University of Leicester

Foundations of Model Transformations

A "lambda calculus" for MDD ?

Reiko Heckel
University of Leicester, UK

Motivation

- At the heart of model-driven engineering are activities like
  - 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
  - Semantic correctness

  of transformations
- This lecture is about graph transformations as one such foundation

Outline

- Graph transformation
  - why it is fun
  - how it works
- Model transformation
- Theory and Tools

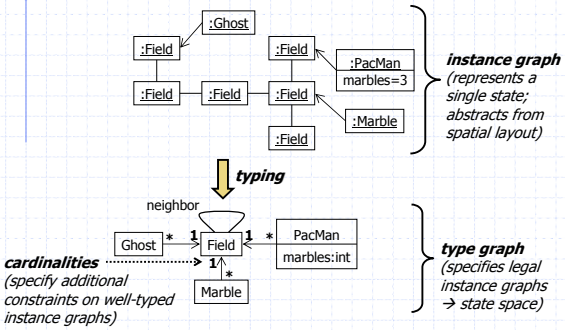
Why it is fun: Programming By Example

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

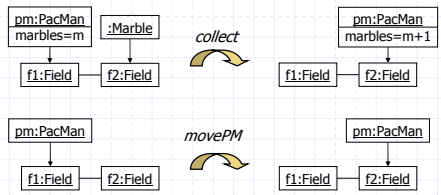
→ intuitive rule-based behavior modelling

Next: abstract from concrete visual presentation

States of the PacMan Game: Graph-Based Presentation

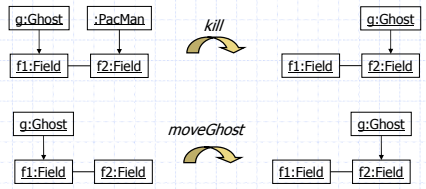


Rules of the PacMan Game: Graph-Based Presentation, PacMan



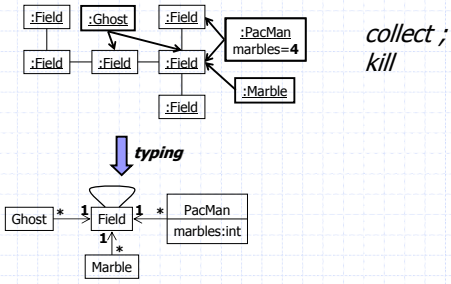
PacMan's rules:
collect has priority over *movePM*

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



Ghost's rules:
kill has priority over *moveGhost*

Graph Transformation



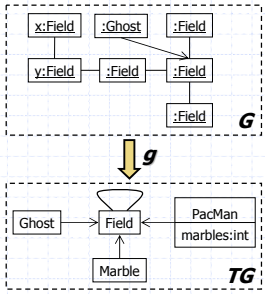
How it works: Typed Graphs

Directed graphs as algebraic structures *G = (V, E, src, tar)*
with *src, tar: E → V*

Graph homomorphism as pair of mappings
h = (h_V , h_E): G_1 → G_2 with
- *h_V: V_1 → V_2*
- *h_E: E_1 → E_2*

preserving *src* and *tar*

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

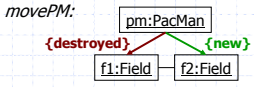


Rules

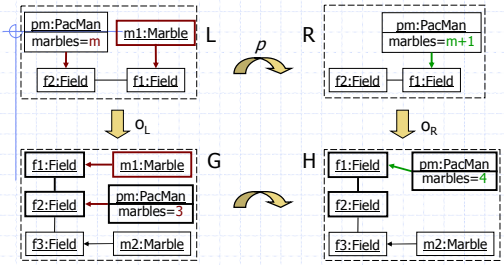
p: L → R with *L ∩ R* well-defined, in different presentations
- like above (cf. PacMan example)
- with *L ∩ R* explicit [DPO]: *L ← K → R*

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]: *L ∪ R* and marking
  - *L - R* as {destroyed}
  - *R - L* as {new}

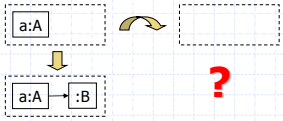


Transformation Step: Operational



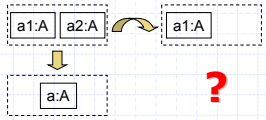
1. select rule *p : L → R*; occurrence *o_L : L → G*
2. remove from *G* the occurrence of *L \ R*
3. add to result a copy of *R \ L*

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

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

Where it comes from and what it is good for

Chomsky Grammars | **Term Rewriting** | **Petri Nets**

Graph Transformation and Graph Grammars

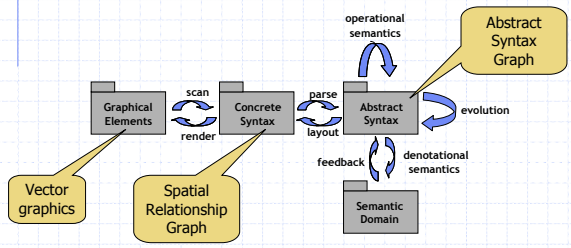
Diagram Languages | Models of Computation | Behaviour Modelling and Visual Programming

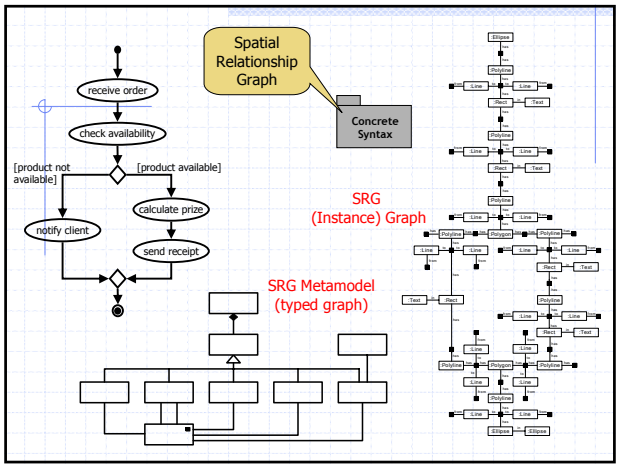
Outline

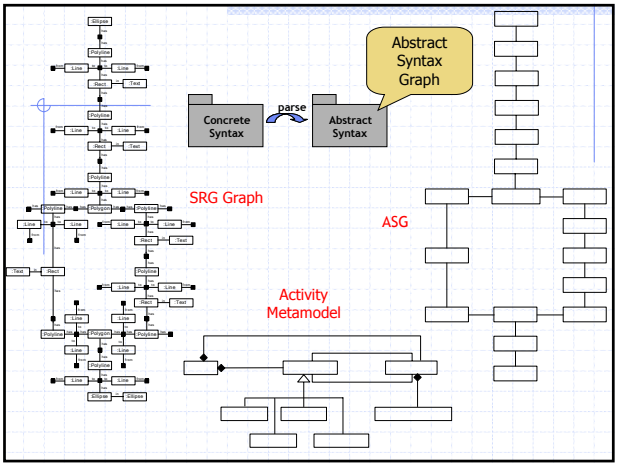
- ✓ Graph transformation
- ✖ Model transformation
  - diagram languages
  - execution
  - translation
- ✖ Theory and Tools

Diagram Languages

- ✖ theory and tools like for textual languages

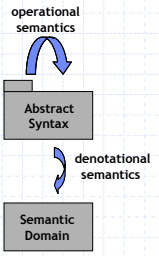




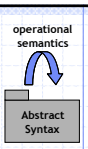


Semantics

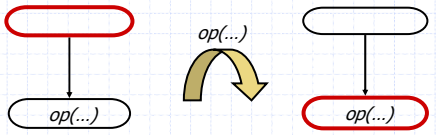
- Operational semantics (execution)
  - concrete syntax: animation rules
  - abstract syntax: graph transformation rules
- Denotational semantics (translation)



Operational Semantics

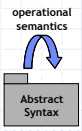


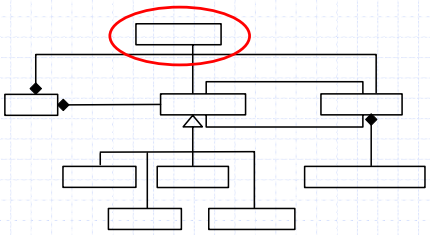
- diagram syntax plus *runtime state*
- visual rules to model state transitions



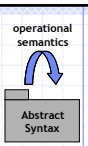
- produce sequences of states / labels, visualized as animations

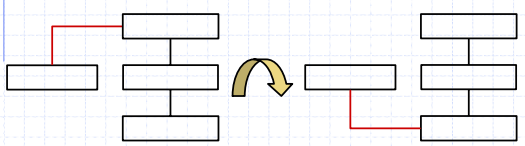
Extended Meta Model: Original plus *Runtime State*





Operational Rules: GT on Meta Model Instances

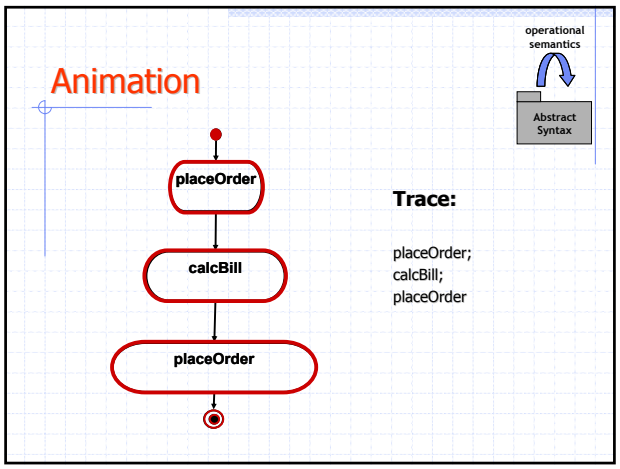




Polyline Line Ellipse Polygon

1 1 1 1 1

Animation

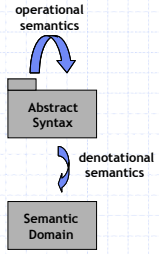


Trace:

placeOrder;
calcBill;
placeOrder

Semantics

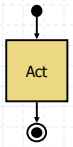
- ✓ Operational semantics (execution)
- ✖ Denotational semantics (translation)
  - analyse model (compiler front-end)
  - generate semantic representation (compiler back-end)



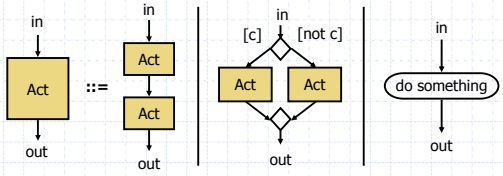
Analysis: *Context-Free* Graph Grammar

Concrete Syntax of *Well-Formed* Activity Diagrams

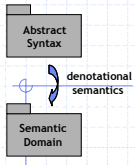
Start Graph:



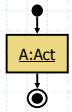
Productions in EBNF-like notation:



Pair Grammar

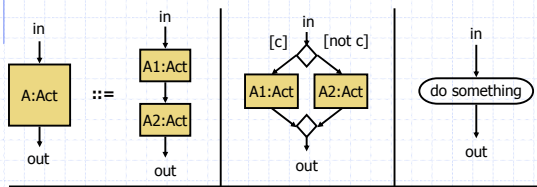


Source: well-structured activity diagrams



Target: CSP

Proc(A)



Proc(A) ::= *Proc(A1)* → *Proc(A2)*

if [c] **then** *Proc(A1)* **else** *Proc(A2)*

do something

Generation

Proc(A0)

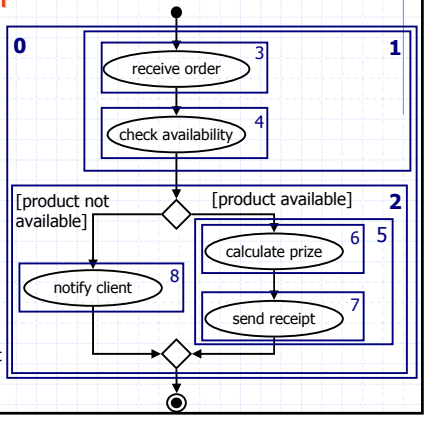
Proc(A1) → *Proc(A2)*

...

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



Good Enough for Model Transformations?

- ✓ Visual
  - abstract syntax or concrete syntax templates
- ✓ Bi-directional
  - inherently symmetry
- ✓ Declarative
- × Expressive ?
  - Context-free graph languages
- × Traceable ?
  - Naming conventions
- × Efficient ?
  - NP complete parsing problem
- × Incremental ?

Cf. OMG's QVT requirements

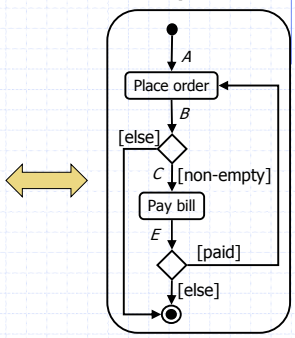
A Non-Well-Structured Example

Actions

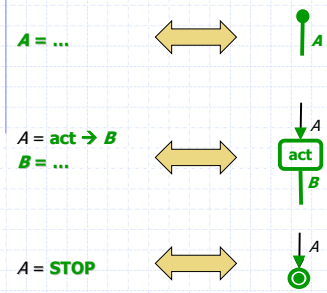
Place_order, Pay_bill

Processes

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

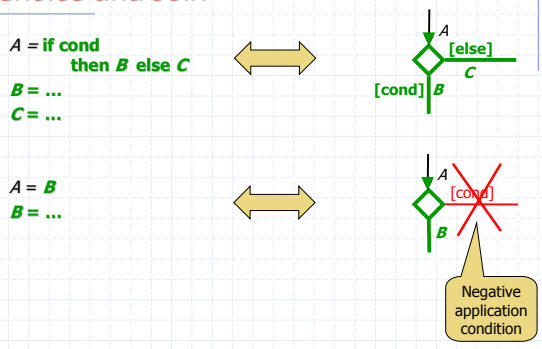


Correspondence Rules: Initial, Action, and Final Nodes

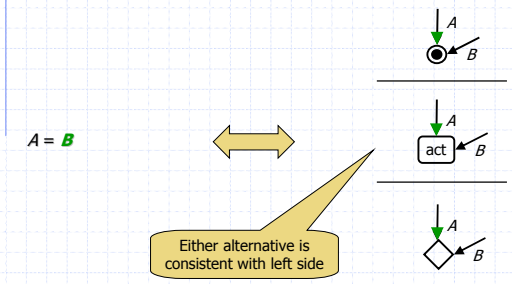


- Rule pairs, in condensed presentation
  - Green/bf → {new}
- No restriction to context-freeness
- Correspondence via common names

Correspondence Rules: Choice and Join

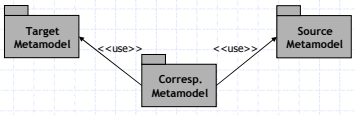


Correspondence Rules: Connection to Existing Nodes



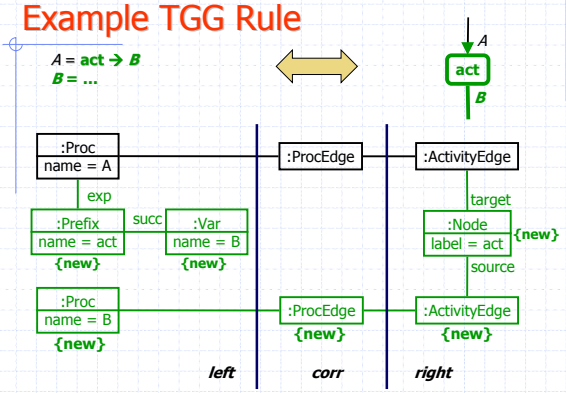
Formally: Triple Graph Grammars

- Meta model for correspondence
  - ✓ traceability

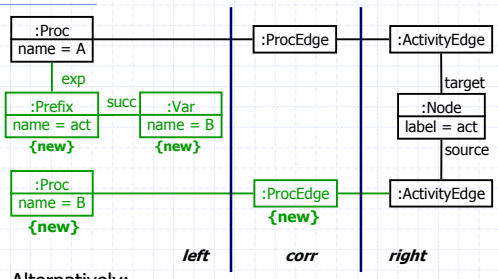


- Symmetric rule triplets (left, corr, right), generating directed rules
  - ✓ Declarative → operational

Example TGG Rule



Derived Operational GT Rule: right → left



Alternatively:

- left → right
- (left, right) → corr

Outline

- ✓ Graph transformation
- ✓ Model transformation
- ✖ Theory and Tools

Theory: Sources of Inspirations

Chomsky Grammars | **Term Rewriting** | **Petri Nets**

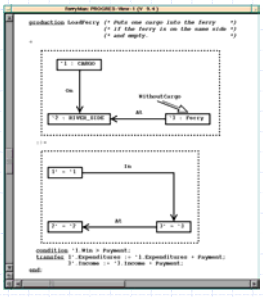
Graph Transformation and Graph Grammars

- Formal language theory of graphs;
- Diagram compiler generators
- Well-definedness
  - Termination
  - Confluence
- Semantics of process calculi
- Concurrency theory
  - Causality and conflict
  - Processes, unfoldings
  - Event-structures

Outline

- ✓ Graph transformation
- ✓ Model transformation
- ✖ Theory and *Tools*
  - General purpose modeling environments
    - PROGRES, AGG, Fujaba, …
  - Environments for specifying visual notations
    - DIAGEN, GENGEd, …
  - Analysis tools
    - Groove, …

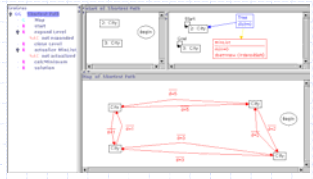
PROGRES
(PROgrammed Graph Rewriting Systems)



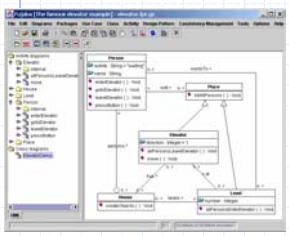
- ✖ Graphical/textual language to specify graph transformations
- ✖ Graph rewrite rules with complex and negative conditions
- ✖ Embedded in programming notation for OO data base transactions
- ✖ Cross compilation in Modula 2, C and Java

AGG
(The Attributed Graph Grammar System)

- ✖ Algebraic approach to graph transformation
- ✖ Attribute computations in Java
- ✖ Efficient graph parsing
- ✖ Analysis facilities for critical pairs, consistency of rules and invariants

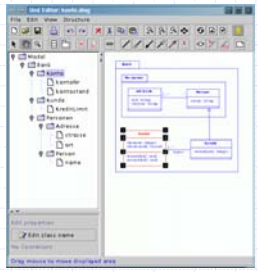


Fujaba
(From UML to Java and Back Again)



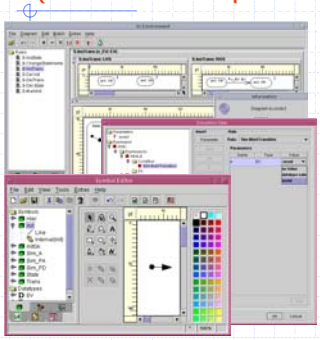
- Round trip engineering with UML, Java, and design patterns
- Class and activity diagrams; GT rules as collaborations
- Generates standalone Java classes
- Various plugins, including triple graph grammars

DiaGen
(The Diagram Editor Generator)



- VL concrete / abstract syntax specified as CF hypergraph grammars
- Generation of diagram editors, parsers, simulators

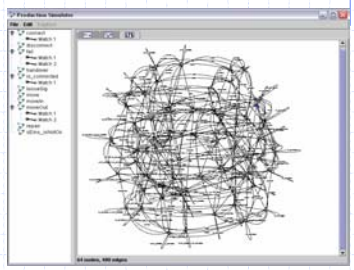
GenGED
(Generation of Graphical Env.s for Design)



- Roughly similar to the above, based on AGG

GRaphs for Object-Oriented VErification (GROOVE)

- (bounded) generation of LTS from GT systems
  - edge-labelled graphs
  - application conditions
  - rule priorities



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

Graph Transformation
A "lambda calculus" for MDD ?

- Expressiveness and complexity
  - CF graph grammars: Rozenberg; Habel
  - Relation with MSO logic: Courcelle
- Execution and optimisation
  - Various tools
  - Graph matching: Zuendorf 96; Varro et al 05
- Well-definedness
  - Term graph rewriting: Plump
  - Confluence, termination of attributed GT: Ehrig et al 04
- Semantic correctness
  - Requires formal semantics of the languages involved
    - operational: Hausmann et al
    - denotational: Kuester et al
  - Verification through
    - proofs: several authors
    - model checking: Varro; Rensink
    - testing: Koenig et al 04

Inspired?
Join us in Vienna!

Fundamental Approaches to Software Engineering

ETAPS 2006

Vienna (Austria) March 27-29, 2006

ETAPS 2006

- The event: 25 March - 2 April 2006
- Abstract submission: Friday 7 October 2005
- Paper submission: Friday 14 October 2005

GT-VMT* 2006

- Paper submission: 12 December 2005

*: 5th Intl Workshop on Graph Transformation and Visual Modelling Techniques

Questions ?