Tutorial Introduction to Graph Transformation: 
A Software Engineering Perspective

Lissa Dora

CSC2125: Modeling Methods, Tools and Techniques
Fall 2012
Overview

- Purpose: to provide a light-weight introduction to graph transformation.
- This presentation will cover some topics about:
  - Graph basic concepts
  - Graph transformation applications and sample figures
  - Some tools to “work” with graphs and graph transformation
- All of the topics are viewed from software engineering, visual modeling techniques.
Visual modeling techniques face complication in their definition and implementation. There are 3 aspects which causes this problem:

1. **Most diagram languages have a graph–like structure.** The usual denotational or operational approach for language definition can’t readily be applied because it works based on terms (abstract syntax trees) as representation of the language structure.

2. **Diversity of modeling languages and dialects in use.** It is not easy to provide language definitions and implementations for these notations.

3. **Consistency of interrelated sub–models in a certain model with different levels of abstractions.** Inconsistent model does not have correct implementation.
There are two uses of graph transformation:

- **As Meta Language**, to supply a means to formally specify the syntax, semantics, and manipulation rules of visual diagrammatic languages.

- **As Semantic Domain**, to supply a specification language and semantic model for reasoning on particular problems. For example: the consistency between functional requirements and software architecture models in concurrent and distributed systems.
Modeling with Graph Transformation

- In typed graph transformation systems have been proposed as a way to specify use cases in a visual, yet formal way with the additional benefit of an executable specification.
- An example for such specification is the rule “payBill” and its application. The figure on the left is called “pre-condition” and the figure on the right is “post-condition”.

Pre–Conditions and Post–Conditions

Pre–Condition

Pre: Account balance = b1
       pays
       to
       b: Bill
       total = a

Post: Account balance = b1 – a

L → G

H → R

A3: Account balance = 4
A1: Account balance = 10
A2: Account balance = 2
C: Client
B: Bill

cpayBill(B)

payBill(b)

Graph–Transformation Rule

Fig. 2. A sample transformation step using rule payBill
Graph transformation rules like “payBill” provide a high-level specification of functional requirements in terms of pre and post conditions, abstracting from intermediate actions and states.

If the more fine-grained structure of an operation is of interest, a rule may be decomposed into more elementary steps.
Decomposition of A Graph-Transformation Rule

Fig. 3. Rule payBill(b) derived from createTransfer(b,t); executeTransfer(t)

\[
G \xrightarrow{\text{payBill(B)}} H \quad \quad \quad \quad G \xrightarrow{\text{createTransfer(B,T)}} X \xrightarrow{\text{executeTransfer(T)}} H
\]
Graph Representing Architectural Style

- A quite different interpretation of nodes and edges in a graph is in terms of components and connectors of a software architecture model.

- Instance of an architecture are:
  - A configuration
  - An architectural style
  - Reconfiguration rules
The main benefit of graph transformation for describing software architectures is the ability to model dynamic reconfigurations in an abstract and visual way.
The static/syntactic integration relating functional requirements and architectures is achieved by defining the abstract syntax of both the functional and the architectural view by means of a meta model.

Generally, a meta model is a model for a modeling language expressed within a simple subset of the language itself.
The meta model allows a uniform representation where elements of different submodels are represented as vertices of the same abstract syntax graph.
In the meta model example before, a meta model has been used to specify the abstract syntax of static diagrams, like class and object diagrams.

More generally, meta modeling techniques may be used to define the concrete syntax, abstract syntax, or semantics of any modeling notation, be it dedicated to static or dynamic aspects.
Visual Modeling Language Layers

Fig. 9. A layered view of a visual modeling language
Concrete Syntax – Definition

- A concrete syntax defines how users perceive different modeling elements supplied by the notation.
- It takes form of bubbles, rectangles, lines, arrows, etc. Each notation element is concretely rendered in terms of geometrical elements that define its appearance.
- For example: states are rectangles with rounded corners, initial states are black bubbles, state transitions are directed edges.
Concrete Syntax Example – StateChart

Fig. 8. The CardReader view of the CashBox behavior as a protocol statechart
The abstract syntax defines the modeling elements supplied by the notation, without the concrete “sugar”, and the relationships among them.

Models at this level can be parsed to check if they are syntactically correct.

Tokens at this level are related to the semantic interpretation, in other words thinking in terms of states transitions, events, and so on.
Abstract Syntax Example

Fig. 10. Abstract syntax graph for the statecharts of Fig. 8
In semantic domain, operational semantics and denotational semantics occur in combination.

For behavioral diagrams like statecharts, an operational semantics can be given directly on the abstract syntax of the language, through yet another grammar. In our example, a formal semantics is desirable to specify precisely the execution of a model.
There are essentially two different ways to define operational semantics by graph transformations:

1. Graph transformation rules can specify an abstract interpreter for the entire language as proposed.
2. Each model can be “compiled” into a set of rules. **We’ll follow the second approach.**

If you want to know more about the first approach, see this the reference: *G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML.*
In this approach, roughly, statecharts are translated to structured graph transformation systems that satisfy the well-formedness rules imposed by the notation.

Fig. 11. Some transitions of Fig. 8 as graph transformation rules.
Semantics defined as a mapping from the abstract syntax into a semantic domain is referred to as **denotational semantics**.

Remember that to define the dynamic semantics of statecharts, the semantic domain itself has to provide an operational model – thus **operational and denotational semantics occur in combination**.
Choose a semantic domain, i.e., a usually simpler formal method whose execution rules are well established, and define the “behavior” of each abstract syntax element through a suitable mapping onto the semantic domain.
Graph transformation tools can be divided into two main groups:

1. **General-purpose environments for modeling graph-centric problems.**
2. **Environments for specifying visual languages**, providing mainly automatic support to the generation of graphical editors by making users define visual languages through suitable grammars.
Graph Transformation Tools Examples

1. Graph Transformation Tools
   - **PROGRES (PRO-grammed Graph Rewriting Systems)**. User specifications can be cross-compiled into Modula 2, C, or Java code to produce “independent” graph manipulation components.
   - **AGG** is a rule-based visual language supporting an algebraic approach to graph transformation which offer similar tool support with PROGRES. The types of rules’ nodes are defined through Java classes and standard Java libraries can be exploited to compute their attributes.
   - **FUJABA**, which is an environment for round trip engineering with UML, Java, and design patterns, based on graph transformation. FUJABA generates Java class definitions from UML class diagrams.

2. Graph Transformation-Enabled Tools
   - **DIAGEN**, supplies a framework of Java classes to provide generic functionality of editing and analyzing diagrams and a generator program to produce Java source code for some aspects that depend on the concrete syntax of the language.
   - **GENGED (Generation of Graphical Environments for Design)** provide similar support as DIAGEN. It support the description of visual modeling languages for producing both graphical editors and simulation environments.

*To know more about these tools, see this reference: R. Bardohl, G. Taentzer, M. Minas, and A. Schurr. Application of graph transformation to visual languages.*
References