4. How to Transform Models with Graph Rewriting

1) Graph Transformations

2) Programmed Graph Rewriting

Prof. Dr. U. Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
Gruppe Softwaretechnologie
http://st.inf.tu-dresden.de
Version 15-1.1, 02.01.16

Part of ST2 and MOST
Obligatory Reading

- Jazayeri Chap 3. If you have other books, read the lecture slides carefully and do the exercise sheets
  - http://www.springerlink.com/content/5742246115107431/
- www.fujaba.de www.moflon.org
Obligatory Literature

- **View models (Wikipedia)**

Further Reading

- **Reducible graphs**

- **Search for these keywords at**
  - http://scholar.google.com
  - http://citeseer.ist.psu.edu
  - http://portal.acm.org/guide.cfm
  - http://ieeexplore.ieee.org/
  - http://www.springer.com/computer?SGWID=1-146-0-0-0
The Problem: How to Master Large Models in Requirements Engineering, Design, Domain Modeling

- Large models have large graphs
- They can be hard to understand -> models must be simplified
  - By transformation
  - By refactoring (behavior-preserving transformation)
  - By structurings

- To this end, we apply graph analysis and rewriting

- Figures taken from Goose Reengineering Tool, analysing a Java class system [Goose, FZI Karlsruhe]
Typical Problems in Software Engineering

• Question: How to Treat the Models of a big Swiss Bank?
  - 25 Mio LOC
  - 170 terabyte databases

• Question: How to Treat the Models of a big Operating System?
  - 25 Mio LOC
  - thousands of variants

• Requirements for Modelling in Requirements and Design
  - We need automatic structuring methods
  - We need help in restructuring by hand...

• Motivations for structuring
  - Getting better overview
  - Comprehensibility
  - Validatability, Verifiability
4.1 Graph Transformations
Model and Code Transformations in General

- Model and code transformations, such as refactorings, lowerings, higherings, optimizers, and other transformations can be specified by graph transformations [Mens]

<table>
<thead>
<tr>
<th></th>
<th>Horizontal</th>
<th>Vertical</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Endogeneous</strong></td>
<td>Structurings, Refactorings</td>
<td>Syntactic and semantic refinement</td>
</tr>
<tr>
<td>(within one language)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Exogeneous</strong></td>
<td>Language migration</td>
<td>Generation of platform-specific models (PSM)</td>
</tr>
<tr>
<td>(crossing languages)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Generation of platform-specific implementation (code generation)</td>
</tr>
</tbody>
</table>
Idea: Structure the Software Systems With Graph Rewrite Systems

- Graph transformations can be specified by graph rewrite systems (GRS)
  - Or by a programming language, of course
- If a graph transformation only manipulate edges, it can be described with Edge Addition Rewrite Systems (EARS)
- Otherwise, we arrive at general GRS
  - Transformation of complex structures to simple ones
  - Structure complex models and systems
Graph Rewrite Systems

- **A graph rewrite system** $G = (S)$ consists of
  - A **set of rewrite rules** $S$
    - A rule $r = (L, R)$ consists of 2 graphs $L$ and $R$ (left and right hand side)
    - Nodes of left and right hand side must be identified to each other
    - $L = \text{“Mustergraphen”} ; R = \text{Ersetzungsgraph”}$
  - An **application algorithm** $A$, that applies a rule to the manipulated graph
    - There are many of those application algorithms...

- **A graph rewrite problem** $P = (G, Z)$ consists of
  - A graph rewrite system $G$
  - A start graph $Z$
  - One or several result graphs
  - A derivation under $P$ consists of a sequence of applications of rules (direct derivations)

- **GRS offer automatic graph rewriting**
  - A GRS applies a set of Graph rewrite rules until nothing changes anymore (to the fixpoint, chaotic iteration)
  - Problem: Termination and Uniqueness of solution not guaranteed
Constant Folding as Graph Rewrite Rule

1 + 2 → Const

Plus

Const

Const

1

2

3

Const
Application of a Graph Rewrite Rule

- **Match** the left hand side: Look for a subgraph $T$ of the manipulated graph: look for a graph morphism $g$ with $g(L) = T$
- **Evaluate side conditions** of the left hand side
- **Evaluate right hand side**
  - Delete all nodes and edges that are no longer mentioned in $R$
  - Allocate new nodes and edges from $R$, that do not occur in $L$
- **Embedding**: redirect certain edges from $L$ to new nodes in $R$
  - Resulting in $S$, the mapping of $g(R)$
- **Evaluate side actions**
  - Assign attributes to nodes

\[
\begin{align*}
\text{L} & \quad \xrightarrow{\text{I}} \quad \text{R} \\
T = g(L) & \quad \xrightarrow{\text{g}} \quad S = g(R) \\
\text{Is replaced to} & \quad \text{manipulated Graph}
\end{align*}
\]
The Firing Rule of Petri Nets is a Graph Rewriting Rule

- Train arrived
  - Passenger at station
  - embarkment
  - Passenger on train
The Firing Rule of Petri Nets is a Graph Rewriting Rule

- Tokens can be modeled as special nodes attached to places
- The application of the rewrite rule models an event
Different Kinds of Graph Transformation Systems

• **Automatic** Graph Rewriting
  – Iteration of rules until termination

• **Graph Reduction**: Reducing a graph; rewrite system only has reductive rules

• **Programmed** Graph Rewriting: The rules are applied by a control flow program. This program guarantees termination and selects one of several solutions
  – Examples: PROGRES from Aachen/München
  – Fujaba on UML class graphs, from Paderborn, Kassel www.fujaba.de
  – MOFLON from Darmstadt www.moflon.org

• **Strategic** Graph Rewriting:
  – The rules are applied by strategies, higher-order functions and recursion strategies, such as bottom-up / top-down

• **Graph grammars** (Graph Recognition)
  – Special variant of automatic graph rewrite systems
  – Graph grammars contain in their rules and in their generated graphs special nodes, so called non-terminals
  – A result graph must not have non-terminals
  – In analogue to String grammars, derivations can be formed and derivation trees
Simpler Forms of Transformation Systems: TRS and Context-Sensitive TRS

• **Term rewriting** replaces terms (ordered trees)
  - right and left hand sides are unordered trees or (ordered) terms
  - Application: XML terms

• **Ground** term rewrite systems, GTRS: only ground terms in left hand sides
  - A GTRS always works bottom-up on the leaves of a tree
  - For GTRS there are very fast, linear algorithms

• **Variable** term rewrite systems, VTRS: terms with variables
  - Replacement everywhere in the tree

• **Dag** rewrite systems (DAGRS)
  - If a term contains a variable twice (non-linear), it specifies a dag
  - Dag rewrite systems contain dags in left and right hand sides (non-linear term rewriting)

• **Context-sensitive** Term Rewriting
  - e.g., remote-attribute controlled-rewriting (RACR)
  - Analyse the context and rewrite trees based on context attributes
4.2 Programmed Graph Rewriting
MOFLON and Fujaba

- MOFLON and Fujaba embed graph rewrite rules into activity diagrams (aka storyboards)
  - A rule set executes as an atomic activity
  - Colors express actions

![Diagram of an activity diagram showing the process `Sum::normalize()` with inputs and outputs, including terms such as `i1:InPort`, `i2:InPort`, `i3:InPort`, `o1:OutPort`, `o2:OutPort`, `I1:Line`, and `s1:Sum` with actions such as `create` and `destroy`.]

[Moflon homepage http://www.moflon.org]
[Fujaba homepage http://www.fujaba.de]
Storyboards are Procedures (Activity Diagrams) with Graph Rewrite Rules In Activities
Analyzer::isIncludeStable (f: File): Boolean ↔

- **true**
- **false**
Works on graphs typed by metamodels, specified in MOF
Other Software Engineering Applications

- Modeling state based systems (such as Petrinets)
- Mapping a PIM to a PSM in Model-Driven Architecture (transformational design)
- Graph Structurings (for simplification of models)
- Refactorings (see also Course DPF)
- Semantic refinements (transformational design)
- Round-Trip Engineering (RTE)
The End: What Have We Learned

- Graph rewrite systems are tools to transform graph-based models and graph-based program representations.
- Petrinets are simple graph rewrite systems.
- Graph rewriting can be automatic, programmed, strategic, reductive, recognizing.
04.A.1
PROGRES, the GRS tool from the IPSEN Project

- PROGRES has been the first tool to model graph algorithms by graph rewriting
- Textual and graphical editing
- Code generation in several languages

query ConsistentConfiguration( out Clause : string ) =

("A configuration is consistent if:
(1) it contains a variant of the system's main module,
(2) it contains a variant for any module which is needed by another included variant, and
(3) it does not contain variants which are not needed by needed variants.
")

use LocalName: string do
  ConfigurationWithMain( out LocalName )
  & not UnresolvedImportExists( LocalName )
  & not ConfigurationWithUselessVariant( LocalName )
  & Clause := LocalName
end

test ConfigurationWithMain( out Clause : string ) =

end:

test UnresolvedImportExists( Clause : string ) =

e
end:

test ConfigurationWithUselessVariant( Clause : string ) =

end:
This example illustrates the possibilities of PROGRES to define *parametrized productions* which must be instantiated (in the sense of a procedure call) with actual attribute values and node types. In this way, a single production may abstract from a set of productions which differ only with respect to used attribute values and types of matched or created nodes. In almost all cases, node type parameters are not used for matching purposes, but provide concrete types for new nodes of the right-hand side.

```
production CreateModule ( Name : string; InterfaceDescription : file; MType : type in MODULE ) =

    l : System contains
        l : MODULE

    name ( MName )

end:

production DeleteModule ( Name : string ) =

    l : MODULE

    name ( MName )

has

    2 : VARIANT

end:

production ChangeModuleType ( Name : string; NewType : type in MODULE ) =

    l : MODULE

    name ( MName )

end:

embedding redirect -n_user-, -has-, -contains-, -n_user-, -v_used- from l to l'

transfer l'.Name := l.Name;
    l'.File := l.File;

end:
```

Fig. 12: Specification of basic graph transformations
Fig. 14: Specification of additionally needed complex productions
Type Scheme of a Graph

Fig. 5: The graph schema of MIL graphs (without derived relationships)

- Boxes with round corners represent node types which are connected to their uniquely defined classes by means of dashed edges representing “type is instance of class” relationships; the type ADTNode belongs for instance to the class MODULE.
- Solid edges between node classes represent edge type definitions; the edge type vUses is for instance a relationship between VARIANT nodes and MODULE nodes and mUses edges connect MODULE nodes with other MODULE nodes.
- Circles attached to node classes represent attributes with their names above or below the circle.