University of Leicester

Foundations of Model Transformations: A Lambda Calculus for MDD?

Reiko Heckel
University of Leicester, UK

GLOBAN Summer School, Warsaw, 22-26 Sept. 2008

A "lambda calculus" for Model-driven Engineering?

- Focus and primary artifacts are models instead of programs
- Core activities include
  - maintaining consistency
  - evolution
  - translation
  - execution

  of models
- These are examples of model transformations
- A math. foundation is needed for studying
  - expressiveness and complexity
  - execution and optimisation
  - well-definedness
  - *preservation of semantics*

  of transformations
- Graph transformations as one such foundation

Why it is fun: Programming By Example

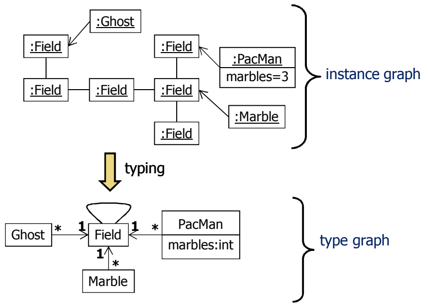
StageCast (www.stagecast.com): a visual programming environment for kids (from 8 years on), based on

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

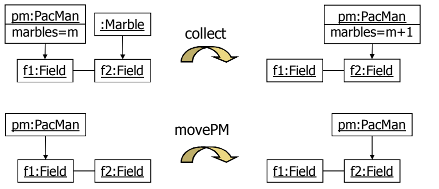
➔ intuitive rule-based behavior modelling

Next: abstract from concrete visual presentation

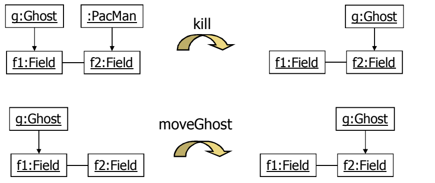
States of the PacMan Game: Graph-Based Presentation

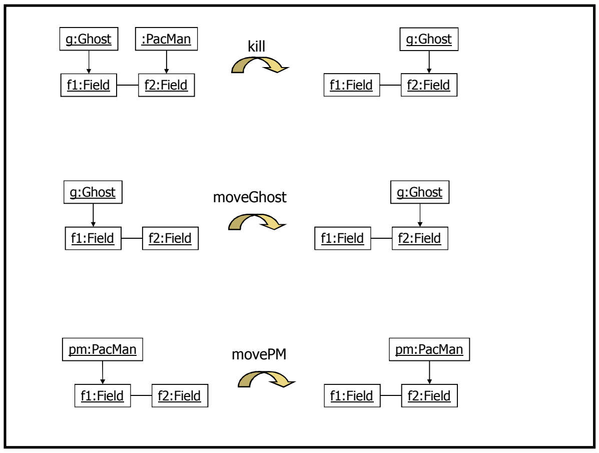


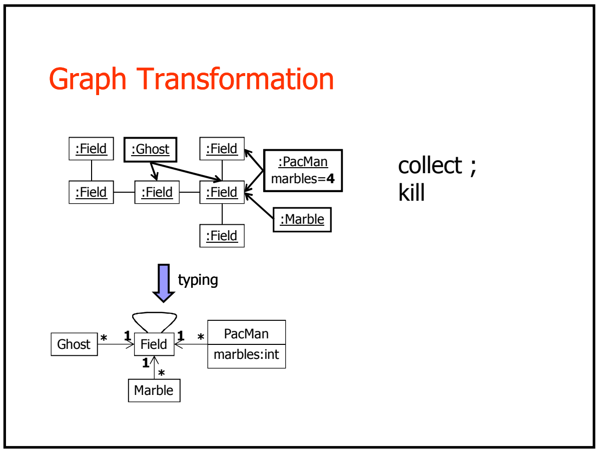
Rules of the PacMan Game: Graph-Based Presentation, PacMan



Rules of the PacMan Game: Graph-Based Presentation, Ghost





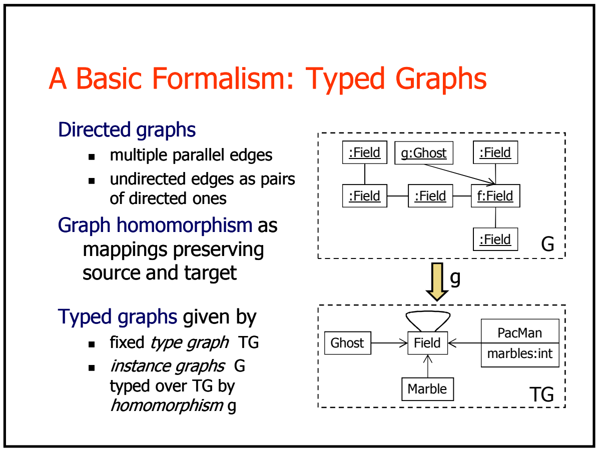


Graph Transformation

collect ;
kill

Outline

- Graph Transformation
  - ✓ why it is fun
  - how it works
- Semantics-preserving Model Transformation

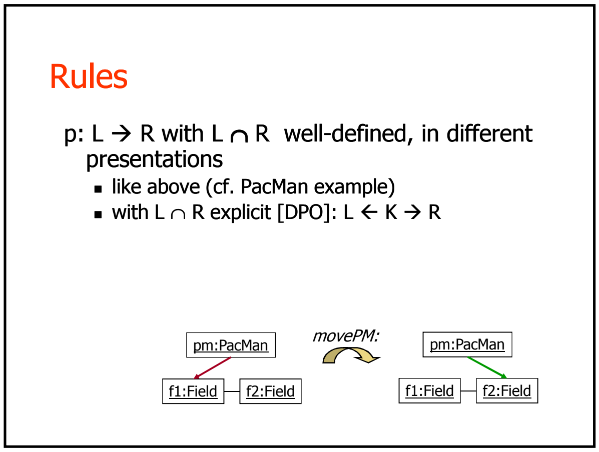


A Basic Formalism: Typed Graphs

Directed graphs
- multiple parallel edges
- undirected edges as pairs of directed ones

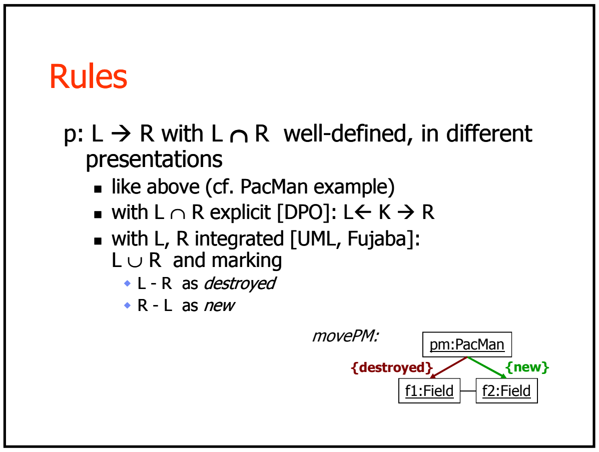
Graph homomorphism as mappings preserving source and target

Typed graphs given by
- fixed *type graph* TG
- *instance graphs* G typed over TG by *homomorphism* g



Rules

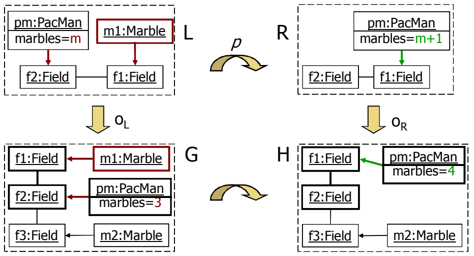
p: L → R with L ∩ R well-defined, in different presentations
- like above (cf. PacMan example)
- with L ∩ R explicit [DPO]: L ← K → R



Rules

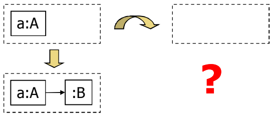
p: L → R with L ∩ R well-defined, in different presentations
- like above (cf. PacMan example)
- with L ∩ R explicit [DPO]: L← K → R
- with L, R integrated [UML, Fujaba]: L ∪ R and marking
  - L - R as *destroyed*
  - R - L as *new*

Transformation Step



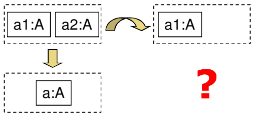
1. select rule p: L → R ; occurrence o_L: L → G
2. remove from *G* the occurrence of L\ R
3. add to result a copy of R \ L

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

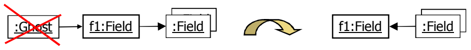


- ✖ conservative solution: application is forbidden
  - invertible transformations, no side-effects
- ✖ radical solution: give priority to deletion
  - more complex behavior, requires explicit control

Advanced Features

Dealing with unknown context

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

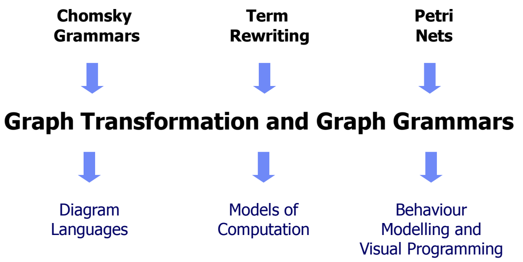


(turns f1 into a trap by reversing all outgoing edges to Field vertices, but only if there is no Ghost)

Control Structures

- priorities
- programmed transformation

A bit of History …



Chomsky Grammars

Term Rewriting

Petri Nets

Graph Transformation and Graph Grammars

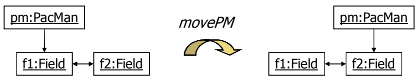
Diagram Languages

Models of Computation

Behaviour Modelling and Visual Programming

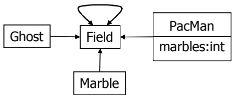
Exercise 1: Packman
a) be a (slightly) more clever player!

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



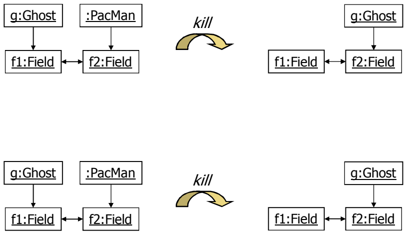
Exercise 1: Packman
b) Give *Pacman* another chance

Let *Pacman* have a counter for his lives.



Next: 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*

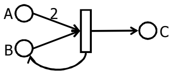


Exercise 2: Roots of GT
a) Chomsky Grammars

Production **A → aAb** as (context-free: one vertex or edge in *L*) graphical production rule

Exercise 2: Roots of GT
b) Petri Nets

A PT net transition as graph transformation rule

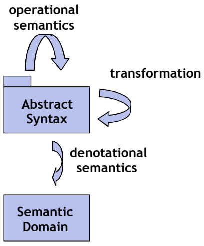


Exercise 2: Roots of GT
c) Term Rewriting

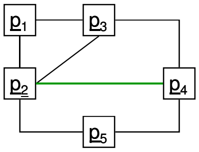
Rule f(s(s(x))) → f(s(x)) + f(x) as graph rewrite rule (tree or DAG)

Outline

- ✓ Graph Transformation
  - ✓ why it is fun
  - ✓ how it works
- ✓ Model Transformation
  - ✓ behavior modeling
  - ▪ operational semantics
  - ▪ denotational semantics



Case Study



Problem:

- no central infrastructure
- unreliable components

➔ removing nodes may disconnect network

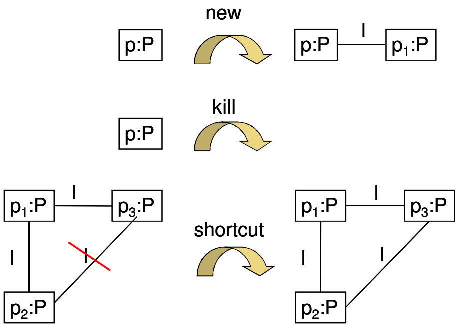
Idea: introduce redundancy!

Question: Which links should be added to guarantee a certain level of reliability ?

a) at random, up to a limit of **n** links

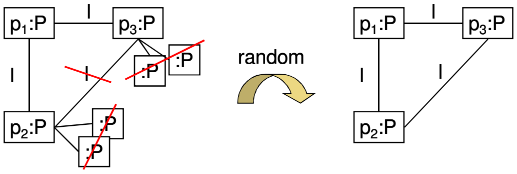
b) so that deletion of node does not increase distance

Modelling Change in the Network: A Graph Transformation System



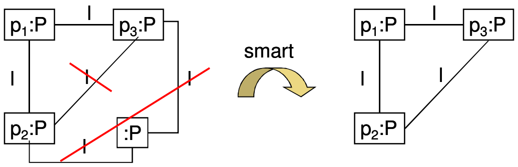
Which shortcuts?

a) At random (limit here: **n = 3** links)



Which shortcuts?

b) So that deletion of node does not increase distance



L. Mariani. Fault-tolerant routing for p2p systems with unstructured topology. Proc. International Symposium on Applications and the Internet (SAINT 2005), Trento, Italy.

Modelling Time: Stochastic Graph Transformation

✖ associate rate **ρ(p)** with every rule **p**

✖ **1/ρ(p)** average delay of p, once enables

$SG_{random,\ x}$

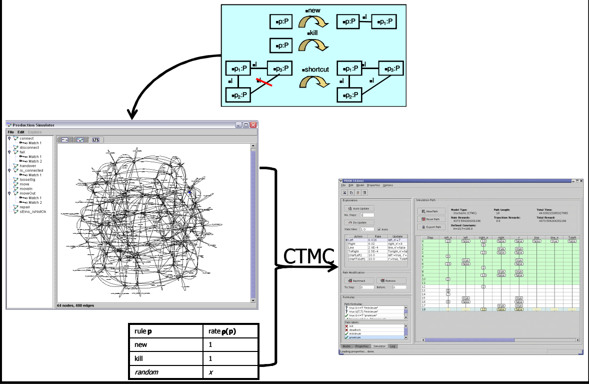
rule **p**	rate **ρ(p)**
new	1
kill	1
random	*x*

x times as fast as new or kill

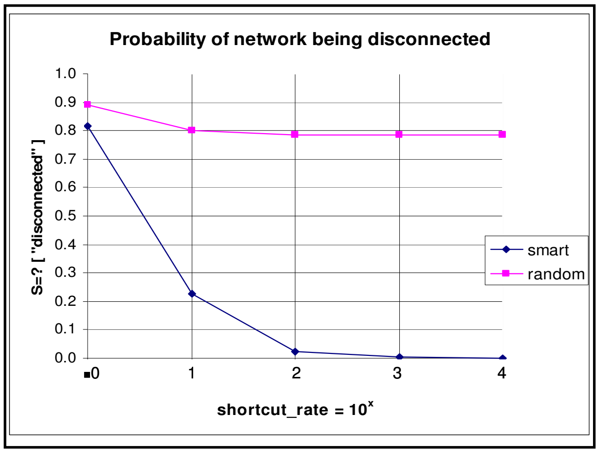
$SG_{smart,\ x}$

rule **p**	rate **ρ(p)**
new	1
kill	1
smart	*x*

Tools for Querying the Model

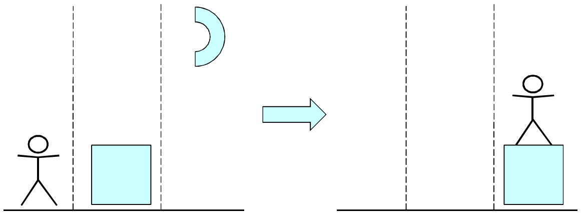


rule **p**	rate **ρ(p)**
new	1
kill	1
random	*x*



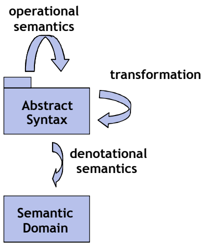
Monkey Business: Model and generate state space (GROOVE)

- A monkey is in a room with a box and a banana suspended from the ceiling.
- Before it can **eat** the banana, the monkey has **move** next to the box, **push** it into position, and **climb** on top.



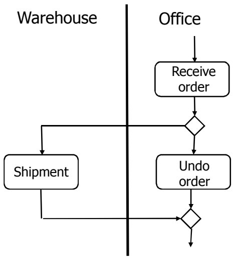
Outline

- ✓ Graph Transformation
  - ✓ why it is fun
  - ✓ how it works
- ✓ Model Transformation
  - ✓ behavior modeling
  - ▪ operational semantics
  - ▪ denotational semantics

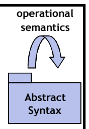


Example: WS Business Process

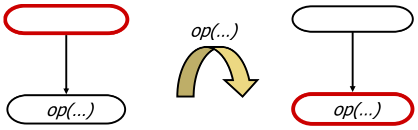
- refactoring of business processes, replacing centralised by distributed execution
- How to demonstrate preservation of behaviour?
  1. specify operational semantics of processes
  2. define transformations
  3. show that transformations preserve semantics



Operational Semantics

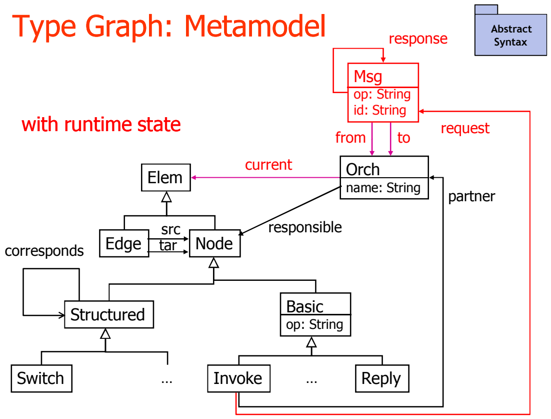


- diagram syntax plus *runtime state*
- GT rules to model transitions → defines labelled transition system

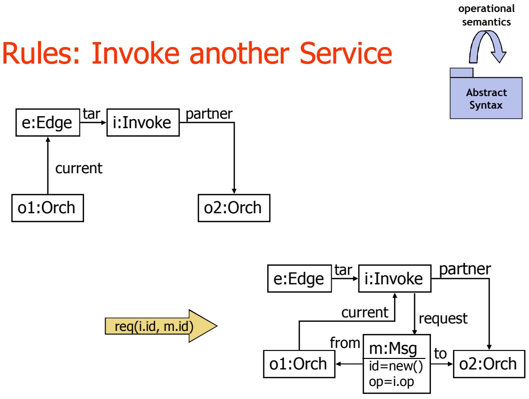


Type Graph: Metamodel

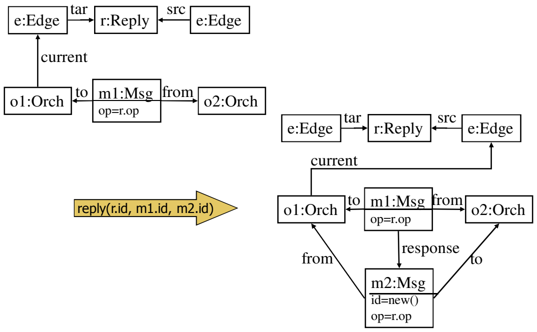
with runtime state



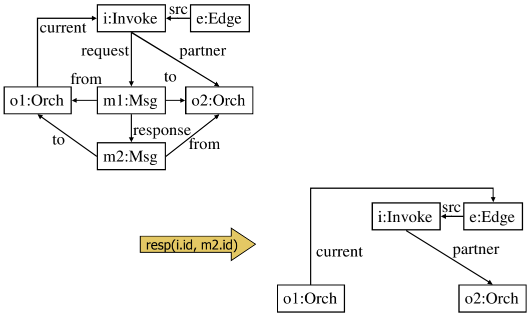
Rules: Invoke another Service



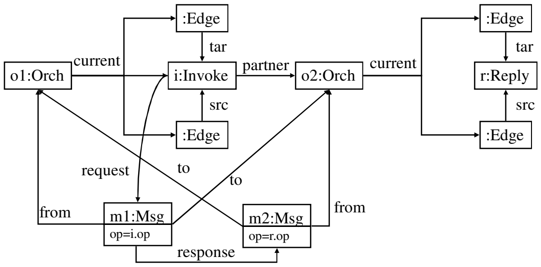
Rules: Answer the Invocation



Rules: Receive the Response

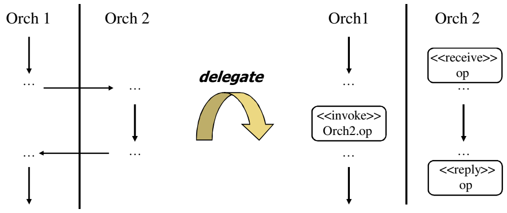


Simulation



Observations: *req(i.id, m1.id); reply(r.id, m1.id, m2.id); resp(i.id, m2.id)*

Refactoring



✖ replace local control flow by message passing

Semantic Compatibility

Processes P_1 and P_2 are compatible if *weakly bisimilar,* hiding labels not in alph(P_1) ∩ alph(P_2)

Show for trafo P_1 ➔ P_2 that P_2 simulates P_1, i.e.

- $P_1 \rightarrow_{obs} Q_1$ implies $P_2 \rightarrow_{obs} Q_2$
- Q_2 simulates Q_1

and vice versa.

Approach:

- mixed (local) confluence
- critical pair analysis

$$\begin{tikzcd} P_1 \arrow[r, "obs"] \arrow[d] & Q_1 \arrow[d] \\ P_2 \arrow[r, "obs"] & Q_2 \end{tikzcd}$$

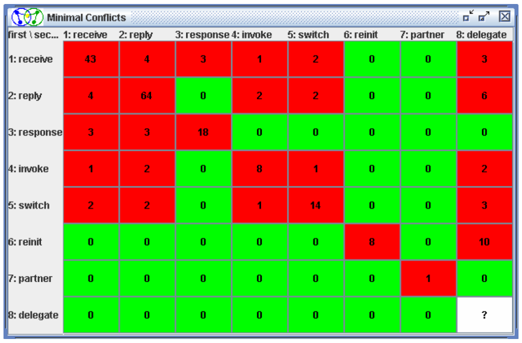
Critical Pairs and Local Confluence

- a pair of rules (p_1, p_2) in a *minimal conflict situation*
- constructed by *overlapping their left-hand sides* to *intersect in items to be deleted*
- system is locally confluent if all critical pairs are

$$\begin{tikzcd} & G \arrow[dl, "p_1"'] \arrow[dr, "p_2"] & \\ G_1 \arrow[dr, "*"'] & & G_2 \arrow[dl, "*"] \\ & H & \end{tikzcd}$$

Critical Pair Analysis in AGG

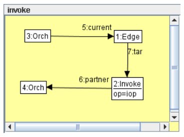
delegate vs *operational rules*



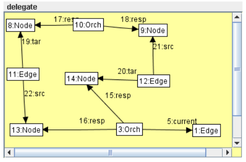
Minimal Conflicts

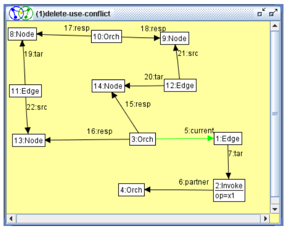
first \ sec...	1: receive	2: reply	3: response	4: invoke	5: switch	6: reinit	7: partner	8: delegate
1: receive	43	4	3	1	2	0	0	3
2: reply	4	64	0	2	2	0	0	6
3: response	3	3	18	0	0	0	0	0
4: invoke	1	2	0	8	1	0	0	2
5: switch	2	2	0	1	14	0	0	3
6: reinit	0	0	0	0	0	8	0	10
7: partner	0	0	0	0	0	0	1	0
8: delegate	0	0	0	0	0	0	0	?

Critical Pair



LHS of
invoke vs delegate

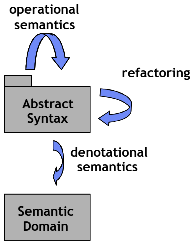




delete-use-conflict

Outline

- ✓ Graph Transformation
  - ✓ why it is fun
  - ✓ how it works
- ✓ Model Transformation
  - ✓ behavior modeling
  - ✓ operational semantics
  - ▪ denotational semantics
    - analysis → synthesis

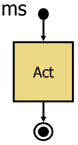


Context-Free Graph Grammar

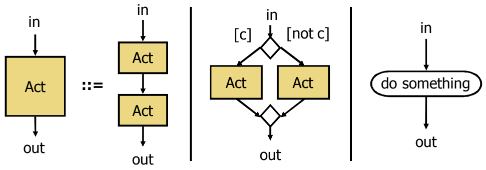
Abstract Syntax

Concrete Syntax of *Well-Structured* Activity Diagrams

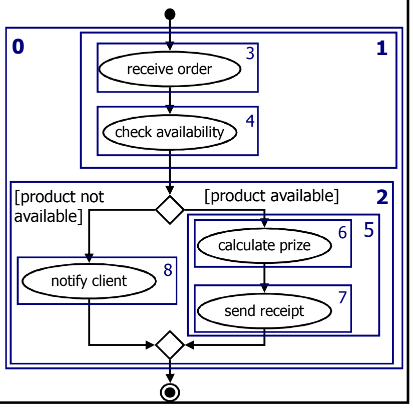
Start Graph:



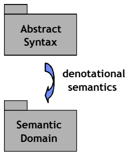
Productions in EBNF-like notation:

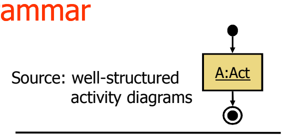


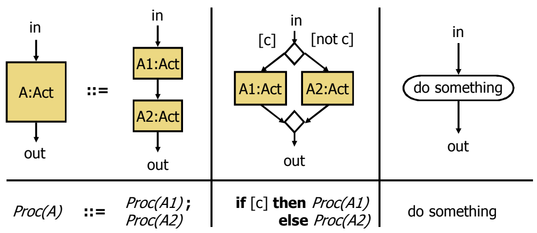
Analysis



Pair Grammar



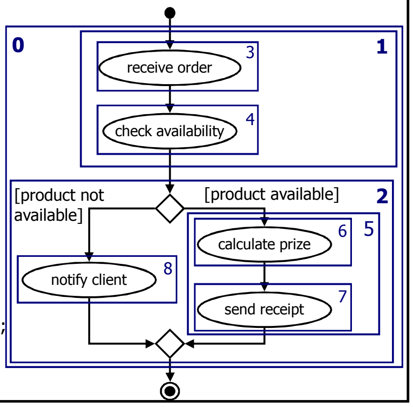




Synthesis

Proc(A0)

Proc(A1) ; *Proc(A2)*
…
Proc(A3) ;
Proc (A4) ;
if [product available]
then *Proc(A5)*
else *Proc(A8)*
…
receive order ;
check availability ;
if [product available]
then calculate prize ;
send receipt
else notify client



Is this Good Enough?

- ✓ Visual
  - abstract syntax or concrete syntax templates
- ✓ Bi-directional
  - swap source and target grammars
- ✓ Declarative
- × Expressive ?
  - context-free graph languages only
- × Traceable ?
  - through naming conventions
- × Efficient ?
  - NP complete parsing problem
- × …

→ Triple Graph Grammars

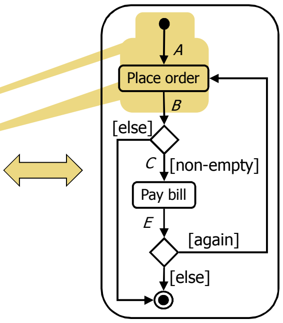
A Non-Well-Structured Example

Actions

Place_order, Pay_bill

Processes

A = Place_order → *B*
B = if 'non-empty'
then *C* else STOP
C = Pay_bill → *E*
E = if 'paid'
then *A* else STOP

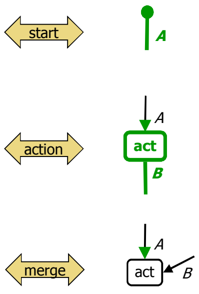


Correspondence Rules:
Initial, Action, and Merge on Action

_A_ = …

A = **act → _B_**
_B_ = …

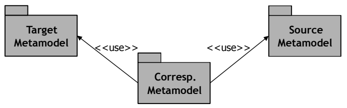
A = **_B_**



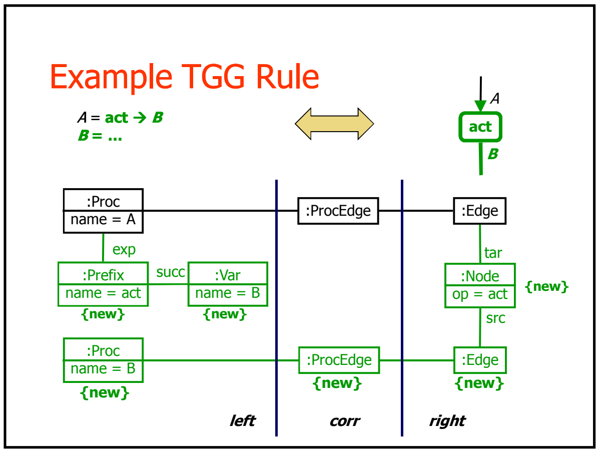
- ✖ Rule pairs, in condensed presentation
  - **Green/bf** → {new}
- ✖ No restriction to context-freeness
- ✖ Correspondence via common names

Formally: Triple Graph Grammars

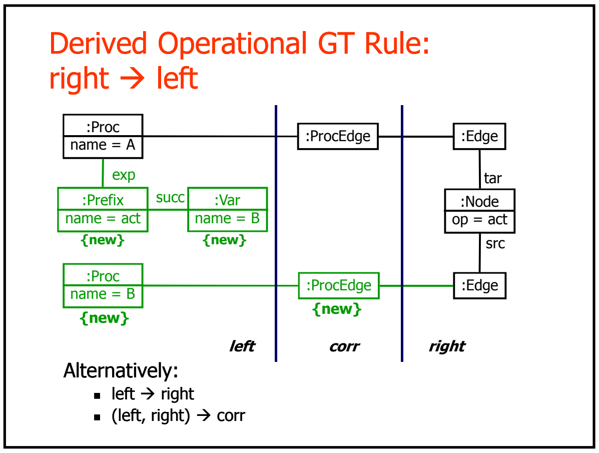
- ✖ Meta model for correspondence
  - ✓ traceability



- ✖ Symmetric rule triplets (left, corr, right), generating directed rules
  - ✓ Declarative → operational



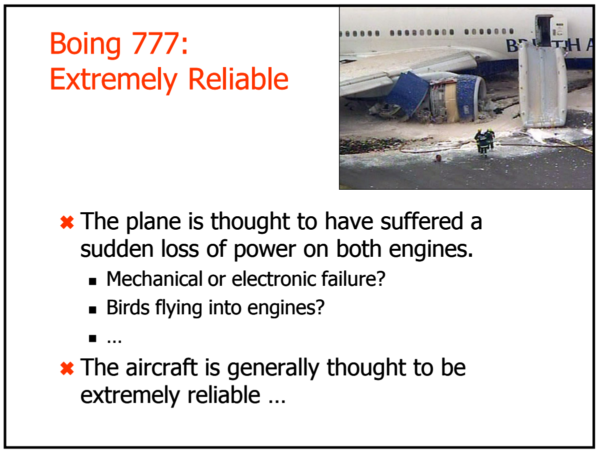
Example TGG Rule



Derived Operational GT Rule: right → left

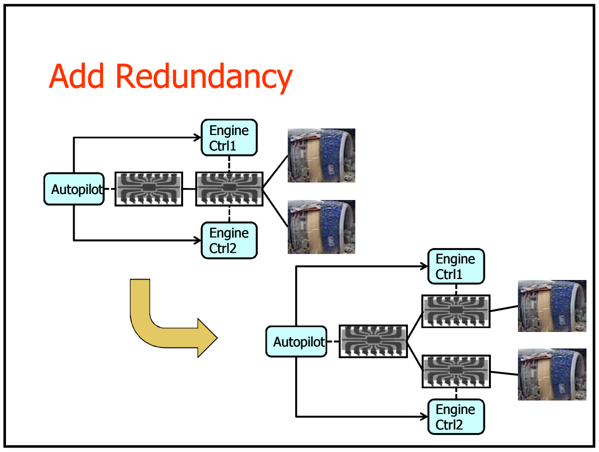
Alternatively:

- left → right
- (left, right) → corr

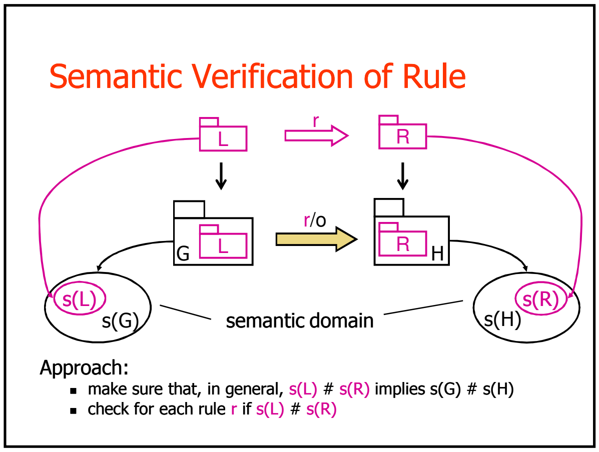


Boing 777: Extremely Reliable

- ✖ The plane is thought to have suffered a sudden loss of power on both engines.
  - Mechanical or electronic failure?
  - Birds flying into engines?
  - …
- ✖ The aircraft is generally thought to be extremely reliable …



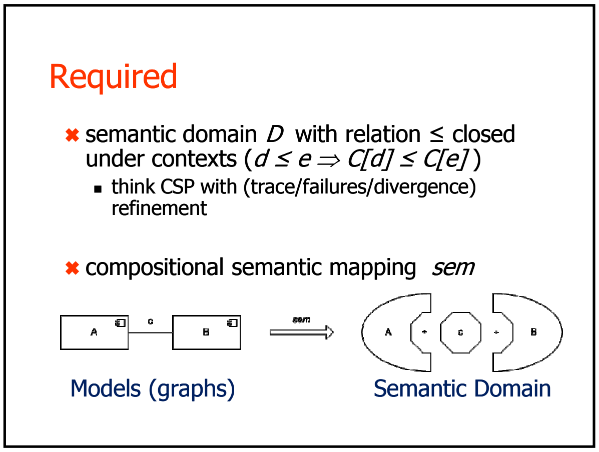
Add Redundancy



Semantic Verification of Rule

Approach:

- make sure that, in general, s(L) # s(R) implies s(G) # s(H)
- check for each rule r if s(L) # s(R)

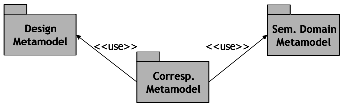


Required

- ✖ semantic domain *D* with relation ≤ closed under contexts ($d \le e \Rightarrow C[d] \le C[e]$)
  - think CSP with (trace/failures/divergence) refinement
- ✖ compositional semantic mapping *sem*

How to ensure compositionality?

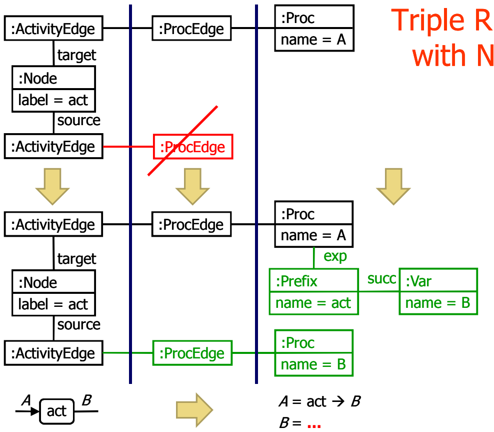
Semantic mapping described by triple graph grammar



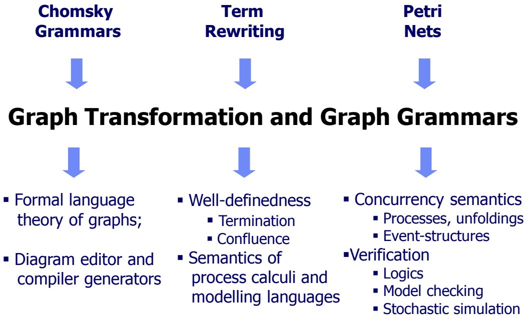
$sem(G_0) ::= G_n \Leftrightarrow G_0 \Rightarrow^* G_n$ terminated

Is compositional if there are no negative preconditions over source elements

Triple Rule with NAC



Roots and Inspirations



Chomsky Grammars

Term Rewriting

Petri Nets

Graph Transformation and Graph Grammars

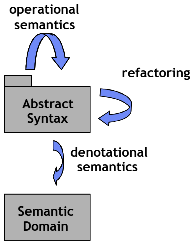
- Formal language theory of graphs;
- Diagram editor and compiler generators

- Well-definedness
  - Termination
  - Confluence
- Semantics of process calculi and modelling languages

- Concurrency semantics
  - Processes, unfoldings
  - Event-structures
- Verification
  - Logics
  - Model checking
  - Stochastic simulation

Outline

- ✓ Graph Transformation
  - ✓ why it is fun
  - ✓ how it works
- ✓ Model Transformation
  - ✓ behavior modeling
  - ✓ operational semantics
  - ✓ denotational semantics



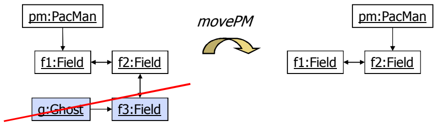
Conclusion

- ✖ The tutorial has
  - Motivated the use of graph transformation in software engineering
  - Introduced the foundations of graph transformation
  - Shown example applications of graph transformation
    - for behavior modeling and analysis
    - for model transformations for translating between languages, execution an refactoring of models
- ✖ Want to know more?
  - visit www.gratra.org, subscribe to gratra@upb.de, or email reiko@mcs.le.ac.uk

Discussion

Solution 1: Packman
a) be a (slightly) more clever player!

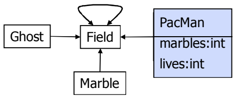
Extend the *movePM* rule so that *Pacman* does not move next to a *Ghost.*



Solution: a negative application condition.

Solution 1: Packman
b) Give *Pacman* another chance

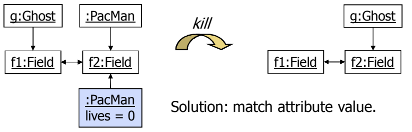
Let *Pacman* have a counter for his lives.



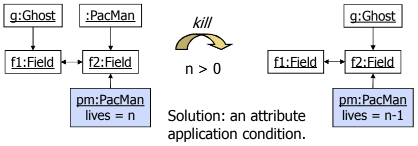
Solution:
add an attribute.

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

Solution 1 b) Refine rule *kill*



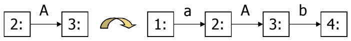
Solution: match attribute value.



Solution: an attribute application condition.

Solution 2: Roots of GT
a) Chomsky Grammars

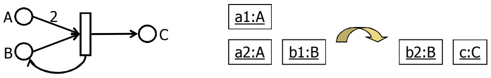
Production **A → aAb** as (context-free: one vertex or edge in *L*) graphical production rule



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

Solution 2: Roots of GT
b) Petri Nets

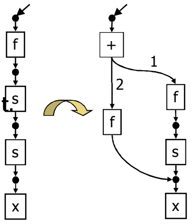
A PT net transition as graph transformation rule



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

Solution 2: Roots of GT
c) Term Rewriting

Rule f(s(s(x))) → f(s(x)) + f(x) as DAG rewrite rule



- ✖ Theory of term graph rewriting (TGR)
  - soundness / completeness w.r.t. TR
  - termination, critical pairs, confluence