Graph Transformation in Software Engineering

Gabriele Taentzer
Philipps-Universität Marburg

Introductory part based on slides by:
Reiko Heckel
University of Leicester
Luciano Baresi
Politecnico di Milano
Overview

March 9 – 11, 2011:
- Introduction to graph transformation
- The basic theory of graph transformation
- Reasoning about graph transformation

In Summer 2011:
- Applications to Software Engineering
  - Model-based Software Development
  - Domain-Specific Language Engineering
Aims

- knowledge of the basic graph transformation theory
- ability to recognize and specify graph structures and their transformations
- ability to reason about graph transformations
- feeling for the potentials and limits of graph transformation in the context of software engineering
Introduction to Graph Transformation

March 9, 2011
Overview

- Pac Man: A first example
- The basic graph transformation concepts
- Connections to other rewriting concepts
  - Chomsky grammars
  - Term rewriting
  - Petri nets
- Tool support for graph transformation
Pac Man: A First Example

What is the idea?
Motivation:
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 internal control structures (priority, sequence, non-determinism, ...)
- external keyboard control

Rule-based behavior modeling is a natural and intuitive paradigm!

Example: A simple PacMan game; concrete presentation

Gabriele Taentzer
States of the PacMan Game: Graph-Based Presentation

instance graph
(represents a single state; abstracts from spatial layout)

typing

cardinalities
(specify additional constraints on well-typed instance graphs)

type graph
(specifies legal instance graphs \(\rightarrow\) state space)

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

PacMan’s rules:

\( \text{collect} \) has priority over \( \text{movePM} \)
Rules of the PacMan Game: Graph-Based Presentation, Ghost

Ghost’s rules: 
\textit{kill} has priority over \textit{moveGhost}
Graph Transformation

collect; kill

typing
The Basic Concepts 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 to other Rewriting Techniques
  - Inspiration for application and theory.
Roots and Sources

Chomsky Grammars

Term Rewriting

Petri Nets

Graph Transformation and Graph Grammars

(Web grammars: Pfaltz, Rosenfeld 68; Montanari 69)
(Grammars for partial orders: Schneider 70)
(λ-graph reduction: Wadsworth 71)

Node label-controlled graph grammars
[ NLC ]

Monadic 2nd Order Logic of Graphs
[ MSO ]

PROgrammed Graph REwriting Systems
[ PROGRES ]

Gabriele Taentzer
Two basic approaches to graph transformation

- The embedding approach

- The gluing approach

We consider the gluing approach in the following.
Graphs and Graphs

- Edges as relations
  - \( G = (V, E, src, tar) \)
  - \( E \subseteq V \times V \)
- No parallel edges possible
- Usual graph definition

- Edges as identities
  - \( G = (V, E, src, tar) \)
  - \( src, tar : E \rightarrow V \)
- Parallel edges possible
- Graph notion in the algebraic approach
A Basic Formalism: Typed Graphs

Directed graphs as algebraic structures
\[ G = (V, E, \text{src}, \text{tar}) \]
with
\[ \text{src}, \text{tar}: E \rightarrow V \]

Graph homomorphism as pair of mappings
\[ h = (h_V, h_E): G_1 \rightarrow G_2 \]
with
\[ h_V: V_1 \rightarrow V_2 \]
\[ h_E: E_1 \rightarrow E_2 \]

preserving \( \text{src} \) and \( \text{tar} \)

Typed graphs given by
- fixed type graph \( TG \)
- instance graphs \( G \) typed over \( TG \) by homomorphisms \( g: G \rightarrow TG \)

UML notation: \( x:t \) means \( x \in G \) with \( g(x) = t \in TG \)

Undirected edges: symmetric pairs of directed ones
Rules

\[ p: L \rightarrow R \text{ with } L \cap R \text{ well-defined, in different presentations} \]

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

\[ p: L \rightarrow R \text{ with } L \cap R \text{ well-defined, in different presentations} \]

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

\[ \text{movePM:} \]

\[
\begin{array}{c}
\text{pm: PacMan} \\
\text{f1: Field} & \text{f2: Field}
\end{array}
\]

\{destroyed\} \rightarrow \{new\}
Graph Transformation

1. select rule $\rho : L \rightarrow R$; occurrence $o_L : L \rightarrow G$
2. remove occurrence of $L \setminus R$ from $G$
3. add a copy of $R \setminus L$ to result graph
Question: What if …

… the deletion of vertices would cause “dangling edges”?  

- conservative answer: Don’t delete the vertex! [DPO dangling condition]
  - invertible transformations, no side-effects
- liberal answer: Delete adjacent edges as well! [SPO, PROGRES, Fujaba]
  - more complex behavior, requires explicit control

Gabriele Taentzer
Advanced Features: Rules

Dealing with unknown context:

- set-nodes (multi-objects): match all nodes with the required connections [NLC, PROGRES]
- explicit (negative) context conditions

(PacMan moves to a field with a set of marbles if there is no ghost)

Priorities:

- movePM only if collect is not possible

Programmed transformation:

- \((collect?; \text{movePM})^* \parallel (kill?; \text{moveGhost})^*\)
Pacman Extension: Give *Pacman* another chance

Let *Pacman* have a counter for his lives.

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

Gabriele Taentzer
Pacman Extension: Give *Pacman* another chance

Let *Pacman* have a counter for his lives.

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

Solution: add an attribute.

Gabriele Taentzer 24
Pacman Extension: Refine rule *kill*

Solution: match attribute value.

Solution: an attribute application condition.
Control of Rule Application

- Non-deterministic choice:
  - selection of next rule
  - selection of next match

- Restriction of non-determinism:
  - control structures, rule priorities, layers, etc.
  - input parameters define partial match
Chomsky Grammars: Rewriting of Strings

Production \( A \rightarrow aAb \) as (context-free) graph grammar
production: one vertex or edge in \( L \)

- Application to specifying and parsing of visual languages
- Theory of graph grammars as formal language theory for more-dimensional structures
  - hierarchies of language classes and grammars
  - decidability and complexity results
  - parsing algorithms

Gabriele Taentzer
Petri Nets: Rewriting of Multisets

A PT net transition as graph transformation rule

- Application to modeling and analysis of concurrent systems
- Concurrency theory for graph transformation
  - independence, causality, and conflicts
  - concurrency semantics
Term Rewriting: Rewriting of Trees or DAGs

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

- Application to implementation of functional languages and process calculi
- Theory of term graph rewriting
  - soundness and completeness w.r.t. TR
  - termination, confluence and critical pairs
Tool support
PROGRES
(PROOgrammed Graph REwriting Systems)

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

Gabriele Taentzer
AGG (Algebraic Graph Grammar System)

- Algebraic approach to graph transformation
- Attribute computation in Java
- Analysis tools
  - graph parsing
  - critical pair analysis
  - termination
- Easy integration with Java code (separate transformation engine)
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
- Dynamic Object Browser
- Reverse engineering

Gabriele Taentzer
ViaTra: VIsual Automated model TRAnsformation

- Model transformation by graph transformation
- Transformations by example
- Incremental pattern matching
Groove: Graphs for Object-Oriented Verification

- SPO approach
- Integrated rules
- Specification of graph transformation systems
- Computation of state spaces
- Model checking of graph transformation systems
Henshin: EMF Model Transformation

- Algebraic graph transformation concepts
- Applied to EMF models
- Reasoning on model transformations
  - Computation of state space plus model checking
  - Conflicts and dependencies based on AGG

Gabriele Taentzer
Further Graph Transformation-Based Tools

- GREaT
- AToM3
- VMTS
- GrGen
- Augur
- MOFLON
- ...
Summary

- Graph transformation: rule-based manipulation of graphs
  - Generalization of other rewrite techniques
- Several approaches to graph transformation:
  - Two basic concepts of graphs
  - Gluing versus embedding of the RHS
  - What to do with dangling edges?
  - Variety of rule application control mechanisms
- Variety of graph transformation tools
  - Specification and execution of graph transformations
  - Reasoning support
  - Code generation facilities
Exercise: Modeling of peer-to-peer architectures

We are discussing a P2P architecture such as used by the Skype IP telephony application. This application allows registered users to make voice-over-IP calls and send messages to other users while storing user and connectivity information in decentralized form, without the use of central infrastructure. Based on a detailed study of the Skype network, Skype network nodes are distinguished into clients and super nodes. Peers equipped with sufficient resources can be promoted to super nodes, thus assuming management functions, while continuing to serve as clients. Super nodes form an overly network amongst themselves, while clients have to connect to super nodes acting as servers. There is a maximal number of clients that can be handled by one super node. Termination of super nodes is allowed, but has to make sure that no client is left without a super node connection.
Exercise: Modeling of peer-to-peer architectures

Tasks:

- Specify an adequate type graph.
- Think about a concrete scenario for a Skype network and model it as instance graph which is correctly typed.
- Specify rules for joining the network and making connections as well as disconnecting and leaving the network. (Keep in mind that clients can become super nodes.)
- Think about how rules can be applied and describe a test sequence of rule applications.
Overview

- Set-theoretical definition of
  - Graphs, graph mappings and operations
  - Graph rules, matches and transformations

- Categories Graph and $\text{Graph}_{TG}$
  - Characterization of graph operations by pullbacks and pushouts
  - Double-pushout transformation

How is algebraic graph transformation really defined?
PacMan-Example

Formally?

movePM

pm: PacMan

f1: Field  f2: Field

pm: PacMan

f1: Field  f2: Field

: Ghost

: Field

: Field

: Field

: Field

: Field

: Field

: Field

: Field

: PacMan

: Field

: Field

: Field

: Field

: Field

: Field

: Field

: PacMan
Graph and subgraph

- **A graph is a tuple**
  
  $G = (G_V, G_E, \text{src}^G, \text{tar}^G)$
  
  with $\text{src}^G, \text{tar}^G : G_E \to G_V$.

- **A graph $G$ is a subgraph**
  
  of graph $H$, $G \subseteq H$, if
  
  $G_V \subseteq H_V$ and $G_E \subseteq H_E$,
  
  such that source and target nodes in $G$ come from $H$:
  
  $src^G = src^H|_{G_E}$
  
  $tar^G = tar^H|_{G_E}$. 

![Diagram of graphs G and H]
Graph morphism

A graph morphism $f$ from graph $G$ to graph $H$, $f: G \rightarrow H$, are two functions $f = (f_V, f_E): G \rightarrow H$ with

- $f_V: G_V \rightarrow H_V$
- $f_E: E_1 \rightarrow E_2$

preserving $\text{src}$ and $\text{tar}$, i.e.

- $\text{src}^H \circ f_E = f_V \circ \text{src}^G$
- $\text{tar}^H \circ f_E = f_V \circ \text{tar}^G$

Example:

- $g_V = \{f \rightarrow f1, f' \rightarrow f2\}$
- $g_E = \{c \rightarrow c1\}$
TG-typed graph

- A TG-typed graph $\langle G, tp^G \rangle$ is a graph $G$ and a graph morphism $tp^G: G \rightarrow TG$ to graph TG, called type graph.

- Example:
  
  $tp^G_V = \{ f \rightarrow Field, f' \rightarrow Field\}$
  
  $tp^G_E = \{ c \rightarrow c\}$

Integrated notion:
Categories **Graph** and **Graph\textsubscript{TG}**

- Graphs and graph morphisms form a category, called **Graph**.
- For each graph there is an identity morphism \( \text{id}_G : G \rightarrow G \).
- For each two graph morphisms \( f : G \rightarrow H \) and \( g : H \rightarrow K \) there is a graph morphism \( k : G \rightarrow K \) with \( g \circ f = k \).

\[
\begin{array}{ccc}
G & \xrightarrow{k} & K \\
\downarrow f & & \downarrow g \\
H & \xrightarrow{tpH} & K \\
\downarrow \text{tp}^G & & \downarrow \text{tp}^K \\
\text{tg} & \text{tg} & \text{tg} \\
\end{array}
\]

- **TG**-typed graphs and **TG**-typed graph morphisms form a category, called **Graph\textsubscript{TG}**.
- For each **TG**-typed graph there is an identity morphism \( \text{id}_G : \langle G, \text{tp}^G \rangle \rightarrow \langle G, \text{tp}^G \rangle \).
- For each two graph morphisms \( f : \langle G, \text{tp}^G \rangle \rightarrow \langle H, \text{tp}^H \rangle \) and \( g : \langle H, \text{tp}^H \rangle \rightarrow \langle K, \text{tp}^K \rangle \) there is a graph morphism \( k : \langle G, \text{tp}^G \rangle \rightarrow \langle K, \text{tp}^K \rangle \) with \( g \circ f = k \), \( \text{tp}^G = f \circ \text{tp}^H \) and \( \text{tp}^H = g \circ \text{tp}^K \).
A TG-typed graph $\langle G, tp^G \rangle$ is a subgraph of a TG-typed graph $\langle H, tp^H \rangle$, i.e. $\langle G, tp^G \rangle \subseteq \langle H, tp^H \rangle$, if $G \subseteq H$ and $tp^G = tp^H|_G$. 
Intersection of graphs

- The intersection (and union) of two graphs $G$ and $H$ is defined, if there is a graph $C$ such that $G$ and $H$ are subgraphs of $C$.

The intersection $G \cap H$ is a subgraph of $C$ defined by

$$(G \cap H)_V = G_V \cap H_V$$

and

$$(G \cap H)_E = G_E \cap H_E.$$
The intersection (and union) of two TG-typed graph $G$ and $H$ is defined, if there is a TG-typed graph $C$ such that $G$ and $H$ are TG-typed subgraphs of $C$.

The intersection $G \cap H$ is a subgraph of $C$ defined by

$$(G \cap H)_V = G_V \cap H_V$$

and

$$(G \cap H)_E = G_E \cap H_E.$$
Characterization of graph intersections by pullbacks

- Given \( g: G \rightarrow C \) and \( h: H \rightarrow C \), the intersection can be characterized by \( i_G: I \rightarrow G \) and \( i_H: I \rightarrow H \), since \( g \circ i_G = h \circ i_H \).
- If there are \( g': X \rightarrow G \) and \( h': X \rightarrow H \) with \( g' \circ g = h' \circ h \), there is a unique graph morphism \( x: X \rightarrow I \) such that \( x \circ i_G = g' \) and \( x \circ i_H = h' \) (universal property of pullbacks).

\[ \begin{array}{ccc}
G & \xrightarrow{g} & X \\
\downarrow{i_G} & \downarrow{=} & \downarrow{=} \\
I & \xrightarrow{x} & I \\
\downarrow{i_H} & \downarrow{=} & \downarrow{=} \\
H & \xrightarrow{h} & X
\end{array} \]

\( I \) is the largest graph which intersects \( G \) and \( H \) according to their inclusion in the common graph \( C \).
Union of graphs

- The union of two graphs $G$ and $H$ is defined, if there is a graph $C$ such that $G$ and $H$ are subgraphs of $C$.
- *Exercise*: Give two graphs $G$ and $H$ such that $C$ does not exist.

The union $G \cup H$ is the subgraph of $C$ which is defined by 

\[ (G \cup H)_V = G_V \cup H_V \]

and 

\[ (G \cup H)_E = G_E \cup H_E \]
Union of typed graphs

- The union of two TG-typed graphs $G$ and $H$ is defined, if there is a TG-typed graph $C$ such that $G$ and $H$ are TG-typed subgraphs of $C$.
- Exercise: Give two graphs $G$ and $H$ such that $C$ exists, without caring about typing, however $C$ as TG-typed graph does not exist.

The union $G \cup H$ is a subgraph of $C$ such that 
$$(G \cup H)_V = G_V \cup H_V$$
and 
$$(G \cup H)_E = G_E \cup H_E$$
Characterization of graph unions by pushouts

- Given $i_G: I \to G$ and $i_H: I \to H$, the union can be characterized by $g: G \to P$ and $h: H \to P$, since $g \circ i_G = h \circ i_H$.
- If there are $g': G \to X$ and $h': H \to X$ with $g' \circ i_G = h' \circ i_H$, there is a unique graph morphism $x: P \to X$ such that $x \circ g = g'$ and $x \circ h = h'$ (universal property of pushouts).

$P$ is the smallest graph which unifies $G$ and $H$ such that they overlap in the images of $I$. 
Graph transformation rule

- A graph transformation rule $p:L \Rightarrow R$ consists of a rule name and two graphs $L$ and $R$ with defined intersection.
- A TG-typed graph transformation rule $p:L \Rightarrow R$ consists of a rule name and two TG-typed graphs $L$ and $R$ with defined intersection.

![Diagram of graph transformation rule]
Application conditions for graph transformation rules

- Is any graph transformation rule applicable to any graph? No
- First condition:
  - There is a match from the left-hand side $L$ of a rule to instance graph $G$, i.e. there is a graph morphism $o:L \rightarrow G$.
- Further conditions:
  - *Problem 1*: Conflict preservation or deletion of node/edge? $\Rightarrow$ Identification Condition
  - *Problem 2*: Deletion of nodes can cause dangling edges $\Rightarrow$ Dangling Condition
  - Dangling Condition + Identification Condition = Gluing Condition
Example: Rule match

Instance graph $G$

Exercise:
How can $L$ be matched to graph $G$?
Gluing Condition

- **Gluing condition:** Match $o:L \rightarrow G$ satisfies the *gluing condition* for $p:L \Rightarrow R$, if the following two conditions are fulfilled:
  - **Identification condition:** Two elements $x$ and $y$ in $L$ are allowed to be mapped non-injectively by match $o$, if they preserved by rule $p$, i.e.
    
    $o(x)=o(y) \Rightarrow x=y$ or $x,y \in L \cap R$
  - **Dangling condition:** The deletion of all elements of $G$ which shall be deleted does not cause dangling edges in the instance graph, i.e.
    
    $\forall v \in o_V (L_V \setminus R_V)$:
    
    $\forall e \in G_E$: $\text{src}^G(e)=v$ or $\text{tar}^G(e)=v$
    
    $\Rightarrow e \in o_E (L_E \setminus R_E)$
    
    Hence: If a node is deleted, all its adjacent edges have to be deleted, too.
Example: Identification Condition

\[ o_V = \{v1 \rightarrow v, v2 \rightarrow v\} \]
\[ o_E = \{\} \]

Rule 1

Match \( o \) does not satisfy the Identification Condition,

since \( o_V(v1) = o_V(v2) \),

but \( v1 \neq v2 \) and \( v2 \in L \setminus R \)

Exercise: Are there matches of rules \textit{movePM}, \textit{collect}, \textit{kill} and \textit{moveGhost} to graph \( G \) such that the Identification Condition is not satisfied?
Example: Dangling Condition

Rule 2

\( o_V = \{ v \rightarrow v_1 \} \)

Match \( o \) does not satisfy the Dangling Condition, since \( src_G(e) = v_1 \), but \( v_1 \not\in o_V(L_V \cup R_V) \)

Exercise: Are there matches of rules movePM, collect, kill and moveGhost to graph G such that the Dangling Condition is not satisfied?
Construction of a direct graph transformation step

Given
- rule \( p: L \Rightarrow R \)
- instance graph \( G \)
- match \( o_L: L \rightarrow G \)

such that
- \( o_L \) satisfies the Gluing Condition for \( p \)
- \( G \cap (R \setminus L) = \emptyset \)

A (direct) graph transformation step \( G \xrightarrow{p(o)} H \) consists of two parts.

In the following, we assume that match \( o \) is injective.
Direct graph transformation: Deletion

Construct intermediate graph $D$ as subgraph of $G$ by
Deleting all matched elements of $L \setminus R$:

- $D_V = G_V \setminus o_L(L_V \setminus R_V)$
- $D_E = G_E \setminus o_L(L_E \setminus R_E)$
- $\text{src}^D = \text{src}^G|_{D_E}$
- $\text{tar}^D = \text{tar}^G|_{D_E}$

The Dangling Condition ensures that $D$ is a graph!
Direct graph transformation: deletion
Direct graph transformation: Insertion

Construct the result graph \( H \) by inserting a new copy of \( R \setminus L \) in \( D \) (i.e. \( G \cap (R \setminus L) = \emptyset \)):

\[
\begin{align*}
\mathcal{H}_V &= \mathcal{D}_V + (\mathcal{R}_V \setminus \mathcal{L}_V) \\
\mathcal{H}_E &= \mathcal{D}_E + (\mathcal{R}_E \setminus \mathcal{L}_E)
\end{align*}
\]

Definition of src and tar dependent on the edge origin (D or R):

\[
src^H(e) = \begin{cases} 
src^D(e), & \text{for } e \in \mathcal{D}_E \\
o_L(src^R(e)), & \text{for } e \in \mathcal{R}_E \text{ and } src^R(e) \in \mathcal{L} \\
src^R(e), & \text{otherwise}
\end{cases}
\]

Analog for tar.
Direct graph transformation: Insertion
Direct graph transformation

Let \( p: L \Rightarrow R \) be a graph transformation rule, \( G \) and \( H \) two instance graphs such that \( G \cap H \) is defined.

\( G \Rightarrow_{p(o)} H \) is a direct graph transformation iff there is a graph morphism \( o: L \cup R \rightarrow G \cup H \) with \( o_L = o|_L \) such that

- \( o(L) \subseteq G \) and \( o(R) \subseteq H \)
- \( o(L \setminus R) = G \setminus H \) and \( o(R \setminus L) = H \setminus G \)
Direct typed graph transformation

- Let \( p: L \Rightarrow R \) be a TG-typed graph transformation rule, \( G \) and \( H \) two TG-typed graphs such that \( G \cap H \) is defined.

\[ G \xrightarrow{p(o)} H \]

is a direct typed graph transformation iff there is a graph morphism \( o: L \cup R \rightarrow G \cup H \) with \( o_L = o|_L \) such that

- \( o(L) \subseteq G \) and \( o(R) \subseteq H \)
- \( o(L \setminus R) = G \setminus H \) and \( o(R \setminus L) = H \setminus G \)
Graph transformation as double pushout

- A graph transformation step $G \xrightarrow{\rho} H$ can be characterized by two pushouts in $\text{Graph}$ where $G$ and $H$ are pushout objects.
- Given rule $p$ and graph morphism $d$, transformation $G \xrightarrow{\rho} H$ is always defined and unique up to isomorphisms.
- Straight forward extensible to category $\text{Graph}_{\text{TG}}$
Graph transformation: Pushout complement construction

- Given rule \( p \) and match \( m \), the left pushout can be constructed as follows:
- Given \( l: K \rightarrow L \) and \( m: L \rightarrow G \), we construct \((D, d: K \rightarrow D, g: D \rightarrow G)\) such that
  - \( \text{PO1} \) is a pushout.
  (\(D,d,g\)) is called \( \text{PO-complement} \).
- If \( p \) and \( m \) fulfill the gluing condition, the \( \text{PO-complement} \) exists.
- Straight forward extensible to category \( \text{Graph}_{TG} \)
Graph transformation system and graph transformations sequence

- A graph transformation system $G = (TG, P)$ consists of
  - a type graph $TG$
  - a set $P$ of $TG$-typed rules

- A graph transformation (sequence) over $G$ is a sequence of direct graph transformation steps

\[
G_0 \xrightarrow{p_1(o_1)} G_1 \xrightarrow{p_2(o_2)} \cdots \xrightarrow{p_n(o_n)} G_n
\]

with
  - $G_0, \ldots, G_n$ being $TG$-typed instance graphs and
  - graph transformation rules $p_1, \ldots, p_n \in P$
Graph transformation and the Pacman game

- graph transformation system $G = (TG, P)$
- set of TG-typed rules $P$
- type graph $TG$

$G_{Pacman} = (TG, P)$

$P = \{movePM, collect, kill, moveGhost\}$

$TG$

- graph transformation over $G$

$movePM; collect; moveGhost; movePM; kill$
Negative application condition

\[ o = \{ pm \to p, f \to f2, f' \to f3 \} \]

does not satisfy the negative application condition.

\[ o = \{ pm \to p, f \to f2, f' \to f1 \} \]

satisfies the negative application condition.
Negative application condition

- A **negative application condition** over $L$ 
  
  $$N = \{N_1, \ldots, N_n\}$$

  is a set of **partial extensions** $N_i$ of $L$

  i.e. graphs $N_i$ such that $N_i \cap L$ is defined.

- Match $o: L \rightarrow G$ **fulfills** $N$ if for all $N_i \in N$ holds:
  
  There is **no injective** graph morphism $q_i: N_i \rightarrow G$

  such that $q_i = o$ for all elements in $N_i \cap L$, i.e.

  $$q_i|_{N_i \cap L} = o|_{N_i \cap L}.$$
Example: Negative application condition

- \( o = \{ pm \rightarrow p, f \rightarrow f_2, f' \rightarrow f_3 \} \) does not satisfy the negative application condition since \( q = \{ g \rightarrow g, f' \rightarrow f_3 \} \).

- \( o = \{ pm \rightarrow p, f \rightarrow f_2, f' \rightarrow f_1 \} \) satisfies the negative application condition, since \( g: \text{Ghost} \) cannot be mapped if \( f' \rightarrow f_1 \).
Outline: Typed attributed graph transformation

A typed attributed graph transformation rule is a typed graph transformation rule where the left and right-hand sides of the rule are attributed over data type algebra $T_\Sigma(X)$, the term algebra over variable set $X$.

A direct typed attributed graph transformation $AG \Rightarrow AH$ by rule $p$ and match $o_L: L \rightarrow AG$ is constructed over underlying typed graphs. $AG$ and $AH$ have the same data type algebra.

Attention: Match $o_L$ is a typed attributed graph morphism, i.e. the data type part of $o_L$ must be a $\Sigma$-morphism.
Summary

- Set-theoretical definition of graphs, subgraphs, union and intersection of typed graphs
- Set-theoretical definition of graph transformation
  - Graph transformation rule
  - Match and gluing condition
  - Transformation step
  - Negative application condition
- Characterization of graph operations by constructions from category theory
- Outlook: attributed graph transformation
Exercise: Category Graph

- Is there a graph from which there exists a graph morphism to any other graph?
- To which graph does there exists a graph morphism from any other graph?
Exercise: Monomorphisms in category Graph

Please show:

- An injective graph morphism $f: H \to K$ is a monomorphism in category **Graph**, i.e. for all graph morphisms $g, h: G \to H$ the following holds:
  \[ f \circ g = f \circ h \text{ implies } g = h \]

- Tipp: A function $f: M \to N$ is injective if for all $x, y$ in $M$: $f(x) = f(y)$ implies $x = y$
Exercise: Union of graphs as pushout in category \textbf{Graph}

Please choose a concrete union of graphs and argue why it is a pushout in category \textbf{Graph}.
Exercise: The gluing condition for P2P-rules

- Consider your graph transformation system for P2P-architectures. Choose pairs of rule and match such that
  - the gluing condition is fullfilled
  - the gluing condition is not fullfilled
Reasoning about Graph Transformation

March 11, 2011
Overview

- Conflicts and causal dependencies of graph transformations
- Critical pairs and local confluence
- Termination criteria
- Application to model transformation
  - Functional behavior of model transformations
**Example: Parallel Independence**

**Example 1:** Two ghosts can kill *Pacman*. If one ghost kills Pacman, the other ghost cannot kill him anymore.

Where is the conflict?

**Example 2:** Between two *ghosts* there is a free field. Both ghosts can move to this free field.

Are these transformations executable in parallel?
Definition: Parallel Independence

- A direct graph transformation $\text{tr}_1: G \xrightarrow{p_1(o_1)} H_1$ is *weakly parallel independent* of $\text{tr}_2: G \xrightarrow{p_2(o_2)} H_2$ if
  - $\text{tr}_2$ preserves match $o_1(L_1)$ of rule $p_1$ ($o_1(L_1)$ does not overlap with elements being deleted by $p_2$.)
  - $\text{tr}_2$ does not create elements which are prohibited by a negative application condition of $p_1$.
  - $\text{tr}_2$ does not change attributes used by $o_1(L_1)$.

- Two direct graph transformations are *parallel independent* if $G \xrightarrow{p_1(o_1)} H_1$ is weakly independent of $G \xrightarrow{p_2(o_2)} H_2$, and vice versa.

- Direct graph transformations are *in conflict* if they are not parallel independent.
Example: Sequential Independence

Example 1: A ghost stands on a field beside Pacman. It moves to the same field and kills Pacman.

Why is the second transformation sequentially dependent on the first one?

Example 2: Between two ghosts there is a free field F2. First ghost G1 moves to field F2 and thereafter ghost G2 moves there.

Why are these transformations sequentially independent?
Definition: Sequential Independence

- Two direct graph transformations \( G \xrightarrow{p_1(o_1)} H_1 \xrightarrow{p_2(o_2)} X \) are **sequentially independent** if match \( o_2(L_2) \) is already existing before \( tr_2 \) has been applied, i.e.
  - \( o_2(L_2) \) does not overlap with elements inserted by \( p_1 \) and
  - negative application conditions of \( p_2 \) are not satisfied by deletions performed by \( tr_1 \).

- \( tr_2 \) preserves match \( o_1(L_1) \) of rule \( p_1 \), i.e.
  - \( tr_2 \) does not delete elements used by \( tr_1 \) and
  - *negative application conditions of rule \( p_1 \) are not satisfied by deletions of \( tr_2 \).*
Local Church-Rosser Theorem

1. Let $G \xrightarrow{p_1(o_1)} H_1$ and $G \xrightarrow{p_2(o_2)} H_2$ be parallel independent direct graph transformations, then there are transformations $H_1 \xrightarrow{p_2(o_2)} X$ and $H_2 \xrightarrow{p_1(o_1)} X$ such that $G \xrightarrow{p_1(o_1)} H_1 \xrightarrow{p_2(o_2)} X$ and $G \xrightarrow{p_2(o_2)} H_2 \xrightarrow{p_1(o_1)} X$ are sequentially independent.

2. Let $G \xrightarrow{p_1(o_1)} H_1 \xrightarrow{p_2(o_2)} X$ be sequentially independent direct transformations, then there are sequentially independent direct transformations $G \xrightarrow{p_2(o_2)} H_2 \xrightarrow{p_1(o_1)} X$ such that $G \xrightarrow{p_1(o_1)} H_1$ and $G \xrightarrow{p_2(o_2)} H_2$ are parallel independent.
Critical Pairs

Problem: How can we find conflicts and causal dependencies of graph transformation systems before it is too late?

Solution: We analyse conflicts not during run time, but search for potential conflicts statically.

Formal concept: Critical pairs of graph transformation systems can be found and analysed statically. They present potential conflicts in minimal contexts.
Definition: Critical Pair

- A **critical pair** is a pair of direct graph transformations $G \xrightarrow{p_1(o_1)} H_1$ and $G \xrightarrow{p_2(o_2)} H_2$ in conflict such that graph $G$ is *minimal*, i.e.
  - an overlapping $o_1(L_1) \cup o_2(L_2)$ of the matches of rules $p_1$ and $p_2$ or
  - an overlapping of a match and a part of a negative application condition.

- *The set of all critical pairs is finite.*
  - An overlapping is interesting only if it contains at least one element which is deleted by one of the rules and both rules are applicable to $G$. 

Gabriele Taentzer
Example: Critical Pairs

- How many overlapping graphs of \( L \) and \( R \) exist, i.e. how many critical pairs exist for rules \((\text{kill}, \text{kill})\)? How do these overlapping graphs look like? Which of these critical pairs can be resolved, i.e. can be transformed to a common graph?
Computing critical pairs with AGG

A discussion about the significance of certain critical pairs is needed. For example, some critical pairs show essentially the same conflict situation. The number of critical pairs gets smaller if the type graph contains additional constraints, e.g. multiplicities.
Completeness of Critical Pair Set

- The set of critical pairs contains all potential conflicts, since there is a critical pair iff there is a direct transformation via $p_1$ in conflict with another via $p_2$.

- Reason: Each pair of direct graph transformations in conflict contains a critical pair such that the overlapping graph is the overlapping of the matches of both direct graph transformations.
Definition: Local Confluence

- A pair of direct graph transformations $G \xrightarrow{p_1(o_1)} H_1$ and $G \xrightarrow{p_2(o_2)} H_2$ is *locally confluent* if there are a graph $X$ and graph transformations $H_1 \xrightarrow{*} X$ and $H_2 \xrightarrow{*} X$.

- A graph transformation system is *locally confluent* if all direct graph transformations are *locally confluent*.

Do you know an example for locally confluent transformations?
Example: Local confluence

Are the direct graph transformations where *Pacman* moves left or right, locally confluent?
Criteria for Local Confluence

- **Strong confluence** (informal): A critical pair is *strongly confluent* if
  - it is locally confluent and
  - the graph part which is preserved by both transformations $G \xrightarrow{p_1(o_1)} H_1$ and $G \xrightarrow{p_2(o_2)} H_2$ of a critical pair, is preserved also by $H_1 \xrightarrow{*} X$ and $H_2 \xrightarrow{*} X$.

- **Critical Pair Lemma**: A graph transformation system is *locally confluent* if all critical pairs are *strongly confluent*. 

Gabriele Taentzer 95
Criteria for Confluence

For replacement systems we have in general:

- A replacement system is confluent if it is *locally confluent* and it *terminates*.

- $\Rightarrow$ A graph transformation system is *confluent* if it is *locally confluent* and it *terminates*.
Application to Model Transformation
Model Transformations in SE

- Analysis model
- Design model
- Code
- Code generation
- Reverse engineering
- Formal model
- Validation
- Forward engineering
- Refactoring

Gabriele Taentzer

98
Correct Model Transformations

- Functional behaviour:
  - There is always a result, i.e. the model transformation terminates.
  - The result is unique.

- Syntactical consistency:
  - The result is an element of the target language.

- Semantical consistency:
  - The semantics of the resulting model corresponds to that of the source model.
Outline

- Example: From Activity Diagrams to Petri Nets
- Model Transformation by Graph Transformation
- Analysis of Graph Transformation Systems
  - Local confluence by critical pairs
  - Termination criteria
- Analysis of Example
- Further Case Studies
- Implementation in AGG
- Related Work and Conclusion
Example: Activities → Petri nets

Gabriele Taentzer
How to transform a model?

- Models are represented by abstract syntax graphs.
- Source and target languages by type graphs +
  - constraints (descriptive)
  - graph grammars (constructive)
- Model transformation is performed stepwise by graph transformation.
Abstract Syntax Graphs

Diagram:
- receive order → check availability
- check availability → [product not available] → notify client → send receipt
- check availability → [product available] → calculate price

Underlying Syntax Graph:
- Activity name="check availability" kind=simple
  begin
  Next inscr=""
  end
- Activity name="notify client" kind=simple
  begin
  Activity name="calculate price" kind=simple
  begin
  Next inscr="product available"
  end
  Activity name="send receipt" kind=simple
  begin
  Activity name="product not available"
  end
  Activity
  begin
Model Transformation by Graph Transformation

Language 1

Type graph 1

G1

Type graph 12

Gi

Type graph 2

Gn

Language 2
Transformation: Activites → Petri nets

source language:
activity diagrams

helper structure

target language:
Petri nets

Gabriele Taentzer
Activities → Petri Net Transformation

Start2Trans

Choose a Train

[no train chosen]

[train chosen]

Get Set of Tracks

Gabriele Taentzer
Activities → Petri Net Transformation

Simple2Trans

Choose a Train

[no train chosen]

Get Set of Tracks

[no train chosen]

Get Set of Tracks

Gabriele Taentzer
Activities → Petri Net Transformation

Dec2Transitions

1: Activity
   kind="decision"
   name=n

2: Next
3: end

Start

Choose a Train

[no train chosen]

[train chosen]

Get Set of Tracks

Place
   token=""
   name=""

Transition
   name=""

ArcPT
   arcPTinscr=""

Dec2Transitions

Gabriele Taentzer
Activities → Petri Nets Transformation

Choose a Train

[no train chosen]

Get Set of Tracks

[no train chosen]

[train chosen]

Gabriele Taentzer
Activities → Petri Nets Transformation

Choose a Train

Get Set of Tracks

[no train chosen]

[train chosen]

Create Arc

Start

Choose a Train

[no train chosen]

[train chosen]

Get Set of Tracks

Gabriele Taentzer
Correct Model Transformations

- **Functional behaviour:**
  - There is always a result, i.e. the model transformation terminates.
  - The result is unique.

- **Syntactical consistency:**
  - The result is an element of the target language.

- **Semantical consistency:**
  - The semantics of the resulting model corresponds to that of the source model.
Conflicts, Confluence and Termination

- **Critical pairs** represent potential conflicts in a minimal context. They can be found by static analysis of the graph transformation system.

- A graph transformation system is **locally confluent**, if all critical pairs are confluent.

- A graph transformation system is **confluent**, if it is locally confluent and it terminates.

- **Termination** of a graph transformation system is undecidable in general. We give sufficient conditions for the termination of layered graph transformation systems.
Termination

- **Termination** of a graph transformation system is undecidable in general.
- A graph transformation system terminates if every graph transformation sequence is finitely long.
- Proof idea: layered graph transformation systems
  - Splitting of rule set in layers
  - Assignment of creation and deletion layers to each node and edge type
    - *Deletion layer*: Each rule deletes at least one element and creates elements of next layer types.
    - *Non-deletion layer*: Each rule cannot be applied at the same match more than once.
Termination Criteria

Layered graph transformation system:

- A graph transformation system with rules \( RULE \) and types \( TYPE \) where each rule \( r \in RULE \) has a layer \( rl(r) = k \) with \( 0 \leq k \leq k_0 \) \( (k,k_0 \in N) \) with

- \( k_0 : \text{number of layers} \) and

- for each type \( t \in TYPE \) there are creation and deletion layers \( cl(t), dl(t) \in N \) with \( cl(t) < dl(t) \).

Example: Activities → PN

Rule Layers:

- Layer 0:
  - translates node elements (start, simple activity, decision, end)

- Layer 1:
  - translates transitions

- Layer 2:
  - deletes the source model
Termination Criteria

Given a layered graph transformation system, each layer $k$ is either a deletion or non-deletion layer:

- **Deletion Layer**
  - Rule $r$ deletes at least one graph item.
  - $0 \leq cl(t) \leq dl(t) \leq k_0$ for all types $t \in \text{TYPE}$
  - Rule $r$ deletes $t \implies dl(t) \leq rl(r)$
  - Rule $r$ creates $t \implies cl(t) > rl(r)$

- **Non-deletion Layer**
  - Rule $r$ is non-deleting, i.e. $r: L \rightarrow R$ is total and injective
  - Rule $r$ has NAC $n: L \rightarrow N$ and there is an injective $n': N \rightarrow R$ with $n' \circ n = r$
  - $x \in L \implies cl(\text{type}(x)) \leq rl(r)$
  - Rule $r$ creates $t \implies cl(t) > rl(r)$
Termination of Layered Graph Grammars

- **Theorem:**
  Each layered graph grammar with injective matches terminates.

- **Proof idea:**
  - **Deletion layer:** Each rule deletes at least one item and can create items of higher layers only.
  - **Non-deletion layer:** Each rule can be applied at most once at the same essential match (due to special NAC form and creation of higher layer items only).
**Example: Layer Assignments**

\[
\begin{align*}
cl(l) &= \text{if } l \in T_0 \text{ (source type graph) then } 0 \text{ else } \max\{ rl(r) \mid r \text{ creates } l \} + 1 \\
dl(l) &= \text{if } l \text{ is deleted by some } r \text{ then } \min\{ rl(r) \mid r \text{ deletes } l \} \text{ else } k_0
\end{align*}
\]

<table>
<thead>
<tr>
<th>Src type</th>
<th>cl</th>
<th>dl</th>
<th>Ref type</th>
<th>cl</th>
<th>dl</th>
<th>Tgt type</th>
<th>cl</th>
<th>dl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>0</td>
<td>2</td>
<td>RefAct</td>
<td>1</td>
<td>2</td>
<td>Place</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Next</td>
<td>0</td>
<td>2</td>
<td>RefNext</td>
<td>1</td>
<td>2</td>
<td>Transition</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ArcTP</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ArcPT</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Example: Termination

- Rule layer: 1
- Activity, Next: $cl = 0, dl = 2$
- RefAct: $cl = 1, dl = 2$
- RefNext: $cl = 1, dl = 2$
- Transition: $cl = 1, dl = 2$
- Place: $cl = 1, dl = 2$
- ArcTP $cl = 2, dl = 2$

**Non-deletion Layer**
- Rule $r$ is non-deleting
- $r$ has NAC $n:L \rightarrow N$ and there is an inj. $n':N \rightarrow R$ with $n' \circ n = r$
- $x \in L \Rightarrow cl(label(x)) \leq rl(r)$
- $r$ creates $l \Rightarrow cl(l) > rl(r)$
Tool support: AGG
Conclusion

- Model transformation by graph transformation is
  - intuitive and precise
  - independent of the source model structure
  - especially suited for visual models

- Correct model transformation
  - type consistent transformation
  - sufficient termination criteria for model transformation
  - critical pair analysis to find potential conflicts

- Tool support AGG: http://tfs.cs.tu-berlin.de/agg

Gabriele Taentzer
Outlook

- **Application to the Eclipse Modeling Framework:**
  EMF model transformations can be defined as restricted graph transformations.
  - graphs with special containment edges, no multiple containment, no containment cycles
  - verification of EMF model transformations

- **Amalgamated graph transformation:**
  - apply kernel rule once and multi rules as often as possible, kernel rule is subrule of each multi rule
  - amalgamated rule reflects individual application situation of multi rules, matches of multi rules overlap in kernel match
  - lift theory on confluence and termination
Exercise:
Analysis of your P2P-example

- Give examples for parallel dependent and independent graph transformations.
- Give examples for sequentially dependent and independent graph transformations.
- Analyse your P2P-graph transformation system concerning conflicts and dependencies of rules.
- Is there a layering such that the termination criteria for layered graph transformation systems are fulfilled? Does your P2P-example terminate?
Exercise: Model transformation from class models to RDBM

- Design a model transformation which translates a class model to a relational data base schema in the following sense:
  - Class to table
  - Attribute to column (primary attribute to key)
  - Association to column where the target end is referred to by the corresponding key
  - No inheritance, all classes persistent

- Analyse the functional behavior of your model transformation system.
Second part of the lecture

Student presentations to following subjects:

- Model-based software development
  - Verifying consistency of refined use case models [HHT02]
  - Matching service specifications [HHL04,JLMTW09]
  - Stochastic Analysis of Peer-to-Peer Architecture [Hec05]
  - Aspect-oriented model weaving [MMT08]
  - Model Checking of Graph Transformation Systems [RSV04]

- Domain-Specific Language Engineering
  - Syntax definition and editor generation [EEHT05]
  - Model-driven software refactoring [MTR07,MTM08]
  - Model-to-model transformations [EEHT07,ABJKT10]