Advanced Topics in Software Engineering (02265)

Ekkart Kindler

DTU Compute
Department of Applied Mathematics and Computer Science
V. Transformations
Overview

1. Model to Text Transformation (M2T) (JET → last week)

2. Model to Model Tranformation (M2M) (TGG → today)

3. Other approaches (QVT, ...)

4. Overview and classification

→ Is there a fundamental difference between M2T and M2M?

→ Differences between different M2M technologies!
2.1. Motivation

Up to now:
- Use of modelling notations (M1)!
- Development of modelling notation (M2)!
- Generate code (or other text) from models!

How do we
- Transform one model into another?
- Transform changes in the target model back to the source model?
- Keep different models consistent (synchronization)?
- Identify changes and transfer them to other models? (Version control, Identify design decisions, ... )

Is there a difference to a transformation to text?
2.2. Background

- Grammars, a reminder!
  (and two different purposes)

- Graph grammars
  (same thing just with graphs)
Example: Grammar

\[
\begin{align*}
E & \rightarrow T \mid E + T \\
T & \rightarrow F \mid T * F \\
F & \rightarrow I \mid N \mid (E) \\
I & \rightarrow a \mid \ldots \mid z \mid Ia \mid \ldots \mid Iz \\
N & \rightarrow 0 \mid \ldots \mid 9 \mid N0 \mid \ldots \mid N9
\end{align*}
\]

Expression

Term

Factor

Identifier

Number
Example: Grammar

E \rightarrow T \mid E + T

T \rightarrow F \mid T \ast F

F \rightarrow I \mid N \mid (E)

I \rightarrow a \mid \ldots \mid z \mid Ia \mid \ldots \mid Iz

N \rightarrow 0 \mid \ldots \mid 9 \mid N0 \mid \ldots \mid N9

Terminal symbols: +, *, (,), a, ..., z, 0, ..., 9

Non-terminal symbols: E, T, F, I, N

Left-hand side (of a rule)

Right-hand side (of a rule)

Identifier

Number

Meta-symbols: \rightarrow, \mid

Meta-meta-symbol: \ldots
Example: Parsing

\[ x + y \ast (x + 1) \]

\[
E \rightarrow E + T \rightarrow T + T \rightarrow F + T \rightarrow I + T \rightarrow x + T \rightarrow \\
\hspace{1cm} x + T \ast F \rightarrow x + F \ast F \rightarrow x + I \ast F \rightarrow \\
\hspace{1cm} x + y \ast F \rightarrow x + y \ast (E) \rightarrow x + y \ast (E + T) \rightarrow \\
\hspace{1cm} x + y \ast (T + T) \rightarrow x + y \ast (F + T) \rightarrow \\
\hspace{1cm} x + y \ast (I + T) \rightarrow x + y \ast (x + T) \rightarrow \\
\hspace{1cm} x + y \ast (x + F) \rightarrow x + y \ast (x + N) \rightarrow \\
\hspace{1cm} x + y \ast (x + 1) 
\]
Grammars

- A grammar consists of
  - rules and
  - one axiom (a non-terminal)
- A rule says how, within a character sequence, a sub-sequence can be replaced by another sequence
  Technically, $E \rightarrow T \mid E + T$ represents two rules!
- Grammars are often restricted to character sequences of non-terminals on the left-hand side
- In context-free grammars (CFG), the left-hand side consists of exactly one non-terminal symbol
Grammars: Use 1

- can be used to define the legal syntax of a programming language or some other textual language
- can be used to build parsers and for building the syntax tree

→ Formal languages
→ Compiler construction
→ Parsing theory

→ Xtext

- In this sense, "grammars are meta-models" or "meta-models are grammars"
Another example: "Grammar"

What does that "grammar" do?

#< \rightarrow #>

a< \rightarrow <a

>\ a \rightarrow a>

>\# \rightarrow <\#
Another example:

\[
\begin{align*}
#\rightarrow & \text{aaaaaa}\# & \rightarrow & #a\rightarrow \text{aaaaa}\# & \rightarrow & #aa\rightarrow \text{aaaa}\# & \rightarrow & #aaa\rightarrow \text{aaa}\# \\
#\text{aaaa}\rightarrow & \text{aa}\# & \rightarrow & #\text{aaaaaa}\rightarrow & \#\text{aaa}\rightarrow & \text{aaa}\# & \rightarrow & #\text{aaaaaa}\leftarrow & \#\text{aaa}\rightarrow & \text{aaa}\# & \rightarrow & #\text{aaaaaa}\leftarrow & \#\text{aa}\rightarrow & \text{aaaa}\# & \rightarrow & #\text{a}\rightarrow & \text{aaaa}\# & \rightarrow & #\leftarrow & \text{aaaaaa}\# & \rightarrow & #\leftarrow & \text{aaaaaa}\# & \rightarrow & ... \\
\end{align*}
\]
"Grammar": Use 2

- No axiom (just a "start configuration")

- No distinction between terminals and non-terminals (conceptually, all symbols can be considered to be terminals; technically, all symbols can be considered to be non-terminals)

- The purpose is not parsing a string; it is about "defining behaviour"; the string is just the current state (→ Markov algorithms)

Traditionally, the "algorithms" would be required to terminate; but, if they don’t, it just defines infinite behaviour (reactive systems).

In this context, the “grammar” is typically called “rewriting system”.
Background

- Grammars, a reminder! (and two different purposes)

- Graph grammars (same thing just with graphs)
Reminder: Firing rule of Petri nets
Firining rule of a Petri net transition

Example

Idea: Replace a subgraph with another one!

What is the same on the left-hand and the right-hand side?
Example

Firing rule of a Petri net transition as a graph grammar rule

Could be indicated by a mapping (also between the arcs). For humans “mostly obvious” 😊
Different representation: single graph, indicating in colours (and labels) what does not change, what is deleted (--) and what is added (++):

This is “Use 2” of graph grammars (defining evolving behaviour).
Use 1:

Defining the syntax of Petri nets:

Axiom: 

Rule 1: 

Rule 2: 

Note: In the tool that we are using, all nodes of the axiom will be “green” (+) nodes. Which interpretation makes more sense, depends on whether you want to consider the axiom as a rule or as a graph.
Use 1:

Defining the syntax of Petri nets

Rule 1:

Rule 2:

Rule 3:

Rule 4:

Rule 5:

Note that this is not the (main) purpose of GGs here; the example should just illustrate the “Use 1” of GGs.
2.3. Triple Graph Grammars (TGGs)

- Using graph grammars for defining the relation between models (in a special way),
- for transforming them accordingly, and
- keeping the resulting models consistent.
Outline

- Example
- Semantics
- Strength
- Problems and Weaknesses
- Extensions and Open Issues
An Example

engineer
„practice“

formal methods
„theory“

Borrowed from ComponentTools
This is very similar to project 1.
Corresponding Petri Net
Petri nets: Meta-model

context Arc inv:
( self.source.oclIsKindOf(Place) and self.target.oclIsKindOf(Transition) )
or
( self.source.oclIsKindOf(Transition) and self.target.oclIsKindOf(Place) )

name: String

Token

Place

Transition

Arc

Node

Object

Petrinet

1 source

1 target
Transformations

:Project

:InPort

:Track

:OutPort

:PetriNet

:Place

:Arc

:Transition
Transformations

- :Project
- :InPort
- :Track
- :OutPort
- :Corresp
- :Corresp
- :Corresp
- :Corresp
- :Petrinet
- :Place
- :Arc
- :Transition
Triple Graph Grammar Rule

:Project

:Corresp

:Petri net

:InPort

:Corresp

:Place

:Track

:Corresp

:Arc

:OutPort

:Corresp

:Transition
TGG-Rule Application

:Project \(\rightarrow\) :Corresp \(\rightarrow\) :Petrinet

:InPort \(\rightarrow\) :Corresp \(\rightarrow\) :Place

:Track \(\rightarrow\) :Corresp \(\rightarrow\) :Arc

:OutPort \(\rightarrow\) :Corresp \(\rightarrow\) :Transition
Transf. of connection
Transf. of connection
TGG-rule: Connection

:OutPort → :Corresp → :Transition
:InPort → :Corresp → :Place
:Connection → :Corresp → :Arc
A rule in practice

A real example from component tools.

TGG-rules in abstract syntax is very verbose!
TGG-Rules in “graphical syntax”

That is the graphical syntax, we used in our example!
Outline

- Example
- Semantics
- Strength
- Problems and Weaknesses
- Extensions and Open Issues
Semantics of TGG-Rules

This is Use 1 of grammars! Just for two/three models in parallel.
“Model-driven Execution”
Outline

- Example
- Semantics
- Strength
- Problems and Weaknesses
- Extensions and Open Issues
Strength of TGGs

- Rules are declarative and local
- Semantics works both ways
- Yet, the transformations are operational (compiler / interpreter approach)
- Transformations are operational in both directions!

Under some reasonable constraints, which are yet to be identified.
Corollary of locality

- Transformations can (in principle) be verified for semantical correctness
- Approach works incrementally!
Incremental application
TGGs are good for

- Defining transformations between models that are structurally similar

- Executing these transformations (in models of reasonable size)
TGGs are not so good for

- defining transformations between models that are very different in structure
- defining the legal syntax for the models on each sides of the transformation

TGGs should be combined with other transformation technologies such as templates! How?

Use UML and OCL for defining the legal “syntax” of source and target (meta-modelling).
TGGs are not so good for

- formulating the rules of real-world examples in abstract syntax

But, this is only a matter of better tool support!
(Could be a nice MSc-project!)

sometimes there are many large but very similar rules

We need mechanisms for reusing and structuring rules (TGG++):
- “inheritance”
- combination and composition of rules (“where” / “when” → QVT)
Outline

- Example
- Semantics
- Strength
- Problems and Weaknesses
- Extensions and Open Issues
Extensions

- **TGG++**
  - Inheritance of rules
  - where-clause
  - other “abbreviations”

- **Negation**
TGG-Rules in “graphical syntax”
“Model-driven Semantics”

ATSE (02265), L06: Triple Graph Grammars
"Incremental approach"

More problematic: Deletion of model elements. But, in principle doable.
Extensions

- **TGG++**
  - inheritance of rules
  - where-clause
  - other “abbreviations”

- **Negation**
  - grammar-style semantics (not what we want?)
  - model-driven semantics (incrementality lost or incompatible)
Re-usable nodes

Note that this example uses a changed meta-model (components refer to their type).
Rules?

straight:type

Does not work!!

strange:type
Rules?

straight: type

Does not work either!!

strange: type
Re-usable nodes

If they exist already, they will be re-used; if not, they will be created (since they are green or black, we sometimes call them grey nodes).
Extensions

- TGG++
- Negation
- Re-usable nodes ("grey nodes" / ##)
Clean definitions

- Attributes
- Inheritance in graph models
Rules in with attributes

number = n

marking = n
Rules in with attributes

Values of attributes are “grey nodes”!
Attributes: General Concept

n:integer
m:integer

number
marking
offset

f

g
Clean definitions

- Attributes
  - are grey nodes
  - problem: operational interpretation needs inverse functions

- Inheritance in graph models
Inheritance

Meta model

Node in a TGG rule

Model node

Does b map to node?

Don’t know! Must be made explicit!

→ In our tool: the property MatchSubtypes of a node defines, what we want.
Research issues

- Good examples
- Benchmarks
- “Theory” of sufficient conditions for deterministic transformations / deterministic “partial transformations”
- Verification techniques
- Uniform interface / integration of strategies
- Efficient transformations / synchronisation
- …

→ Some of the concepts discussed here, are not implemented in the TGG interpreter we use in our tutorial. Values to attributes are assigned via constraints (see examples in tutorial).
TGGs: Summary

- (Often) elegant way of defining the relation between two kinds of models
- Based on this definition, models can be
  - transformed in either direction (different approaches: compile rules, interpret rules)
  - corresponding models can be kept consistent (synchronization)
- Good for defining the relation between structurally similar models
TGGs: Literature


→ We did NOT invent TGGs (that was Andy Schürr more than 20 years ago)

→ Due to their nice concepts we are enthusiastic about them anyway and try to promote them.