attribute application condition.

3- Applications of Graph Transformation

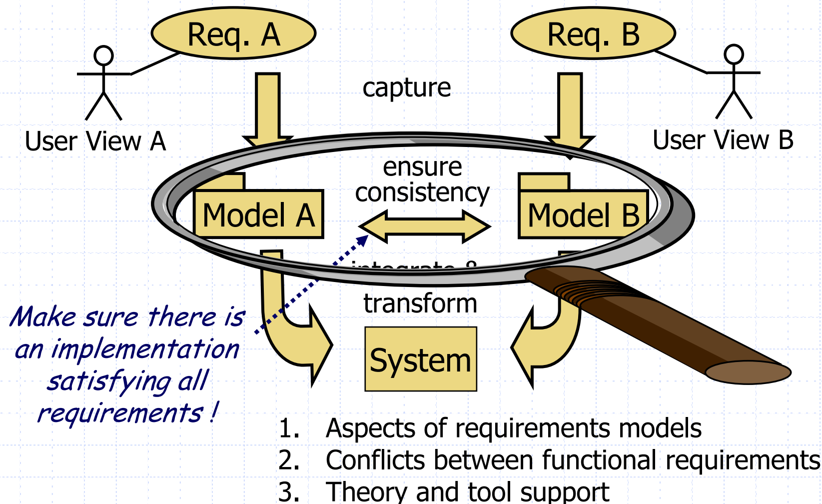
What it is all good for
(except video games).

Outline



- As semantic domain for behavior modeling
  - capturing and analyzing functional requirements
  - reconfiguration, mobility, and evolution of soft- and hardware architectures
- As a meta language for visual modeling techniques

Motivation: Software Development as Integration of Views



1. Aspects of requirements models
2. Conflicts between functional requirements
3. Theory and tool support

Aspects of Requirements Models

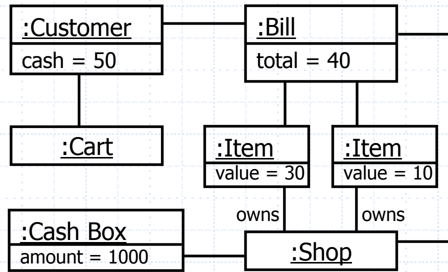
Model A

Model B

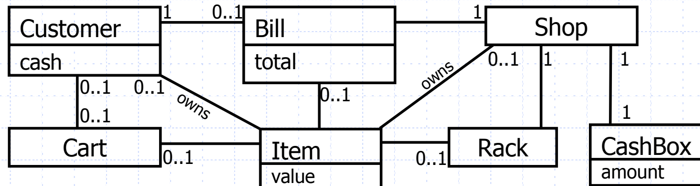
1. Static domain model: Agree on vocabulary first !
   → class and object diagrams
2. Business process model: Which actions are performed in which order ?
   → use case description in natural language, activity or sequence diagrams, etc.

Structure: Class and Object Diagrams

- ✓ formal, e.g., attributed graphs at the type and instance level
- ✓ established techniques for view integration

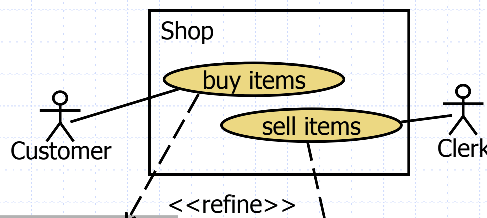


typing



Behaviour: Use Case Description by Structured Text

- ✓ based on vocabulary of integrated domain model



- take shopping cart
- select items from rack
- take items out of cart
- pay required amount
- collect items

- create empty bill for new customer
- take items out of customer's cart
- add them to the bill
- collect payment
- pack and give items to customer

- ✗ no way to tell if views are consistent

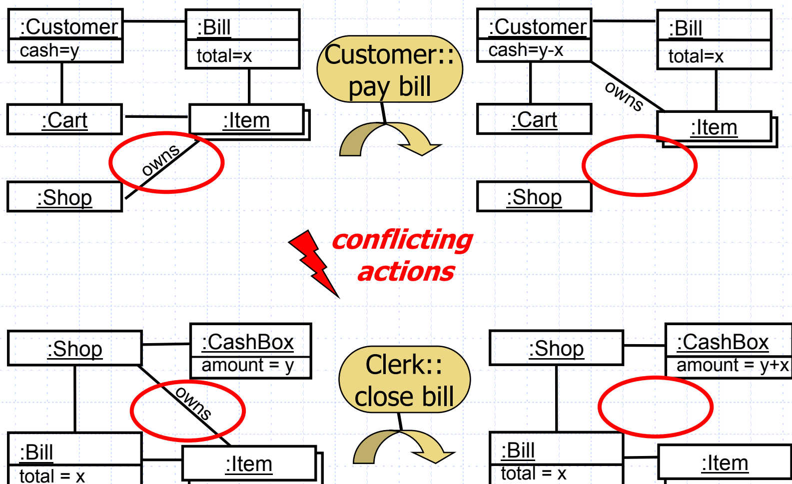
Aspects of Requirements Models

Model A

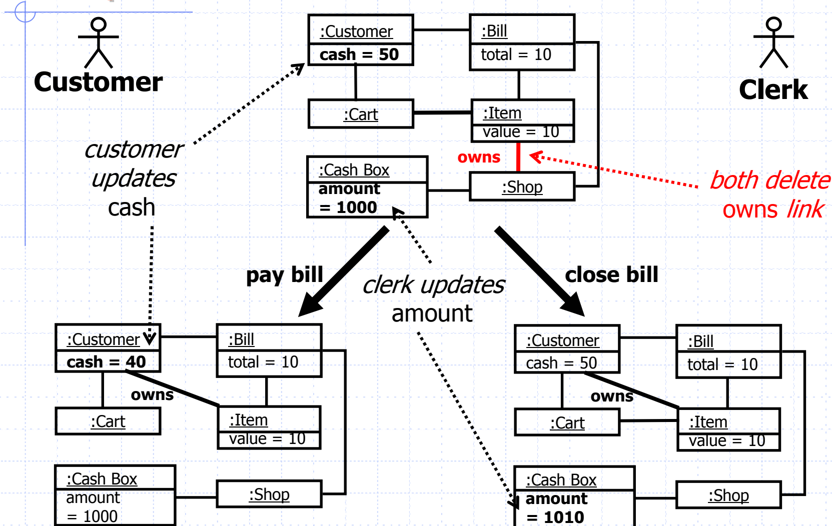
Model B

- ✓ Static 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 or sequence diagrams, etc.
3. Functional model: What happens if an action is performed ?
  → pre-/post conditions as logic constraints
  → transformation rules on object diagrams (Fusion, Catalysis, Fujaba, formally: graph transformations)

Function: Transformation Rules on Object Diagrams



Conflicts Between Functional Requirements



Theory: Independence, Causality and Conflicts in Graph Transformation

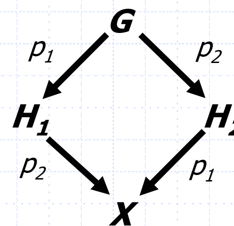
- Alternative steps are *parallel independent* if they do not disable each other.

  Otherwise they are *in conflict*.

- Consecutive steps are *sequentially independent* if they may be swapped without affecting the result.

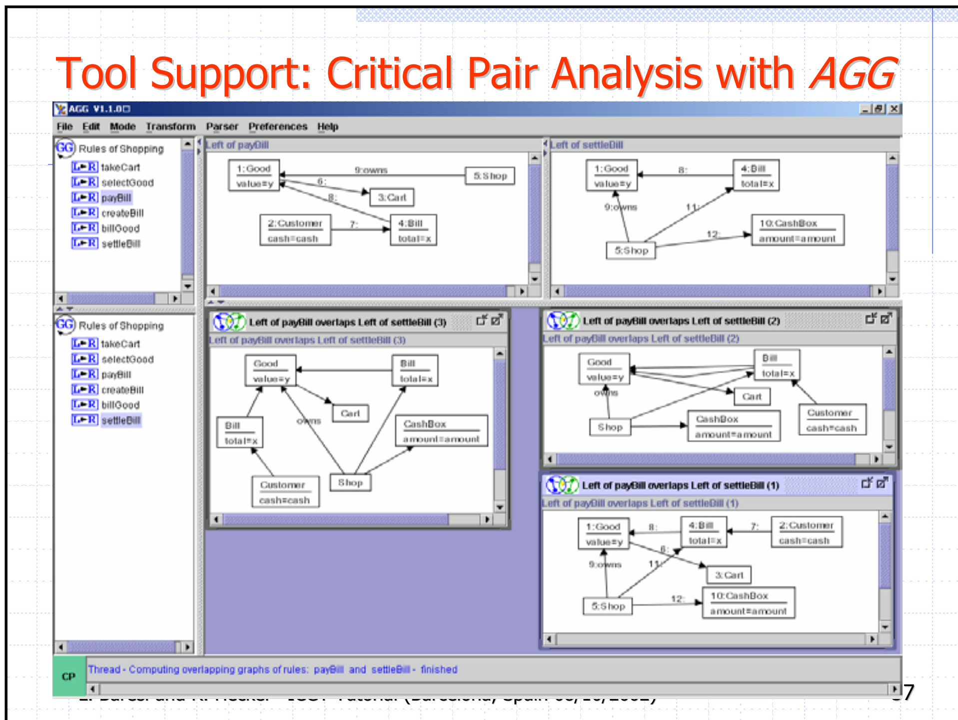
  Otherwise they are *causally dependent.*

Idea: Find *potential* conflicts and causal dependencies between rules by **critical pair analysis**

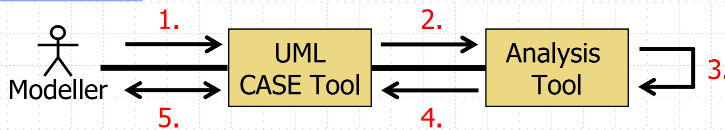


Characterization [EPS73]:
Two (alternative or consecutive) steps are independent **iff** all commonly accessed items are in read-access only.

Tool Support: Critical Pair Analysis with *AGG*



Usage Scenario

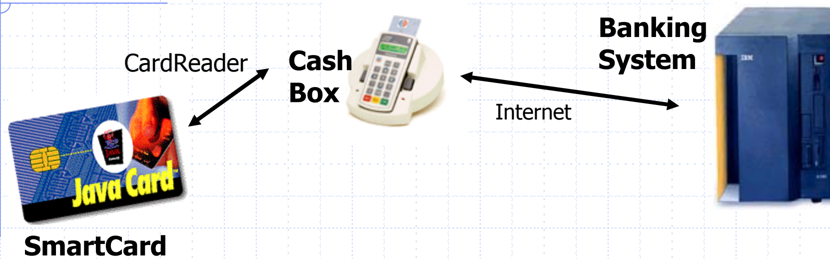


1. input model to CASE tool
2. import model by analysis tool
3. analyze model for conflicts
4. back annotate models with conflicts
5. interprete and improve models

Domain expert: "*buy items and sell items should not be in conflict*"

Modeller: *"inconsistency between views"*

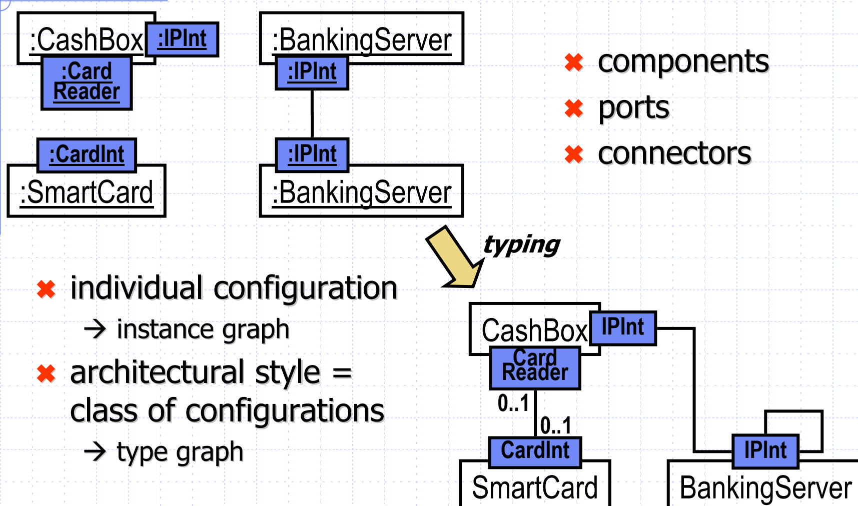
Another Interpretation: Graphs as Architectures



1. Type vs. instance level
2. Generation and reconfiguration
3. Integration with functional requirements
   - static and dynamic
   - consistency issues

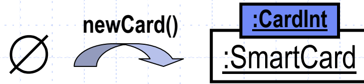
Architecture: Type and Instance Graphs

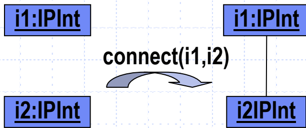
- components
- ports
- connectors

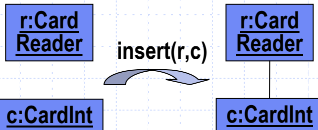


- individual configuration
  → instance graph
- architectural style = class of configurations
  → type graph

Reconfiguration: Typed Graph Transformation



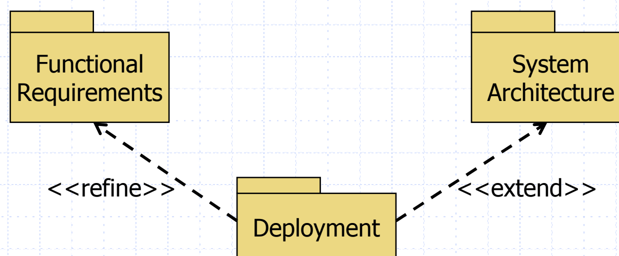




- Elementary operations
  - creating (destroying) new component instances
  - establishing (removing) connectors
- Complex reconfigurations
  - local (CF) rules with synchronization
  - global (non-CF) rules

Also: architectural style as graph grammar

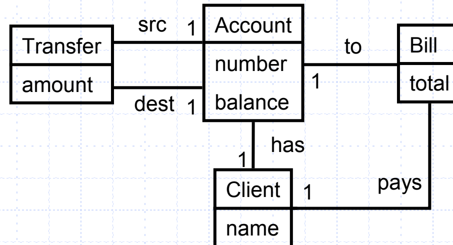
Development Problem: Implement functionality on a given architecture

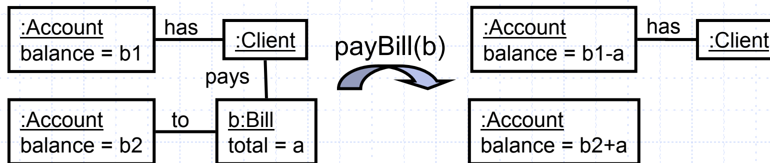


- static: deploy classes at components and objects at component instances
  → define a typed relation
- dynamic: distribute functionality
  → decompose global operations
  → add communication and reconfiguration

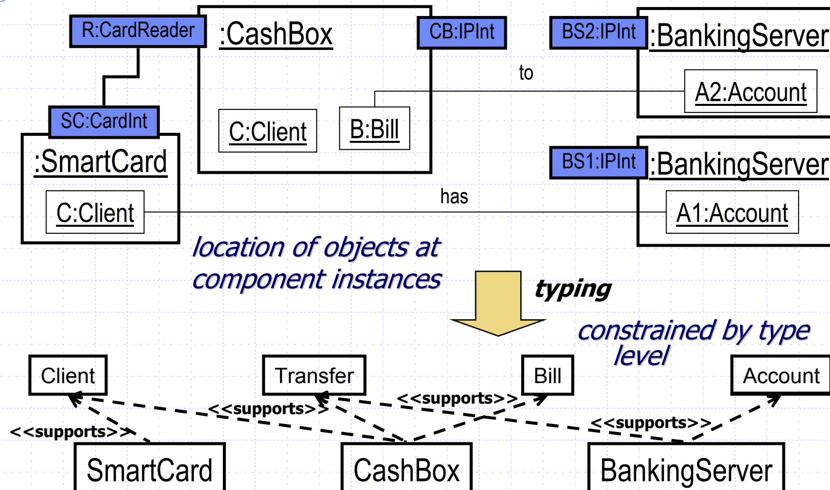
Functional Requirements

Payment of Bills
- static: type graph
- dynamic: typed graph transformation rule





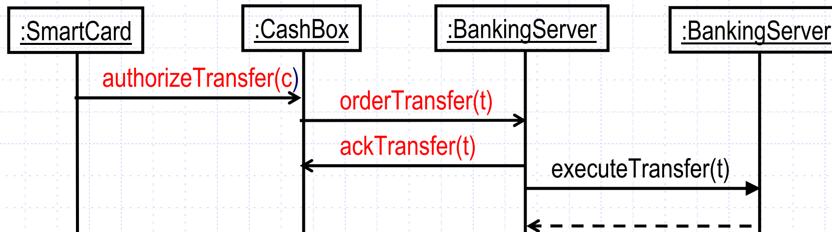
Static Integration: Instance and Type Level



Dynamic Integration: Three views

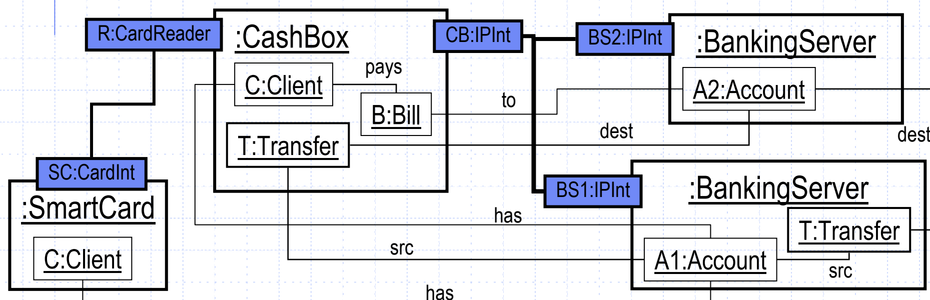
Rules for

- computation: what happens inside components
- reconfiguration: how the architecture is transformed
- interaction: how components communicate

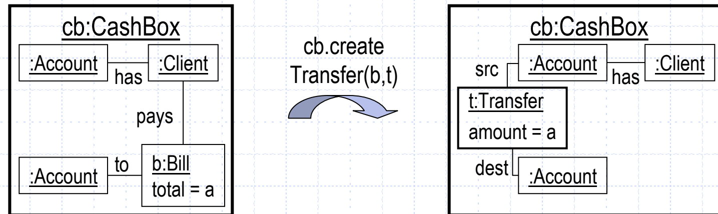


Implementing payBill(B)

1. insert(R,SC); authorize(C); eject(R,SC);
2. createTransfer(B,T);
3. connect(CB,BS1); orderTransfer(T); ackTransfer(T); disconnect(CB,BS1);
4. connect(BS1,BS2); executeTransfer(T); disconnect(BS1,BS2)

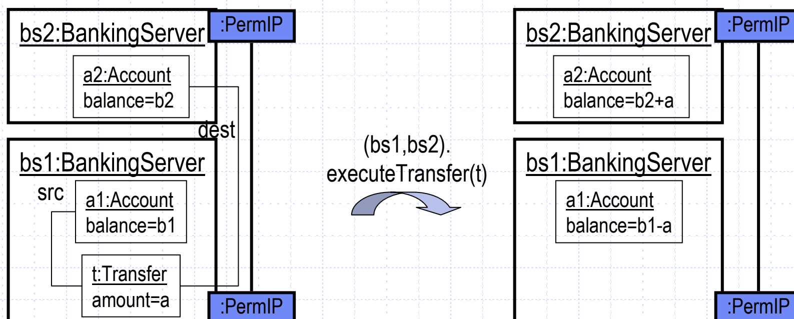


Computation rule: createTransfer



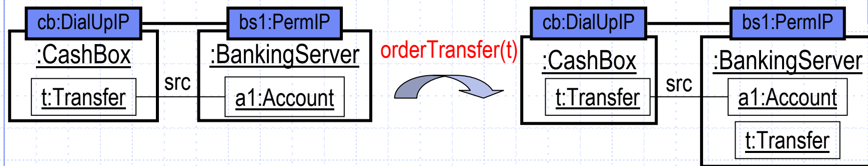
- local to *CashBox* instance

Computation Rule: executeTransfer



- non-local operation
- synchronized between two *BankingServer* instances

Communication Rule: orderTransfer



- shows effect of communication
  - transmission of *Transfer* instance
- abstracts from communication protocol
  - sending and reception of messages

Consistency Problem: Partial Correctness ?

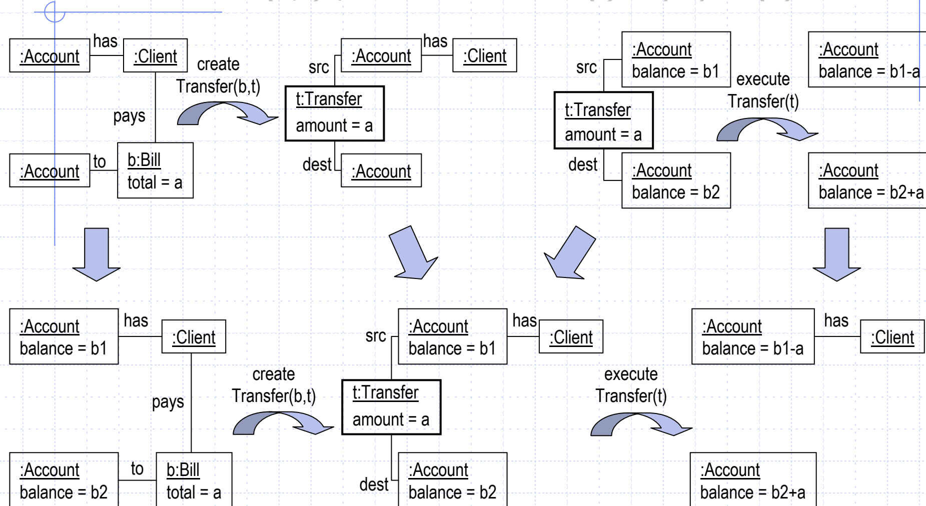
Do executions of the rule sequence implement the requirements expressed by the global rule ?

1. Project sequence to the functional view
   - hide all communication and reconfiguration rules
   - remove all components, connectors, and ports
   - identify shared objects
2. Minimize the resulting sequence
   - clip off unnecessary context
   - skip idle steps
3. Compare reduced sequence s to the original rule r

Thm [embedding]: If *r* can be embedded into (is equal to) the *derived rule* of the sequence *s*, each execution of *s* implements at least (exactly) the effects specified by *r*.

Example: Reduced Sequence Equals Global Rule

createTransfer(b,t) ; executeTransfer(t) = payBill(b)



Summary

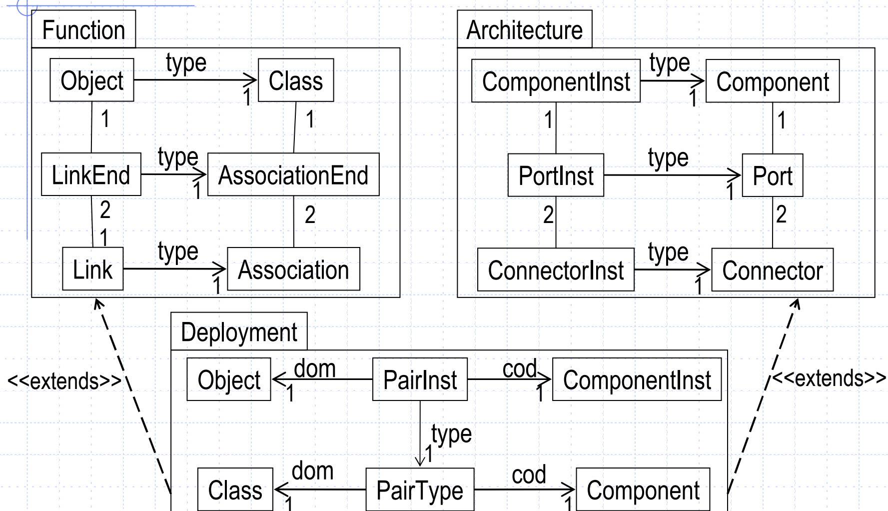
- Specification of changes by means of graph transformation rules on object structures, architectures, ...
  - formal, yet visual and intuitive
  - integrates structural and behavioral aspect
- Relevant graph transformation theory
  - independence and local Church-Rosser; critical pair analysis: detect potential conflicts between views
  - embedding of transformation sequences: consistency of implementation and requirements

  → *theory of concurrency and rewriting*

Exercise 2

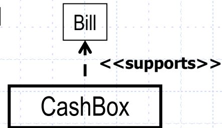
Meta modelling.

One formalism – two interpretations: Integration at the Meta Level

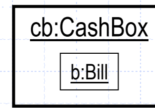


Represent as meta model instance!

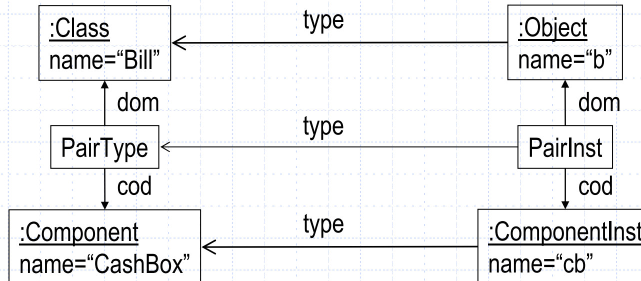
type level diagram:



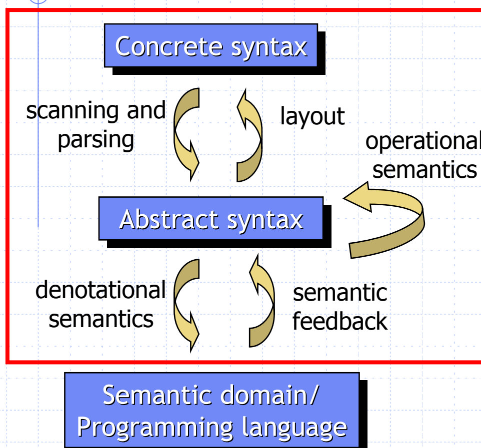
instance level diagram:



Hint: Assume that every meta type has a *name* attribute.



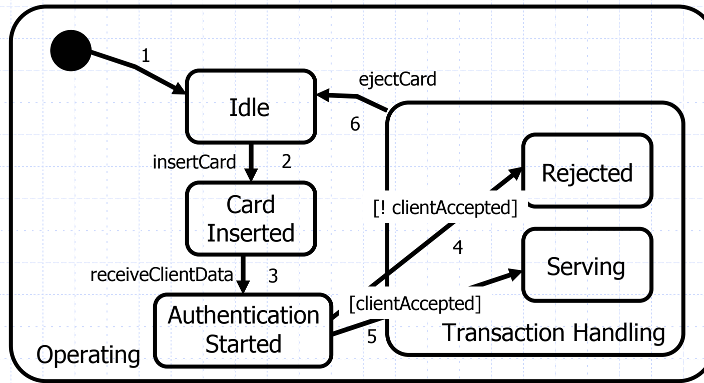
Meta language



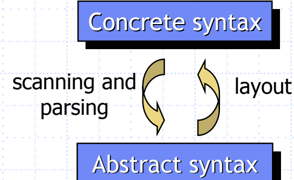
- as semantic domain for behaviour modelling
- as a meta language for visual modelling techniques
  - syntax definitions:
    - editors, parsers, ...
  - semantics definitions:
    - interpreters, compilers,
  - syntactic and semantic integration of VMTs
    - CASE tools

Running example

Cashbox



Syntax



- Concrete syntax
  - Describes how graphical elements are represented and their actual shapes
    - Open, closed lines, and labels are specialized in kinds of lines and closed shapes
    - Concepts are modeled through spatial relationship graphs
- Abstract syntax
  - Describes the sentence by the structure it has according to the language and fully abstracts away from the visual representation
    - Sentences (diagrams/models/graphs) are defined by composing shapes (concrete elements)
    - Sentences are rendered as abstract syntax graphs (close to OMG metamodels)

Scanning and parsing

Concrete syntax

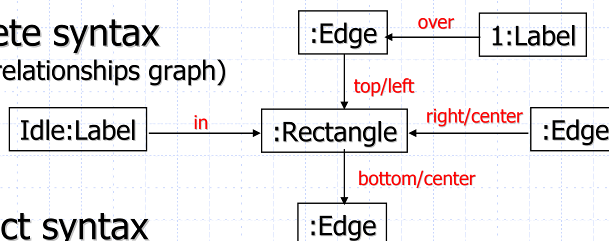
scanning and parsing

Abstract syntax

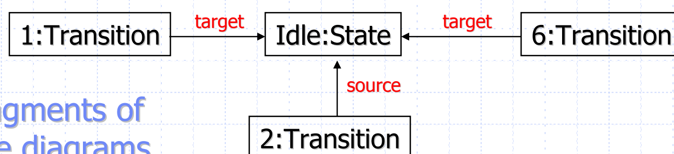
- User models are scanned to build spatial relationship graphs and abstract syntax graphs
  - The grammar predicates in terms of lines, bubbles, labels, etc
- Abstract syntax graphs can be parsed to validate that they represent valid elements of the language
  - The grammar predicates in terms of language's tokens
  - It could be used to define allowed steps in syntax-directed editors

Example
(Cashbox)

- Concrete syntax
(Spatial relationships graph)



- Abstract syntax
(Abstract syntax graph)



Only fragments of complete diagrams

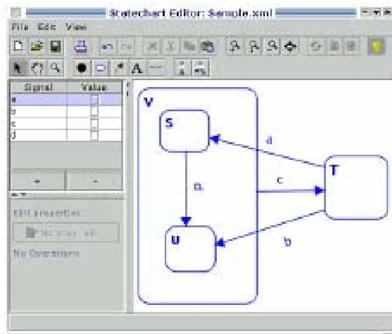
Layout

Concrete syntax

Abstract syntax

- Abstract syntax graphs can be used also to define automatic layouts through special-purpose grammars that embody layout algorithms
  - Textual attributes can be used to compute correct positions for concrete shapes
  - Almost all algorithms employ general purpose rules. Subtle positioning cannot be rendered
- Example (toy)
  - The first edge always leaves for the center of the left side
  - The second edge from the center of the top side
  - ...

Example
(DiaGen)



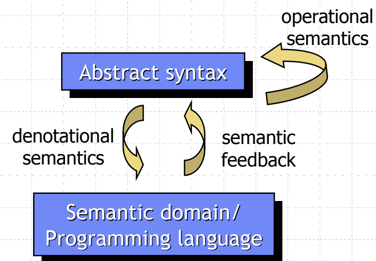
From specifications of

- abstract and concrete syntax
  - by graph grammar rules
- free-hand and syntax-directed editing operations
  - by editing rules
- operational semantics
  - by animation rules

DIAGEN generates standalone editors as Java classes

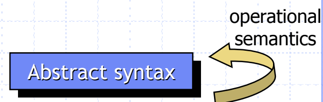
Live demo

Semantics



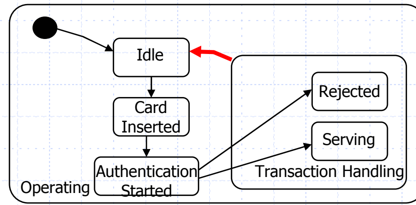
- It defines the meaning of language sentences
- Semantics can be given in two different ways
  - Operational semantics
    - Directly on the abstract syntax graph through another grammar
  - Denotational semantics
    - Through a mapping from the abstract syntax to an external semantic domain
    - In this case the role played by the grammar depends on the chosen domain
- It depends on what we want to communicate with the language

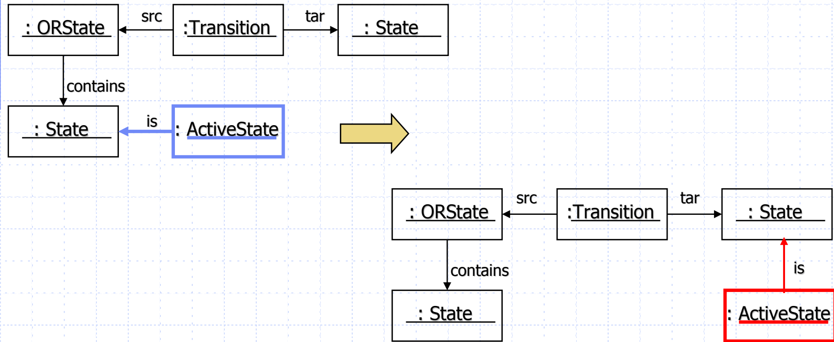
Operational semantics



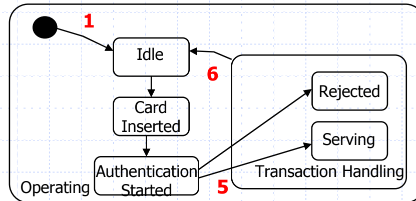
- Mainly two different ways
  - The grammar transformation rules can specify an abstract interpreter for the language
    - The interpreter is for the notation and can be applied on all correct models
  - Each model can be "compiled" into a set of rules
    - The set of rules is specific to the particular model
    - Theoretically, each model generates a different set of rules

Abstract Interpreter

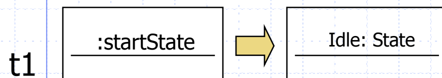




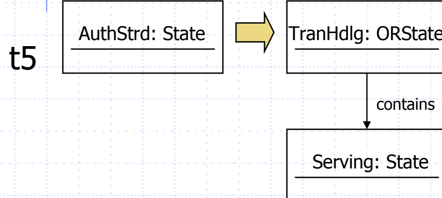
Ad hoc rules



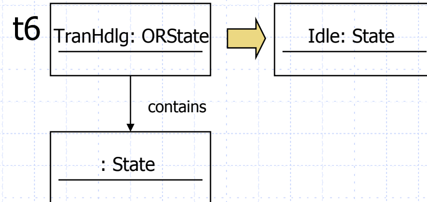
t1



t5

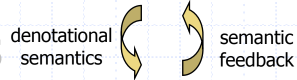


t6



- Transitions as rules

Denotational semantics



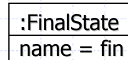
Semantic domain/ Programming language

- The identification of the right semantic domains impacts what sentences can be represented
  - It is extremely complex for practical modeling languages
    - Usually it is defined only informally
    - Or it captures only certain crucial aspects
- The semantic domain itself needs
  - a syntactic representation to “denote” the meaning of models
  - Tool support for analysis and reasoning
- Good examples are formal methods, like algebraic specification, Petri nets, CSP, Z, Alloy

Statecharts to CSP

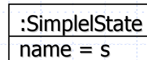
Rules for state decomposition and behavior

1



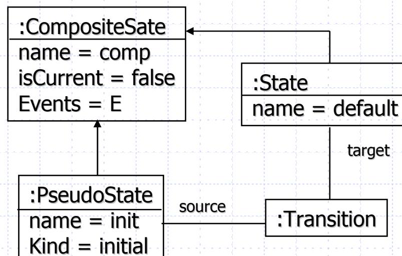

```
State(fin) =
  stop
```

2




```
State(s) =
  beh(s)
```

3



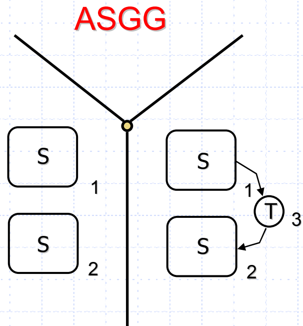

```
State(comp) =
  State(default)
```

....

Statecharts to Petri nets

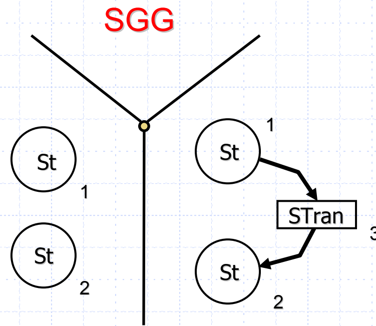
Rules are pairs of productions

ASGG



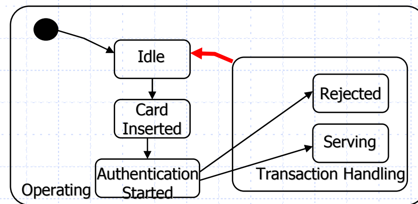

```
3.id =
3.name =
3.type = "Transition"
3.event =
3.action =
```

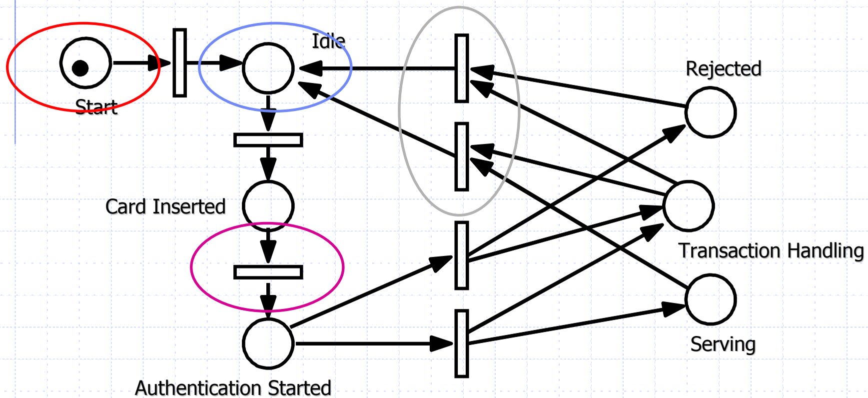
SGG



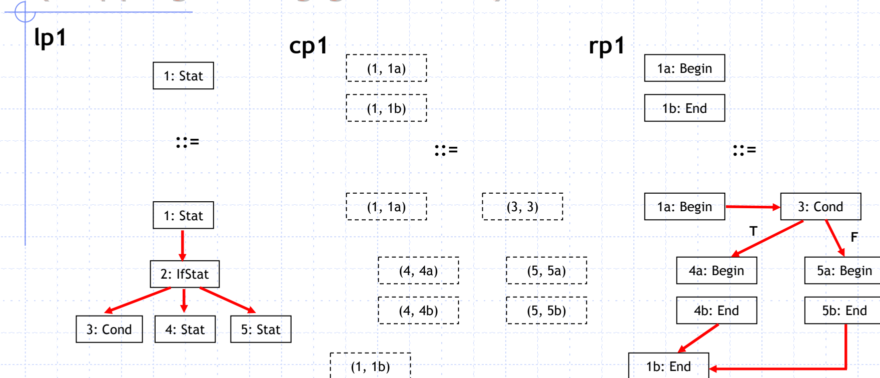

```
3.name =@3.name@
3.type = "STran"
3.predicate = @3.event@
3.action = @3.action@
```


A possible net





Integration
(Mapping among grammars)

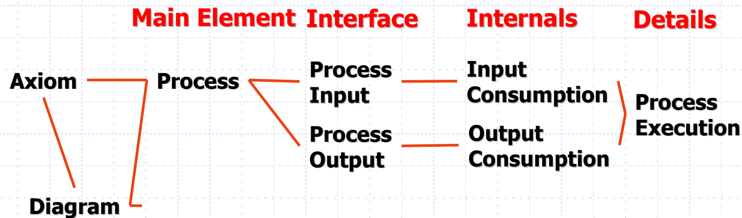


The triple GG approach

Semantic feedback

- One-way mapping
  - The formal model is visible to users
  - The visual notation is supplemented with the semantic domain
- Two-way mapping
  - The formal model remains hidden to users
  - The formal method is used to interpret visual sentences
  - Feedback on the formal model must be mapped back onto the visual model
    - Ad-hoc graph grammars
    - Simple textual rules

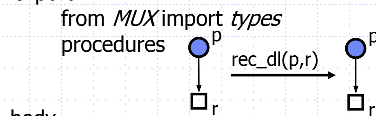
Separation of concerns at the meta level



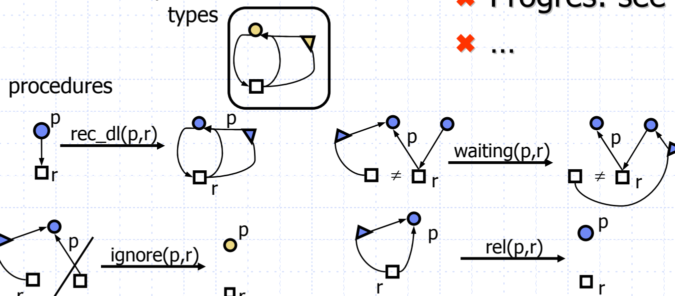
Modularity in Graph Transformation

DR =
export
from *MUX* import *types*
procedures p rec_dl(p,r) r

body
from *MUX* import *rel(p,r)*
implement
types



procedures



For example:

- Grace: an approach-independent initiative
- Progres: see above
- ...

Summary

- GT for syntax, semantics, and integration of VMT's
- Relevant graph transformation theory
  - generative power of graph grammars and parsing of graph languages: specifying and recognizing the syntax of visual languages
  - confluence and termination: translation of models into semantic domains

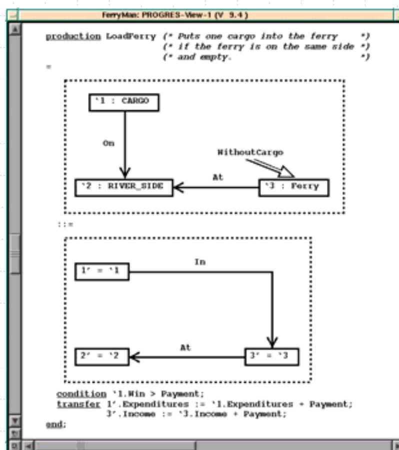
  → *theory of formal languages and rewriting*

4- Tool support

Outline

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

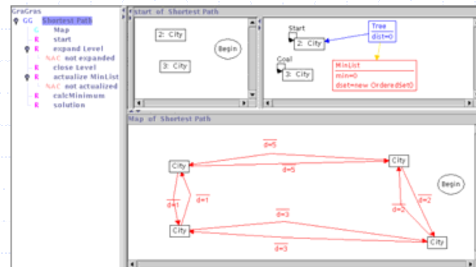
PROGRES
(PROgrammed Graph Rewriting Systems)



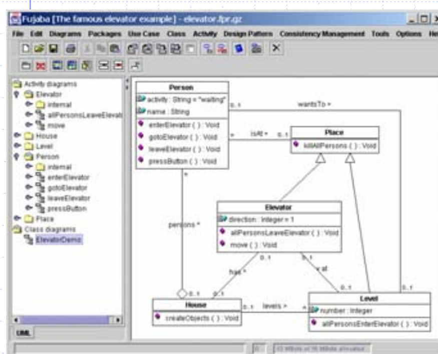
- Graphical/textual language to specify graph transformations
- Graph rewrite rules with complex and negative conditions
- Cross compilation in Modula 2, C and Java

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

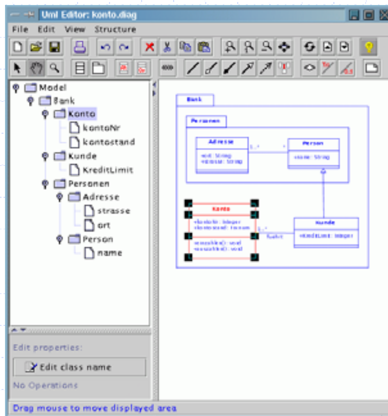


Fujaba
(From UML to Java and BAck)



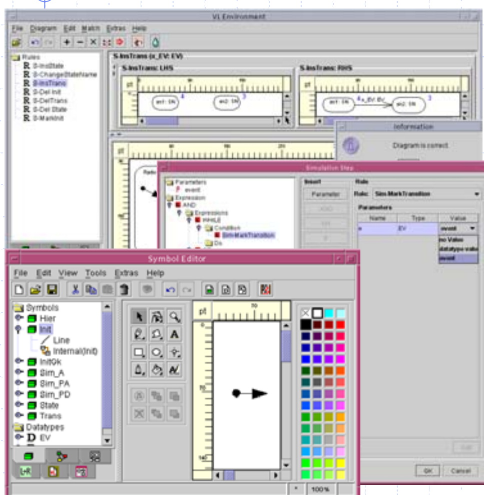
- Round trip engineering with UML, Java, and design patterns
- Class, collaboration and activity diagrams for story diagrams
  - Dynamic behavior
  - Automatic generation
- Reverse engineering

DiaGen
(The Diagram Editor Generator)



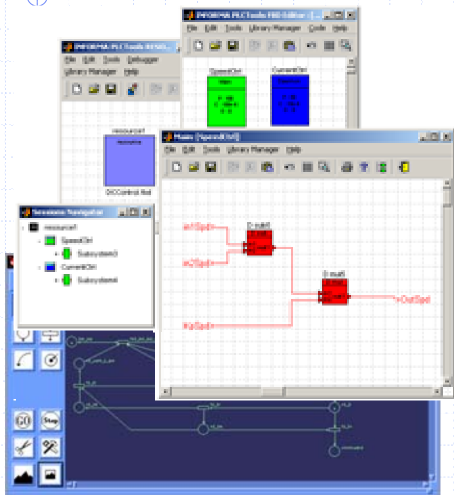
- Notations are specified through hypergraphs
- Framework of Java classes
  - to provide basic functionality
- Generator program
  - to produce Java source code

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

5- 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 should 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 cooperations 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
  - → theory of concurrency and rewriting
- semantics-preserving transformations
  - → evolution / refactoring of diagrams
- refinement and modularity of graph transformations
- relation with other areas like
  - → process calculi (e.g., Robin Milner's talk): proof techniques of algebraic and logic methods; *Can we adopt them?*
  - → tree- and term-based rewriting techniques (in compilers, XML, etc.): efficiency of special-purpose tools vs. usability of graph-based specification; *Can't we have both?*

Research Training Network *SegraVis* (10/2002 – 9/2006)

Syntactic and Semantic Integration of Visual Modeling Techniques

12 European partners offer grants for young researchers (< 36) with interest in visual modeling techniques

Paderborn	Leiden
Antwerp	London
Barcelona	Milan
Berlin	Darmstadt
Bremen	Pisa
Canterbury	Rome

Objectives: to develop meta-level techniques for defining syntax, semantics, analysis, transformation, … of UML and other visual models

Contact: Reiko Heckel

A few basic references

- HANDBOOK OF GRAPH GRAMMARS AND COMPUTING BY GRAPH TRANSFORMATION
  - Volume 1: foundations
    edited by Grzegorz Rozenberg (Leiden University, The Netherlands)
  - Volume 2: Applications, Languages and Tools
    edited by H Ehrig (Technical University of Berlin, Germany), G Engels (University of Paderborn, Germany), H-J Kreowski (University of Bremen, Germany) & G Rozenberg (Leiden University, The Netherlands)
  - Volume 3: Concurrency, Parallelism, and Distribution
    edited by H Ehrig (Technical University of Berlin, Germany), H-J Kreowski (University of Bremen, Germany), U Montanari (University of Pisa, Italy) & G Rozenberg (Leiden University, The Netherlands)

Web sites

- 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/
- Graph Grammar Bibliography
  - www.informatik.uni-bremen.de/theorie//appligraph/bibliography.html

6- Open discussion

Our Addresses

- Luciano Baresi
  - Politecnico di Milano
    Dipartimento di Elettronica e Informazione
    Piazza L. da Vinci, 32 – I20133 Milano (Italy)
    baresi@elet.polimi.it
- Reiko Heckel
  - Dr. Reiko Heckel
    University of Paderborn
    Mathematics/Computer Science Department
    Warburger Str, 100 - D33098 Paderborn (Germany)
    reiko@upb.de