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

Outline
- Graph Transformation
  - why it is fun
  - how it works
- Applications and Theory
- Tools
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

Rules

$p: L \to R$ with $L \cap R$ well-defined, in different presentations
- like above (cf. PacMan example)
- with $L \cap R$ explicit [DPO]: $L \leftarrow K \to R$
- with $L, R$ integrated [UML, Fujaba]:
  - $L \cup R$ as (destroyed)
  - $R - L$ as (new)

Transformation Step

1. select rule $p: L \to R$; occurrence $q_0: L \to G$
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
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
\[\text{Graph Transformation and Graph Grammars}\]

Control Structures
- priorities
- programmed transformation

Outline

✓ Graph Transformation
  ✓ why it is fun
  ✓ how it works
✓ Applications and Theory
  ✓ Modelling and Analysis of Functional Requirements
  ✓ Model Transformation and Semantics
✓ Tools

Motivation: Software Development as Integration of Views

```
Model A
```

```
Model B
```

Structure: Class and Object Diagrams

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

Aspects of Requirements Models

Model A

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.

Model B

```
Consecutive steps
- 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

Alternative steps are
- select items from rack
- pack and give items to customer’s cart
- take items out of customer’s cart
- add them to the bill
- collect payment
- pack and give items to customer

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.

Aspects of Requirements Models
- 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.
- 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

Tool Support: Critical Pair Analysis with AGG

Reiko Heckel, Univ. of Leicester
Focus and primary artifacts

These are examples of model transformations. A math. foundation is needed for studying programs.

Core activities include:
- maintaining consistency
- evolution
- translation
- execution of models

These are examples of model transformations.

Graph transformation
- why it is fun
- how it works

Applications and Theory
- Modelling and Analysis of Functional Requirements
- Model Transformation and Semantics
- Tools

Outline

- Model transformation
  - denotational semantics
  - operational semantics
  - refactoring

Model-driven Development

- 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

Concrete Syntax of Well-Formed Activity Diagrams

Productions in EBNF-like notation:

Context-Free Graph Grammar

Start Graph:
**Analysis**

```
0:receive order
1:check availability
2:if [product available] then calculate prize
   out
   then send receipt
   else notify client
3:[product not available]
4:[product available]
```

**Synthesis**

```
Proc(A0)
Proc(A1) => Proc(A2)
... Proc(An) =>
if [product available] then Proc(A5)
else Proc(A8)
... receive order =>
check availability => if [product available] then calculate prize =>
send receipt
else notify client
```

**Pair Grammar**

```
A:Act := Proc(A) :=
if [c] then Proc(A1) => Proc(A2)
else Proc(A3)
out
in
Proc(A) :=
A:Act =>
A:Act =>
A:Act =>
A:Act =>
do something
```

**Is this Good Enough?**

- **Visual**
  - abstract syntax or concrete syntax templates
- **Bi-directional**
  - swap source and target grammars
- **Declarative**
- **Efficient**
- **Expressive**
- **Traceable**
  - through naming conventions
- **Context-free**
- **NP complete parsing**

→ Triple Graph Grammars

**A Non-Well-Structured Example**

```
Actions
- Place_order, Pay_bill
Processes
A = Place_order \rightarrow B
B = if 'non-empty' then C else STOP
C = Pay_bill \rightarrow E
E = if 'paid' then A else STOP
```

**Correspondence Rules: Initial, Action, and Final Nodes**

- **Rule pairs, in condensed presentation**
  - Green/bf \rightarrow \langle new \rangle
- **No restriction to context-freeness**
- **Correspondence via common names**
Correspondence Rules:
Split and Join

\[ A = \text{if cond then } B \text{ else } C \]
\[ B = \ldots \]
\[ C = \ldots \]

Correspondence Rules:
Connection to Existing Nodes

\[ A = B \]

Formally: Triple Graph Grammars
- Meta model for correspondence
  - traceability
- Symmetric rule triplets (left, corr, right), generating directed rules
  - Declarative \(\rightarrow\) operational

Derived Operational GT Rule:
right \(\rightarrow\) left

Example TGG Rule

Outline
- Model transformation
  - denotational semantics
  - operational semantics
  - refactoring
Example: Executable 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: Idea

- diagram syntax plus runtime state
- GT rules to model state transitions

Operational Semantics: Formally

\[ GTS = (TG, P) \] with start graph \( G_0 \)
defines transition system

\[ LTS(GTS, G_0) = (S, L, \rightarrow) \]

taking
- as states \( S \) all graphs reachable from \( G_0 \)
- observations on rules as labels
- transformations as transitions

Rules: Invoke another Service

Rules: Answer the Invocation

Type Graph: Metamodel

with runtime state
**Foundations and Applications of Graph Transformation**

**SMF / ICGT 2006 Tutorial**

**September 2006, Natal, Brazil**

---

**Rules: Receive the Response**

```plaintext
i:Invoke

m1:Msg

m2:Msg

to from

current

partner

1. Invoke 2. Edge

rep(i.id, m1.id)

a1.Orch a2.Orch
```

**Simulation**

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

---

**Outline**

- Model transformation
  - denotational semantics
  - operational semantics
  - refactoring

**Example: Executable 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

**Preservation of Semantics**

Show for each refactoring $P \Rightarrow P'$ that $P'$ simulates $P$, i.e.

- $P \Rightarrow_{obs} Q$ implies $P' \Rightarrow_{obs} Q'$
- $Q'$ simulates $Q$ and vice versa.

**Approach:**

- mixed (local) confluence
- critical pair analysis

---

**Reiko Heckel, Univ. of Leicester**
Foundations and Applications of Graph Transformation

Reiko Heckel, Univ. of Leicester

Relevant Theory

- Chomsky Grammars
- Term Rewriting
- Petri Nets

Graph Transformation and Graph Grammars

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

Outline

- Graph Transformation
  - why it is fun
  - how it works
- Applications and Theory
  - Modelling and Analysis of Functional Requirements
    - Model Transformation and Semantics
- Tools

Tools

- Two main groups:
  - General purpose modeling environments
    - PROGRES, AGG, Fujaba, ...
  - Environments for specifying visual notations
    - DIAGEN, GENGEd, MetaEnv, ConWork, ...

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

September 2006, Natal, Brazil
### 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, IBM Zurich
  - JKI@zurich.ibm.com

### Visual JML Tool
- GT rules as visual contracts
- maps into JML for run-time monitoring
- Contact:
  - M. Lohmann, Paderborn
  - lohmann@tulip.de
  - (see ICGT talk)

### 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

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
    - GT for behavior modeling and analysis
    - GT for model transformation
  - Presented available tools

Future work

- Dissemination to potential users
  - this tutorial
  - examples and case studies
  - co-operations with domain experts
- More user-friendly and efficient tools (we have come a long way already)

- Analysis and verification
- Refinement / modularity
- relation with other areas
  - process calculi
  - DNA computing
  - XML, Meta data, Semantic Web (Rising)

Some 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
- Tutorial Introduction to Graph Transformation: A Software Engineering Perspective

Discussion