# Application of Graph Grammars to Model Transformations

Technical Report 2013-604

Francisco de la Parra

School of Computing
Queen's University
Kingston, Canada
parra@cs.queensu.ca

July 9, 2013

# Contents

# List of Figures

# 1 Introduction

## 1.1 Overview

Model Driven Engineering (MDE) has become in recent years a sound and efficient alternative for the design and development of complex software of the kind found in industrial and networked applications [10, 50, 52]. It encourages the use of diagram and language-based models to create system representations and designs, at different abstraction levels [8, 45, 57, 58, 70]. One typical example of this paradigm consists of creating platform-independent models (PIMs) representing system level requirements, which are later transformed into platform-specific models (PSMs), at a lower abstraction level, to support system implementation and testing activities [52]. By following MDE's practices, analysts and developers can produce powerful and reusable designs, as well as readily verifiable program code, in an environment that promotes enhanced visibility of the software artifacts and extensive task automation via tool support [50, 74].

In the MDE approach, model-to-model transformations are first-class entities and their classification, formalization and implementation have been the subject of intensive research [9, 10, 15, 52, 67, 87]. The studies have primarily focused on: 1) analyzing their intrinsic properties: nature, scope, directionality, scheduling, technical environment, etc. [16, 17, 67, 69, 88]; 2) defining formal syntactic and semantic aspects of their realization [31, 42, 49, 70, 89]; and 3) developing approaches for their implementation (i.e., rule-based, procedural, hybrid, graph-transformation-based, etc.) [1, 3, 11, 12, 31, 32, 39, 48, 68, 91].

Given the two-dimensional, diagram-like nature of these models, assembled from sets of objects that likely have multiple types of relations among them, the use of graphs, composed of nodes and edges, is a natural choice for representing their structural and behavioural aspects [7, 44]. A model's structure or current state can be abstracted into the definition of a given graph. Its structural and behavioural changes along a time coordinate can be represented by rule-based evolution of graphs, in other words, by a graph rewriting system [1, 7, 11, 32, 36, 39, 44, 48, 66, 68].

This work surveys the characteristics and properties of graph rewriting systems, also known as:

- *graph grammars* [12, 18, 20, 21, 22, 23, 24, 25, 26, 28, 29, 33, 47, 54, 62, 71, 72, 77, 78, 79, 80, 82, 83, 84, 85, 86, 94] if the emphasis is on studying the types of graphs they generate (i.e., *graph languages*), and

- *graph transformation systems* [4, 7, 21, 30, 36, 44, 51, 66, 68, 90, 91] if the primary interest is in describing the graph rewriting mechanisms they use,

and their application to formalize, specify and implement transformations on models of the type described above.

The primary focus is to describe core theoretical aspects of these systems, such as the structure and properties of the grammar productions, aspects of graph transformations, types of graphs generated, and others, from the viewpoint of their suitability and potential for conceptualizing and supporting the required formal structures and processes, as well as the desired tools to transform diagram-based models.

In pursuing this analysis goal, the general exposition gradually aggregates graph rewriting concepts, beginning with the mostly graph generation approach of graph grammars in Section 2, and continuing in Section 3 with the more versatile graph transformation approach which applies the software engineering oriented concepts of graph transformation units and systems. Section 4 outlines a set of model transformations recently researched and considered potential candidates to take advantage of the graph transformation framework. Section 5 presents a brief summary of open problems related to graph grammars and graph transformations, and also provides an initial insight to using them in building integrated model transformation tools.

The remainder of Section 1 establishes a general conceptual context used in this work for the analysis of applying graph transformations to model transformations.

## 1.2 Modeling and Graph Rewriting Background

Overall, graph grammars and graph transformations have been used in software engineering as formalisms for representing design and computational aspects of systems at a very high level of abstraction, which normally requires as a prerequisite the use of sound methodologies to analyze the problem domain and a reasonably structured approach to systems development (e.g., MDE). The following subsections provide a summary introduction to the modeling process, its elements, and the relations that can be established with graph rewriting systems.

### 1.2.1 Application Domains and Modeling

Sound application of the MDE's techniques has highlighted the need for knowledge and design reusability, which in turn requires organizing similar problems, solutions and software applications in the context of a well defined application domain with a substantially focused goal set. Correspondingly, domain analysis [81], as a process that promotes abstraction by identifying, creating, classifying and organizing useful modeling entities (i.e., standards, taxonomies, structural and procedural models, domain languages, etc.) in a given application domain, has become more a mainstream activity rather than an optional task. It is in this structured environment where one can envision using graphs and graph transformation as potentially effective and useful tools to describe, formalize and structure some of the activities and objects resulting from the analysis, mainly those that lend themselves to diagram-like representations, rather than applying them in analyzing a one of a kind problem and developing a single-use solution.

Modeling, as an activity that strives for extracting useful abstractions from "originals" such as real systems, subjects, or even other models (i.e., meta-models), is language-oriented in the context of MDE [45, 52, 57, 58]. One builds descriptive or prescriptive models using general modeling languages such as the UML [8] or domain-specific languages with formal syntax and semantics [42, 64]. The main concern is whether, given the requirements of an application, these languages allow the building of models that contain accurate and capable representations of the entities and relationships identified in the domain analysis stage, which can be incorporated into tools for analysis and verification. In other words, important questions arise about completeness and expressiveness of the syntax and soundness of the semantics associated with the elements of the modeling language(s) used to build those models and the ability to build tool support for them. In light of this situation, graph grammars and graph transformations haven been referred to as a promising alternative in order to provide more precise syntactic and semantic definitions, for example in diagrammatic languages, with potential for supporting tool construction (e.g., diagram editors, configuration management and integration tools) [52, 64].

### 1.2.2 Models and their Transformations

Research to characterize MDE in the context of the OMG's MDA architectural standard has suggested that encouraging a mindset of "everything is a model" [10] could well be the key to further developing this approach. In addition to realizing that models in the context of MDE approach are language-based and that their presence is ubiquitous, it is also necessary to briefly discuss some basic modeling concepts that appear very relevant to an interest in applying graph grammars to model transformations.

According to Kühne [57] "a model is an abstraction of a real or language-based system allowing predictions or inferences to be made". In the present work, it will be assumed that the abstraction component of this definition, arguably the most controversial [45, 58, 70], represents "a convenient and pragmatic mapping between an existing original (system, subject, or another model) and a language-based (visual or textual) representation (model) which facilitates further computerized processing associated with the original due to complexity reduction, standardized entity representation, and the use of well-known information structures". Additionally, it will be considered that the prediction and inference capabilites of a model are properties that accept appropriate and accurate characterization when defined in the context of specific application domains. In other words, general domain-independent analysis of these properties does not really contribute to the effectiveness of prescriptive and descriptive models of interest resulting from applying the abstraction process as described above.

In attempting to determine the usefulness and applicability of different models, MDE researchers have suggested various classification schemes. Among them, the views of "token versus type models" [57] and "platform-independent (PIM) versus platform-specific (PSM) models" [52] have gained prominence due to extensive theoretical, standardization, and system implementation work done around them. In this context, graphs and graph transformations appear to offer a great deal of descriptive and precision power when formalizing and describing, for example, many structural aspects of type models [64] , or formal aspects of the transformations required to convert PIMs into PSMs and vice versa [49].

In the overall modeling context described so far, model transformations have become primary and absolutely necessary entities that make possible a powerful and flexible approach based on creating domain specific language-based models, at different levels of abstraction and representing different views of the same system, which can be translated into each other for more convenient analysis or application implementation. To this end, graph grammars and graph transformations have been proposed as concise and expressive formalisms not only to describe syntactic and semantic aspects of PIMs to PSMs transformations but also to include transformations between different domain-specific modeling languages (DSML) [49].

### 1.2.3 Graph Grammars

Graph grammars were first introduced in the late 1960's as an extension to graphs, composed of nodes and directed edges, of the grammar concepts used to generate formal string-based languages [79]. Since then, intensive research has developed their theoretical foundation into a number of different grammar analysis and construction techniques: algebraic techniques, set theoretic approach, node-label controlled (NLC) approach, etc. [23, 25, 26, 28, 47, 71, 72, 77, 78, 80, 82, 83, 84, 86] and extended their application to fields in science and engineering: pattern recognition, natural computing, visual languages, software engineering, etc. [29] From a software engineering perspective, which is the analysis viewpoint exercised in this work, the initial focus on graph grammars has shifted from understanding the properties of the graph languages they generate to an interest in describing and exploiting the graph transformation mechanisms they use [7, 44] and their integration into graph rewriting systems [36, 51, 85], and how they can be applied to formalize computational structures, models and processes, and used for software specification and very high level rule-based programming [1, 3, 4, 33, 34, 39, 48, 49, 54, 55, 61, 62, 66, 68, 85, 91, 94].

Over the years, the basic conceptual idea of a graph, consisting of labeled nodes and directed edges, has been extended to include attributed graphs, typed graphs, hypergraphs, and others. By the same token, the basic idea of a graph grammar (GG) as a structure having:

- An initial graph $G_0$.
- A finite set $P = \{p_1, p_2, ..., p_n\}$ of productions (i.e., rules).
- Each production $p_i$ has a Left Side Graph ($LSG$) and a Right Side Graph ($RSG$).
- A production is applied (expressed as $LSG \Longrightarrow RSG$), in an exhaustive, either deterministic or non-deterministic fashion, to graph $G_i(i = 0, 1, 2, ...)$ (i.e., to $G_0$ or one of its descendants), and involves searching for an isomorphic subgraph occurrence of $LSG$ in graph $G_i$. All these occurrences of $LSG$ in $G_i$ are completely replaced by $RSG$ using a sub-graph "connecting" or "gluing" approach to produce graph $G_{n+1}$.

where the successive applications of productions generate a graph language $L(GG) = \{G_0, G_1, G_2, ...\}$, has evolved to include rule application conditions [27], programmed rule application approaches [33, 34, 35, 85],and a number of graph rewriting techniques [4, 7, 44]. Section 2 provides a more detailed overview of the different types of graph grammars.

### 1.2.4 Using Graph Rewriting in Model Transformations

Research that developed during the mid-to-late 1990's to integrate graph grammars into graph rewriting systems [85, 90], in order to enhance programmability and enable control flow in graph transformation-based software applications, along with a growing interest in using diagram-based modeling languages (e.g., UML) to specify and build software systems [8, 36], established the foundation for new research and development efforts geared to demonstrate the use of graphs and graph grammars / graph transformations as suitable formalisms to describe and specify syntactical and semantic aspects of software models and model transformations [39, 62, 66]. Their application opened the possibility to generate, for example, model visualization and transformation tools in a fairly automated fashion from generic graph-based specifications [75, 91]. In general, ideas for designing those specifications have been extracted from other areas where graph rewriting techniques have been used, such as in specification of program transformations, semantic definition of well known modeling formalisms (e.g., state charts, Petri nets), and tool integration [1, 61].

Applying graphs to models and model transformations elaborates on the central idea of abstracting the generic entities of a specific abstract or concrete domain into a metamodel, which in turn is defined using a highly descriptive modeling language (e.g., UML class diagrams), and considering that the models created within that domain, which conform to the metamodel, can be represented by graphs [1, 64]. Model transformations can occur between models of the same domain or different domains and involve graph rewritings that can be specified as instantiations of the language generated by a graph grammar or the operations in a graph transformation system [48, 60].

Section 3 presents an overview of the characteristics that have evolved in graph grammars and graph rewriting, in general, to constitute graph transformation systems of interest to the MDE research community. Section 4 explains how core types of model transformations can be supported by graph grammars and graph transformations, highlighting the special case of triple graph grammars as a useful formalism to specify incremental model transformations and model synchronizations.

### 1.2.5 Graph Rewriting Tools

As theory evolved into a number of different approaches to build graph grammars and the interest in applying them to software engineering problems became more prominent, a number of prototyping tools to investigate their properties and use them as software specifications started to appear. Graph$^{Ed}$ [46] represents an early tool effort with a rather basic objective of providing an interactive environment for graph drawing/parsing and the representation of productions in context-free graph grammars (e.g., 1-NCE, NLC, and BNLC grammars [82]). The Algebraic Graph Grammar (AGG) system [65, 90], which supports the Single Pushout (SPO) graph rewriting approach to build graph grammars, since its initial versions implemented a more powerful graphical editor to allow experimentation with grammar productions, subgraph morphisms and graph rewriting results. It also included a derivation component to allow direct execution of user-selected grammar productions and subgraphs. Graph$^{Ed}$ and AGG allow the working with graphs and prototype graph grammars in isolation, and do not provide integrated environments for the application of graph rewriting to software systems.

In the early 1990's, the PROGRES tool introduced one of the first integrated environments for interactively prototyping graph grammars and graph transformations based on systemic specifications and very high level programming [73, 85]. The environment consisted of: 1) a visual/operational specification language for graphs and graph rewriting rules, 2) a set of tools for editing, analyzing and executing specifications in this language, and 3) a methodology associated with the use of the language and tools, denominated graph grammar engineering approach, intended to support the entire development cycle, from requirements to code implementation, for graph grammar-based software applications, and characterized by techniques to deal with complex data structures (i.e., graph-like structures, individual objects, and the relationships between them), complex graph pattern matching and transformation, and appropriate user interfaces. PROGRES, although a generic environment for application specification and development using graph grammars, can be considered in many ways a precursor for a number of more specialized prototyping tools dealing with models and model transformations such as the following:

- AToM$^3$ is a multi-paradigm modeling and metamodeling framework that allows the building of models for complex systems using different modeling formalisms (i.e., metamodels). It relies on graph grammars and graph transformations to implement automated transformations between formalisms, and to perform visualization and code generation tasks [60, 62].

- The Graph Rewriting and Transformation (GReAT) system is a graph-based language and tool suite for the specification of graph transformations between DSMLs. The GReAT language allows the specification of patterns to match on input graphs, transformation rules and rule sequencing control. The metamodels for the DSMLs are specified using UML class diagrams [1, 5, 48].

- FUJABA is a suite of tools for round-trip software engineering that allows code generation and software artifact recovery between UML diagrams and Java code using graph transformation techniques. Graph-like *story diagrams* are the main artifacts used in this framework for modeling application behaviour [37, 75].

- MOFLON is built on top of FUJABA as a metamodeling framework (compliant with the OMG standards for the MDA paradigm) for specifying system integration tools and model transformations which makes use of story driven modeling (SDM) (i.e., story diagrams) and graph transformations for synchronization of tools and models [2, 93].

- VIATRA2 is a generic model transformation framework based on metamodeling and graph transformations. It provides a declarative language for manipulating graph models, including their transformations, and abstract state machines to produce metamodeling, model transformation and template-based code generation specifications [6, 13].

- MATE is a tool for performing automated analysis and repair of MATLAB Simulink models using metamodeling techniques and graph transformations. It uses FUJABA's SDM to define visual graph transformations that make models compliant with standard guidelines [63].

# 2 Graph Grammar Approaches

## 2.1 General Concepts

In order to briefly formulate a clear description of graph grammars as operational structures providing the mechanisms to generate, transform and parse graphs and graph-like structures, it is helpful to identify their key components from the outset. The following is an abbreviated description of these elements.

### 2.1.1 Graphs and Hypergraphs

A *simple labeled directed graph* $G$ over a pair of label alphabets $(\Sigma_V, \Sigma_E)$ (where $\Sigma_V = \{vl_1, ..., vl_p\}$: node label alphabet, and $\Sigma_E = \{el_1, ..., el_q\}$: edge label alphabet) consists of a sextuple $(V, E, s_V, t_V, l_V, l_E)$ where $V = \{v_1, ..., v_n\}$ (set of nodes; or vertices), $E = \{e_1, ..., e_m\}$ (set of directed edges), $s_V : E \to V$ assigns an edge's source node, $t_V : E \to V$ assigns an edge's target node, $l_V : V \to \Sigma_V$ assigns node labels, $l_E : E \to \Sigma_E$ assigns edge labels (fig. 2.1-(a)).

From the previous definition of a simple labeled graph, one can derive various specific classes of graphs, which the different graph grammar approaches [82], context-free and context-sensitive, can generate:

- In an *undirected graph* if there exists a directed edge between nodes $v_i$ and $v_j$ $(i \neq j)$, then there also exists a directed edge between $v_j$ and $v_i$.
- A *multigraph* can contain multiple directed edges between a given pair of nodes $v_i$ and $v_j$ $(i \neq j)$.
- In an *unlabeled graph* the label alphabets $\Sigma_V$, $\Sigma_E$ are empty sets.
- In a *typed graph*, the sets of node and edge labels $\Sigma_V$ and $\Sigma_E$ respectively, are partitioned into classes called *types*. Edges identified with a certain label type can only be incident to source and target nodes identified with specific label types.
- An *attributed graph* contains attributes on nodes which can have any of the numeric and text data types, or some other complex types (e.g., an expression, list, or graph).
- *Star graphs* consist of nodes labeled as nonterminal and terminal (i.e., $\Sigma_V = \Sigma_{VN} \cup \Sigma_{VT}$ with $\Sigma_{VN} \cap \Sigma_{VT} = \phi$, $\Sigma_{VN}$: nonterminal labels, $\Sigma_{VT}$: terminal labels) with the following restrictions on edges: 1) the target node of an edge between a nonterminal and a terminal node must be a terminal node, 2) edges between nonterminal nodes are not allowed (fig. 2.1-(b)).

*Hypergraphs* represent an extension of the structural concept of a graph to contain *hyperedges*, instead of edges, connecting to multiple source and target nodes through links denominated "tentacles" (fig. 2.1-(c)). Context-free *hyperedge replacement* (HR) grammars are commonly applied to generate and parse these structures [41]. The algebraic transformation based *hypergraph replacement* (HGR) grammars represent a more complex generalization of HR-grammars. When limiting hyperedges to having one source and one target node, these grammars generate simple graphs of the type defined above.

*Triples of graphs* extend the idea of a single graph as the unit of analysis to a triple (LG, CG, RG) consisting of three graphs: LG (left graph), RG (right graph), and CG (connection graph), of the same class, where graph CG represents associations between the elements of graphs LG and RG. *Triple graph grammars* are the structures associated with generating and parsing these graph triples [84, 86].

*Hierarchical graphs* constitute a most general structure with *higher-order nodes* that are abstractions of graphs and *higher-order edges* that represent relations between graphs which are represented as graphs again. With these structures, it is possible to abstract an entire sub-graph to a node or to a single edge between two abstracted nodes [19].

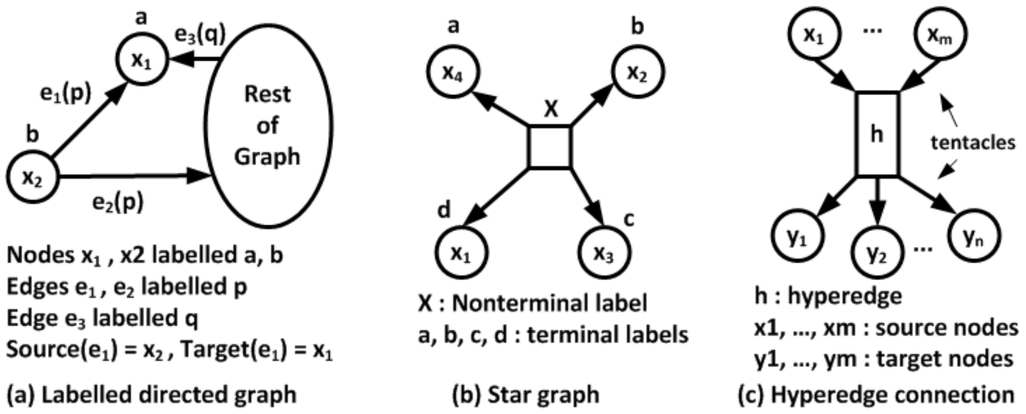

Figure 2.1: Graphs and Hypergraphs

## 2.1.2 Graph Rewriting Productions

The sequential and iterative application of graph rewriting productions (i.e., rules) to an existing graph(s) of a certain class is the mechanism used to generate a complete set of graphs of that same class (i.e., graph language). The following is a general definition of a production [4] that is applicable to most graph transformation (i.e., graph grammar) approaches found in the literature.

A graph rewriting production is a sextuple $p = (LG, K, RG, ac, gl, em)$ that allows the transformation of a *host graph* $G$ into a *direct derivation graph* $H$, whose components are defined as follows:

- $LG$ is the production's left graph. Applying a production to a host graph $G$ involves searching for one or more isomorphic occurrences of $LG$ in $G$ which will eventually be replaced by instantiations of graph $RG$.

- $K$ is the production's interface graph which is a sub-graph (i.e., no dangling edges: all its edges have a target and a source node) of $LG$ and $RG$ contains an isomorphic occurrence of it. This graph is only used in the *double pushout approach* to graph grammars to replace $LG$ by $RG$ via a graph gluing operation.

- $RG$ is the production's right graph which will be attached to an intermediate graph $(G - LG) \cup K$, in the more common case, using graph gluing or embedding, or both operations, to produce a direct derivation graph $H$ as the overall result of applying the production.

- $ac$ are the application conditions under which a production will be applied or not. They could represent the existence or non-existence of certain nodes, edges, or subgraphs in the host graph $G$, as well as embedding restrictions of $LG$ in $G$ or of $RG$ in $H$.

- $gl$ is the gluing operation which consists of three steps:

  - Build a context graph $D$ which contains the graph $G$, minus the nodes and edges of the isomorphic occurrence of $LG$ in $G$ that are not preserved in $RG$.
  - Glue graphs $D$ and $LG$ through $K$ to generate the host graph $G$. This operation is the disjoint union of graphs $D$ and $LG$ (i.e., corresponding nodes and edges of the occurrences of $K$ in $D$ and $LG$ are amalgamated into one).
  - Glue graphs $D$ and $RG$ through $K$ to generate the direct derivation graph $H$. This operation is the disjoint union of graphs $D$ and $RG$.

- $em$ is the embedding (or connecting) operation, which creates edges between designated nodes in the context graph $D$ and nodes in the graph $RG$, to produce the final direct derivation graph $H$. Its definition can, for example, specify that dangling edges (i.e., edges with a source or target node, but not both) resulting from removing an isomorphic occurrence of $LG$ in $G$, be recreated, or that new edges be created between nodes in $D$, determined by path expressions [72] or connection relations [47], and specific nodes in $RG$. Embedding is mostly used in single-node replacement productions (i.e., $LG$ is a single node).

The application of a context-free production $p_i$ from a set $P = \{p_1, ..., p_n\}$ to a host graph $G$ has a strictly local scope to the isomorphic occurrence of the production's $LG$ graph - which is a single element: a node, a handle (i.e., single edge with source and target nodes), or hyperedge - in the host graph $G$. Context-sensitive productions can involve nodes and edges not local to this occurrence, and are characterized by the existence of at least one element $n$ in $LG$ with the property that $LG - \{n\}$ is a subgraph of $G$ (i.e., arbitrary graphs).

### 2.1.3 Graph Grammars

A graph grammar is a quadruple $GG = (T, S, P, A)$ where:

- $T$ is the type or class of graph, as defined in Section 2.1.1 (i.e., unlabeled, labeled, hypergraph, etc.), that can be generated, modified and recognized by applying a set $P$ of productions.

- $S$ is the start graph to which a set $P$ of productions will be initially applied. Typical start graphs for context-free grammars [1] are: a single node, a handle, a hyperedge. Context-sensitive grammars can have arbitrary start graphs.

- $P$ is a set of productions $P = \{p_1, ..., p_n\}$ with each production $p_i$ defined as in Section 2.1.2.

- $A$ is a set of additional specifications extending the scope and nature of applying the set of productions $P$ to graphs of type $T$ (i.e., attributes, programmed rules, global application conditions, structural conditions, etc.).

The language $L(GG)$ generated by graph grammar $GG$ is the set of terminal graphs (i.e., graphs without non-terminal nodes) that can be generated from the *start graph $S$* by repeated application of the graph rewriting productions. All graph languages generated by graph grammars acting upon the same graph type $T$ constitute a *class of languages*.

Applying a set of productions $P = \{p_1, ..., p_n\}$ to one or more graphs involves scheduling the application of the individual productions in a non-deterministic or deterministic manner. Deterministic approaches can utilize ad-hoc programmed rules, prioritization schemes, application conditions and hybrid schemes to schedule productions. It is well known that these extensions found in deterministic approaches do not increase the generative power of a graph grammar (i.e., do not extend or create new classes of graphs not generated in a base grammar) [71, 72]; they rather reduce it, to make the parsing and recognition of the graphs in the generated language a more manageable task. In general, they facilitate using graph grammars in applications such as subject modeling, entity description, process specification and pattern recognition, by increasing their descriptive power [21, 22].

## 2.2 Context-Free Graph Grammars

Informally, a graph grammar is said to be context-free if its graph rewriting mechanism is:

- *local* to the isomorphic occurrence of each production's $LG$ graph in the host graph $G$.

- *confluent* - two productions affecting different locations of a host graph $G$ can be applied in any order giving the same derived graph.

- *associative* - as in ordinary algebraic associativity, productions can be applied in different associative groups resulting in the same derived graph.

Context-free grammars are important in the sense that graph derivations can be modelled by derivation trees which, as a whole, produce the same set of graphs for any different ordering used to apply the productions [14]. In general, they are specially useful in describing and generating graphs with recursive properties.

[1] A grammar is *context-free* if all its productions are context-free. Otherwise, it is *context-sensitive*.

### 2.2.1 Early Approaches

Graph grammars were first studied in [79] under the name of web grammars, with the analysis concentrating on comparing the generative properties of context-free and context-sensitive grammars, using simplified models of ordered acyclic directed graphs as natural representations of strings, aiming to imitate Chomsky's most applicable grammar types. This work emphasized the application of context-free grammars to generate node-labeled graphs representing simple and very regular directed networks. It also introduced the concept of embedding a production's replacement graph without giving any details of this operation.

In another interesting development, Pratt [80] introduced the concept of pair grammars, where correspondence mappings are established between the productions and the non-terminals in the productions of two graph grammars defined over the same label alphabet(s). The language generated by this type of grammar is a set of ordered pairs of graphs. Similarly, the analysis in this case focused on considering both grammars of the pair as being context-free, capable of generating directed graphs representing strings. From this viewpoint, the pair grammar could be used as a high level specification for string-to-graph and graph-to-string conversions (i.e., program code to graph representation and vice versa).

Both examples represent the early efforts to emulate the common properties of string grammars in graph approaches, which is the preamble to grammars with more advanced graph rewriting mechanisms, the topic of the subsequent sections.

### 2.2.2 Algorithmic Approaches

These approaches to graph grammars, also known as: set theoretic, connecting, and node replacement, implement a graph rewriting mechanism which replaces a node $v_l$, labeled with the symbol $l$ from an alphabet $\Sigma$, in a host graph $G$, by reconnecting to the context graph $D = G - v_l - E_d$ ($E_d$: set of dangling edges in $G$ after removing $v_l$) an isomorphic copy of a grammar production's $RG$ graph through edges determined by path expressions [72], connection relations [47, 82], or some other algorithm.

The following three sections introduce the grammar types: NLC, NCE and edNCE, which have completely local graph rewriting mechanisms, where a production's $RG$ graph is reconnected only to the neighbour nodes of the node being replaced. This restriction makes these grammars easier to further constrain to produce grammar classes with the desirable property of being confluent, hence context-free.

#### 2.2.2.1 NLC Grammars

A *node-label-controlled* (NLC) graph grammar is a quadruple $(NU, P, C, S)$ where:

- $NU = (V, E, \Sigma, L_v)$ is a *node-labeled undirected graph* definition as follows:
  - $V = \{v_1, ..., v_n\}$ is a finite set of nodes.
  - $E = \{e_1, ..., e_m\}$ is a set of undirected edges (i.e., $e_i = (v_j, v_k) = (v_k, v_j)$; $j \neq k$; $j = 1, ..., m$; $k = 1, ..., n$).
  - $\Sigma = \Sigma_N \cup \Sigma_T$ is a finite alphabet of symbols , with $\Sigma_N$: non-terminals, $\Sigma_T$: terminals.

  - $l_v : V \to \Sigma$ is the labelling function which assigns a terminal or non-terminal symbol to each node in $V$.

- $P = \{p_1, ..., p_n\}$ is a finite set of productions, with each $p_i = (L_i, RG_i, C_i)$, $L_i$: non-terminal symbol in $\Sigma_N$ (left side of production), $RG_i$: graph of type $NU$ (right side of production), $C_i$: connection relation of $p_i$.

- $C = \{(\rho, \sigma) \,|\, \rho, \sigma \in \Sigma\} = \bigcup_{1 \leq i \leq n} C_i$ ($C \subseteq \Sigma \times \Sigma$) is the connection relation. A pair $(\rho, \sigma)$ in this relation indicates that an edge should be created between each node labeled $\sigma$ in graph $RG$ and each node labeled $\rho$ in the context graph $D$.

- $S$ is the start graph. This graph is usually, but not necessarily, a single node labeled with a non-terminal symbol (i.e., $S$).

Consistent with these definitions, the graph replacement and embedding mechanisms in NLC grammars are strictly local to the vicinity of a single labeled node. These grammars generate undirected node-labeled graphs, usually with some type of recursive structure and properties (i.e., examples in [47, 82]), and generally are not confluent. However, applying certain restrictions to their base definition can produce truly context-free grammars. A boundary restriction is one way to obtain grammars with this property, producing the so-called B-NLC grammars. This structural restriction dictates that there can not be an edge between two non-terminal nodes in either the start graph or in a production's $RG$ graph.

#### 2.2.2.2 NCE Grammars

A *neighbourhood controlled embedding* (NCE) grammar represents a refinement of a NLC grammar to provide enhanced control over the local graph embedding mechanism. Formally, a NCE graph grammar is a quadruple $(NU, P, C, S)$ where:

- $NU$ as in a NLC grammar.

- $P = \{p_1, ..., p_n\}$ is a finite set of productions whose left and right hand sides are defined as in the case of NLC grammars, however, the right-hand-sides of the pairs in the connection relation $C_i$ of each production now refer to specific nodes in graph $RG$.

- $C = \{(\rho, v) \,|\, \rho \in \Sigma, v \in V\} = \bigcup_{1 \leq i \leq n} \boldsymbol{C_i}$ ($C \subseteq \Sigma \times V$) is the connection relation. A pair $(\rho, v)$ in this relation indicates that an edge should be created between a specific node labeled $v$ in one of the productions and any node labeled $\rho$ in the *context graph*.

- $S$ is the start graph as defined in a NLC grammar.

These grammars still generate undirected node-labeled graphs with unlabeled edges.

#### 2.2.2.3 edNCE Grammars

A further increase in the generation power of NLC grammars requires that, in addition to enhancing their graph embedding mechanism, new structural rules be aggregated to their baseline definitions which would allow them to produce and recognize more general classes of graphs. *Edge-labeled directed-graph neighbourhood controlled embedding* (edNCE) grammars represent one of such improved approaches. Formally, an edNCE grammar is a quadruple $(ND, P, C, S)$ where:

- $ND$ similar to $NU$ is the definition of a graph with a finite set of nodes $V = \{v_1, ..., v_n\}$, with the exception that $E = \{e_1, ..., e_m\}$ is now a *set of directed labelled edges*. That is, if $e_{jk} = (v_j, v_k)$ and $e_{kj} = (v_k, v_j)$ then $e_{jk} \neq e_{kj}$; $j \neq k$; $j = 1, ..., m$; $k = 1, ..., n$. Also, both mappings: $l_V : V \to \Sigma_V$ (assignment of node labels), and $l_E : E \to \Sigma_E$ (assignment of edge labels) are defined.

- $P = \{p_1, ..., p_n\}$ is a finite set of productions whose left-hand-sides are defined as in the case of NLC grammars, whose right-hand-sides are now graphs of type $ND$, and the right-hand-sides of the pairs in the connection relation $C_i$ refer now to specific nodes in graph $RG$.

- $C = \{(\rho, v, d) \mid \rho \in \Sigma, v \in V, d \in \{in, out\}\} = \bigcup_{1 \leq i \leq n} C_i$ $(C \subseteq \Sigma \times V)$ is the connection relation. A triple $(\rho, v, d)$ in this relation indicates that an incoming $(d = in)$ or outgoing $(d = out)$ directed edge should be created between a specific node labeled $v$ in one of the productions and any node labeled $\rho$ in the context graph.

- $S$ is the start graph as defined in a NLC grammar.

*Dynamic edge relabelling* is one additional graph embedding enhancement that can also be implemented in edNCE grammars to provide more discerning connection control when replacing a node whose neighbourhood has some nodes with identical labels. In this case, the connection relation $C$ is composed of quadruples of the form $(\rho, v, d, p/q)$. The additional component $p/q$ indicates that a directed edge between $v$ and $\rho$ should be labeled $q$ if the edge connecting $\rho$ to the node being replaced by the grammar production was labeled $p$ before the graph replacement.

It is well known that some other possible extensions to the graph embedding mechanism, for example, allowing flipping directions on edges, or allowing multiple edges in the tuples of the connection relation, do not increase the generation power of an edNCE grammar [82].

#### 2.2.2.4 Grammars with Arbitrary Embedding

In general, the embedding mechanism of a production's $RG$ graph can be extended to an arbitrary degree of complexity, beyond local schemes, allowing reconnecting it to not only neighbour nodes of the node being replaced but also to any node in the context graph $D$. This would allow a further increase in the generation power of a grammar using the node reconnecting approach. However, the trade-off is increased complexity of the generated graphs which could make parsing them a very difficult or even intractable task. Furthermore, applying these grammars to modeling aspects of a specific application could become cumbersome and devoid of clarity.

One example is the expression approach [72], which implements an arbitrary reconnection scheme, based on set theory and algebraic expressions, applied to node- and edge-labeled graphs with directed edges. In this scheme, the nodes in a production $p_i$'s $RG$ graph are identified by numbers and the nodes in the context graph $D$ that will be used for reconnecting $RG$ are identified by algebraic set expressions $\lambda_{ij}$. Each production $p_i$ has an associated set $T_i$ (embedding transformation) of $(\lambda_{ij}, \mathit{number})$ (incoming edges to $RG$) and $(\mathit{number}, \lambda_{ij})$ (outgoing edges from $RG$) pairs that defines the edges to be created when reconnecting graph $RG$ to the context graph $D$ (embedding transformation). This approach extends the locality of the graph embedding found in NLC and NCE grammars to a global embedding that can include reconnection to any node in the context graph $D$.

### 2.2.3 Hypergraph Grammars

#### 2.2.3.1 Hypergraphs

The central element of a hypergraph is the *hyperedge*, which is an abstraction of a single entity $h$ (i.e., a block in a flowchart, an event in a Petri net, an operator in a functional expression graph, etc.) with $N$ links ("tentacles") that are connected to a set $S(h) = \{v_{s1}, ..., v_{sm}\}$ of source nodes and a set $T(h) = \{v_{t1}, ..., v_{tn}\}$ of target nodes, with $N = m+n$ (the pair $(m, n)$ is known as the type of the $(m, n)$-hyperedge) and $S(h) \cap T(h) = \phi$. $S(h)$ and $T(h)$ are ordered sequences of nodes.

Similar to the definition of a labeled directed graph (Section 2.1.1), one can define a *directed labeled hypergraph* $H$ over a pair of label alphabets $(\Sigma_V, \Sigma_E)$ ($\Sigma_V$ [2]: node label alphabet, $\Sigma_E$: hyperedge label alphabet) as a sextuple $(V, E, s_V, t_V, l_V, l_E)$ where $V$: set of nodes, $E$: set of hyperedges, $s_V : E \to \mathcal{P}_s(V)$ [3] assigns a hyperedge's source nodes, $t_V : E \to \mathcal{P}_s(V)$ assigns a hyperedge's target nodes, $l_V : V \to \Sigma_V$ assigns node labels, $l_E : E \to \Sigma_E$ assigns hyperedge labels.

An extension to this definition is the concept of a *directed labeled multi-pointed hypergraph* $H_M$, which is a hypergraph with additional external nodes organized in two sets of labeled sequential nodes: $\mathit{begin} = \{b_1, ..., b_m\}$ and $\mathit{end} = \{e_1, ..., e_n\}$, which are the source and target nodes to the hypergraph, respectively. Formally, $H_M$ consists of the octuple $(V, H, s_V, t_V, l_V, l_E, \mathit{begin}, \mathit{end})$. Let $m = |\mathit{begin}|$ and $n = |\mathit{end}|$ be the cardinalities of sets *begin* and *end*, respectively, then the pair $(m, n)$ is the type of $H_M$ which qualifies it as an $(m, n)$-hypergraph. This type of structure provides all the elements required to implement hyperedge replacement grammar productions using a node gluing approach to replace a single hyperedge labeled with a non-terminal symbol by a specified multi-pointed hypergraph $RH$. The gluing operation occurs between the hyperedge's $S(h)$ and $T(h)$ nodes and sets *begin* and *end* of $RH$, respectively.

#### 2.2.3.2 Hyperedge-Replacement Grammars

A *hyperedge replacement* (HR) grammar is a quadruple $(H_M, P, C, S)$ where:

- $H_M$ is a directed labeled multi-pointed hypergraph specification as described in 2.2.3.1.

[2] $\Sigma_V = \Sigma_T \cup \Sigma_N$; $\Sigma_T \cap \Sigma_N = \phi$, $\Sigma_T$: terminal labels, $\Sigma_N$: non-terminal labels

[3] $\mathcal{P}_s(V)$: power set of V, where each subset is a labeled sequence of nodes

- $P = \{p_1, ..., p_n\}$ is a finite *set of productions*, with each $p_i = (L_i, RH_i, C_i)$, $L_i \in \Sigma_N$ (left side of production) is the label of a single hyperedge $h$ with $S(h)$ source nodes and $T(h)$ target nodes, $RH_i$: hypergraph of type $H_M$ (right side of production) with $|begin| = |S(h)|$ and $|end| = |T(h)|$ , $\boldsymbol{C_i}$: re-connection mapping of $p_i$. Similar to the procedure applied for graphs, a production $p_i$ is applied to a host hypergraph $GH$, and in this case a labeled hyperedge is replaced by hypergraph $RH_i$ with an embedding defined by node-gluing mapping $C_i$.

- $C = \{(\delta_b, \delta_e) \mid \delta_b : \pi(begin) \to S(h),\ \delta_e : \pi(end) \to T(h);\ \delta_b, \delta_e$ surjective functions$\}$ ($\pi(begin)$, $\pi(end)$: permutations of *begin* and *end*, respectively. $C = \bigcup_{1 \le i \le n} C_i$) is the node-gluing mapping.

- $S$ is the start hypergraph. This graph is usually, but not necessarily, a single handle labeled with a non-terminal symbol. A handle consists of a single $(m, n)$-hyperedge with its tentacles connected to external nodes: the source tentacles are connected to a *begin* set of nodes ($|begin| = m$), and the target tentacles are connected to an *end* set of nodes ($|end| = n$) .

HR-grammars have convenient context-free properties, mostly due to the fact that their hypergraph replacement mechanism is completely local to a single hyperedge, whose structure and connection to the rest of the host hypergraph cannot be further decomposed. They are quite flexible structures with a wide spectrum of generation power, considering that they can generate graphs and hypergraphs of varying complexity just by changing the *type* (i.e., number of tentacles) on hyperedges. The hyperedge concept is a fundamental structure in many different types of diagram-based models [41], hence it is applicable to consider HR-grammars as having a wide range of descriptive power. From a formal language viewpoint, HR-grammars are a powerful tool to generate string graphs, hence string languages with specific properties.

#### 2.2.3.3 Hypergraph Rewriting Extensions

Although, in principle, HR-grammars could generate all recursively enumerable hypergraphs, the granularity of the derivations to obtain a given hypergraph from another one could be high or rather inconvenient. One solution to this situation is to consider replacing arbitrary sub-hypergraphs of a given host hypergraph, instead of just single hyperedges. *Hypergraph replacement* (HGR) grammars implement this rewriting mechanism using a hypergraph counterpart of the graph gluing technique employed in the algebraic approach to graph grammars (Section 2.3). HGR-grammars are context-sensitive in general, which can revert to the special case of context-free HR-grammars when the hypergraph replaced is a single hyperedge. A HGR-grammar hypergraph production $p = (LH, K, RH)$ consists of a left ($LH$) and a right ($LH$) hypergraphs, and a gluing hypergraph ($K$) such that $K \subseteq LG$, $K \subseteq RG$. Applying a production $p$ involves the following steps:

- If $h$: $LH \to GH$ is a hypergraph morphism, then find an occurrence $h(LH)$ in $GH$.

- Check that the gluing conditions for hypergraph $K$ are met:

  - If for $v \in V_{LH}$, $h(v)$ contacts a hyperedge $e \in E_{GH} - E_{h(LH)}$, then $v \in K$.
  - If two different elements $x$, $y$ (nodes or edges) in $LH$ map to the same element in $GH$ under morphism $h$ (i.e., $h(x) = h(y)$), then $x, y \in K$.

- Remove the occurrence $h(LH)$, as well as $h(K)$, from $GH$ to obtain a context hypergraph $DH = GH - (h(LH) - h(K))$.
- Embed hypergraph $RH$ in $DH$ to produce the direct derivation [4] hypergraph $HH = DH + (RH - K)$.

Although the generative power of HGR-grammars and the classes of languages they generate have been extensively studied in the past [41, 82], the attention has gradually shifted to the rewriting mechanisms of specific types of hypergraphs. Rewriting of *jungles*, which are forests of coalesced trees with shared structures that can be represented by acyclic hypergraphs, is an important application of this concept [29]. The nodes and edges of a jungle represent the terms and operations of functional or algebraic expressions. Jungle rewriting rules manipulate and evaluate the transformation of jungles, or more precisely, they establish the term rewrite rules for those expressions. This approach allows a more efficient evaluation of expressions by avoiding the multiple representation, hence the re-calculation, of repeated terms.

## 2.3 Algebraic Approach to Graph Grammars

### 2.3.1 Graph Transformation

The algebraic approach to graph grammars focuses the analysis on the rewriting mechanism of direct derivations of graphs (i.e., on graph transformations), rather than on the study of general constructs to obtain graph languages [25, 30, 82]. Its basic operation to induce graph transformations is graph gluing, where common nodes and edges in a production's $LG$ (left) and $RG$ (right) graphs play a key role in maintaining graph consistency when replacing $LG$ by $RG$ in a host graph $G$. A general outline of the approach would be, for example, given a node- and edge-labeled simple (i.e., no single-node loops, no multiple edges between two nodes) graph $G$, the application of a graph rewriting production $p$ (Section 2.1.2) yields a graph $H$ under the following conditions:

- The definition of production $p$ is based on partial graph morphisms (mainly monomorphisms) between its $LG$ (left graph) and its $RG$ (right graph) in the case of the single pushout approach (SPO) (fig. 2.2-(b)), or is based on total graph morphisms between $LG$, $RG$ and an auxiliary interface graph $K$, containing the nodes and edges required for gluing $RG$ to a context graph $D$ (fig. 2.2-(a)), in the case of the double pushout approach (DPO).

[4]A direct derivation is denoted $GH \overset{p}{\Longrightarrow} HH$. A sequence of derivations is denoted by $H_0 \overset{p_1}{\Longrightarrow} H_1 ... \overset{p_n}{\Longrightarrow} H_n$, or just $H_0 \overset{*}{\Longrightarrow} H_n$.

- The application of production $p$ is based on total graph morphisms between $LG$, $RG$ and $G$ and $H$ (i.e., match $m$ and co-match $m$*, respectively; fig. 2.2). The overall effect, represented by pushout constructions (1) and (2), or (3) in the DPO and SPO approaches, respectively, is to remove the nodes and edges of $G$ that can be mapped to $LG$ but not to $RG$, and to add to $G$ the new nodes and edges of $RG$ to produce $H$. Ignoring morphisms and just applying structural considerations, one can broadly say that $H$ is the result of the operation $(G - (LG - RG)) \cup (RG - LG)$.

Similar to all approaches, a graph grammar GG consists of a start graph $G_0$ and a finite set of productions $P = \{p_1, ...p_n\}$, each production as defined above. The language $L$(GG) generated by the grammar is the set $\{G_n \mid G_0 \overset{*}{\Rightarrow} G_n; n = 1, 2, ...\}$.

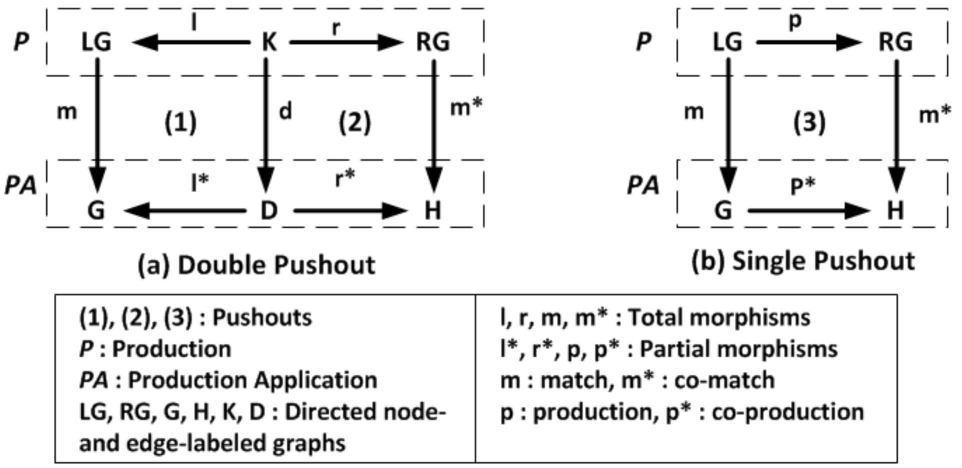
P
LG
l
K
r
RG
m
(1)
d
(2)
m*
PA
G
l*
D
r*
H
(a) Double Pushout
P
LG
p
RG
m
(3)
m*
PA
G
p*
H
(b) Single Pushout
(1), (2), (3) : Pushouts
P : Production
PA : Production Application
LG, RG, G, H, K, D : Directed node- and edge-labeled graphs
l, r, m, m* : Total morphisms
l*, r*, p, p* : Partial morphisms
m : match, m* : co-match
p : production, p* : co-production

Figure 2.2: Single and Double Pushout Production Application

The grammars obtained using this approach, generally context-sensitive, can be highly tuned to specific graph modeling situations with arbitrary start graphs and application-specific graph derivation steps. Application conditions and imperative programing structures on the scheduling and application of productions provide additional control over graph derivations and the structure of the graphs produced. The abstractions of category theory: *categories* having graphs as *objects* and derivations as *arrows*, *functors* to establish mappings between categories, and specially the *pushout* construction, are the essential tools used in the literature concerning this approach for reasoning and presenting results [23, 24, 25, 30, 82] in a very compact theoretical fashion. This categorical framework, to a certain extent, hinders the applicability of the algebraic approach to concrete problems in areas outside graph grammar theory, mainly because it overrides most of the self-expression that the grammar formalism might have and replaces it by very abstract concepts that are hard to map to real systems.

### 2.3.2 Extended Analysis

The focus on direct graph derivations, typical of the algebraic approach, facilitates detailed analyses of comparative properties, structural transformations, and semantics associated with specific subsets of productions in a grammar and on specific graphs of its generated language, which can be used to find optimized ways to transform graphs of a given class. The following is a brief overview of the properties that could be used to adapt grammar productions to specific modeling situations.

#### 2.3.2.1 Independence and Embedding of Derivations

Given a graph $G$ and productions $p_1$, $p_2$, then direct derivations $G \overset{p_1}{\Rightarrow} H_1$ and $G \overset{p_2}{\Rightarrow} H_2$ are *parallel independent* iff there exist direct derivations $H_1 \overset{p_1}{\Rightarrow} X$ and $H_2 \overset{p_2}{\Rightarrow} X$. Also, derivation $G \overset{p_1}{\Rightarrow} H_1 \overset{p_2}{\Rightarrow} X$ is *sequentially independent* iff there exists an equivalent derivation $G \overset{p_2}{\Rightarrow} H_2 \overset{p_1}{\Rightarrow} X$. Finding derivations with these properties is known as the Church-Rosser problem.

If both derivations are generated in parallel and, for example, $G \overset{p_1}{\Rightarrow} H_1$ preserves an item (i.e., node or edge) that is deleted by $G \overset{p_2}{\Rightarrow} H_2$, then one can delay applying the second direct derivation and say that $G \overset{p_2}{\Rightarrow} H_2$ is weakly parallel independent of $G \overset{p_1}{\Rightarrow} H_1$.

In a sequential derivation, for example $G \overset{p_1}{\Rightarrow} H_1 \overset{p_2}{\Rightarrow} X$, if the *match* of $H_1 \overset{p_2}{\Rightarrow} X$ depends on an item generated by $G \overset{p_1}{\Rightarrow} H_1$ or deletes an item accessed by it, then $H_1 \overset{p_2}{\Rightarrow} X$ is weakly sequentially independent of $G \overset{p_2}{\Rightarrow} H_1$.

Given two derivations $\alpha \equiv G_0 \overset{p_1}{\Rightarrow} ... \overset{p_n}{\Rightarrow} G_n$ and $\beta \equiv H_0 \overset{p_1}{\Rightarrow} ... \overset{p_n}{\Rightarrow} H_n$, one says that $\alpha$ is embedded into $\beta$, denoted by $\alpha \overset{e}{\rightarrow} \beta$, if there exists family $e = \{G_i \overset{e_i}{\rightarrow} H_i \mid i = 1, ..., n\}$ of $n$ injective morphisms. If embedding $e$ exists, then it is determined by the first injective morphism: $G_0 \overset{e_1}{\rightarrow} H_0$.

Studying the independence and embedding of derivations in known sets of graphs could be a very effective tool when attempting to infer grammars with desirable properties such as confluence and termination.

#### 2.3.2.2 Structural Manipulation of Productions

If a derivation frequently occurs in some modeling situation and the interest concentrates on the initial and final graphs, it seems natural to search for an optimization of the productions that would bypass the derivation of the intermediate graphs. This problem translates to finding a derived production $p^* : G_0 \rightsquigarrow G_n$. This production exists if, given a derivation $\alpha \equiv G_0 \overset{p_1}{\Rightarrow} ... \overset{p_n}{\Rightarrow} G_n$ and an injective morphism $G_0 \overset{e_0}{\rightarrow} H_0$, $e_0$ induces an embedding $e : \alpha \rightarrow \beta$ (where $\beta \equiv H_0 \overset{p_1}{\Rightarrow} ... \overset{p_n}{\Rightarrow} H_n$) and $H_0 \overset{p^*}{\Rightarrow} H_n$. Derived production $p^*$ for derivation sequence $\alpha$ is given by $p^* = p_1^*; ...; p_n^*$ (i.e., sequential application of co-productions $p_i^*$ (fig. 2.2-(b))).

If two productions $p_1$ and $p_2$ must work cooperatively on graphs to model, for example, a concept of synchronized execution of tasks or updates to systems states, then they must synchronize their application to common items accessed by both on those graphs. One solution to this problem is to amalgamate the shared effects of both productions into a common subproduction $p_s$, embedded in $p_1$ and $p_2$. The productions synchronized along $p_s$ are indicated as $p_1 \overset{h_1}{\leftarrow} p_s \overset{h_2}{\rightarrow}$ ($h_1$, $h_2$: injective morphisms). If $p_1$ and $p_2$ are also glued along $p_s$ (i.e., along their $LG$ and $RG$ graphs) one calls $p_s \equiv p_1 \oplus p_2$ the amalgamated production. Consistently, an amalgamated direct derivation of a graph $H$ (from a graph $G$) is denoted as $G \overset{p_1 \oplus p_2}{\Rightarrow} H$. This type of derivation is useful in transforming distributed graphs whose composition is $DG \equiv G_1 \overset{h_1}{\leftarrow} G_0 \overset{h_2}{\rightarrow} G_2$, where $G_1$ and $G_2$ are local graphs and $G_0$ is a common interface graph ($G_1 \supseteq G_0 \subseteq G_2$).

## 2.4 Triple Graph Grammars

The formalism of triple graph grammars (TGG) was devised [84] to generate and transform pairs of related graphs (usually called source and target graphs) under a well-formed synchronization mechanism (i.e., a connection graph) which would preserve the relations in the pair after applying a transformation. The initial motivation of the formalism was to provide a high-level graph-based modeling and specification tool for problems involving related diagrams (i.e., syntax trees, control flow diagrams) and information structures (i.e., requirements, design and traceability documents) requiring simultaneous update for consistency purposes. More recently, key concepts found in TGGs have been used in the OMG's QVT model transformation language standard [76] to define the "mappings" of a transformation. Also, applications of TGGs to the specification of bidirectional and incremental model transformations are becoming more common [17, 18, 86, 88, 89].

The categorical framework [30] is the standard mathematical tool used to describe TGGs: their elements and operations. TGG's elementary unit of transformation is the *graph triple*, denoted as $GT \equiv (LG \overset{gl}{\leftarrow} CG \overset{gr}{\rightarrow} RG)$, where $LG$: left graph, $CG$: connection graph, $RG$: right graph (usually node- and edge-labeled directed graphs, however, other types of graphs can be used), and $gl$, $gr$ are graph morphisms (fig. 2.3). The transformation operator is the *triple production*, denoted as $p \equiv (lp \overset{lr}{\leftarrow} cp \overset{rr}{\rightarrow} rp)$, and consisting of three monotonic single productions [5]: $lp \equiv (LL, LR)$, $cp \equiv (CL, CR)$ and $rp \equiv (RL, RR)$ (with graph morphisms $lp : LL \rightarrow LR$, $cp : CL \rightarrow CR$ and $rp : RL \rightarrow RR$, respectively, and the induced graph morphisms $lr|_{CL} : CL \rightarrow LL$, $rr|_{CL} : CL \rightarrow RL$[6], $lr : CR \rightarrow LR$ and $rr : CR \rightarrow RR$), which are applied simultaneously (fig. 2.3).

Applying a triple production $p$ to a graph triple $GT$ produces a directly derived triple $HT \equiv (LH \overset{hl}{\leftarrow} CH \overset{hr}{\rightarrow} RH)$ (fig. 2.3), derivation denoted as $GT \overset{p}{\rightsquigarrow} HT$, where the application of each of its component productions $lp$, $cp$ and $rp$ are modeled by the single pushout categorical construct described in fig. 2.2-(b). That is, productions $lp$, $cp$ and $rp$, match $LL$, $CL$ and $RL$ in graphs $LG$, $CG$ and $RG$ through total morphisms $lg$, $cg$ and $rg$, respectively. Then, they transform the latter triple into graphs $LH$, $CH$ and $RH$, which are related to the applied production and original graphs by the morphism pairs ($lh$, $lp$*), ($ch$, $cp$*), and ($rh$, $rp$*), respectively (fig. 2.3).

The basic definition of TGGs with monotonic productions, although partially restrictive, makes it more feasible to specify translator tools (i.e., graph-based model translators) based on inter-graph relationships by simplifying the manipulations (i.e., graph parsing and reconstruction) required to obtain a triple's unknown target graph that should be the matching counterpart of a known arbitrary source graph, as shown below [84].

[5]Production $p \equiv (L, R)$ is monotonic if $L \subseteq R$.

[6]$lr|_{CL}$ and $rr|_{CL}$ indicate that $lr$ and $rr$ are total morphisms over graph CL

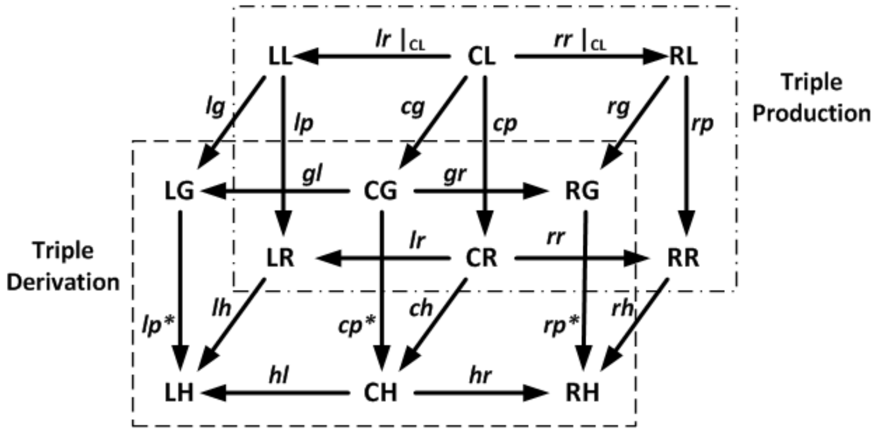

Figure 2.3: TGG Production Application

A triple production $p \equiv ((LL, LR) \overset{lr}{\leftarrow} (CL, CR) \overset{rr}{\rightarrow} (RL, RR))$ can be split into a pair of equivalent productions: a left-local production $p_L$ and a left-to-right production $p_{LR}$, with:

- $p_L \equiv ((LL, LR) \overset{\epsilon}{\leftarrow} (\phi, \phi) \overset{\epsilon}{\rightarrow} (\phi, \phi))$ [7]
- $p_{LR} \equiv ((LR, LR) \overset{lr}{\leftarrow} (CL, CR) \overset{rr}{\rightarrow} (RL, RR))$

When $p$ is applied to a triple $GT$, of which, only graph $LG$ is known, denoted by $GT \overset{p(lh)}{\rightsquigarrow} HT$ ($HT$: derived triple), if its component productions are monotonic, the following equivalence holds:

$$GT \overset{p(lh)}{\rightsquigarrow} HT \iff \exists\, XT \mid GT \overset{p_L(lh)}{\rightsquigarrow} XT \wedge XT \overset{p_{LR}(lh)}{\rightsquigarrow} HT$$

This result extends to the application of a sequence of triple productions $p_1, ..., p_n$ in the form of the following equation:

$$p_1(lh_1) \circ ... \circ p_n(lh_n) = (p_{1L}(lh_1) \circ ... \circ p_{nL}(h_n)) \circ (p_{1LR}(lh_1) \circ ... \circ p_{nLR}(lh_n))$$

which translates to say that the application of a *sequence of triple productions* is equivalent to the application of a sequence of left-local productions followed by the application of a sequence of left-to-right productions. The previous analysis is also valid for right-local and right-to-left productions.

In summary, if a triple's $LG$ and $RG$ graphs represent left (L) and right (R) information structures, respectively, then this monotonic property of TGG productions makes it tractable to specify LR- and RL-translators and L versus R correspondence analysis tools.

[7] $\phi$: empty graph; $\epsilon$: inclusion of empty graph in any graph.

## 2.5 Other Approaches

Overall, the grammar approaches presented to this point appear to be the most likely to be applied in modeling languages, and they have also served as structural foundations for building other types of grammars. However, the literature of this area includes many other alternatives ranging from studies to handle very specific graph models, for example: star grammars [20, 21, 22] and fuzzy graph grammars [78], to attempts to provide meta-generation of grammars [28].

Star grammars are a representative example of research efforts to extend desirable context-free properties found in node and hyperedge replacement grammars with additional structural properties that would increase their descriptive power. They were devised with the specific goal of being able to generate graphs that not only capture context-free properties of object-oriented programs and transformation languages, but are also useful to express other elements such as scope rules, references to variables and method overriding. A production in this grammar replaces a non-terminal node and its outgoing edges (i.e., a star) with a new graph glued to the target nodes of the replaced structure (i.e., the star's border nodes). It is possible to designate each of the border nodes as a multiple node, in which case, these nodes of the host graph can be cloned, along with its incident edges, any number of times, before applying a graph transformation rule. Overall, a production in this grammar is a schema composed of a graph replacement rule and a configuration of multiple nodes with a cloning scheme (i.e., eager or lazy cloning) [20].

Fuzzy grammars are an attempt to define transformations of fuzzy graphs, which are constructed from fuzzy points (i.e., nodes) with assigned membership functions and ordinary edges between fuzzy points, using mostly elements of category theory [78]. They are claimed to be a theoretical generalization of ordinary graph grammars given comparable fuzzy and non-fuzzy graph structures.

Adhesive high level replacement systems are an effort to define, using elements of category theory, transformations of complex structures in a rather generic manner. This highly abstract area has shown a steady progression over the years [30], with theoretical applications, for example, to the generation of multilevel graph representations [77].

# 3 Graph Transformation Systems

As described in Section 2.1, a significant body of concepts are common to all the graph grammar approaches found in the literature. Graph types, common components in grammar productions and the general structure of graph grammar themselves are some of the most representative examples of very useful abstractions in the application of grammars. When extending these ideas to applications involving large numbers of rules (i.e., productions) and complex systems, resorting to proven software engineering principles (i.e., classification, encapsulation, reuse and module-based semantics) to further structure the graph transformation process is essential if the requirement is to produce manageable and scalable graph-based systems.

Graph transformation systems [29, 59] is a formalism that represents research efforts in this direction with its focus on defining encapsulation abstractions such as transformation units and modules, using precise graph-based semantics, which can be reused and aggregated to build larger systems. Its basic structuring generalization, a *graph transformation approach* $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{E}, \mathcal{C})$, where:

- $\mathcal{G}$ is a class of graphs of a specified type (i.e., labelled graphs, hypergraphs, etc.).
- $\mathcal{R}$ is a class of rules with the same structure (i.e., single pushout, double pushout, etc.).
- $\Rightarrow$ is a rule application operator that for each rule $r \in \mathcal{R}$ produces a relation $\overset{r}{\Rightarrow} \equiv \{(g_1, g_2) \in \mathcal{G} \times \mathcal{G} \mid g_1 \overset{r}{\Rightarrow} g_2\}$.
- $\mathcal{E}$ is a class of *graph class expressions*, where each one of them can be defined by: a finite enumeration of graphs, or a set theoretic definition (i.e., directed node- and edge-labelled graph, reduced graphs for a rule set $R$, etc.), or a graph theoretic property, or the exhaustive output of a given transformation unit, or some graph meta-definition scheme (i.e., schemata), or as the result of boolean operations on graph classes.
- $\mathcal{C}$ is a class of *elementary control conditions* over a set $ID$ of identifiers, which are implemented as rule execution schedules, priority schemes, expressions (i.e., boolean operations) and languages (i.e., regular expressions). For a given environment $E$, defined by the mapping $E : ID \to 2^{\mathcal{G} \times \mathcal{G}}$ [8], each $c \in \mathcal{C}$ specifies a relation $SEM_E(c)$ consisting of pairs $(G, G') \in \mathcal{G} \times \mathcal{G}$, where $G$: initial graph, $G'$: derived graph.

allows one to refer to the multiple grammar types in an approach-independent manner that facilitates the analysis of similarities and differences in properties, structures and rules.

Given a graph transformation approach, a *transformation system* in this environment consists of a set of transformation units with the capability to import each other forming acyclic or cyclic import paths. Each transformation unit produces an output terminal graph from an initial graph by interleaving rule applications with calls to other transformation units, where the latter are executed in an atomic manner (i.e., the whole transformation unit is executed or not).

[8]Given a set A, $2^A$ denotes its power set

## 3.1 Transformation Units

A transformation unit (TU) consists of a (possibly empty) set of graph transformation rules conforming to a graph transformation approach $\mathcal{A}$ and a (possibly empty) set of atomic-execution calls to imported transformation units, where elements of both sets execute interleaving each other (”interleaving semantics”) to transform an initial graph of a given class into a specific terminal graph of the same or another class. The calls to other import transformation units can be cyclic or acyclic, depending on whether a “called unit” calls one of its “calling units” in a given calling path or not.[9] The input and output graph classes can be defined by class expressions as described in $\mathcal{A}$. The set of all transformations over a graph transformation approach $\mathcal{A}$ is denoted as $\tau_{\mathcal{A}}$.

A TU can also include a *control condition* (*cc*), composed of boolean expressions on elementary control conditions of a class $\mathcal{C}$ (as specified in $\mathcal{A}$), whose purpose is to regulate, restrict, and possibly eliminate the inherent non-determinism appearing in graph transformations due to the fact that: 1) multiple rules can simultaneously be applicable to an existing graph, 2) there can exist multiple places in a graph where a given rule could be applied. A *cc* can have properties of minimality (i.e., the graphs transformed by a TU are only the ones allowed by the *cc*), invertibility (i.e., for a given *cc* $C$, $C^{-1}$ exists) and continuity (i.e., given a TU with a *cc* $C$ over a sequence of environments $\{E_1, ..., E_n\}$, then $SEM_{E_1 \cup ... \cup E_n}(C) = SEM_{E_1}(C) \cup ... \cup SEM_{E_n}(C)$). The following are the most common types of control conditions:

- *Control conditions of language type* which only allow interleaving sequences of rules and imported TUs that can be represented by strings belonging to a specific control language $L$ (i.e., formal language generated by a grammar, automaton or regular expression). The alphabet used to build those strings consists of the identifiers $l_i$ assigned to rules and imported TUs (i.e., for each $w \in L$, $w = l_1, ..., l_n$, $n = 1, 2, ...$) and the left-to-right parsing of each string represents an interleaving execution sequence of rules and imported TUs.

- *Priorities* assigned to the rules of a TU can represent control conditions that allow only specific pairs $(G, G') \in SEM_E(\boldsymbol{C})$.

- *Reduced graphs* with respect to a control condition (denoted $C!$) is a type of control condition where the only pairs $(G, G') \in \mathcal{G} \times \mathcal{G}$ allowed, have the property that there is no graph $G''$ such that $(G', G'') \in SEM_E(C!)$.

- *Rules applied as-long-as-possible* (denoted $R!$) for a given rule set $R$ constitutes a special type of reduced graphs control condition.

- A TU can be a control condition as it defines a binary relation on graphs.

As a generative structure, a TU over a graph transformation approach $\mathcal{A}$ is a 5-tuple $TU = (IG, L, R, CC, TG)$, where:

- $IG$ and $TG$ are the initial and terminal graph class expressions, respectively.

[9]This analysis assumes acyclic calling paths.

- $L$ is a finite set of identifiers to refer to imported transformation units.
- $R$ is a finite set of labelled rules ($R \subseteq \mathcal{R}$), whose identifiers are not in $L$.
- $CC$ is a control condition.

with a graph-based interleaving operational semantics, denoted as $INTER_{SEM}(TU)$ or just $SEM(TU)$, and implemented by: 1) a function $SEM : L \to 2^{\mathcal{G}\times\mathcal{G}}$ capable of assigning a set of graphs to each identifier associated with a TU, either a graph transformation rule or an imported transformation unit, 2) a set of graphs resulting from applying the rules from $R$. The overall semantics $SEM(TU)$ of a TU is also a binary relation on graphs having pairs $(G, G')$ where $G$ and $G'$ are compliant with the initial and terminal graph class expressions, respectively, and the control condition $CC$.

TUs can also be parameterized with graph class expressions defined as parameters of specified types (i.e., all graphs, hypergraphs, etc.) which can be instantiated with expressions of the same type or any of its subtypes. This parameterization allows to specify TUs very compactly, specially in situations in where they will be imported multiple times with different graph class expressions of the same type.

## 3.2 Transformation Modules

Efficient specification of very large systems using languages based on graph transformation would require structuring the transformation units in a rather hierarchical and encapsulated way, so as to be able to identify, for example, the ones that are central to a given graph-related semantic aspect of an application from the ones that represent the interfacing to other systems, and therefore, to provide information hiding. The definition of a transformation module [29], as a set of transformation units, some hidden to the rest of the specification and some visible through an export interface, with the capability of importing external transformation units as instantiations of parameters and a binary graph-based semantics comparable to the one defined for TUs, attempts to capture these concepts. The following describes one formalization of this concept based on a proposal for providing structuring facilities in a rule- and graph-based specification and programming language which is independent of any specific graph transformation approach [29, 43].

A *transformation module* over a graph transformation approach $\mathcal{A}$ [10] is a triple $MOD = (IMPORT, BODY, EXPORT)$ where:

- $IMPORT$ is a set of *imported names* referring to external TUs.
- $BODY$ and is a finite set of local named TUs, each of which can use TUs from sets $BODY$ and $IMPORT$ only, and named rules over some set $L$ of names.
- $EXPORT$ is a set of exported names referring to TUs in $BODY$ and/or $IMPORT$.

with the following restrictions:

[10]The set of all transformation modules over a graph transformation approach $\mathcal{A}$ is denoted $\tau_{\mathcal{A}}$

- Imported names cannot be reused for a local rule or TU (i.e., $L \cap IMPORT = \phi$).
- Local TUs use only names that are imported or represent local TUs to refer to imported TUs.

The semantics of a module over a graph transformation approach $\mathcal{A}$, denoted as $SEM_{E_{mod}}(MOD)$, defines an environment $E_{mod}$ that associates binary relations on graphs to each imported name, each local rule, each locally defined TU, and each exported name in the $EXPORT$ set. The export semantics of a module is obtained as the restriction of $E_{mod}$ to names in $EXPORT$ only.

There have been other studies to provide modularization to graph-transformation-based systems (e.g., PROGRES) which have used, for example, encapsulation concepts from object-oriented systems or the package concept of UML [29]. However, they have focused on using specific graph transformation approaches as opposed to approach independent formalisms, perhaps envisioning optimizations in the specification of particular types of systems (i.e., embedded, distributed systems).

## 3.3 Model Transformation Units

As indicated in Section 1.2.4, the application of graph rewriting to model transformations revolves around the idea of mapping models onto graphs and then applying graph transformation rules to obtain the graph representations of the transformed models. One could establish a parallel of this paradigm on a TU and consider that its initial and terminal graphs are the representations of input and output models respectively, which would make it a type of *model transformation unit* (MTU) instead. However, although MDA oriented models are generally diagrammatic or textual, hence reasonably representable by graphs, other types of models, containing tuples, sequences and sets of models, cannot be represented in this manner. Therefore, the definition of a MTU as a type of TU requires some further inclusive abstraction of the latter formalism such as the one proposed in [56]:

- Models have a type that could be one of: 1) basic: class $\mathcal{G}$ of graphs, set $ID$ of identifiers, boolean, or a natural number, 2) cartesian product $T_1 \times ... \times T_k$, where each $T_i$ $(i = 1, ..., k)$ is a model type, 3) sequence $T^*$ of models with a model type, 4) set of models with a model type.
- Actions are applied to models. An action is a tuple $a = (a_1, ..., a_k)$ where each $a_i$ $(i = 1, ..., k)$ is one of: 1) a void action causing no change on an input model, 2) a graph transformation rule, 3) a rename of an identifier, 4) an operation on natural numbers, 5) a boolean operation, 6) a recursive action on a tuple.
- Similar to the specification of a TU, a MTU is a tuple $(ITD, OTD, WT, A, C)$ where: 1) $WT$ is a product type (working type), 2) $ITD$ is the *input type declaration* which consists of a constrained product type and a mapping to initial models, 3) $OTD$ is the *output type declaration* which consists of a constrained product type and a mapping to terminal models, 4) $A$ is the set of actions, and 5) $C$ is the control condition.

# 4 Graph-Based Model Transformation

The suitable use of graphs and graph transformations for representing models and model transformations, respectively, in an environment where accurate and thorough domain analysis is a common practice, could potentially provide the tools to define domain modeling and transformations languages with very precise syntax and semantics. The following sections provide an overview of some common types of transformations and how they can be supported by the graph modeling paradigm.

## 4.1 Model Transformations

A great deal of theoretical and applied research on model transformations has focused on studying specific approaches, design tools and techniques that apply within rather specialized scopes with less emphasis on interoperability or integration considerations [1, 6, 63]. In contrast, the OMG's MDA standard encourages unification principles for modeling languages, models and their transformations [76]. Independently from the type and intended purpose of model transformations, a significant set of their properties and features such as rules, scope, directionality and architecture, are common to all initiatives and warrant further analysis. The goal is to visualize how the model transformation problem can be decomposed into manageable parts, where techniques such as graph transformations can be applied to produce overall solutions characterized by accurate syntactic translations, reliable preservation of model semantics and efficient implementations.

### 4.1.1 Core Concepts

Reliable transformations on language-based models, as processes, have to correctly interpret the syntax and semantics of the modeling language(s) (i.e., metamodels) defining the source models, and at the same time, correctly reproduce the syntax and semantics of the modeling language(s) defining the target models , with the additional task of accurately reconstructing what a user-specified environment would be for the latter models. During software development, the views of the system, represented by models, can change (i.e., model-to-model transformations), and models are refined to lower abstraction levels (i.e., PIM-to-PSM transformations) to generate system implementations, ideally altering neither the type of representations they are in (i.e., token or type models) nor their scopes. The following model, adapted from [69], provides a unifying and systemic understanding of these multiple options occurring in MDE development, which would include graph languages:

- A modeling language (ML) is a formalism which precisely defines the notation and meaning of a model (fig. 4.4).
- The ML's notation consists of concrete syntax (i.e., user readable elements) and abstract syntax (i.e., abstract notational elements).

- The ML's meaning consists of the static semantics (i.e., well-formedness of structural elements) and dynamic semantics (i.e., behavioural elements restricting the set of valid models under the formalism).

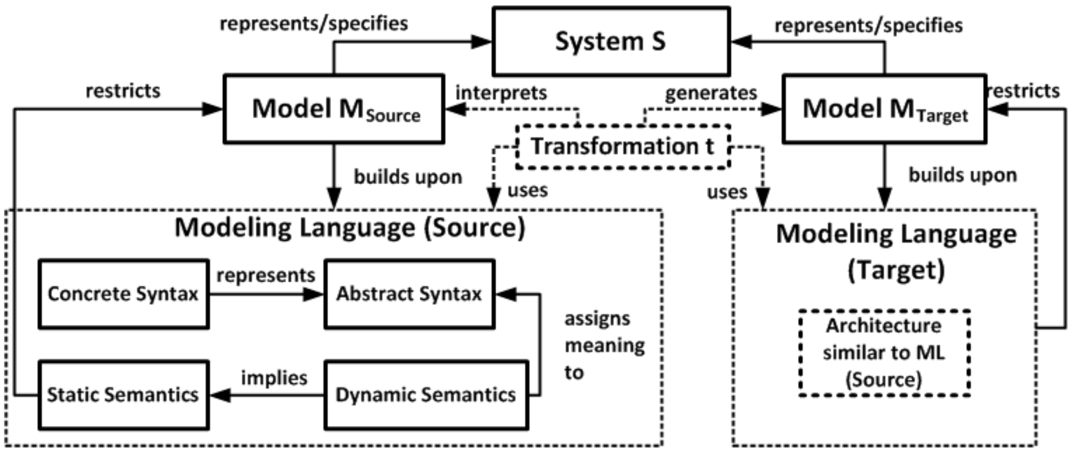

Figure 4.4: Modeling and Model Transformation Architecture

Using the conceptual model of fig. 4.4, one can define a model transformation as a mapping:

$$f : M_{Source}(S)|_{ML_{Source}} \rightarrow M_{Target}(S)|_{ML_{Target}}$$

where $ML_{Source}$ and $ML_{Target}$ are the modeling languages defining the source $ML_{Source}(S)$ and target $M_{Target}(S)$ models of the system $S$, respectively.

Visualizing and interpreting some of the existing relationships and elements in fig.4.4, one can provide a brief taxonomy of model transformations, based on the more relevant properties, as follows:

- *Directionality*: a unidirectional transformation only interprets its source and reproduces its target models. A bidirectional transformation can also switch interpretation to target and generation to source models.

- *Degree of evolution*: a horizontal transformation basically generates an equivalent view of the system at the same abstraction level, often using the same source and target modeling languages (i.e., endogen transformation). A vertical transformation generally uses different modeling languages for source and target models (i.e., exogen transformation), for example in code generation, and reduces the abstraction level.

- *Implementation type*: a transformation can be implemented using a declarative approach based on rules and pre/post-conditions, or an operational approach which explicity specifies the sequence of actions to transform a model.

- *Atomicity*: a transformation can be executed as a single non-decomposable step or as multiple steps allowing for partial rollbacks.

- *Degree of automation*: the configuration and execution of a transformation can be completely automated by a software tool, or it can be partially automated, requiring user intervention to complete some manual steps.

Other classifications of transformations exist [16, 69, 67, 87], but the properties considered tend to be refinements of the five above.

### 4.1.2 Bidirectional Transformations

The development of complex and large software systems using the MDE approach usually requires designing, for a given set of specifications, at different levels of abstraction and applying separation of concerns techniques [53]. This approach can produce sets of of highly interrelated models of dissimilar sizes and nature, some being the direct output of applying a transformation to other models in a set, some requiring partial or total synchronization with syntactic, or semantic, or both aspects of other models [38]. Given the iterative nature of the software development process, it might not be practical or even efficient in this environment to have a scheme of only batch-oriented unidirectional incremental transformations to restore consistency to a set of models after each cycle of changes. Bidirectional transformations, implemented as declarative programs based, for example, on the triple graph grammar formalism [31, 38] or a specific language such as QVT-Relations [89], can capably generate from formal specifications the operations necessary to support the automated reconciliation, verification and traceability of model synchronization changes. Having been incorporated to the OMG's QVT standard, they are currently the subject of active research [17, 88, 89].

When a transformation of this type is formally specified in a declarative (relational) manner, without any hand-crafted functionality, the following definition accurately describes it:

Given the sets $\mathcal{M}_1$ and $\mathcal{M}_2$ of *source* and *target* models respectively, a bidirectional transformation between them is a pair $\mathcal{T} = (t, t^{-1})$, where $t : \mathcal{M}_1 \to \mathcal{M}_2$ is an injective function (forward transformation) and its inverse $t^{-1} : \mathcal{M}_2 \to \mathcal{M}_1$ (backward transformation) exists. In other words, for any model $m_1 \in \mathcal{M}_1$, there exists only one $m_2 \in \mathcal{M}_2$ such that $m_2 = t(m_1)$ and $t^{-1}(m_2) = m_1$. Any pair of models $m_1 \in \mathcal{M}_1$, $m_2 \in \mathcal{M}_2$ related by this functionality are in a consistency relation or contract that would make them synchronized system abstractions accepted by all their stakeholders [38, 88].

When a bidirectional transformation also includes ad-hoc programming, usually, the notion of consistency is limited to specific instance pairs $(m_1, m_2)$ of models, and to specific elements in the models of the pair. A transformation $t : m_1 \to m_2$, emphasizing a relation between model *elements* needs to be injective or bijective so that $t^{-1}$ exists, and consequently one is able to define a bidirectional transformation between $m_1$ and $m_2$.

Suitable support for these types of transformation schemes has been built into bidirectional transformation languages (BTLs) such as QVT-Relations [89], which allows one to develop programs (i.e., specifications) capable of describing a forward transformation and its associated backward transformation simultaneously, therefore guaranteeing their compatibility by construction. It is possible to use BTLs to build bijective transformations, with forward and inverse injective functions, and also generically bidirectional ones, where arbitrary forward transformations may drop information and reverse transformations process original and updated models.

## 4.2 Model Transformations Using Graphs

Models in MDE routinely express complex structural and dynamic aspects of a system by using fairly intuitive visual languages (i.e., diagrams). If one formalizes syntactic (i.e., presentation an logical structure) and semantic (i.e., interpretation) aspects of these languages using graphs, then one can make use of graph grammars and graph transformations (Sections 2 and 3) to specify precisely how these models should be built and how they should be transformed [7, 39]. The main reason proposed in the literature for pursuing this formalization approach is to avoid using graphs in a restrictive ad-hoc manner to address problems that have known solutions in more general contexts.

Graph transformations, with their defined graph types and rule sets, provide a solid foundation for reasoning about structural, semantic and operational aspects of model transformations in the following roles:

- as a *semantic domain* that directly provides a generic specification language and semantic model for very high level definitions of application domains, their abstractions (i.e., models) and the modifications of those abstractions (i.e., transformations) to support software development activities such as specification of functional requirements (e.g., object dynamics) and description of architectural changes [11, 39, 44].

- as a *meta-language* to be used in the formal specification of the syntax, semantic and manipulation rules of the DSLs (i.e., metamodels) and model transformation languages (MTLs) defining source/target models and model transformations, respectively [1, 48, 61].

### 4.2.1 Graph-Based Metamodeling

Metamodeling is a widely accepted technique used to define the rules (e.g., abstract syntax of static diagrams) of visual/diagrammatic languages, which provides a flexible configuration environment, especially useful when working with DSLs, and allows the checking of whether a model produced in a specific language is valid or not [3, 64]. The construction of a metamodel requires two basic elements: 1) a meta-language capable of describing the metamodel's structure and relations between modeling items (e.g., class diagrams, graphs) and 2) an instantiation relationship indicating how model instances will be generated as a result of using the metamodel.

In the context of graphs one can formally think of metamodels as sets of typed graphs and of models as sets of instance graphs obtained from enacting type-instance mappings from the former sets. Furthermore, given a model of a certain type (e.g., diagrams, object structure, architectures), one can specify its transformations (e.g., visual/notational representation changes, functional requirements, architectural changes) as graph transformation rules over the typed graphs of the associated metamodel. These graph-based transformations are sufficiently generic to: 1) specify syntactic changes when using different source and target modeling languages and 2) define semantic mappings when the target language is used to define a semantic domain for the source language.

### 4.2.2 Triple Graph Grammars and Transformations

Having many similarities with the model transformation languages specified in the OMG's QVT standard, triple graph grammars (TGG) represent an important alternative for specifying automated PIM-to-PIM and PIM-to-PSM incremental model transformations in a *declarative* manner, with an implicit control structure over the execution of the transformations. The basic graph structure and transformation scheme they support - a graph triple consisting of a source, a target and a correspondence graph, where the latter graph acts as a *mapping* between the elements of the first two, along with three single-graph grammars operating in parallel over the triple (Section 2.4) - makes them a very suitable tool for specifying horizontal and vertical incremental transformations to synchronize models [38], and also transformations to just translate models in a forward or backward direction while maintaining an overall logical relationship between source and target patterns [92].

Similarly to the case of single graphs, one can define a triple metamodel embodied in a typed triple consisting of typed source, target and correspondence graphs, that determines the nature of the graphs allowed in an instance triple [91].

A number of software tools (Section 1.2.5), mainly research prototypes, have included support for working with TGGs on tasks such as grammar prototyping and grammar-based software specification:

- PROGRES, as a general-purpose graph rewriting system, supports the specification of TGGs through its graph schema definition language facility, which also allows one to define typed triples as well as attach negative application conditions (NACs) to grammar rules [39, 85].

- AToM$^3$, as a multi-formalism and metamodeling tool, provides built-in support for defining and using TGGs as driving engines for model transformations [40, 60].

- The FUJABA tool suite allows the definition of typed triples and additional execution controls on rules in its graph schemata system, in this case using UML class diagrams and story diagrams [37, 75].

- MOFLON makes use of story driven modeling (SDM) and TGGs to specify model transformation and synchronization operations. It also provides a TGG editor and can generate SDM rules from TGG specifications [2, 93].

# 5 Open Problems and Conclusions

## 5.1 Open Problems

Research on the application of graph grammars and graph transformations to model transformations is currently fairly active, showing two main polarizing trends: 1) on the one hand, graph grammar researchers continue to increase the abstraction level of the grammar formalisms by basing their analyses on purely theoretical fields (i.e., category theory) and 2) on the other hand, researchers in software engineering are interested in automating the different aspects of model transformations using a graph transformation approach considered generically compatible, although not obviously the most suitable for the transformation problems at hand. There is a noticeable gap between both trends that gives way to an interesting set of open problems. The following is a small representative sample of open problems:
1) Related to graph grammars.

- At the present time, the DPO approach to graph transformations and triple graph grammars appear to have a higher potential to support complex model transformations. Therefore any studies of these approaches, involving confluence and termination of specific types of grammars, would make graph-based unidirectional and bidirectional model transformations a more viable declarative alternative.
- Undoubtedly, studies on more specialized types of grammars such as stochastic and fuzzy grammars that could also support model transformations of a similar nature, would be most interesting.

2) Related to graph transformation.

- The computational complexity of the graph transformation has long been an issue when considering its use in large and computationally intensive systems. It is well known that the sub-graph isomorphism problem, resulting from matching sub-graph structures within graphs using a "brute-force" approach, is NP-complete. However, the literature describes some algorithms for efficient graph pattern matching in specific situations. These studies open possibilities for research into determining which graph structures, and in which situations, should be in the language generated by a graph grammar or a set of graph transformations, which is intended to be an efficient modeling formalism of a given application domain.
- Research on algorithms that would improve the parallelism and distribution in the processing of the graph transformations would certainly alleviate the addressing of problems such as the sub-graph isomorphism.
- Finding specialized graph structures that would perform well in the specification and implementation of graph- and metamodel-based model transformations would make this declarative approach more suitable for building complex model transformations.

3) Related to integrated graph-based transformation tools.

- The research community has built mostly prototyping tools that are often cumbersome to use, offer solutions for very narrow paradigms and offer incomplete support for systems integration and management. Solving these deficiencies to make graph transformations a mainstream technology, from the software engineering viewpoint, requires further research into the requirements that industrial strength graph-based tools should have, given the nature of the upcoming applications in science, engineeering and business.

- The trend in academic research is and will continue to be towards building graph-based architectures and systems compatible with standards to ensure a more synergistic effect of the results obtained from the different studies.

## 5.2 Conclusions

In general, this work has found that there is a large gap between a theoretical body of numerous graph grammar and graph transformation approaches and its application to practical problems in software engineering. Their use in formalizing syntactic and semantic aspects of software artifacts still remains far removed from proven software engineering methodologies and tools that have produced reliable, efficient, user-friendly and maintainable applications. The graph transformation research community has recently become aware of these deficiencies and has responded with studies placing the graph transformation in a more systemic context of transformation units and systems, although with a marked theoretical flavour.

In addition, if the graph transformation is to fit better as a foundation for building applications, then many factors indicate that there exists an imperative need to evaluate and manage its performance (i.e., address computational complexity issues) on relevant graph instances, before continuing to explore additional formalizations that tend to clutter the effectiveness of the existing approaches.

## 5.3 Future Research

The continuation of this research will mainly focus on potential applications of the graph transformation to the "Model and Pattern Formalization Project", sponsored by the canadian research network NECSIS to study model-driven software solutions for the automotive sector. Within this context, pattern languages, tools, evaluation and management of graph transformation complexity issues, domain-specific applicability and standardization issues will be investigated.

# Bibliography

[1] A. Agrawal, G. Karsai, and F. Shi. Graph Transformations on Domain-Specific Models. Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, USA, 2003.

[2] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMA-FA 2006)*, LNCS, vol. 4066, pages 361-375. Springer, 2006.

[3] C. Amelunxen, E. Legros, A. Schürr, and I. Stürmer. Checking and Enforcement of Modeling Guidelines with Graph Transformations. In *3rd International Workshop on Application of Graph Transformations with Industrial Relevance (AGTIVE'07)*, LNCS, vol. 5088, pages 241-255. Springer, 2008.

[4] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump d, A. Schürr, and G. Taentzer. Graph Transformation for Specification and Programming. Science of Computer Programming, vol. 34, pages 1-54. Elsevier, 1999.

[5] D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai. The Graph Rewriting and Transformation Language: GReAT. In *3rd International Workshop on Graph Based Tools (GraBats 2006)*, Electronic Communications of the EASST, vol. 1.

[6] A. Balogh and D. Varró. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In *2006 ACM Symposium on Applied Computing (SAC'06)*, New York, USA, 2006.

[7] L. Baresi and R. Heckel. Tutorial Introduction to Graph Transformation: a Software Engineering Perspective. In *1st International Conference on Graph Transformations (ICGT 2002)*, LNCS, vol. 2505, pages 402-429. Springer, 2002.

[8] J. Bézivin. UML: The Birth and Rise of a Standard Modeling Notation. In *1st International Workshop on the Unified Modeling Language (UML '98)*, LNCS, vol. 1618. Springer, 1999.

[9] J. Bézivin and O. Gerbe. Towards a Precise Definition of the OMG/MDA Framework. In *16th IEEE international conference on Automated Software Engineering (ASE '01)*, pages 273-280, Washington DC, USA, 2001. IEEE Computer Society.

[10] J. Bézivin. In Search of a Basic Principle for Model Driven Engineering. *UPGRADE -The European Journal for the Informatics Professional*, vol. 2, pages 21-24, April 2004.

[11] T. Buchmann, A. Dotor, S. Uhrig, and B. Westfechtel. Model-Driven Software Development with Graph Transformations: A Comparative Case Study. In *3rd International Workshop on Application of Graph Transformations with Industrial Relevance (AGTIVE '07)*, LNCS, vol. 5088, pages 345-360. Springer, 2008.

[12] T. Buchmann, A. Dotor, S. Uhrig, and B. Westfechtel. Triple Graph Grammars or Triple Graph Transformation Systems? A Case Study for Software Configuration Management. In *Models in Software Engineering - Workshops and Symposia at MODELS 2008*, LNCS, vol. 5421, pages 138-150. Springer, 2009.

[13] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In *17th IEEE international conference on Automated Software Engineering (ASE '02)*, pages 267-270, Washington DC, USA, 2002.

[14] B. Courcelle. An Axiomatic Definition of Context-Free Rewriting and its Application to NLC Graph Grammars. Theoretical Computer Science, vol. 55, issues 2-3, pages 141-181. Elsevier, 1987.

[15] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, Anaheim, CA, USA, 2003.

[16] K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. IBM Systems Journal, vol. 45, issue 3, pages 621-645, 2006.

[17] K. Czarnecki, J. N. Foster, Z. Hu, R. Lammel, A. Schurr, and J. F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Pespective. In *Theory and Practice of Model Transformations: Second International Conference (ICMT2009)*, LNCS, vol. 32, pages 260-283. Springer, 2009.

[18] D. Dang and M. Gorgolla. On Integrating OCL and Triple Graph Grammars. In *Models in Software Engineering - Workshops and Symposia at MODELS 2008*, LNCS, vol. 5421, pages 124-137. Springer, 2009.

[19] T. Dean and J. Cordy. A Syntactic Theory of Software Architecture. IEEE Transactions on Software Engineering, vol. 21, issue 4, pages 302 -313, 1995.

[20] F. Drewes, B. Hofmann, D. Janssens, M. Minas, and N. Van Eetvelde. Adaptive Star Grammars. In *3rd International Conference on Graph Transformations (ICGT 2006)*, LNCS, vol. 4178, pages 77-91. Springer, 2006.

[21] F. Drewes, B. Hofmann, D. Janssens, M. Minas, and N. Van Eetvelde. Shaped Generic Graph Transformation. In *3rd International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007)*, LNCS, vol. 5088, pages 201-216. Springer, 2008.

[22] F. Drewes, B. Hofmann, and M. Minas. Adaptive Star Grammars for Graph Models. In *4th International Conference on Graph Transformations (ICGT 2008)*, LNCS, vol. 5214, pages 442-457. Springer, 2008.

[23] H. Ehrig, M. Pfender, and H. J. Schneider. Graph Grammars: an Algebraic Approach. In *14th Annual Symposium on Automata and Switching Theory*, pages 167-180. IEEE, 1973.

[24] H. Ehrig. Embedding Theorems in the Algebraic Theory of Graph Grammars. In *1977 International FCT-Conference on Fundamentals of Computation Theory*, LNCS, vol. 56, pages 245-255. Springer, 1977.

[25] H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars (A Survey). In *International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, LNCS, vol. 73, pages 1-69. Springer, 1979.

[26] H. Ehrig. Aspects of Concurrency in Graph Grammars. In *2nd International Workshop on Graph Grammars and their Application to Computer Science*, LNCS, vol. 153, pages 58-81. Springer, 1983.

[27] H. Ehrig, A. Habel. Graph Grammars with Application Conditions. In G. Rozenberg, A. Salomaa, eds., The Book of L, pages 87-100, Springer-Verlag, 1986.

[28] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. From Graph Grammars to High Level Replacement Systems. In *4th International Worshop on Graph Grammars and their Application to Computer Science*, LNCS, vol. 532, pages 269-291. Springer, 1991.

[29] H. Ehrig, G. Engels, H. J. Kreowski, and G.Rozenberg, editors. Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages and Tools. World Scientific Publishing, 1999.

[30] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Fundamentals of Algebraic Graph Transformation. EATCS Series. Springer, 2006.

[31] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer. Information Preserving Bidirectional Model Transformations. In *10th International Conference on Fundamental Approaches to Software Engineering (FASE 2007)*, LNCS, vol. 4422, pages 72-86. Springer, 2007.

[32] H. Ehrig, C. Ermel, F, Hermann, and U. Prange. On-the-Fly Construction, Correctness and Completness of Model Transformations Based on Triple Graph Grammars. In *12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*, LNCS, vol. 5795, pages 241-255. Springer, 2009.

[33] G. Engels, R. Gall, M. Nagl, and W. Schäfer. Software Specification Using Graph Grammars. Computing, vol. 31, pages 317-346. Springer, 1983.

[34] E. Engels and W. Schäfer. Graph Grammar Engineering: a Method for the Development of an Integrated Programming Support Environment. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, LNCS, vol. 186, pages 179-193. Springer, 1985.

[35] E. Engels, C. Lewerentz and W. Schäfer. Graph Grammar Engineering: A Software Specification Method. In *3rd International Workshop on Graph Grammars and their Application to Computer Science*, LNCS, vol. 291, pages 186-201. Springer, 1987.

[36] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *6th International Workshop on Theory and Application of Graph Transformations*, LNCS, vol. 1764, pages 157-167. Springer, 2000.

[37] C. Fuss, C. Mosler, Ulrike Ranger, and E. Schultchen. The Jury is still out: A Comparison of AGG, Fujaba, and PROGRES. In *6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, Electronic Communications of the EASST, vol. 6, 2007.

[38] H. Giese and R. Wagner. From Model Transformation to Incremental Bidirectional Model Synchronization. Journal of Software and Systems Modeling, vol. 8, number 1, pages 21-43. Springer, 2009.

[39] L. Grunske, L. Geiger, A. Zundorf, N. van Eetvelde, P. van Gorp, and D. Varro. Using Graph Transformation for Practical Model-Driven Software Engineering. Model-Driven Software Development, pages 91-117. Springer, 2005.

[40] E. Guerra and J. de Lara. Event-Driven Grammars: Towards the Integration of Metamodelling and Graph Transformation. In *Graph Transformations*, LNCS, vol. 3256, pages 54-69. Springer, 2004.

[41] A. Habel. Hyperedge Replacement: Grammars and Languages. LNCS, vol. 643. Springer, 1992.

[42] D. Harel and B. Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff. Technical Report MCS00-16, Weizmann Institute of Science, 2000.

[43] R. Heckel, B. Hoffmann, P. Knirsch and S. Kuske. Simple Modules for GRACE. In *Theory and Applications of Graph Transformations*, LNCS, vol. 1764, pages 383-395. Springer, 2000.

[44] R. Heckel. Graph Transformation in a Nutshell. In *School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004)*. Electronic Notes in Theoretical Computer Science, vol. 148, issue 1, pages 187-198. Elsevier, 2006.

[45] W. Hesse. More matters on (meta-)modelling: remarks on Thomas Kuhne's "matters". Journal of Software and Systems Modeling, vol. 5, number 4, pages 387-394. Springer, 2006.

[46] M. Himsolt. Graph$^{Ed}$: An Interactive Tool for Building Graph Grammars. In *4th International Worshop on Graph Grammars and their Application to Computer Science*, LNCS, vol. 532, pages 61-65. Springer, 1991.

[47] D. Janssens and G. Rozenberg. On the Structure of Node-Label-Controlled Graph Languages. Information Sciences, vol. 20, pages 191-216. Elsevier, 1980.

[48] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. Journal of Universal Computer Science, vol. 9, issue 11, pages 1296-1321, 2003.

[49] G. Karsai and A. Agrawal. Graph Transformations in OMG's Model-Driven Architecture. In *2nd International Workshop on Application of Graph Transformations with Industrial Relevance (AGTIVE '03)*, LNCS, vol. 3062, pages 243-259. Springer, 2004.

[50] G. Karsai. Automotive Software: A Challenge and Opportunity for Model-Based Software Development. In *Automotive Software - Connected Services in Mobile Networks*, LNCS, vol. 4147, pages 103-115. Springer, 2006.

[51] G. Karsai. Lessons Learned from Building a Graph Transformation System. In *Graph Transformations and Model-Driven Engineering*, LNCS, vol. 5765, pages 202-223. Springer, 2010.

[52] S. Kent. Model Driven Engineering. In *3rd International Conference on Integrated Formal Methods*, LNCS, vol. 2335, pages 286-298. Springer, 2002.

[53] F. Klar, A. Königs and A. Schürr. Model Transformation in the Large. In *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium of the Foundations of Software Engineering*, pages 285-294. ACM, 2007.

[54] A. Königs and A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. In *School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004)*. Electronic Notes in Theoretical Computer Science, vol. 148, issue 1, pages 113-150. Elsevier, 2006.

[55] A. Königs. Model Integration and Transformation - A Triple Graph Grammar-based QVT Implementation. PhD thesis, Darmstadt University of Technology, 2009.

[56] H.-J. Kreowski, S. Kuske, and C. von Totth. Stepping from Graph Transformations Units to Model Transformation Units. In *International Colloquium on Graph and Model Transformation (GraMot 2010)*, Electronic Communications of the EASST, vol. 30, 2010.

[57] T. Kühne. Matters of (Meta-) Modeling. Journal of Software and Systems Modeling, vol. 5, number 4, pages 369-385. Springer, 2006.

[58] T. Kühne. Clarifying matters of (meta-) modeling: an author's reply. Journal of Software and Systems Modeling, vol. 5, number 4, pages 395-401. Springer, 2006.

[59] S. Kuske. Transformation Units - A Structuring Principle for Graph Transformation Systems. PhD thesis, University of Bremen, 2000.

[60] J. de Lara and H. Vangheluwe. AToM$^3$: A Tool for Multi-Formalism and Meta-Modelling. In *5th Conference on Fundamental Approaches to Software Engineering (FASE '02)*, LNCS, vol. 2306, pages 174-188. Springer, 2002.

[61] J. de Lara, E. Guerra, and H. Vangheluwe. Meta-Modeling, Graph Transformation and Model Checking for the Analysis of Hybrid Systems. In *2nd International Workshop on Application of Graph Transformations with Industrial Relevance (AGTIVE '03)*, LNCS, vol. 3062, pages 292-298. Springer, 2004.

[62] J. de Lara, H. Vangheluwe, and M. Fonseca. Meta-Modeling and Graph Grammars for Multi-paradigm Modelling in AToM$^3$. Journal of Software and Systems Modeling, vol. 3, number 3, pages 194-209. Springer, 2004.

[63] E. Legros, W. Schäfer, A. Schürr, and I. Stürmer. MATE - A Model Analysis and Transformation Environment for MATLAB Simulink. In *International Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*, LNCS, vol. 6100, pages 323-328. Springer, 2011.

[64] T. Levendovzky, L. Lengyel, and T. Mézáros. Supporting Domain-Specific Model Patterns with Metamodeling. Journal of Software and Modeling Systems, vol. 8, number 4, pages 501-520. Springer, 2009.

[65] M. Löwe, M. Beyer. AGG - An Implementation of Algebraic Graph Rewriting. In *5th International Conference on Rewriting Techniques and Applications*, LNCS, vol. 690, pages 451-456. Springer, 1993.

[66] T. Mens. On the Use of Graph Transformations for Model Refactoring. In *Generative and Transformational Techniques in Software Engineering (GTTSE 2006)*, LNCS, vol. 4143, pages 219-257. Springer, 2006.

[67] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. In *International Workshop on Graph and Model Transformation (GraMoT 2005).* Electronic Notes in Theoretical Computer Science, vol. 152, pages 125142. Elsevier, 2006.

[68] T. Mens, P. Van Gorp, D. Varró and G. Karsai. Applying a Model Transformation Taxonomy to Graph Transformation Technology. In *International Workshop on Graph and Model Transformation (GraMoT 2005).* Electronic Notes in Theoretical Computer Science, vol. 152, pages 143-159. Elsevier, 2006.

[69] A. Metzger. A Systematic Look at Model Transformations. Model-Driven Software Development, pages 19-33. Springer, 2005.

[70] P.-A. Muller, F. Fondement, and B. Baudry . Modeling Modeling. In *12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*, LNCS, vol. 5795, pages 2-16. Springer, 2009.

[71] M. Nagl. A Tutorial and Bibliographical Survey on Graph Grammars. In *International Workshop on Graph Grammars and their Application to Computer Science and Biology*, LNCS, vol. 73, pages 70-126. Springer, 1979.

[72] M. Nagl. Set Theoretic Approaches to Graph Grammars. In *3rd International Workshop on Graph Grammars and their Application to Computer Science*, LNCS, vol. 291, pages 41-54. Springer, 1987.

[73] M. Nagl, A. Schürr. A Specification Environment for Graph Grammars. In *4th International Worshop on Graph Grammars and their Application to Computer Science*, LNCS, vol. 532, pages 599-609. Springer, 1991.

[74] S. Neema and G. Karsai. Software for Automotive Sytems: Model-Integrated Computing. In *Automotive Software - Connected Services in Mobile Networks*, LNCS, vol. 4147, pages 116-136. Springer, 2006.

[75] U. Nickel, J. Niere, and A. Zundorf. The FUJABA environment. In *22nd International Conference on Software Engineering (ICSE'00)*, pages 742-745, New York, NY, USA. ACM, 2000.

[76] Object Management Group. Meta Object Facility (MOF) 2.0 Query / View / Transformation Specification, version 1.1, January 2011.

[77] F. Parisi-Presicce. Transformations of Graph Grammars. In *5th International Worshop on Graph Grammars and their Application to Computer Science*, LNCS, vol. 1073, pages 428-442. Springer, 1996.

[78] N. Parasyuk and S. V. Yershov. Categorical Approach to the Construction of Fuzzy Graph Grammars. Journal of Cybernetics and Systems Analysis, vol. 42, number 4, pages 570-581. Springer, 2006.

[79] J. L. Pfaltz and A. Rosenfeld. Web Grammars. In *1st International Joint Conference On Artificial Intelligence*, pages 609-620, Washington, D.C., USA, 1969.

[80] J. T. W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. Journal of Computer and Systems Science, vol. 5, issue 6, pages 560-595. Elsevier, 1971.

[81] R. Prieto-Díaz. Domain Analysis: an Introduction. ACM SIGSOFT Software Engineering Notes, vol. 15, issue 2, pages 47-54, 1990.

[82] G. Rozenberg, editor. Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1: Foundations. World Scientific Publishing, 1997.

[83] H. J. Schneider. Graph Grammars. *1977 International FCT-Conference on Fundamentals of Computation Theory*, LNCS, vol. 56, pages 314-331. Springer, 1977.

[84] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *20th International Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS, vol. 903, pages 151-163. Springer, 1995.

[85] A. Schürr, A. J. Winter, and A. Zündorf. Graph Grammar Engineering with PROGRES. In *5th European Software Engineering Conference (ESEC '95)*, LNCS, vol. 989, pages 219-234. Springer, 1995.

[86] A. Schürr and F. Klar. 15 Years of Triple Graph Grammars: Research Challenges, New Contributions, Open Problems. In *4th International Conference on Graph (ICGT 2008)*, LNCS, vol. 5214, pages 411-425. Springer, 2008.

[87] S. Sendall and W. Kozaczynski. Model Transformation - the Heart and Soul of Model-Driven Software Development. Software, IEEE, vol. 20, number 5, pages 42-45, 2003.

[88] P. Stevens. A Landscape of Bidirectional Model Transformations. In *Generative and Transformational Techniques in Software Engineering II (GTTSE 2007)*, LNCS, vol. 5235, pages 408-424. Springer, 2008.

[89] P. Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. Journal of Software and Systems Modeling, vol. 9, number 1, pages 7-20. Springer 2010.

[90] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *2nd International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003)*, LNCS, vol. 3062, pages 446-453. Springer, 2004.

[91] G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, D. Varro, and S. Varro-Gyapay. Model Transformation by Graph Transformation: A Comparative Study. In *Model Transformation in Practice (MTiP'05) Workshop at MODELS'05*, Jamaica, 2005.

[92] D. Varró, M. Asztalos,D. Bisztray, A. Boronat, D. Dang, R. Geiß, J. Greenyer, P. Van Gorp, O.Kniemeyer, A. Narayanan, E. Rencis, and E. Weinell. Transformation of UML Models to CSP: A Case Study for Graph Transformations Tools. In *3rd International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007)*, LNCS, vol. 5088, pages 540-565. Springer, 2008.

[93] I. Weisemöller, F. Klar, and A. Schürr. Development of Tool Extensions with MOFLON. In *International Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*, LNCS, vol. 6100, pages 337-343. Springer, 2011.

[94] A. Zündorf, L. Geiger, R. Gemmerich, R. Jubeh, J. Leohold, D. Müller, C. Reckord, C. Schneider, and S. Semmelrodt. Using Graph Grammars for Modeling Wiring Harnesses - An Experience Report. In *Graph Transformations and Model-Driven Engineering*, LNCS, vol. 5765, pages 512-532. Springer, 2010.