

Fundamentals of Algebraic Graph Transformation

Bearbeitet von
Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, Gabriele Taentzer

1. Auflage 2006. Buch. xiii, 390 S. Hardcover
ISBN 978 3 540 31187 4
Format (B x L): 15,5 x 23,5 cm

Weitere Fachgebiete > EDV, Informatik > Informatik > Logik, Formale Sprachen,
Automaten

Zu Inhaltsverzeichnis

schnell und portofrei erhältlich bei


DIE FACHBUCHHANDLUNG

Die Online-Fachbuchhandlung beack-shop.de ist spezialisiert auf Fachbücher, insbesondere Recht, Steuern und Wirtschaft. Im Sortiment finden Sie alle Medien (Bücher, Zeitschriften, CDs, eBooks, etc.) aller Verlage. Ergänzt wird das Programm durch Services wie Neuerscheinungsdienst oder Zusammenstellungen von Büchern zu Sonderpreisen. Der Shop führt mehr als 8 Millionen Produkte.

General Introduction

The main idea of graph grammars and graph transformation is the rule-based modification of graphs, where each application of a graph rule leads to a graph transformation step. Graph grammars can be used to generate graph languages similar to Chomsky grammars in formal language theory. Moreover, graphs can be used to model the states of all kinds of systems, which allows one to use graph transformation to model state changes in these systems. This enables the user to apply graph grammars and graph transformation systems to a wide range of fields in computer science and other areas of science and engineering. A detailed presentation of various graph grammar approaches and application areas of graph transformation is given in the three volumes of the *Handbook of Graph Grammars and Computing by Graph Transformation* [Roz97, EEKR99, EKMR99].

1.1 General Overview of Graph Grammars and Graph Transformation

The research area of graph grammars or graph transformation is a discipline of computer science which dates back to the 1970s. Methods, techniques, and results from the area of graph transformation have already been studied and applied in many fields of computer science, such as formal language theory, pattern recognition and generation, compiler construction, software engineering, the modeling of concurrent and distributed systems, database design and theory, logical and functional programming, artificial intelligence, and visual modeling.

This wide applicability is due to the fact that graphs are a very natural way of explaining complex situations on an intuitive level. Hence, they are used in computer science almost everywhere, for example for data and control flow diagrams, for entity relationship and UML diagrams, for Petri nets, for visualization of software and hardware architectures, for evolution diagrams

of nondeterministic processes, for SADT diagrams, and for many more purposes. Like the token game for Petri nets, graph transformation allows one to model the dynamics in all these descriptions, since it can describe the evolution of graphical structures. Therefore, graph transformations have become attractive as a modeling and programming paradigm for complex-structured software and graphical interfaces. In particular, graph rewriting is promising as a comprehensive framework in which the transformation of all these very different structures can be modeled and studied in a uniform way.

Before we go into more detail, we discuss the following basic question.

1.1.1 What Is Graph Transformation?

Graph transformation has at least three different roots:

- from Chomsky grammars on strings to graph grammars;
- from term rewriting to graph rewriting;
- from textual description to visual modeling.

We use the notion of graph transformation to comprise the concepts of graph grammars and graph rewriting. In any case, the main idea of graph transformation is the rule-based modification of graphs, as shown in Fig. 1.1.

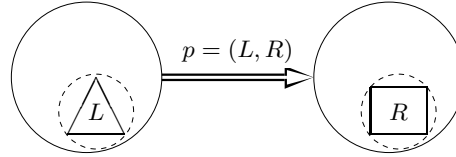


Fig. 1.1. Rule-based modification of graphs

The core of a rule or production, $p = (L, R)$ is a pair of graphs (L, R) , called the left-hand side L and the right-hand side R . Applying the rule $p = (L, R)$ means finding a match of L in the source graph and replacing L by R , leading to the target graph of the graph transformation. The main technical problems are how to delete L and how to connect R with the context in the target graph. In fact, there are several different solutions to how to handle these problems, leading to several different graph transformation approaches, which are summarized below.

1.1.2 Aims and Paradigms of Graph Transformation

Computing was originally done on the level of the von Neumann Machine which is based on machine instructions and registers. This kind of low-level computing was considerably improved by assembler and high-level imperative

languages. From the conceptual point of view – but not necessarily from the point of view of efficiency – these languages were further improved by functional and logical programming languages. This newer kind of computing is based mainly on term rewriting, which, by analogy with graphs and graph transformations, can be considered as a concept in the field of tree transformations. Trees, however, unlike graphs, do not allow sharing of common substructures, which was one of the main reasons for the efficiency problems concerning functional and logical programs. This motivates us to consider graphs rather than trees as the fundamental structures of computing.

The main idea is to advocate graph transformations for the whole range of computing. Our concept of computing by graph transformations is not focused only on programming but includes also specification and implementation by graph transformation, as well as graph algorithms and computational models, and computer architectures for graph transformations.

This concept of computing by graph transformations was developed as a basic paradigm in the ESPRIT Basic Research Actions COMPUGRAPH and APPLIGRAPH and in the TMR Network GETGRATS during the years 1990–2002. It can be summarized in the following way.

Computing by graph transformation is a fundamental concept for the following items:

- *Visual modeling and specification.* Graphs are a well-known, well-understood, and frequently used means to represent system states. Class and object diagrams, network graphs, entity-relationship diagrams, and Petri nets are common graphical representations of system states or classes of system states; there are also many other graphical representations. Rules have proved to be extremely useful for describing computations by local transformations of states. In object-oriented modeling, graphs occur at two levels: the type level (defined on the basis of class diagrams) and the instance level (given by all valid object diagrams). Modeling by graph transformation is visual, on the one hand, since it is very natural to use a visual representation of graphs; on the other hand, it is precise, owing to its formal foundation. Thus, graph transformation can also be used in formal specification techniques for state-based systems.

The aspect of supporting visual modeling by graph transformation is one of the main intentions of the ESPRIT TMR Network SEGRAVIS (2002–2006). In fact, there are a wide range of applications in the support of visual modeling techniques, especially in the context of UML, by graph transformation techniques.

- *Model transformation.* In recent years, model-based software development processes (such as that proposed by the MDA [RFW⁺04]) have evolved. In this area, we are witnessing a paradigm shift, where models are no longer mere (passive) documentation, but are used for code generation, analysis, and simulation as well. An important question is how to specify such model transformations. Starting from visual models as discussed above, graph

transformation is certainly a natural choice. On the basis of the underlying structure of such visual models, the abstract syntax graphs, the model transformation is defined. Owing to the formal foundation, the correctness of model transformations can be checked. More precisely, correctness can be formulated on a solid mathematical basis, and there is a good chance of verifying correctness using the theory of graph transformation. The first steps in this direction have been taken already (see Chapter 14).

- *Concurrency and distribution.* When graph transformation is used to describe a concurrent system, graphs are usually taken to describe static system structures. System behavior expressed by state changes is modeled by rule-based graph manipulations, i.e. graph transformation. The rules describe preconditions and postconditions of single transformation steps. In a pure graph transformation system, the order of the steps is determined by the causal dependency of actions only, i.e. independent rule applications can be executed in an arbitrary order. The concept of rules in graph transformation provides a clear concept for defining system behavior. In particular, for modeling the intrinsic concurrency of actions, graph rules provide a suitable means, because they explicate all structural interdependencies.

If we stick to sequential execution, parallel transformations have to be modeled by interleaving their atomic actions arbitrarily. This interleaving leads to the same result if the atomic actions are independent of each other. Simultaneous execution of actions can be modeled if a parallel rule is composed from the actions.

Parallel and distributed graph transformation both offer structured rule applications, in both temporal and spatial dimensions. Distributed graphs contain an additional structure on the graphs. Graphs are allowed to be split into local graphs and, after local transformations, local graphs are joined again to one global graph. Parallel graph transformation can be considered as a special case of distributed graph transformation, where the host graph is nondistributed.

- *Software development.* In software development, a large variety of different structures occur on different levels, which can be handled as graphs. We distinguish architectural and technical structures from administrative configurations and integration documents. All this structural information evolves, i.e. it changes during the software development process. This includes editing of documents, execution of operations, modification (optimization) of programs, analysis, configuration and revision control, etc. Graph transformation techniques have been used to describe this structural evolution in a rule-based way.

For software development purposes, graphs have several advantages over trees. Graphs are a powerful description technique for any kind of structure. This means that all structural information can be expressed by graphs and does not have to be stored outside the structural part, as is done

in the case of trees. Attributes are important for storing data information. In this way, structural information is separated from nonstructural.

Graph transformations are used to describe certain development processes. Thus, we can argue that we program on graphs. But we do so in a quite abstract form, since the class of structures is the class of graphs and is not specialized to a specific class. Furthermore, the elementary operations on graphs are rule applications. Mostly, the execution order of rule applications relies on structural dependencies only, i.e. it is just given implicitly. Alternatively, explicit control mechanisms for rule applications can be used.

A state-of-the-art report on applications, languages, and tools for graph transformation on the one hand and for concurrency, parallelism, and distribution on the other hand is given in Volumes 2 and 3 of the *Handbook of Graph Grammars and Computing by Graph Transformation* [EEKR99, EKMR99].

1.1.3 Overview of Various Approaches

From an operational point of view, a graph transformation from G to H , written $G \Rightarrow H$, usually contains the following main steps, as shown in Fig. 1.2:

1. *Choose* a production $p : L \Rightarrow R$ with a left-hand side L and a right-hand side R , and with an occurrence of L in G .
2. *Check* the application conditions of the production.
3. *Remove* from G that part of L which is not part of R . If edges dangle after deletion of L , either the production is not applied or the dangling edges are also deleted. The graph obtained is called D .
4. *Glue* the right-hand side R to the graph D at the part of L which still has an image in D . The part of R not coming from L is added disjointly to D . The resulting graph is E .
5. If the production p contains an additional embedding relation, then *embed* the right-hand side R further into the graph E according to this embedding relation. The end result is the graph H .

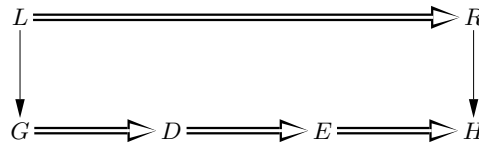


Fig. 1.2. Graph transformation from an operational point of view

Graph transformation systems can show two kinds of nondeterminism: first, several productions might be applicable and one of them is chosen ar-

bitrarily; and second, given a certain production, several matches might be possible and one of them has to be chosen. There are techniques available to restrict both kinds of choice. Some kind of control flow on rules can be defined for applying them in a certain order or by using explicit control constructs, priorities, layers, etc. Moreover, the choice of matches can be restricted by specifying partial matches using input parameters.

The main graph grammar and graph transformation approaches developed in the literature so far are presented in Volume 1 of the *Handbook of Graph Grammars and Computing by Graph Transformation* [Roz97]:

1. The *node label replacement approach*, developed mainly by Rozenberg, Engelfriet, and Janssens, allows a single node, as the left-hand side L , to be replaced by an arbitrary graph R . The connection of R with the context is determined by an embedding relation depending on node labels. For each removed dangling edge incident with the image of a node n in L , and each node n' in R , a new edge (with the same label) incident with n' is established provided that (n, n') belongs to the embedding relation.
2. The *hyperedge replacement approach*, developed mainly by Habel, Krewski, and Drewes, has as the left-hand side L a labeled hyperedge, which is replaced by an arbitrary hypergraph R with designated attachment nodes corresponding to the nodes of L . The gluing of R to the context at the corresponding attachment nodes leads to the target graph without using an additional embedding relation.
3. The *algebraic approach* is based on pushout constructions, where pushouts are used to model the gluing of graphs. In fact, there are two main variants of the algebraic approach, the double- and the single-pushout approach. The double-pushout approach, developed mainly by Ehrig, Schneider, and the Berlin and Pisa groups, is introduced in Section 1.2 and presented later in Part I of this book in more detail. In both cases, there is no additional embedding relation.
4. The *logical approach*, developed mainly by Courcelle and Bouderon, allows graph transformation and graph properties to be expressed in monadic second-order logic.
5. The *theory of 2-structures* was initiated by Rozenberg and Ehrenfeucht, as a framework for the decomposition and transformation of graphs.
6. The *programmed graph replacement approach* of Schürr combines the gluing and embedding aspects of graph transformation. Furthermore, it uses programs in order to control the nondeterministic choice of rule applications.

1.2 The Main Ideas of the Algebraic Graph Transformation Approach

As mentioned above, the algebraic graph transformation approach is based on pushout constructions, where pushouts are used to model the gluing of

graphs. In the algebraic approach, initiated by Ehrig, Pfender, and Schneider in [EPS73], two gluing constructions are used to model a graph transformation step, as shown in Fig. 1.4. For this reason, this approach is also known as the double-pushout (DPO) approach, in contrast to the single-pushout (SPO) approach. Both of these approaches are briefly discussed below.

1.2.1 The DPO Approach

In the DPO approach, roughly speaking, a production is given by $p = (L, K, R)$, where L and R are the left- and right-hand side graphs and K is the common interface of L and R , i.e. their intersection. The left-hand side L represents the preconditions of the rule, while the right-hand side R describes the postconditions. K describes a graph part which has to exist to apply the rule, but which is not changed. $L \setminus K$ describes the part which is to be deleted, and $R \setminus K$ describes the part to be created.

A *direct graph transformation* with a production p is defined by first finding a match m of the left-hand side L in the current host graph G such that m is structure-preserving.

When a direct graph transformation with a production p and a match m is performed, all the vertices and edges which are matched by $L \setminus K$ are removed from G . The removed part is not a graph, in general, but the remaining structure $D := (G \setminus m(L)) \cup m(K)$ still has to be a legal graph, i.e., no edges should be left dangling. This means that the match m has to satisfy a suitable gluing condition, which makes sure that the gluing of $L \setminus K$ and D is equal to G (see (1) in Fig. 1.3). In the second step of a direct graph transformation, the graph D is glued together with $R \setminus K$ to obtain the derived graph H (see (2) in Fig. 1.3). Since L and R can overlap in K , the submatch occurs in the original graph G and is not deleted in the first step, i.e. it also occurs in the intermediate graph D . For gluing newly created vertices and edges into D , the graph K is used. This defines the gluing items at which R is inserted into D . A *graph transformation*, or, more precisely, a graph transformation sequence, consists of zero or more direct graph transformations.

More formally, a direct graph transformation with p and m is defined as follows. Given a production $p = (L \leftarrow K \rightarrow R)$ and a context graph D , which includes also the interface K , the source graph G of a graph transformation $G \Rightarrow H$ via p is given by the gluing of L and D via K , written $G = L +_K D$, and the target graph H is given by the gluing of R and D via K , written $H = R +_K D$. More precisely, we shall use graph morphisms $K \rightarrow L$, $K \rightarrow R$, and $K \rightarrow D$ to express how K is included in L , R , and D , respectively. This allows us to define the gluing constructions $G = L +_K D$ and $H = R +_K D$ as the pushout constructions (1) and (2) in Fig. 1.3, leading to a double pushout. The resulting graph morphism $R \rightarrow H$ is called the comatch of the graph transformation $G \Rightarrow H$.

In order to apply a production p with a match m of L in G , given by a graph morphism $m : L \rightarrow G$ as shown in Fig. 1.3, we first have to construct a

context graph D such that the gluing $L +_K D$ of L and D via K is equal to G . In the second step, we construct the gluing $R +_K D$ of R and D via K , leading to the graph H and hence to a DPO graph transformation $G \Rightarrow H$ via p and m . For the construction of the first step, however, a *gluing condition* has to be satisfied, which allows us to construct D with $L +_K D = G$. In the case of an injective match m , the gluing condition states that all dangling points of L , i.e. the nodes x in L such that $m(x)$ is the source or target of an edge e in $G \setminus L$, must be gluing points x in K .

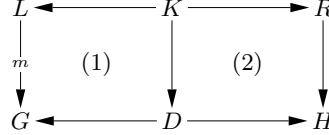


Fig. 1.3. DPO graph transformation

A simple example of a DPO graph transformation step is given in Fig. 1.4, corresponding to the general scheme in Fig. 1.3. Note that in the diagram (PO1), G is the gluing of the graphs L and D along K , where the numbering of the nodes indicates how the nodes are mapped by graph morphisms. The mapping of the edges can be uniquely deduced from the node mapping. Note that the gluing condition is satisfied in Fig. 1.4, because the dangling points (1) and (2) of L are also gluing points. Moreover, H is the gluing of R and D along K in (PO2), leading to a graph transformation $G \Rightarrow H$ via p . In fact, the diagrams (PO1) and (PO2) are pushouts in the category **Graphs** of graphs and graph morphisms (see Chapter 2).

For technical reasons, the morphisms $K \rightarrow L$ and $K \rightarrow R$ in the productions are usually restricted to injective graph morphisms. However, we allow noninjective matches $m : L \rightarrow G$ and comatches $n : R \rightarrow H$. This is especially useful when we consider a parallel production $p_1 + p_2 : L_1 + L_2 \leftarrow K_1 + K_2 \rightarrow R_1 + R_2$, where $+$ denotes the disjoint union. Even for injective matches $m_1 : L_1 \rightarrow G$ of p_1 and $m_2 : L_2 \rightarrow G$ of p_2 , the resulting match $m : L_1 + L_2 \rightarrow G$ is noninjective if the matches $m_1(L_1)$ and $m_2(L_2)$ are overlapping in G .

1.2.2 The Algebraic Roots

In Chapter 2, we shall see that a graph $G = (V, E, s, t)$ is a special case of an algebra with two base sets V (vertices) and E (edges), and operations $s : E \rightarrow V$ (source) and $t : E \rightarrow V$ (target). Graph morphisms $f : G_1 \rightarrow G_2$ are special cases of algebra homomorphisms $f = (f_V : V_1 \rightarrow V_2, f_E : E_1 \rightarrow E_2)$. This means that f_V and f_E are required to be compatible with the operations, i.e. $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$. In Fig. 1.4, all arrows

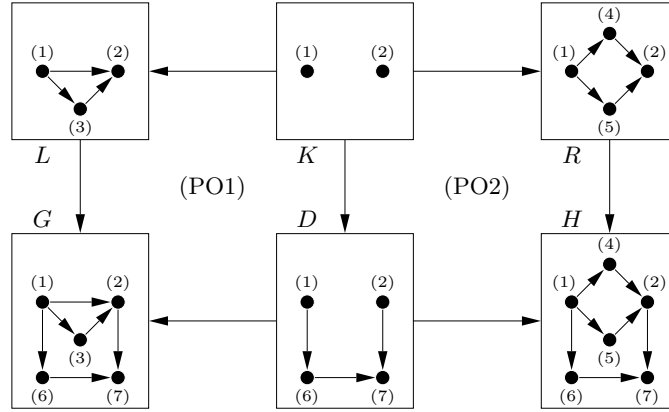


Fig. 1.4. Example of DPO graph transformation

between the boxes are graph morphisms. Moreover, the gluing construction of graphs can be considered as an algebraic quotient algebra construction. This algebraic view of graphs and graph transformations is one of the main ideas of the algebraic graph transformation approach introduced in [EPS73, Ehr79].

1.2.3 From the DPO to the SPO Approach

As pointed out already, the gluing constructions in the algebraic approach are pushouts in the category **Graphs** based on (total) graph morphisms. On the other hand, the production $p = (L \leftarrow K \rightarrow R)$ shown in Fig. 1.4 can also be considered as a partial graph morphism $p : L \rightarrow R$ with domain $\text{dom}(p) = K$. Moreover, the span $(G \leftarrow D \rightarrow H)$ can be considered as a partial graph morphism $s : G \rightarrow H$ with $\text{dom}(s) = D$. This leads to the diagram in Fig. 1.5, where the horizontal morphisms are partial and the vertical ones are total graph morphisms. In fact, Fig. 1.5 is a pushout in the category **PGraphs** of graphs and partial graph morphisms and shows that the graph transformation can be expressed by a single pushout in the category **PGraphs**. This approach was initiated by Raoult [Rao84] and fully worked out by Löwe [Löw90], leading to the single-pushout approach.

From the operational point of view, the SPO approach differs in one main respect from the DPO approach, which concerns the deletion of context graph elements during a graph transformation step. If the match $m : L \rightarrow G$ does not satisfy the gluing condition with respect to a production $p = (L \leftarrow K \rightarrow R)$, then the production is not applicable in the DPO approach. But it is applicable in the SPO approach, which allows dangling edges to occur after the deletion of $L \setminus K$ from G . However, the dangling edges in G are also deleted, leading to a well-defined graph H .

If, in Fig. 1.4, vertex (2) were to be deleted from K , the gluing condition would not be satisfied in the DPO approach. In the SPO approach, this would mean that vertex (2) is not in the domain of p , leading to a dangling edge e in G after deletion of $L \setminus \text{dom}(p)$ in Fig. 1.5. As a result, edge e would be deleted in H .

A detailed presentation and comparison of the two approaches is given in Volume 1 of the *Handbook of Graph Grammars and Computing by Graph Transformation* [Roz97]. In this book, however, we consider only the DPO approach as the algebraic graph transformation approach.

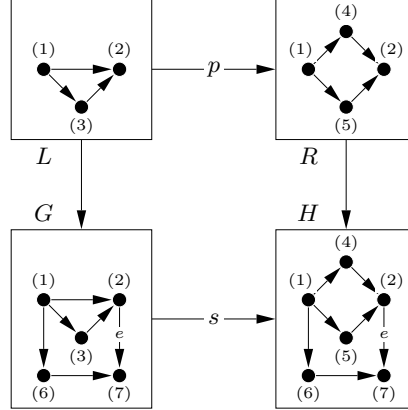


Fig. 1.5. Example of SPO graph transformation

1.2.4 From Graphs to High-Level Structures

The algebraic approach to graph transformation is not restricted to the graphs of the form $G = (V, E, s, t)$ considered above, but has been generalized to a large variety of different types of graphs and other kinds of high-level structures, such as labeled graphs, typed graphs, hypergraphs, attributed graphs, Petri nets, and algebraic specifications, which will be considered in later chapters of this book. This extension from graphs to high-level structures – in contrast to strings and trees, considered as low-level structures – was initiated in [EHKP91a, EHKP91b] leading to the theory of high-level replacement (HLR) systems. In [EHPP04], the concept of high-level replacement systems was joined to that of *adhesive categories* introduced by Lack and Sobociński in [LS04], leading to the concept of adhesive HLR categories and systems, which will be described in Chapters 4 and 5. The theory of adhesive HLR systems, developed in Part II of this book, can be instantiated in particular to typed attributed graph transformation systems, described in Part III, which are especially important for applications to visual languages and software engineering. In Part I of this book, we give an introduction to classical graph

transformation systems, as considered in [Ehr79]. Hence the fundamentals of algebraic graph transformation are given in this book on three different levels: in Part I on the classical level, in Part II on the level of adhesive HLR systems, and in Part III on the level of typed attributed graph transformation systems. A more detailed overview of the four Parts is given below.

1.3 The Chapters of This Book and the Main Results

The chapters of this book are grouped into Parts I–IV and three appendices.

1.3.1 Part I: Introduction to Graph Transformation Systems

Part I of this book is an introduction to graph transformation systems in general and to the algebraic approach in the classical sense of [Ehr79] in particular. In Chapter 2, we introduce graphs, graph morphisms, typed graphs and the gluing construction for sets and graphs. In order to show that the gluing construction is a pushout in the sense of category theory, we also introduce some basic notions of category theory, including the categories **Sets** and **Graphs**, and pullbacks as dual constructions of pushouts.

This is the basis for introducing the basic notions of graph transformation systems in Chapter 3. As mentioned above, a direct graph transformation is defined by two gluing constructions, which are pushouts in the category **Graphs**. The first main results for graph transformations discussed in Chapter 3 are concerned with parallel and sequential independence of graph transformations. The Local Church–Rosser Theorem allows one to apply two graph transformations $G \Rightarrow H_1$ via p_1 and $G \Rightarrow H_2$ via p_2 in an arbitrary order, provided that they are *parallel independent*. In this case they can also be applied in parallel, leading to a parallel graph transformation $G \Rightarrow H$ via the *parallel production* $p_1 + p_2$. This second main result is called the Parallelism Theorem. In addition, in Chapter 3 we give a detailed description of the motivation for and an overview of some other main results including the Concurrency, Embedding, and Extension Theorems, as well as results related to critical pairs, confluence, termination, and the Local Confluence Theorem. Finally, we discuss graph constraints and application conditions for graph transformations. All these results are stated in Chapter 3 without proof, because they are special cases of corresponding results in Parts II and III.

1.3.2 Part II: Adhesive HLR Categories and Systems

In Part II, we introduce adhesive HLR categories and systems, as outlined in Subsection 1.2.4 above. In addition to pushouts, which correspond to the gluing of graphs, they are based on pullbacks, corresponding to the intersection and homomorphic preimages of graphs. The basic axioms of adhesive

HLR categories stated in Chapter 4 require construction and basic compatibility properties for pushouts and pullbacks. From these basic properties, several other results, called HLR properties, can be concluded, which allow us to prove the main results stated in Part I on the general level of high-level replacement systems. We are able to show that there are several interesting instantiations of adhesive HLR systems, including not only graph and typed graph transformation systems, but also hypergraph, Petri net, algebraic specification, and typed attributed graph transformation systems. The HLR properties allow us to prove, in Chapter 5, the Local Church–Rosser and Parallelism Theorems, concerning independent transformations, as well as the Concurrency Theorem, concerning the simultaneous execution of causally dependent transformations.

Some further important results for transformation systems are the Embedding, Extension, and Local Confluence Theorems presented in Chapter 6. The first two allow us to embed transformations into larger contexts, and with the third one we are able to show local confluence of transformation systems on the basis of the confluence of critical pairs.

In Chapter 7, we define constraints and application conditions for adhesive HLR systems, which generalize the graph constraints and application conditions introduced in Chapter 3. We are able to show, as the main results, how to transform constraints into right application conditions and, further, how to transform right to left application conditions. A left or right application condition is a condition which has to be satisfied by the match $L \rightarrow G$ or the comatch $R \rightarrow H$, respectively, of a DPO transformation, as shown for graphs in Fig. 1.3.

1.3.3 Part III: Typed Attributed Graph Transformation Systems

In Part III, we apply the theory of Part II to the case of typed attributed graph transformation systems outlined in Subsection 1.2.4. In Chapter 8, we introduce attributed type graphs ATG , typed attributed graphs, and typed attributed graph morphisms leading to the category $\mathbf{AGraphs}_{ATG}$ of typed attributed graphs and the construction of pushouts and pullbacks in this category. In Chapter 9, we define the basic concepts of typed attributed graph transformations. Moreover, we extend some of the main results from the case of graphs considered in Chapter 3 to typed attributed graphs. In particular, this leads to the important result of local confluence for typed attributed graph transformation systems based on confluence of critical pairs. All the results suggested in Chapter 3 and presented in the general framework of Part II are instantiated for typed attributed graph transformation in Chapters 9 and 10. They are proven in Chapter 11 by showing that $\mathbf{AGraphs}_{ATG}$ is an adhesive HLR category with suitable additional properties, which allows us to apply the general results from Part II. In Chapter 12, we apply the categorical theory of constraints and application conditions to typed attributed graph transformation systems and discuss termination in addition. In Chapter 13, we introduce attributed type graphs with inheritance in order to model type

inheritance in the sense of class inheritance in UML. The main result shows the equivalence of concrete transformations without inheritance and abstract transformations with inheritance. However, the use of inheritance leads to a much more efficient representation and computation of typed attributed graph transformations.

1.3.4 Part IV: Case Study and Tool Support

In Part IV, we show how the theory of Part III can be applied in a case study and what kind of tool support can be offered at present. A case study of model transformation from statecharts to Petri nets using typed attributed graph transformation systems is given in Chapter 14. In Chapter 15, we show how typed attributed graph transformation has been implemented in the AGG tool environment developed at TU Berlin [AGG, ERT99].

1.3.5 Appendices

In Appendix A, we give a short introduction to category theory summarizing the main categorical concepts introduced in Parts I–III together with some technical results. Since, on the one hand, typed attributed graphs as considered in Part III are based on algebraic signatures and algebras, and, on the other hand, algebraic specifications themselves and in connection with Petri nets are interesting instantiations of adhesive HLR systems, we review the corresponding algebraic concepts from [EM85] in Appendix B. Finally, we present in Appendix C all of the proofs that were postponed in Parts I–III.

1.3.6 Hints for Reading This Book

For those readers who are interested mainly in the concepts and results of transformation systems for classical and typed attributed graphs, but not so much in the general theory and in the proofs, it is advisable to read Part I but to skip Part II and continue immediately with Parts III and IV.

1.4 Bibliographic Notes and Further Topics

In this last section of the introduction, we present some bibliographic notes and an overview of further topics concerning concepts, applications, languages, and tools for graph transformation systems.

1.4.1 Concepts of Graph Grammars and Graph Transformation Systems

Graph transformation originally evolved in the late 1960s and early 1970s [PR69, Pra71, EPS73] as a reaction to shortcomings in the expressiveness

of classical approaches to rewriting, such as Chomsky grammars and term rewriting, to deal with nonlinear structures. The main graph grammar and graph transformation approaches that are still popular today are presented in Volume 1 of the *Handbook of Graph Grammars and Computing by Graph Transformation* [Roz97] and have been mentioned in Subsection 1.1.2 already. In contrast to the algebraic approach, which is based on the concept of gluing (see Section 1.2), the NLC approach [JR80] and other ones such as that of Nagl in [Nag79] can be considered as embedding approaches. In this case, the embedding (of the right-hand side into the context graph) is realized by a disjoint union, with as many new edges as needed to connect the right-hand side with the context graph. Nagl's approach has been extended by Schürr to programmed graph replacement systems [Sch97], leading to the PROGRES approach in [SWZ99]. In the FUJABA approach [FUJ], graph transformations are used in order to define transformations from UML to Java and back again. Both approaches allow the replacement of substructures in an unknown context using the concept of *set nodes* or *multiobjects*.

Concerning the algebraic approach described in Parts I and II, most of the concepts in the classical case of graph transformation systems were developed in the 1970s [EPS73, Ehr79]. Application conditions were first considered in the 1980s [EH86], and negative application conditions together with graph constraints in the 1990s [HW95, HHT96], including the important result of transforming graph constraints into application conditions.

The main parts of the theory for the DPO and SPO approaches are presented in Volume 1 of the *Handbook of Graph Grammars and Computing by Graph Transformation* [Roz97]. In addition to our presentation in Part I, the presentation in the *Handbook* includes an abstract concurrent semantics of graph transformation systems in the DPO case. This is based on the shift equivalence of parallel graph transformations, developed by Kreowski in his Ph.D. thesis [Kre78]. Further concepts concerning parallelism, concurrency, and distribution for the algebraic and other approaches are presented in Baldan's Ph.D. thesis [Bal00] and in Volume 3 of the *Handbook of Graph Grammars and Computing by Graph Transformation* [EKMR99]. For the concepts of term graph rewriting, hierarchical graph transformation systems, graph transformation modules and units, and for the first approaches to the analysis, verification, and testing of graph transformation systems, we refer to the proceedings of the first and second International Conferences on Graph Transformation ICGT 2002 [CEKR02] and ICGT 2004 [EEPR04].

The first approach to HLR categories and systems, as presented in Part II, was described in [EHKP91b] and was joined to adhesive categories [LS04] in [EHPP04] and [EEHP04]. Attributed and typed attributed graph transformations, as described in Part III, were considered explicitly in the 1990s, especially in [LKW93, CL95, HKT02]. A fundamental theory for these important kinds of graph transformation systems was first developed in [EPT04] as an instantiation of adhesive HLR systems.

1.4.2 Application Areas of Graph Transformation Systems

At the very beginning, the main application areas of graph transformation systems were rule-based image recognition [PR69] and translation of diagram languages [Pra71]. Later on, graph transformations have been applied to several areas in computer science, biology, chemistry, and engineering, as documented in Volume 2 of the *Handbook of Graph Grammars and Computing by Graph Transformation* [EEKR99]. More recently, graph transformations have been applied most successfully to the following areas in software engineering, some part of which have been mentioned already in Subsection 1.1.2:

- model and program transformation;
- syntax and semantics of visual languages;
- visual modeling of behavior and programming;
- modeling, metamodeling, and model-driven architecture;
- software architectures and evolution;
- refactoring of programs and software systems;
- security policies.

Other important application areas have been term graph rewriting, DNA computing, Petri nets, process algebras and mobile systems, distributed algorithms and scheduling problems, graph theory, logic, and discrete structures. These application areas have been subject of the international conferences ICGT 2002 [CEKR02] and ICGT 2004 [EEPR04], together with the following satellite workshops and tutorials, most of which have been published in Electronic Notes in Theoretical Computer Science (see e.g. [GTV03, SET03]):

- Graph Transformation and Visual Modeling Techniques (GTVMT);
- Software Evolution through Transformations (SETRA);
- Petri Nets and Graph Transformations (PNGT);
- DNA Computing and Graph Transformation (DNAGT);
- Graphs, Logic and Discrete Structures;
- Term Graph Rewriting (TERMGRAPH).

1.4.3 Languages and Tools for Graph Transformation Systems

In Chapter 15, we discuss the implementation of typed attributed graph transformation using the AGG language and tool [AGG, ERT99]. A basic version of AGG was implemented by Löwe et al. in the early 1990s, and later a completely redesigned version was implemented and extended by Taentzer, Runge, and others. Another general-purpose graph transformation language and tool is PROGRES [Sch97, SWZ99]. Quite different support is offered by FUJABA [FUJ], an environment for round-trip engineering between UML and Java based on graph transformation.

Two examples of more application-specific tools are DIAGEN [Min97] and GenGed [BE00, Bar02]. These provide support for the generation of graphical

editors based on the definition of visual languages using graph grammars. In addition, several model transformation tools based on graph transformation have been developed, e.g. ATOM³ [dLV02a], MetaEnv [BP02], and Viatra [VP03]. More detailed presentations of languages and tools are given in Volume 2 of the *Handbook of Graph Grammars and Computing by Graph Transformation* [EEKR99] and in the proceedings [MST02, MST04] of the GraBats workshops on graph-based tools held as satellite events of ICGT 2002 and 2004 [CEKR02, EEPR04].

1.4.4 Future Work

The aim of this book is to present the fundamentals of algebraic graph transformation based on the double-pushout approach. As discussed above, there are several other topics within the DPO approach which have been published already, but are not presented in this book. There are also several interesting topics for future work. First of all, it would be interesting to have a similar fundamental theory for the single-pushout approach. A comparative study of the DPO and SPO approaches is presented in Volume 1 of the *Handbook of Graph Grammars and Computing by Graph Transformation* [Roz97]. Another important topic for future research is the DPO approach with borrowed context [EK04], motivated by process algebra and bigraphical reactive systems in the sense of Milner [Mil01] and Sobociński [Sob04]. Concerning the main topics of this book, it is open to extend the main results for graph transformations to the case of negative and general application conditions and also to study new kinds of constraints for graphs and typed attributed graphs. Concerning typed attributed graph transformation, we have introduced type inheritance, but it remains open to extend the theory in Chapters 9–12 to this case. Finally, the theory has to be extended to meet the needs of several interesting application domains. In particular, in the area of model transformations (Chapter 14), it remains open to develop new techniques in addition to the analysis of termination and confluence, to show the correctness of model transformations.