Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems

vorgelegt von
Diplom-Informatikerin
Gabriele Taentzer
aus Berlin

Vom Fachbereich 13 – Informatik –
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktorin der Ingenieurswissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuß:
Vorsitzender: Prof. Dr. H. Weber
Berichter: Prof. Dr. H. Ehrig
Berichter: Prof. Dr. H.-J. Kreowski

Tag der wissenschaftlichen Aussprache: 29. März 1996

Berlin 1996
D 83

Abstract

The overall aim of this thesis is the development of new concepts for parallel and distributed graph transformation to increase their applicability to communication-based systems. All concepts for parallel and distributed graph transformation presented are formalized and compared with respect to their expressive power.

Based on parallel graph grammars introduced by Ehrig and Kreowski the main concepts of existing approaches to parallel graph replacement are integrated into a new approach and generalized concerning flexible synchronization mechanisms for parallel actions and restriction concepts for un-bounded parallel execution of actions. Distributed graph transformation as developed in this thesis combines structured graph transformation on two abstraction levels, the network and the local level, with the concept of synchronization by interface graphs into a new approach. In this new approach the main distribution concepts of categorical graph grammars presented by Schneider are combined with the algebraic approach to distributed graph transformation introduced by Ehrig et.al. Modeling of distributed systems by this new kind of distributed graph transformation offers a clear and elegant description of dynamic networks, distributed actions as well as communication and synchronization based on graph transformation.

The formal description of parallel and distributed graph transformation is based on the double-pushout approach to graph transformation including all concepts presented above. Especially distributed graph transformation can be very clearly described by category theory. The parallel graph transformation types developed in this thesis are compared with each other and with sequential graph transformation concerning their expressive power. Moreover, distributed graph transformation is shown to be closely related to parallel graph transformation if parallel graph transformation is extended by adding distribution structure compatible with the application structure of parallel productions.

Zusammenfassung

Das Gesamtziel dieser Arbeit ist die Entwicklung neuer Konzepte für parallele und verteilte Graphtransformation, um ihre Anwendbarkeit auf kommunikationsbasierte Systems zu verbessern. Alle vorgestellten Konzepte paralleler und verteilter Graphtransformation werden formal beschrieben und hinsichtlich ihre Ausdruckskraft verglichen.

Basierend auf parallelen Graphgrammatiken von Ehrig und Kreowski werden die wichtigsten Konzepte der existierenden parallelen Graphersetzungsansätze in einen Ansatz integriert und um flexible Synchronisationsmöglichkeiten für parallele Aktionen sowie Kontrollmöglichkeiten für unbeschränkte parallele Ausführung von Aktionen erweitert. Die in dieser Arbeit entwickelte verteilte Graphtransformation kombiniert strukturierte Graphtransformation auf zwei Abstraktionsebenen, der Netzwerk- und der lokalen Ebene, mit dem Konzept der Synchronisation über Schnittstellengraphen in einen neuen Ansatz. In diesem neuen Ansatz werden wesentliche Verteilungskonzepte der kategoriellen Graphgrammatiken von Schneider mit denen des algebraischen Ansatzes für verteilte Graphtransformation von Ehrig et.al. kombiniert. Modellierungen verteilter Systeme mit diesem neuen Ansatz für verteilte Graphtransformation ermöglichen eine klare und elegante Beschreibung dynamischer Netzwerke, verteilter Aktionen sowie Kommunikation und Synchronisation von verschiedenen Systemkomponenten basierend auf Graphtransformation.

Die formale Beschreibung der parallelen und verteilten Graphtransformation basiert auf dem Doppel-Pushout-Ansatz für Graphtransformation und behandelt alle oben genannten Konzepte. Insbesondere die verteilte Graphtransformation kann mit Hilfe von Kategorientheorie sehr klar beschrieben werden. Die in dieser Arbeit entwickelten parallelen Graphtransformationsarten werden miteinander und mit sequentieller Graphtransformation bezüglich ihrer Ausdrucksmöglichkeiten verglichen. Darüberhinaus wird gezeigt, daß verteilte Graphtransformation mit paralleler Graphtransformation vergleichbar ist, falls die parallele Graphtransformation um zusätzliche Verteilungsstruktur, die mit der parallelen Regelanwendung kompatibel ist, erweitert wird.

Contents

Introduction

Motivation When reading a text about some parallel or distributed systems and their description, you usually find some graphical representations, in the belief that "a graph says more than 1000 words". At least the topological structure of a distributed system or the architecture of parallel processor systems are presented graphically. But normally, the graphical representations are only used for system description. The exact specification or implementation of the system is nearly entirely based on textual representations which are not the best choice for describing structures, at least those which are neither string nor tree like. *Graphical representations* can make complex structures easier to survey.

Parallelism is hard to understand and parallel system specification is usually very complex. One reason may be the fact that parallel process control has been much less investigated than sequential control constructs. Some sequential control constructs have parallel counterparts, e.g. the for-loop, where parallelism is expressed *explicitly*. On the other side it is widely accepted that rule-based systems are a good means to support parallel system specification. They go the opposite way by generally not supporting control constructs to order actions (described by rules). Nevertheless, *implicit* causal dependencies may occur ordering different actions according to their problem-inherent parallelism. A good parallel specification technique has to support the designer as well as possible to find out and to describe the problem-inherent parallelism and to avoid unnecessary sequentializations.

Graph transformation provides rule-based manipulation of graphs. Thus, graph transformation combines the properties of both, graphical representations and rule-based character, into one computational paradigm. Therefore, it seems to be promising to use graph transformation to specify communication-based system parts. Distributed system structures can be described by graphs and parallel execution can be modeled by graph transformation.

Moreover, graph transformation is a formally defined calculus based on set-theory, algebra or category theory. For communication-based systems which are often infinite/unbounded or have non-deterministic behavior we can argue that it is very important to understand the systems since sufficient testing is impossible. Using graph transformation offers the possibility to describe parallelism and distribution aspects of a system precisely. This description serves as a good basis for understanding and analyzing the system's behavior.

Aims of this thesis This thesis tries to bring the theoretical concepts of graph transformation and practical aspects of communication-based systems closer together. The long-term goal of this line of research is to provide a comprehensive specification technique for communication-based systems well-founded on graph transformation.

The starting point for the investigations are parallel graph grammars [EK76b] and distributed graph transformation [EBHL88] in the double-pushout approach to graph transformation [Ehr79]. One aim is to increase the expressive power of parallel and distributed graph transformations by inventing significant concepts in order to improve their applicability to communication-based systems. In this thesis we concentrate on using graph transformation for the design specification of the communication-based system parts.

The other aim of this thesis is to provide a formal description of parallel and distributed graph transformation which integrates the new concepts and is based on the double-pushout approach to graph transformation. Furthermore, parallel and distributed graph transformations shall be investigated according to their relations to each other and to sequential graph transformation.

Main results Since the aims of this thesis are two-fold concerning the formalization of parallelism and distribution concepts for graph transformation and their improved applicability to communication-based systems, the main results are divided into *main conceptual results* and *main technical results*.
Main conceptual results: The main concepts of existing approaches to parallel graph replacement [EK76b, Nag79, Kre92, DDK93] are combined and generalized concerning flexible and powerful synchronization possibilities for parallel actions and restriction possibilities for un-bounded parallel execution of actions.

Existing graph transformation models for distributed systems mostly concentrate on the topological aspects of a distributed system [DM87, JR91, KLG91]. In contrast, the algebraic approach to distributed graph transformation [EBHL88, EL93] describes mostly local actions by graph transformation related to others by interface graphs. The operations "split" and "join" switching between global and local states of a system are introduced to change network structures. The categorical approach [Sch93] uses graph transformation on the network level and the local level. Distributed graph transformation as presented in this thesis combines structured graph transformation on two levels introduced in the categorical approach with the synchronization possibilities via interface graphs of the algebraic approach. Modeling of distributed systems by distributed graph transformation supports a clear and elegant description of dynamic networks, local actions, network administration, communication and synchronization of system components.

Application of parallel and distributed graph transformation to communication-based systems leads to the conclusion that graph transformation has a good potential to describe communication aspects although there are still deficiencies. Graph transformation offers description concepts for problem-oriented parallelism by parallel application of graph productions, communication and synchronization by graph sharing and flexible system structures by arbitrary underlying graph structures modified by graph transformation.
Main technical results: This thesis provides a formal description of parallel and distributed graph transformation based on the double-pushout approach to graph

transformation including all extensions presented above. The theory about parallel graph transformation is extended and related to sequential graph transformation. Since the main concepts are characterized by categorical notions based on the category **GRAPH**, parallel graph transformation can be generalized to *high-level replacement systems* [EHKP91a]. High-level replacement systems are an axiomatic framework based on double-pushouts in order to unify replacement systems like grammars for different kinds of graphs and relational structures or other types of structures like algebraic specifications. For this generalization see [ET92, Tae92]. Different parallel graph transformation concepts are formally compared with each other and with sequential graph transformation concerning their expressive power.

Distributed graph transformation is formalized by a new approach based on the double-pushout approach to graph transformation. A distributed graph can be described as a diagram of graphs and graph morphisms. A transformation step is characterized by a double-pushout in the category **DISTR(GRAPH)** which consists of distributed graphs as objects and distributed graph morphisms. Thus, it is shown that distributed graph transformation fits into high-level replacement framework in [EHKP91a]. It can be expected that results for high-level replacement systems concerning parallelism, concurrency and embedding can be carried over to distributed graph transformation.

Distributed graph transformation is closely related to parallel graph transformation if parallel graph transformation is extended by adding distribution structure compatible with the application structure of parallel productions.

Organization of the chapters Since graphical representations can make complex structures easier to survey, the overall structure of this thesis (not being a hypertext) is shown by the graph in figure 0.1. Each chapter is shown as a box with its title inside. The different kinds of relations are indicated by arrows equipped with the type of the corresponding relation.

This thesis is concerned with two areas: graph transformation and communication-based systems indicated by the outer large dashed boxes. In *chapter 1* communication-based systems are introduced using typical examples, main aspects and requirements. It serves as a basis for further chapters where graph transformation concepts are classified and compared on the basis of the contents of chapter 1.

In *chapter 7* parallel and distributed graph transformation as introduced in chapters 3 to 6 are applied to communication-based systems of different application fields. Although the applications of parallel and distributed graph transformation differ heavily, the underlying ideas of application show similarities. They are summarized and future work towards a comprehensive graph-transformation-based specification technique for communication-based systems is discussed. Although there is still little tool support for graph transformation, the different kinds of parallel and distributed graph transformation can be directly implemented due to their operational and very constructive character.

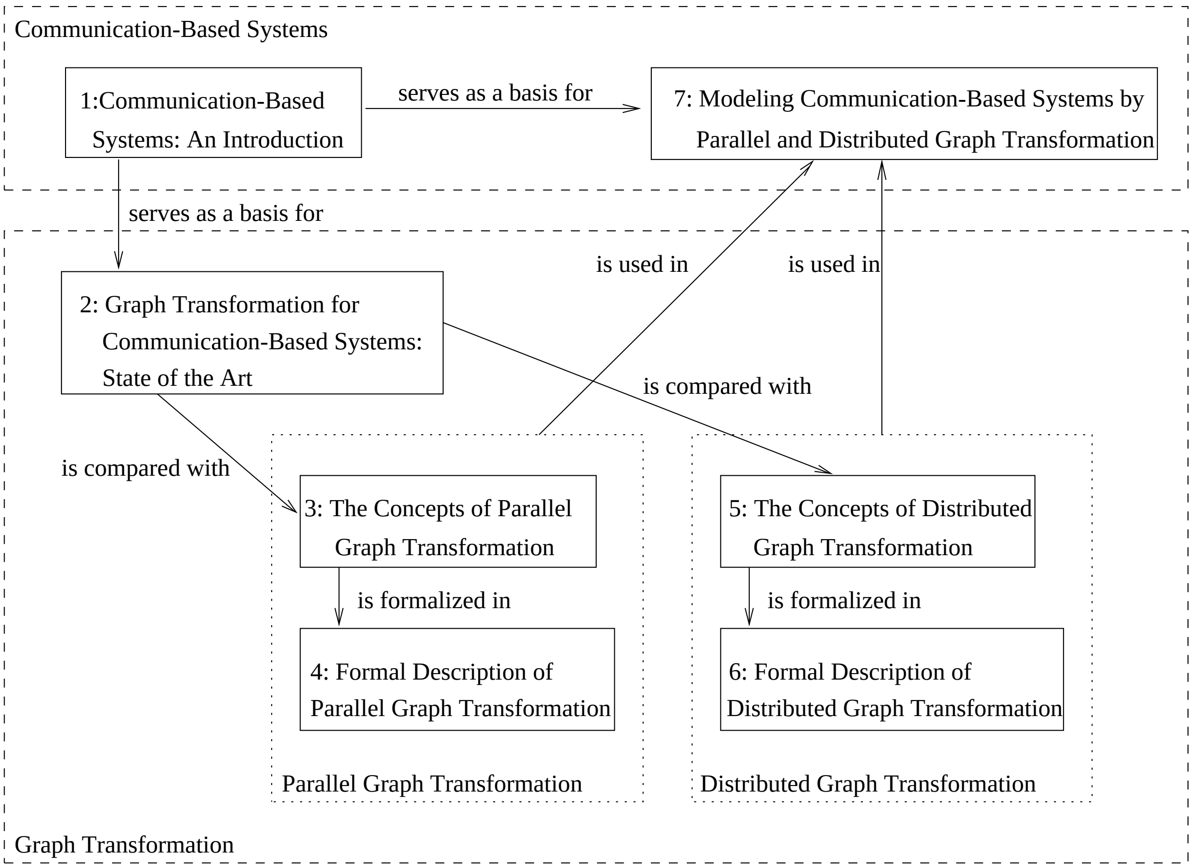

Figure 0.1: The overall structure of this thesis

Chapters 2 to 6 are concerned with graph transformation.

Chapter 2 gives an informal introduction to graph transformation independent of a special graph transformation approach. The state of the art of parallel graph replacement and modeling of distributed systems by graph transformation are reviewed in particular. The different approaches presented there are compared according to the requirements for communication-based systems in chapter 1. Since this thesis should also be understandable for people outside the graph transformation community, this chapter should provide those readers with the operational understanding of the main graph transformation concepts.

Chapters 3 and 4 present the concepts of parallel graph transformation (indicated by the left dotted box).

Chapter 3 introduces the concepts of parallel graph transformation in an informal way. A parallel replacement on graphs is introduced combining and extending different concepts from existing parallel replacement approaches. On comparison, this approach is at least as powerful as existing approaches.

Chapter 4 contains the formalization of the parallelism concepts given in chapter 3 based on the double-pushout approach to graph transformation. The double-pushout approach provides a kind of cautious graph rewriting where conflicting transformations or those destroying the graph structure are not allowed. The preservation of structural consistency is especially required by parallel and distributed systems which often contain safety-critical parts. On the other hand, double-pushout graph transformation provides a flexible and formal basis for description and comparison (using category theory). Different parallel graph transformation concepts are formally defined and compared with each other and with sequential graph transformation concerning their expressive power. This chapter presents the formal description of all the concepts introduced in chapter 3 and shows how the desired relation results between graph transformation types are achieved.

Chapters 5 and 6 are concerned with distributed graph transformation (indicated by the right dotted box).

Chapter 5 introduces the distribution concepts for graph transformation informally. Graph transformation is proposed to describe the dynamic topology of a system as well as local and communicating actions. The conceptual ideas presented in this chapter lead to more elaborated and structured descriptions of essential communication concepts compared with those of existing graph transformation models for distributed systems.

In *chapter 6* distributed graph transformation is formally defined based again on the double-pushout approach to graph transformation. But here, the transformed structures are distributed graphs instead of graphs. Cautious rewriting preserves these structured graphs with all their object interrelations. Distributed graph transformation is compared to parallel graph transformation with additional distribution structure compatible with parallel matches.

After completing this thesis, the reader will know about parallel and distributed graph transformation and their application to communication-based systems. A reader may leave out chapters 4 and 6 where parallel and distributed graph transformation are formally defined and compared. Nevertheless, he or she gets a deep insight into the graph transformation paradigm and its expressive power. A reader familiar with graph transformation, especially parallel and distributed graph transformation, may just skim chapter 2 where the state of the art is summarized, although the comparison parts of this chapter may be of interest.

Acknowledgements

I like to thank my supervisor Hartmut Ehrig for his constructive support concerning the writing of this thesis. He gave me a lot of valuable hints to improve draft versions of this work. Furthermore, I thank my co-supervisor Hans-Jörg Kreowski who was immediately willing to report on this thesis although the time was very short.

Special thanks to Annika Wagner. She read this thesis, especially the theoretical parts in chapters 4 and 6, very carefully and had many valuable comments. Last but not least, I thank Ralph Betschko, Rosi Bardohl, Martin Große-Rhode, Antje Lennartz, Leila Ribeiro, Andy Schürr and Holger Schween for proof reading this thesis and their numerous suggestions.

Chapter 1

Communication-Based Systems: An Introduction

This chapter serves as a short survey of the main features of communication-based systems and their reflection in some of the various specification calculi and programming languages. This survey is mainly oriented at the work of Bal, Steiner and Tanenbaum in [BST89] which presents a comprehensive and structured overview over communication-based systems seldom found in the literature.

Chapter 1 functions as a basis for further chapters, especially chapter 7 where we apply parallel and distributed graph transformation to communication-based systems.

Readers of this chapter can get an overview over communication-based systems. They are referred to further literature (referenced below) to get a deeper insight.

1.1 Communication-Based Systems

Communication-based systems distinguish themselves from sequential systems by allowing multiple components to act in parallel which can communicate with each others. They include *parallel systems* where parallel actions are allowed to modify global system states in parallel whereas *distributed systems* consist of autonomous components with their own states and data. Communication in distributed systems is realized over a network whereas communication in parallel systems can be based on the global state.

Communication-based systems have become more and more popular in computer science and various application areas. This section deals with the reasons for designing communication-based systems (in subsection 1.1.1) and typical examples (see subsection 1.1.2). The reasons leading to the design of communication-based systems are *speed up* of the execution of actions, increasing *reliability* and *availability* (essential for fault-tolerant systems), the *inherent distribution* of an application and *reuse possibilities* of system parts.

1.1.1 Classes of Communication-Based Systems

Massively parallel systems The speedup of computation through parallelism is a common, maybe the main, reason for the design of a parallel system. By allowing different parts of a system to be performed simultaneously some systems will finish faster. The underlying *parallel architectures* for massively parallel systems are mostly of some regular structure where the parallelism of the systems has to fit in. For an abstract problem-oriented modeling technique the architecture should be very flexible. A parallel programming language has to be adapted to the executing machines which usually restrict this flexibility.

Fault-tolerant systems Safety-critical applications gain more reliability by replicating critical parts and performing them in parallel. Furthermore, important parts of a system which must not crash should be controlled intensively by others. This requirement leads to parallel parts controlling critical executions.

Application-inherent distributed systems Inherently distributed applications are geographically distributed, i.e. their parts run at different sites. These applications are usually loosely coupled systems which are connected by physical communication lines such as telephone lines.

Reusing system parts If a system can be described by a set of modules each offering some functionality the reuse possibilities of those modules can be increased to some extent if the system is designed to be distributed. Modules requiring special resources offer their functionality via some interface and do not have to be reconstructed in a slightly different environment. Distributed systems have a *component architecture* if it is possible to erase and integrate components into an existing system. Distributed systems are called *transparent* , if details and mechanism caused by the distribution of the system are hidden. Systems are called *open* if their interfaces are published such that other people (companies) can reuse them. Systems are called *free* if not only the interfaces but the entire system is published.

1.1.2 Parallel and Distributed Applications

In the following, we sum up some typical examples of parallel and distribute applications. Their development is due to one or more of the reasons described above.

A very popular application field for parallel systems is that of *pattern generation*. Especially patterns with some regular shape such as *fractals* [Man82] allow a massively parallel generation. Parallel pattern generation aims for speeding up the process by executing the same instruction on multiple data (SIMD-parallelism).

Another type of massively parallel systems are those based on *genetic algorithms* simulating the life of a population by the selection, mutation and cross-over mechanisms. They are well-known from biology and applied, for example, to simulate the growth of plants. *Neural networks* [Day90] also have a biological motivation, our own brain and nerve cells. This kind of parallel system relies on a high amount of interconnections and simple processors. Neural networks are applied in robotics, speech systems, medicine, etc.

Examples of fault-tolerant systems are the controlling of an aircraft, an automated factory, lift controlling, etc. Also a highly reliable banking system has to be fault-tolerant.

Inherently distributed are all kinds of mailing systems where communication is realized via a physically distributed network like the Internet. Applications such as video conferencing also belong to this kind.

An arbitrary software system can be implemented as a distributed application to increase reusability, modularity and portability of software. Most of the existing software systems can also be designed as *client/server systems* where the servers offer some services the clients request for. The functionality of client/server systems is extendable by replacing application components with either newly developed parts or by integrating already existing ones.

1.2 Requirements and Important Aspects of Modeling Support

In the previous section some examples of parallel and distributed applications have been presented. We discuss now how these applications can be modeled as parallel or distributed systems and how this modeling can be supported. The design technique for communication-based systems should be problem-oriented and sufficiently abstract to concentrate on the main aspects dealing with *parallelism* and *distribution*. There are four main issues distinguishing the description of communication-based systems from sequential ones: *parallel execution*, *communication* and *synchronization* among different parts, *communication network* and *handling of partial failures*.

1.2.1 Parallel Execution

Parallelism can be expressed on different abstraction levels, also called *grain*, leading to different kinds of parallel units. The grain is the amount of execution time between communications. Large-grain parallelism can be described by parallel execution of actions most of the time and little amount of communication whereas fine-grain parallelism means closely coupled parallel actions increasing the amount of communication.

In a sequential system, the parallel unit is the whole system running in parallel with the outside world (not modeled). Parallel units can be *parallel statements* and *actions*, *functions*, *processes* or *objects*. If the system architecture consists of only one processor, *true parallel* execution of parallel units is not possible. Another form of parallelism has been evolved instead, called *interleaving*, where some kind of independence of executions is specified such that the result is independent of the execution order.

Parallel statements and actions The probably simplest way to express parallelism is to group parallel statements or actions together that are to be executed in parallel. The corresponding concept of *parallel composition* is supported by, for example, CSP [Hoa85], CCS [Mil80], state charts [Har87] or Petri nets [Rei85]. Parallel composition is usually defined for a fixed number of units. Another method for creating a problem-dependent amount of parallel units is the parallel processing of actions in any possible part as offered by *L-systems* [RS86, RS92]. Some restricted form of this unbounded parallelism is the *for-loop*. Several sequential programming languages are extended by parallel constructs as parallel composition or parallel for-loops.

Functional and logical parallelism The computation of a function usually does not have any side-effect. Thus, it should not make any difference in which order several functions are executed. All functions can be computed in parallel except the case that a function using another function has to wait for the result to be further executed. This parallelism can be large or fine-grained (dependent on the meaning of the functions), problem-oriented and unbounded. The computation of functions is well suited for data-flow computers. *Higher-order functions* such as "map" ([PM91]), i.e. the application of a function to a sequence of elements, provide a compact way of describing systems that work in parallel (similarly to for-loops).

Logic programming offers so-called *AND/OR-parallelism* in form of independent *clauses* worked on in two different parallel ways. Clauses cannot be evaluated in parallel if they share some variables which may be instantiated differently.

Processes There exist many different definitions of a process in the literature. In this thesis we regard a process as some kind of execution unit with its own state and behavior. Several processes are allowed to run in parallel. A process itself proceeds either sequentially or in parallel using *threads*. The main difference between processes and threads concerns the encapsulation of data. Threads usually do not have their own state and data. Usually, processes can create new processes, but they terminate themselves if they ever terminate. Processes may be connected by channels used for passing parameters to other processes. Notions of processes can be found in CSP, CCS, Ada, Linda [ACG86], Petri nets [BD87], etc.

Objects An *object* is a self-contained unit which encapsulates data and behavior, and that communicates with other objects by *message passing.* The behavior is described by its *class* which documents all the methods that can be started on the object. *Inheritance* allows a class to be defined as an extension of another class. Object models supporting inheritance are called *object-oriented* otherwise they are *object-based.* A method can be executed by sending a message to an object. Parallelism on objects can be obtained by allowing an object to be active without having received a message and to continue after returning the result. Furthermore, the sender is allowed to send messages to several objects in parallel and to proceed in parallel with the receiver. Objects with these properties are called *active* in contrast to *passive* objects not allowing parallelism. Active objects, also called *agents*, are supported by actor systems [Agh86], ConcurrentSmalltalk [YT87], etc.

1.2.2 Communication and Synchronization

Communication and synchronization concepts between different system parts follow either the *message passing* or the *data/object sharing* paradigm. Parallel systems which are non-distributed rely on shared data whereas most of the distributed specification techniques use message passing to model communication.

Message passing Sending a message we can distinguish four modes: *point-to-point messages*, *rendezvous*, *remote procedure calls* and *broadcasting.* Message passing is usually *named*, i.e. the sender names the receiver and the receiver usually knows from whom the message comes. The sender starts the interaction by sending a message or invoking a remote procedure. The receiver may receive the message either implicitly or explicitly executing some kind of accept operation. Naming of the sender or the receiver may be indirect which means that an intermediate *mailbox* is involved.

Message passing may be *synchronous* or *asynchronous.* Synchronous message passing means that the sender has to wait until the receiver has accepted the message. Thus, message passing is also a means for synchronization. Asynchronous message passing means that the sender does not have to wait for acceptance and can continue immediately after sending.

The *rendezvous* concept is similar to that of *remote procedure calls (RPC's)* in some sense. When a process calls a remote procedure provided by some server the parameters are sent to the server which executes the contents and returns the result back to the caller. During the execution the caller is blocked, thus an RPC is a fully synchronous interaction. In contrast to RPC's the rendezvous concept provides an accept statement for accepting a call. After the execution of an accept the caller is unblocked again although the request might be still processed.

Broadcasting means to send a message to all components of a system. This facility has the power to speedup communication because several copies of the message are sent in parallel.

Message passing concepts are supported by most of the distributed specification and programming methods such as actor systems, Ada, etc.

Data/object sharing If two system parts overlap in some objects they can share them, i.e. an object may be changed by one part while the other has to notice that. This is true whether the system is distributed or not. Although it might look unnatural to use shared objects in distributed systems, such a communication type has some advantages. Whereas a message basically transfers information between two specific parts shared objects are accessible by all parts which are involved. Changing a shared object has immediate effect in contrast to message passing where a certain amount of time has to pass between sending and receiving. However, object sharing requires precautions to prevent multiple parts from simultaneously changing the same data or object. Data/object sharing usually supports *asynchronous communication* by one part changing an object and another part recognizing it some time later. Synchronization would be performed if the part transforming the object has to wait with the transformation as long as all other parts sharing that object are ready to listen.

Sharing without distribution has been used, for example, by logical programming languages. *Distributed data structures* are the basis of Linda's underlying structure, the *tuple space*.

1.2.3 Communication Networks

The communicaton network is the medium for transfering information between different local system parts. There exist many different types of networks, e.g. hypercubes, local-area networks (LAN), wide-area networks (WAN). The network determines the speed and reliability of the communication and the spatial distribution of system parts.

Massive parallelism is most easily achieved by the composition of similar system components communicating across regularly structured networks. This regularity can be exploited for designing systems in-the-small and then scaling upward to obtain the performance of massive parallelism. The most general, but most expensive network is that of a complete graph, i.e. each component is connected to each other one.

Most of the distributed systems are systems with component architecture, i.e. their network structures are dynamic. Networks for interprocess or object communication are usually dynamic, i.e. they allow the setting up of communication channels. Some precautions may be needed to prevent processes from trying to communicate with terminated processes. Objects communicate by invoking each other's operations.

1.2.4 Handling of Partial Failures

In parallel and distributed systems partial failures may occur meaning that parts of the system do not run at all, run with failures or become disconnected from the system. If the system can react by shifting the tasks of the failed part to others it increases reliability as well as availability. A system is called *fault tolerant* if it still continues functioning properly after a partial failure. The following concepts to achieve fault tolerance are quite common: *replication of tasks*, *transactions* and *exception handling*.

Replication A common method to ensure more reliability of the system is to replicate system parts. Critical parts of a system have to be controlled by others, i.e. their behavior is checked from time to time. If they failed in the meantime their tasks are taken over by other parts. This method causes some information overhead since the state of the faulty part has to be known and consistency of the faulty part has to be assumed.

Transactions A transaction is a group of actions which has to be performed in an atomic way. This means that a transaction is *indivisible*, i.e. either all or none of its actions have to be executed. If the transaction fails the initial system state has to be recovered such that the transaction has no effect at all.

Recoverability can be achieved by locking all required resources of the transaction. If none of its actions failed a *commit* action is performed where all locks are removed. Otherwise an *abort* action sets all resources to their initial state.

Transaction models are distinguished into *sequential transactions* and *distributed transactions*. In contrast to sequential transactions a finer grain of parallelism can be achieved by using distributed transaction models where, for example, transactions can be divided into subtransactions allowed to be performed in parallel. For this kind of transactions it is much more difficult to preserve recoverability.

Transactions originally stem from the database world, but are realized to be very useful for specification and programming parallel and distributed systems with high reliability.

Exception handling Fault tolerance of a system can also be achieved by starting an exception handling in case of partial failures. The exception handling states what has to be done if certain actions fail. It may include the possibility of rerunning the failed system part, replication of tasks, etc. Exception handling is supported by, e.g. Ada and C++, offering special constructs.

1.3 Examples of Specification Techniques

A formalism which is widely used to model and reason about the behavior of parallel and distributed applications are *Petri nets*. It is especially the graphical representation and the operational view of the behavior which supports modeling. Furthermore, a large amount of analysis techniques are available for Petri nets. In this section we present Petri nets together with other specification techniques for parallel or distributed systems, namely L-systems and actor systems, which are interesting in connection with parallel and distributed graph transformations. Main features of these techniques are also essential for graph transformation, like graphical representation of states and event or rule-based character. Of course, several other specification techniques for communication-based systems have been introduced some of them already mentioned above, like CSP, CCS, process algebra [BK86], state charts and transition systems [Kel76].

1.3.1 L-Systems

Lindenmayer-systems (L-systems) introduced by A. Lindenmayer describe a kind of context-free parallel replacement on strings. A short survey is given in [LJ92]. For a large overview over different kinds of L-systems see [RS86] and [RS92]. The introduction of L-systems was motivated by modeling the development of organisms.

L-systems support a discrete model of evolution. They describe state transitions in discrete subunits (cells), each subunit passes a finite number of states and the transformations are performed in discrete time steps.

A *string OL-system*, the simplest form of an L-system with "zero-sided interactions", consists of a finite *alphabet* modeling the states, a finite set of *state transition rules* (productions) and a *starting string*. The productions are of the form $a \to w$ where a is one element of the alphabet and w is a string (word) over the alphabet. Such a production represents the transformation of a cell state a into the string w of cell states. State transitions allow a cell to remain in its state, to change it, to divide it into several new cells with some state or to disappear. Development starts with a given cell system in some state. This system can evolve in *transformation steps*. A transformation step performs the parallel replacement of each given cell state by a string of cell states specified by the productions. The resulting strings have to be ordered along the given cell ordering. Different selections of suitable productions may lead to some non-determinism. A *transformation* consists of consecutive transformation steps. If there is exactly one production for each state symbol the L-system is called deterministic since it produces a single infinite transformation modeling the *global* behavior of the system.

A variety of extensions have been proposed for OL-systems. We shortly discuss some of them: *TOL-systems* contain a structured set of productions consisting of *tables*. Each table being a set of productions describes a global operation on a cell system state. *EOL-systems* incorporate a terminal alphabet well-known from Chomsky grammars. Switching from strings to *maps* multi-dimensional structures like cells can be better modeled. Maps are graphs with certain regions. *Graph L-systems* are described in section 2.2.

L-systems are useful to describe massively parallel systems simulating the application-inherent parallelism of growing plants. Parallelism occurs on the level of parallel actions. Communication is described by sharing.

1.3.2 Petri nets

C. Petri introduced Petri nets in 1962 to describe the behavior of non-sequential processes. A technical introduction to Petri nets is given in [Rei85]. In the beginning, *condition/event nets* have been introduced which describe the behavior in form of events that may occur satisfying certain conditions.

The operational behavior of Petri nets is described by the *token game*. Condition/event nets resemble *data flow diagrams*. If all its conditions are satisfied (indicated by a token on each node modeling a precondition) an event may occur leading to another state satisfying other conditions and so on. Event handling is shown by removing the tokens from preconditions and putting them to postconditions of the event.

If more than one token on one node (called *place* in the following) is allowed we turn to *place/transition nets* describing processes on sets of objects. Two different types of nodes are distinguished: *places* and *transitions*. Places model passive parts like buffer, stores, memories, etc. while transitions describe actions on objects being buffered or stored. The set of tokens at one place is indistinguishable. Transitions have *input* and *output places*. The token game describes when and how actions may be performed on these objects. For each input place (output place) of a transition the number of objects needed (produced) by the transition is specified. Firing a transition, the indicated number of tokens is removed (added) on each input place (output place). Since the state of a net is composed of the state of each place (modeled by the number of tokens on it) a Petri net can model some kind of distributed systems. The communication can be described by interacting transitions shifting tokens between places. There exist a large amount of analysis techniques for Petri nets investigating the operational behavior of the net, causal dependencies of events or actions as well as liveness and safety properties.

Many extensions of place/transiton nets have been presented allowing distinguishable tokens on one place. Nets with this property are called *high-level nets* (in contrary to *low-level nets* presented above) and include *predicate/transition nets* ([Rei85]), *colored nets* [Jen92], *algebraic high-level nets* [EPR94], etc. *Stochastic Petri nets* [Mar89] have been introduced to study time-dependent behavior of real-time systems.

Parallelism is expressed by parallel firing of transistions modeling parallel actions.

Petri nets may be equipped with a process semantics allowing true parallel processes. A Petri net itself may describe a process. Composing Petri nets parallel processes may be modeled. Communication is simulated by sharing of places, transitions or tokens.

1.3.3 Actor Systems

The actor model (following W. Clinger [Cli81] and G. Agha [Agh86]) is a concurrent and object-based computation paradigm. The active communicating objects are called *actors*. Communication is purely based on asynchronous *message passing*. The transmission order of messages need not to be preserved in their arrival order reflecting corresponding problems in distributed environments. The actor model requires fairness in the sense that a message which can be delivered will be delivered after some finite amount of time. Each actor has a *mail address* and a private part modeling the actor's *behavior*. An actor usually does not know all the actors within the system but only a subset, called *(actor) acquaintances*. A *message* has a target actor, the receiver, and may have a value and a list of actors, the *message acquaintances*. An *event* is the receiption of a message by an actor. After being processed the message disappears. New messages and new actors may be created and the receiving actor may update its acquaintance list as well as its private state. Actor systems have been formally defined by Clinger in [Cli81] providing a *partial ordering semantics* for actor systems.

Graph transformation models for actor systems are reviewed in section 2.3.

Several extensions of the actor model have been studied in the literature due to the incorporation of inheritance and modularization.

Chapter 2

Graph Transformation for Communication-Based Systems: State of the Art

This chapter reviews the main features and results of graph transformation for communication-based systems independently of a specific approach such as the algebraic or algorithmic approaches. A large survey of contributions in the graph transformation field including a variety of different applications can be found in [ENRR87, EKR91, SE94]. The main graph transformation approaches are presented in [Roz].

The main approaches to modeling communication-based systems by graph transformations are presented and compared with each oters in sections 2.2 and 2.3. Their advantages as well as disadvantages for modeling parallel and distributed systems are shortly discussed along the requirements for these systems stated in chapter 1.

A reader not familiar with graph transformation can get an overview and operational understanding of sequential and parallel graph transformation as well as the various applications to distributed system modeling.

2.1 Informal Approach to Graph Transformation

In this section the basic features of graph transformation are reviewed. The presentation follows an informal style and combines several aspects of different graph transformation approaches. It is mainly oriented at the work by Kreowski and Rozenberg in [KR90] and Andries et.al. in [AEH+96].

Based on graphs as underlying structure the concept of rule-based graph transformation is presented. It covers many of the major features one can find in the graph transformation literature. Furthermore, several special cases are discussed closely related to current graph transformation approaches.

Graph transformation is described by the application of *graph productions* which model the permitted actions on graphs which represent states of a system. Graph transformation defines a relation on graphs which can be iterated arbitrarily yielding the transformation process. In this way a set of productions gets an operational semantics.

First we present different kinds of graphical structures suitable for graph transformation, afterwards the transformation process is explained and finally we discuss some properties of graph transformation which are of interest if graph transformation is considered as specification and programming paradigm for parallel and distributed systems.

2.1.1 Graphs

As basic structures of graph transformation we choose directed, node- and edge-labeled graphs. Such a graph consists of a set of nodes, a set of edges, two mappings assigning a source and a target node to each edge, and two labeling (or color) mappings assigning a node and an edge label (color) of a given label (color) alphabet to each node and each edge, respectively.

Often edges are just used for relating several nodes. In this case edges do not need to have an own identity but are defined as binary relation on nodes. For the presentation of the following concepts we assume separate identities of edges described above.

An edge relates its source node with its target node and is called *incident* or *adjacent* with these nodes. In graphical representations, nodes are drawn as points (circles, boxes, etc.) and the edges are drawn as arrows from the source point to the target point. Node and edge labels are written beside the points and arrows or inside the circles, boxes, etc. or distinguished by different shapes, colors, fill styles, etc.

A graph G is a *subgraph* of graph H if the node set of G is a subset of the node set of H, the edge set of G is a subset of the edge set of H, for all edges e in G, the source and target of e coincides with the source and target of e in G, and the labeling of all nodes and edges in G coincides with the corresponding labeling in H, respectively.

Graph G has an *occurrence* in graph H, denoted by $G \to H$, if there are two mappings (f^N, f^E) for nodes and edges which preserve source, target and labeling mappings. If the mappings are bijective, then G and H are *isomorphic*. If function f^N (f^E) is not injective different nodes (edges) are *identified*. The occurrence of G in H is also called *homomorphic image* of G in H.

Variations and Extensions of the Graph Concept There are a number of variations and simple extensions of the graph concept. The main concepts are listed below. *Directed graphs in the classical sense* (i.e. graphs where the edge set is given by a binary relation on the set of nodes) coincide with directed graphs without parallel edges. *Undirected graphs* may be represented as directed graphs by replacing each undirected edge by two directed edges in opposite direction. *Hypergraphs* generalize graphs in the sense that an edge has a sequence of source nodes and a sequence of target nodes. *Hierarchical graphs* are graphs where a subgraph can be abstracted to one node and a bunch of edges between two abstracted subgraphs to one edge.

In the area of specification and programming the concept of distinguishing objects by types is important. In the following it is transfered to graphs.

Typed graphs [CMR96a] are graphs where the nodes and edges can be of different types. The source and target mappings are structured according to the types, i.e. all source and target mappings operate on the given node and edge types. Typing mechanisms for graphs can also be presented by *graph signatures* ([Löw93]) or *schema graphs* ([Sch91b]).

Attributed graphs ([Sch91b], [LKW93]) are graphs where node and edges can be equipped with attributes. An attribute can be a number, a text, an expression, a list or even a graph. Attributes can be of different types and attribute operations which are compatible with these types are available to manipulate the attributes.

2.1.2 Graph Transformation

Graph transformation generalizes the well-known concept of string transformation. In the string case, a production (u, v) is a pair of strings and the application of the production to a string w means to choose an occurrence of u in w and to replace u by v in w.

In the graph case, a production consists of a pair (L, R) of graphs together with some additional information about the embedding of R. The application of the production to a graph G means — more or less — to choose an occurrence of L in G and to replace L by R in G, i.e. to remove the occurrence of L and to insert R. In many cases, the removal has to be done in a controlled way. So application conditions can be formulated to avoid a destructive removal of L together with all incident edges. In contrast to string replacement, the insertion point of R is not uniquely determined, but has to be further specified. This is done by a gluing and an embedding component specifying the gluing of R, described the required context, and the embedding of R into the context of L, specifying the optional context of R.

Productions A *production* $p = (L, R, K, glue, emb, appl)$ consists of two graphs L and R, called the *left-hand side* and the *right-hand side* of production p, respectively, a subgraph K of L called the *intermediate graph*, an occurrence *glue* of K in R, relating the intermediate graph with the right-hand side, an *embedding relation emb*, relating nodes of L to nodes of R, and a set *appl* specifying the *application conditions* for the production.

Application of Productions An *application* of a production $p = (L, R, K, glue, emb, appl)$ to an *original graph* G yields a *resulting graph* H, denoted by $G \stackrel{p}{\Longrightarrow}_d H$, provided that H can be obtained from G as result of the following five basic steps:

1. CHOOSE an occurrence of the left-hand side L in G.
2. CHECK the application conditions according to *appl*. If they are fulfilled the occurrence of L in G is called *match* .
3. REMOVE the occurrence of L (except of K) from G as well as all *dangling edges*, i.e. all edges which have lost their source and/or their target. This yields the *context graph* C of L which still contains an occurrence of K.
4. GLUE the context graph C and the right-hand side R according to the occurrences of K in C and R, that is, construct the disjoint union of C and R and, for every item in K, identify the occurrence of the item in C with the occurrence of the item in R.
5. EMBED the right-hand side R into the context graph C according to the embedding relation *emb*: For each removed edge incident with a node v in C and the image of a node v' of L in G, called *embedding edge*, and each node v'' in R, a new embedding edge (with the same label) incident with the image of v and the node v'' is established provided that (v', v'') belongs to *emb*.

Example 2.1.1 (Application of a graph production) *Figure 2.1 illustrates the steps which have to be performed when applying a production. The solid, bold arrows between different graphs indicate occurrences. The thin dashed arrows between graphs should clarify the transformation process. Preserved nodes are drawn as black circles. The bold dotted arrows inside the graphs are embedding edges. The embedding edge in graph G is a dangling edge after removal of L and therefore, deleted. Embedding edges in H are inserted along the embedding relation emb.*

Graph Transformation Approaches For readers familiar with graph transformation, it shall be mentioned that the framework presented above combines the gluing aspects of the algebraic approach ([Ehr79, Löw93]), the embedding aspects of the set-theoretic approaches ([Roz87],[Nag87]) in the 1-context, i.e. the direct neighborhood of the occurrence and application conditions for productions in different approaches (see [KLG91], [HHT96]). The *double-pushout approach* [Ehr79], the *single-pushout approach* [Löw93], the *node label controlled replacement (NLC)* [Roz87], the Δ-*approach* [KLG91] as well as the *set-theoretic approach* used in PROGRES [Sch91a], can be seen as special cases of this framework. To make this more transparent, let us consider some special cases.

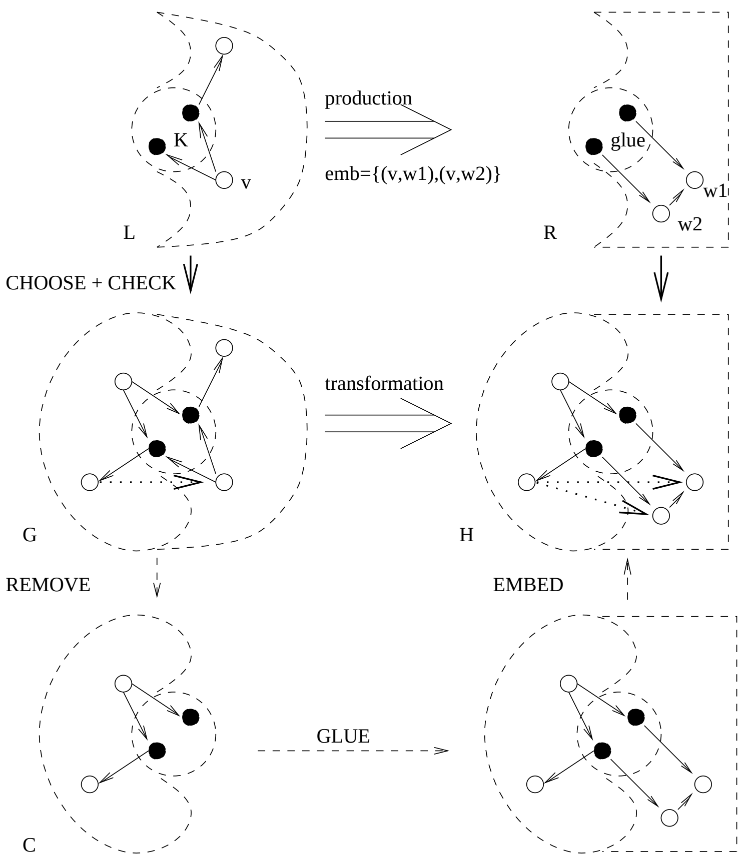

Figure 2.1: Illustration of the application of a production

1. CHOOSE+CHECK. In general, the occurrences of L in G are allowed to identify nodes and edges. In practice, one often use the *injectivity condition* which requires that there is a subgraph L' of G isomorphic to L. In general, the occurrences of L in G are allowed to be non-injective, i.e. nodes as well as edges can be identified.

   Special application conditions are the *contact* or (*dangling*) *condition* and the *identification condition*. The contact condition assures that no dangling edge will arise in REMOVE. It requires that whenever the image of a node in L contacts some edge not in the image of L, then the node has to be preserved (in K). The identification condition requires that the occurrence of L in G may only identify nodes and edges which are preserved. Other interesting application conditions are context conditions. They require or forbid the existence of nodes, edges, or certain subgraphs in the given graph. A general treatment of application conditions can be found in [EH85, KLG91, HHT96].

2. REMOVE. In the special case that the contact condition is satisfied, no dangling edges arise (and have to be removed) when the occurrence of L (up to K) is removed. So the removal of items is completely specified by the production. If the contact condition is not required, the application of a production may remove dangling edges from the context of its occurrence.

3. GLUE. In the special case that the occurrence of K in R is isomorphic to K, the result of the gluing step may be obtained by adding R (except of K) to C.

4. EMBED. In the special case where emb is the empty relation, no additional edges are inserted. In general, emb allows the embedding in 1-context: An edge incident with a node v in the direct neighborhood of the occurrence of L and a node v' in the occurrence of L is redirected to the node v'' of R if (v', v'') is in emb.

A production in the presented approach describes a production in the double-pushout approach if the embedding relation is chosen to be empty and the application conditions to be the contact condition and the identification condition. Similarly, it describes a production in the single-pushout approach if the embedding relation as well as the set of application conditions are empty. (Note that the production application in example 2.1.1 does not fit into the pure algebraic double- or single-pushout approach since emb is not empty.) A production in the NLC-approach corresponds to a production in this approach if the left-hand side L is exactly one node, the intermediate graph is empty and the set of application conditions is empty. Δ-productions correspond to productions in this approach with empty emb relation but are differently represented (compare figure 2.2 where the parts in brackets are of this framework). Finally, a production in (a restricted version of) PROGRES can be described by a production with injective gluing component and injectivity condition.

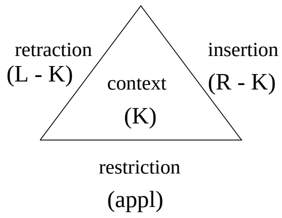

Figure 2.2: The scheme of a Δ-production

Extensions of the Graph Transformation Concept Transforming typed graphs we have to ensure that the types are not changed during the transformation, i.e. all nodes and edges which are preserved keep their types. In a production respecting typed graphs all the nodes and edges of the image of the gluing graph in the right-hand side keep their types. The same holds for the occurrence of the left-hand side L of a production in a typed graph G. Here, L has to be mapped type compatibly into graph G. Looking at the embedding mechanism there are new possibilities of declaration using types. Instead of indicating the embedding relation by nodes it can be described on the basis of node and edge types (like in NLC).

The transformation of attributed graphs contains the transformation of attributes used in the original graph. Depending on the types of attributes the rules may contain variables which have to be instantiated in the occurrence. This means that an occurrence of the left-hand side has to be attributed compatibly according to a suitable substitution of possibly existing variables. Moreover, such an occurrence can be restricted by additional conditions formulated for attributes. During the transformation new attributes can be computed from the original ones using attribute operations defined for the corresponding types.

2.1.3 Graph Transformation Systems

Given the notions of a production and a graph transformation, graph transformation systems, graph grammars, and generated languages can be defined (as usual). A set $\mathcal{P}$ of productions together with a label alphabet C for nodes and edges forms a *graph transformation system* GTS; if GTS is extended by a *start graph* S and a set T of *terminal* symbols a *graph grammar* is described.

The set of all possible graph transformations using $\mathcal{P}$ and C is the operational semantics of GTS. The set of all graphs derived from the start graph S by productions of $\mathcal{P}$ which are labeled by symbols of T is said to be the *language* generated by $\mathcal{P}$, S, and T and is denoted by $L(\mathcal{R}, \mathcal{S}, \mathcal{T})$.

Note that there is some non-determinism: Given a graph G, we have to choose a production from a set of possible applicable productions. Moreover, for a chosen production, there may be several occurrences of the left-hand side. Thus, the result of a graph transformation depends on the chosen production and the chosen occurrence. Furthermore, there is no control structure on the set of productions in a graph transformation system: A production can be applied whenever there is an occurrence of the left-hand side and the application conditions are satisfied. If one wants to apply productions in a specific way, one has to restrict the transformation process: prescribing the sequence of productions in a transformation sequence, determining the next production by the previous one(s), or choosing the production in dependence of its priority. Prototypes of these transformation systems are matrix systems as considered in formal language theory, programmed graph transformation systems or transactions(see e.g.[Bun82] and [Sch95]), and graph transformation systems with priorities (see [LM93]).

2.1.4 Properties of Graph Transformation

In the following, different properties of graph transformation are discussed which are of interest if graph transformation is considered as a specification and programming method for parallel and distributed systems. These properties can be made precise since graph transformation can be described formally using, for example, set theory, algebra or category theory.

The application of graph productions has a local character, thus unwanted side-effects of operations are not modeled. Many operations come up in pairs where one is the inverse of the other. Such undo/redo possibilities, desired for e.g. databases or graphical tools, are also supported by graph transformation. Specifying operations by graph transformation, invariant consistency conditions may be stated and checked automatically. Usually graph transformation is highly non-deterministic. To bound this non-determinism and offer parallel as well as functional behavior several results concerning independence and confluence of graph transformation are discussed. Furthermore, the complexity of transformations is examined to offer efficient tool support.

Locality of transformation The rule-based character of graph transformation ensures a certain degree of locality of action by treating mainly the occurrence part of a work graph. This principle is weakened in some respects. These may be non-local application conditions and a non-local computation of new attributes. The deletion of dangling edges and the embedding mechanism as described above do not destroy the locality property since they concern the direct neighborhood of the occurrence only. Using programmed graph transformation or transactions the principle of locality is removed.

A side-effect of such a locality property is that the sequential, possibly dependent application of several productions may be composed into one new production being a shortcut of the whole transformation sequence and only applicable in a bigger context.

Invertibility of transformation A production may be inverted if it does not identify nodes or edges, i.e. morphism $glue$ is injective, satisfies the contact condition and if the embedding relation is empty (see [Ehr79]). Furthermore, all application conditions given for the left-hand side have to be transformed into equivalent right-hand side conditions. Moreover, the transformation on the attribute part has to be invertible.

Preservation of graph structure and consistency conditions Transforming graphs, the resulting structure is again a graph since dangling edges are either prohibited or deleted and conflicts between preservation and deletion of items are either allowed and solved (in both ways) or forbidden. Using typed graphs the types are not changed during the transformation. Results in this direction are achieved in each graph transformation approach. All added parts have to respect the given graph type. Thus, the resulting graph is of the same graph type ([CMR96a]).

Moreover, consistency conditions concerning the existence and non-existence of subgraphs, cardinality restriction for the number of in- and outgoing edges, etc. can be proven for all resulting graphs. Some of the consistency conditions can be transformed into equivalent application conditions in the sense that given a consistent graph a transformation where all application conditions are satisfied produces a consistent resulting graph ([HW95]). Consistency conditions including attributes can be checked if they are decidable for the attributes.

Independence and confluence of transformations Transformations are said to be independent of each other if the effect of the first one does not affect the result of the second one, and vice versa. This property can be characterized by transformations which do not overlap in deleting elements or affect the opposite application condition. Independent transformations can be executed in either order which leads to equal results. Such an execution models some kind of interleaving of actions. Independent productions may also be applied in parallel offering true parallelism. First results in this direction have been achieved in [EK76a].

Based on independency some criteria for confluence of graph transformation systems have been developed based on the idea of critical pairs ([Plu94]).

Complexity of transformation The central problem concerning complexity of graph transformation is the problem of efficiently matching the left-hand side of a production to a subgraph of the current working graph. Similar problems can also occur when testing the application conditions. For an efficient implementation of the matching problem various ideas have been developed: reduction of costs by using node and edge types where the rare ones are treated first, indication of partial matches, early evaluation of conditions, etc. (compare [Zün95a]). If transformation sequences shall be executed occurrences can be passed to the resulting graphs such that small differences have to be computed only ([BGT91]).

2.1.5 Languages and Tool Support

Until now there are only few languages and tools available which are based on graph transformation. One of the very early specification languages based on graph transformation has been PLAN2D [DFS75]. Tools based on graph transformation are e.g. PAGG [Göt88], GraphEd [Him91], PROGRES [Sch91a], PLEXUS [Wan92], DACTL [GKS91], AGG [LB93] and GradeOne [San93]. Most of them contain a graph editor for drawing graphs and an interpreter for graph transformation. The editors usually support graphical representations, mouse/menu-driven interfaces and different views on graphs, like different sections of a graph, different layout or different levels of abstraction as in the AGG-System. The interpreters mostly offer the performance of production application at user or automatically selected occurrences. If the language offers programmed graph transformation as in PROGRES the interpreter may provide a back-tracking mechanism to perform the prescribed application of productions. Moreover, some of them offer a type checker, a checker for graph-theoretic properties, automatic layout possibilities or cross-compilers to Modula-2 and C.

2.2 Survey of Existing Approaches to Parallel Graph Replacement

In the following, we consider the state of the art concerning parallel graph replacement where most of the approaches are motivated by L-systems leading to *graph L-systems*. This subsection contains also the description of *amalgamated graph transformation* describing a synchronization mechanism between dependent actions. Papers presenting parallel graph transformation concepts as introduced in chapters 3 and 4 are not mentioned here but in those chapters.

Graph L-systems are a direct generalization of L-systems which describe a kind of context-free parallel replacement on strings. (For an overview over L-systems see section 1.3.) The introduction of L-systems was motivated by modeling the development of organisms. The starting point for the step to graph replacement in L-system manner was the fact that L-systems mostly describe the development of multi-dimensional structures like cells. Most of the applications code the neighborhood relations of cells into strings which seems to be unnatural and causes complex development steps with additional restrictions on them. Graphs can model multi-dimensional structures in a simple and clear way. A similar approach to solve this problem are map grammars presented by Lindenmayer in [Lin87] which resemble parallel edge replacement systems. Maps are graphs with certain regions.

Several papers show the usefulness of graph L-systems in computer science, for example, for semantic description of programming languages in [Nag79], pattern generation in [HK92, Kre92, DDK93] or the specification of the structure of parallel algorithms in [BCF91].

In the following, we show the main approaches to graph L-systems and discuss shortly their similarities and differences. The basic actions of a transformation step in graph L-system are the following: Given a set of graph productions the whole given (mother) graph is replaced by these productions in parallel at all possible matches. The resulting (daughter) graph consists of the connected right-hand sides of the given productions. The connection mechanism dependents heavily on the graph transformation approach used.

Nearly all the approaches discussed below describe graph L-systems except the part about amalgamation of graph transformation which presents a concept of synchronizing dependent graph transformation by subtransformations. This concept seems to be extremely useful to describe synchronization of parallel actions.

2.2.1 The Node Replacement Approaches

Nagl ([Nag79]) studies graph L-systems where all graph productions are context-free, i.e. node replacement productions. After the application of these productions at all possible matches in the *mother graph* the right-hand sides of them (*daughter graphs*) which are allowed to be arbitrary graphs have to be connected somehow in the resulting *daughter graph*. This is done by so-called *connection rules* where an edge between two nodes of the mother graph is replaced by one or more edges between nodes in the daughter graphs. (We consider the approach with explicit connection rules here. For the implicit case the reader is referred to [Nag79].)

A connection rule consists of two graphs where the left-hand side is a handle (an edge with its source and target nodes) and the right-hand side is a stencil graph. These graphs are bipartite distinguishing source and target nodes with connecting edges between these two node sets. Similarly to the graph productions the connection rules are applied at all edges of the mother graph where they are applicable. Applying a connection rule to an edge of the mother graph the corresponding daughter graphs of its source and target nodes are connected by the stencil in the right-hand side of the connection rule if the stencil is applicable. Although a connection rule can be applied to a given edge in the mother graph the situation can occur that the stencil of the connection rule cannot be established between the corresponding daughter graphs because the graphs are too "small".

A mother graph can only be replaced in parallel if there is a graph production given for each node. If more than one production is given one of them is chosen arbitrarily. This construction leads to some kind of non-determinism.

Example 2.2.1 (Parallel Replacement with Connection Rules) *In figure 2.3 a parallel replacement on graphs is shown where three nodes are replaced in parallel using the three graph productions beneath it. The daughter graphs indicated by dashed ellipses are connected by the connection rules also given in figure 2.3. The source and target nodes of their stencils are separated by dashed lines. The source*

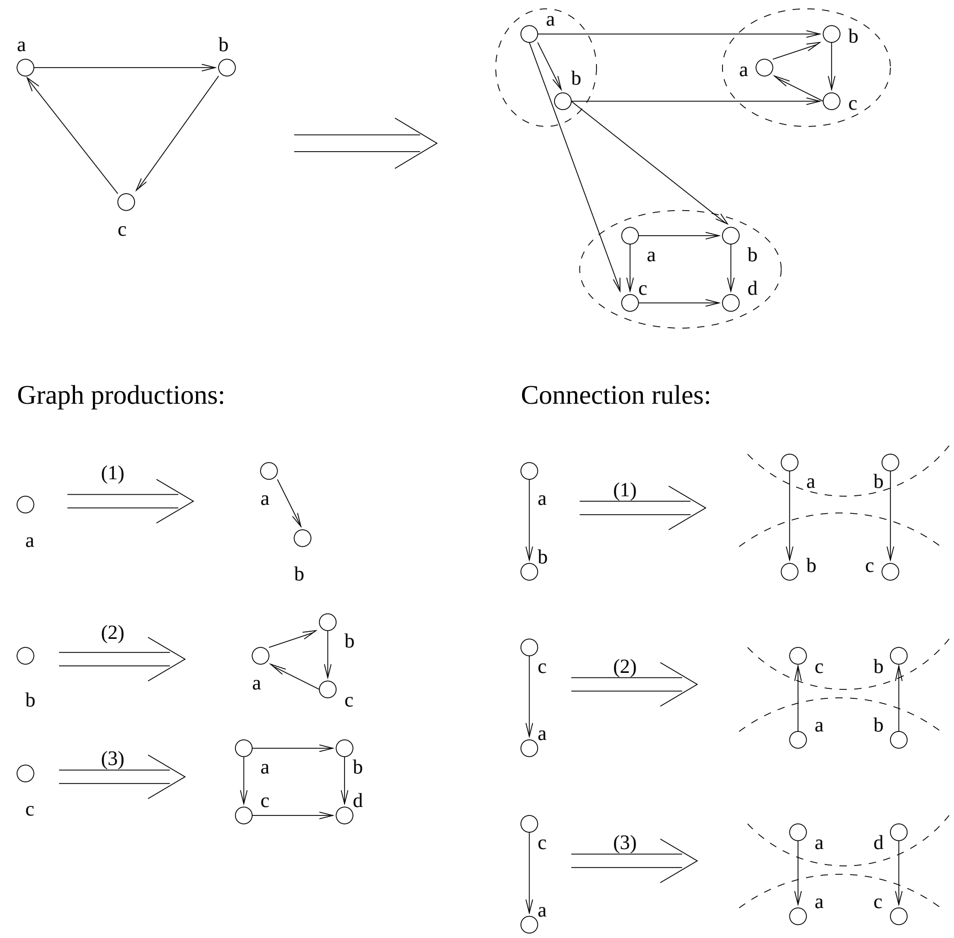

Figure 2.3: Parallel replacement on graphs with connection rules

nodes are drawn above and the target nodes below these lines. Connection rules (1) and (2) can be applied but (3) not. Although a corresponding handle can be found in the mother graph, the stencil of (3) cannot be applied because there is no node with label c in the daughter graph corresponding to the replacement of node with label a.

A *graph L-system with explicit connection rules* (e-GL-system) consists of labeling alphabets, a start graph, a finite set of graph productions for each node label one, a finite set of connection rules for all possible relations of node and edge labels and a replacement mechanism as described above.

Extended e-GL-systems are, for example, *table-e-GL-systems* where instead of one set of graph productions several sets (tables) are given, but for a parallel replacement step only one table is allowed to be used. Further types of e-GL-systems can be found in [Nag79]. There, different kinds of e-GL-systems are compared with each other concerning their *generative power*. Furthermore, it is shown that string-L-systems are contained.

A type of graph replacement that combines sequential and parallel rewriting is *mixed graph rewriting* where more than one production can be applied in parallel, but their matches do not have to cover the whole mother graph. The context is left unchanged.

Mixed graph rewriting has been used to describe program graphs and their manipulation as semantic description of programs. Transformation of program graphs is described by mixed graph rewriting and models (parallel) program execution.

In [NA83] and [AN92] Nakamura and Aizawa study the relation between table-e-GL-systems with terminal labels and picture languages for two-dimensional pattern generation. It is shown that picture languages are captured by these kinds of graph L-systems.

Until now, the connection rules describe direct connections between nodes modeled by one edge. The *path controlled embedding graph grammars* presented in [AN92] allow connecting edges between nodes in the daughter graph if their corresponding nodes in the mother graph are not directly connected but by a path, i.e. a sequence of edges connected by nodes.

A slightly different approach to parallel node replacement is presented by Bailey, Cuny and Fisher in [BCF91], called *aggregate rewriting graph grammars*. This approach differs mainly in the description of the connection of daughter graphs. A so-called *inheritance function* regulates the embedding of the daughter graphs into the context graph and the connection between different daughter graphs. For a clear description of such an inheritance the concept of a junction is introduced which consists of an edge together with either its source or its target node. A graph production consists then of a node replacement production together with all junctions its nodes can have. The inheritance is described for each junction of the daughter graph.

Aggregate rewriting graph grammars have been used to develop a tool for specifying scalable graph description for parallel algorithms. It supports the specification of *massively parallel programs* as annotated graphs.

2.2.2 The Hyperedge Replacement Approach

Another kind of parallel graph replacement in the L-system like way is *parallel hyperedge replacement* as introduced by Habel and Kreowski in [Hab92] and [Kre92]. Hyperedge replacement is a kind of context-free graph replacement where a *hyperedge*, i.e. an edge which does not necessarily attach exactly two nodes but an arbitrary finite set of nodes, is replaced by a *hyperedge-decorated graph*. Such a graph contains edges as well as hyperedges.

Transforming a hyperedge-decorated graph by a parallel hyperedge replacement step all hyperedges of the graph are replaced simultaneously if the label of each of the replaced hyperedges and the replacing decorated graph form a production of the grammar. The resulting graph is obtained by inserting the right-hand side of the corresponding production at each place where its hyperedge was taken out. If

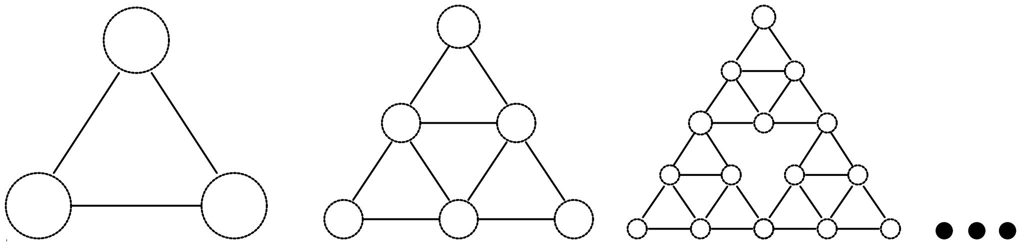

Figure 2.4: Some Sierpinski triangles

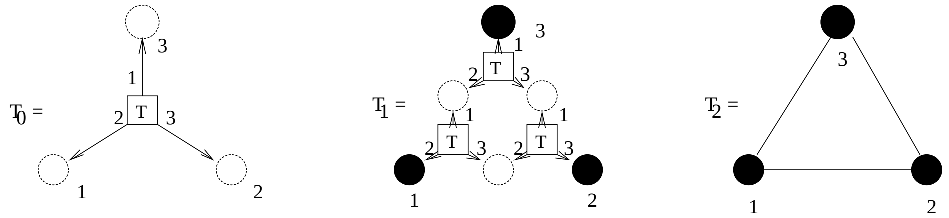

Figure 2.5: Generating Sierpinski triangles by hyperedge replacement

one of the right-hand sides used is hyperedge-decorated the resulting graph is again hyperedge-decorated. Nodes and ordinary edges in a graph are never replaced. Hyperedges are replaced by graphs along so-called *external nodes*. These nodes are fused with the corresponding nodes formerly attached to the replaced hyperedge.

Example 2.2.2 (Sierpinski triangles) *Sierpinski triangles are a rather common kind of fractals where in each step a triangle is divided in four smaller triangles. Three of them have the same orientation as the original one and they are on the border. The fourth is sited in the middle of them with the opposite orientation. The three outer triangles are replaced further. See figure 2.4 for some Sierpinski triangles.*

Starting with a graph consisting of only one hyperedge S without any attachments to nodes it can be replaced by graph T_0. Hyperedge T can be replaced then by graph T_1 providing the pattern for the next Sierpinski triangle in the family or by graph T_2 generating a Sierpinski triangle. Using this production all hyperedges are vanished and an ordinary graph has been produced which means that the replacement process stops. The nodes drawn black are the external nodes. Their numbers indicate an ordering on these nodes which has to coincide with that of the attachments of the replaced hyperedge.

A *hybrid table-controlled OL hyperedge replacement grammar* (HTOL HR grammar) consists of a set of nonterminal labels, a finite set of tables (being again a finite set of productions) and a starting axiom.

For parallel hyperedge replacement composition and decomposition of replacement processes and the decidability of the emptiness problem are shown. Moreover,

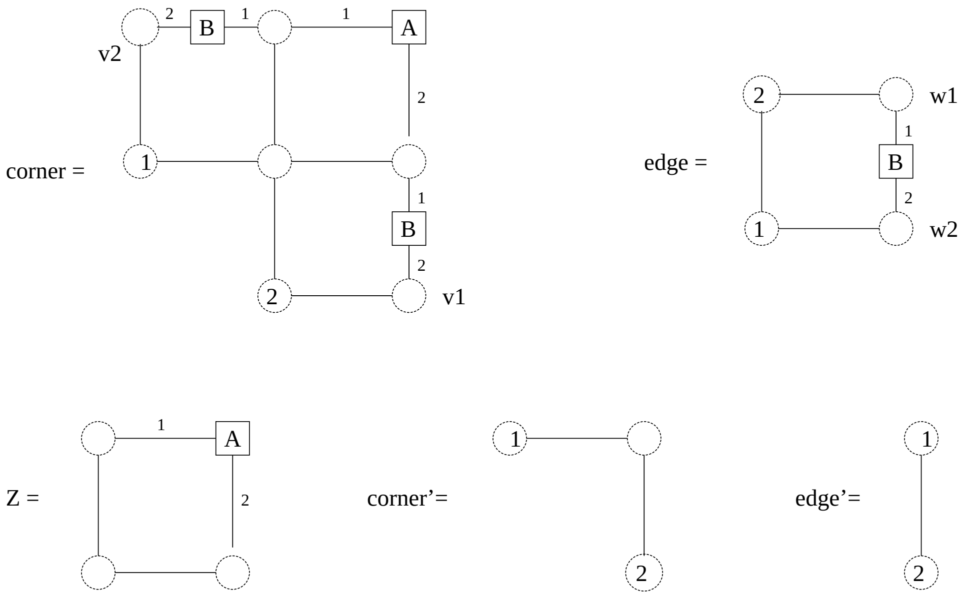

Figure 2.6: Generation of square grids

the generative power of different kinds of hyperedge replacement grammars including the sequential mode of rewriting is compared in [Kre92].

Hyperedge replacement with rendezvous is presented by David, Drewes and Kreowski in [DDK93] which describes some kind of synchronization on parallel replacement. The rendezvous mechanism allows some gluing of the replacing graphs if the attachment of the replaced hyperedges shares some nodes.

Example 2.2.3 (Square grids) *The next example of generating grids is rather popular in the field of parallel graph grammars (see e. g. [NA83], [DDK93]). Grids are especially interesting as network models in massively parallel systems ([BCF91]). Since grids, more concrete square grids, are very symmetrical their generation seems to be predetermined to be done in parallel. Unlike the node replacement approaches the generation of grids can be modeled quite clear with parallel hyperedge replacement using the additional rendezvous mechanism.*

Starting with the graph Z in figure 2.6 hyperedge A can be replaced by graph corner*. Replacing, furthermore, hyperedge B by graph* edge *the next larger square grid can be generated. These two hyperedge replacement productions are denoted by $p_c = (A, \text{corner})$ and $p_e = (B, \text{edge})$. Applying these productions in parallel without synchronization, a replacement step looks like that in figure 2.7. To get a real square grid the rendezvous mechanism has to be applied. This mechanism has to synchronize the parallel application of production p_c and p_e and the multiply application of production p_e. Synchronizing p_c and p_e nodes $v1$ and $w1$ as well as $v2$ and $w2$ have to be glued if attachment 1 of hyperedge B is adjacent to any attachment of A. Synchronizing two applications of p_e nodes $w1$ and $w2$ have to*

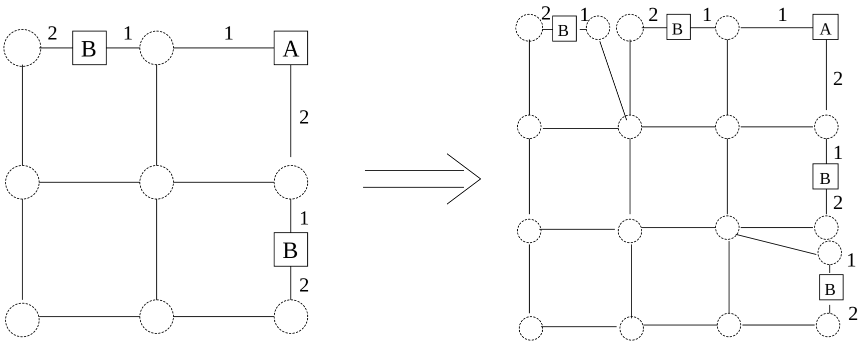

Figure 2.7: A parallel hyperegde replacement without rendezvous

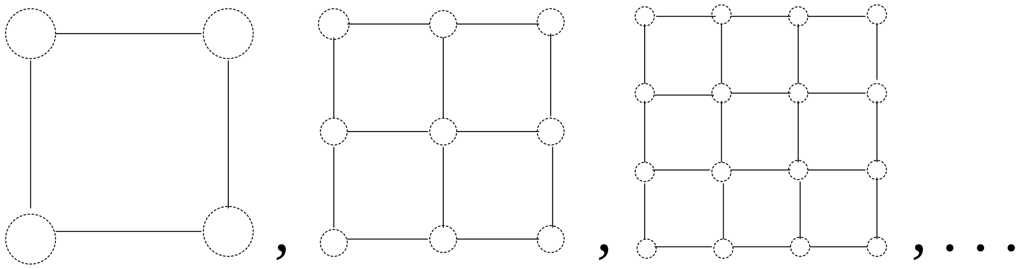

Figure 2.8: Some square grids

be glued if attachment 1 of B is adjacent to attachment 2 of another hyperedge labeled by B. Otherwise nothing is synchronized. The above described synchronized productions generate square grids as depicted in figure 2.8.

Hyperedge replacement with rendezvous increases the generative power of parallel hyperedge replacement.

Similarly to parallel hyperedge replacement Habel and Kreowski introduce *parallel collage grammars* in [HK92] where *collages* instead of graphs are replaced. Collages consist essentially of a set of parts being geometric objects and a set of hyperedges being subjects of further replacement. They seem to resemble maps in some sense which are described in connection with L-systems in section 1.3.

2.2.3 Parallel Graph Grammars in the Algebraic Approach

Based on the *stencil approach* in [ER76] by Ehrig and Rozenberg which differs only slightly from the node replacement approach of Nagl [Nag79] the *covering approach* has been developed by Ehrig and Kreowski in [EK76b]. This approach allows general noncontext-free graph productions to be used for parallel replacement. Applying these productions at all possible matches the mother graph is covered by copies of the corresponding left-hand sides which may overlap in arbitrary subgraphs. These subgraphs are not allowed to be changed during the transformation. They have to be used to glue the daughter graphs. Thus, the main steps of this kind of transformation are first the construction of a *mother covering*, transforming it into a *daughter covering* and gluing it eventually to get the daughter graph.

Considering this way of parallel replacement the problem arises how a *covering* of a graph by a finite set of subgraphs (called *elementary graphs* furtheron) has to be constructed. Searching for *homomorphic images*, i.e. copies where nodes or edges may be identified, their overlapping parts have to be determined. By gluing a covering the original graph should be reconstructed. Thus, the overlapping parts have to be part of the covering. Taking auxiliary graphs isomorphic to the overlapping parts into the covering they have to be related to their corresponding elementary graphs. This means that each two elementary graphs can have an overlapping part (or *interface graph*). Such a structure looks like a star, thus, it is also called *graph star*.

The gluing of a graph star, called *star gluing* of graphs, consists of the disjoint union of all elementary graphs where nodes and edges related by interface items are identified.

A special case of star gluing is the *handle star gluing* where handles (,i.e. an edge with its source and target nodes) are glued such that they overlap completely in their interface graphs. Based on handle star gluing locally catenative formulas defined for L-systems can be generalized to *locally star gluing formulas* presented in [Ehr79].

Example 2.2.4 (Covering approach to parallel replacement) *In figure 2.9 a parallel replacement of mother graph G is shown. Three productions $p1$, $p2$ and $p3$ are applied to G in parallel. The occurrences of productions $p1$ and $p2$ overlap in the grey nodes with the edge in between. The occurrences of productions $p2$ and $p3$ overlap in the black nodes with the edge in between. These overlapping parts are preserved during the replacement. The occurrences of productions $p1$ and $p3$ do not overlap. Their common interface graph is empty and therefore, not drawn. The daughter graphs are glued again using the same interface graphs.*

A *parallel graph grammar* consists of a label alphabet, a start graph and a finite set of productions.

Different properties of coverings are stated in [EK76b], for example, the uniqueness of star gluing is shown.

2.2.4 Amalgamation of Graph Transformation

In their work on *amalgamation of graph transformation* Böhm, Fonio and Habel in [BFH87] do not consider some kind of graph L-system but model a *synchronization* mechanism between parallel actions. Unlike all approaches above they allow

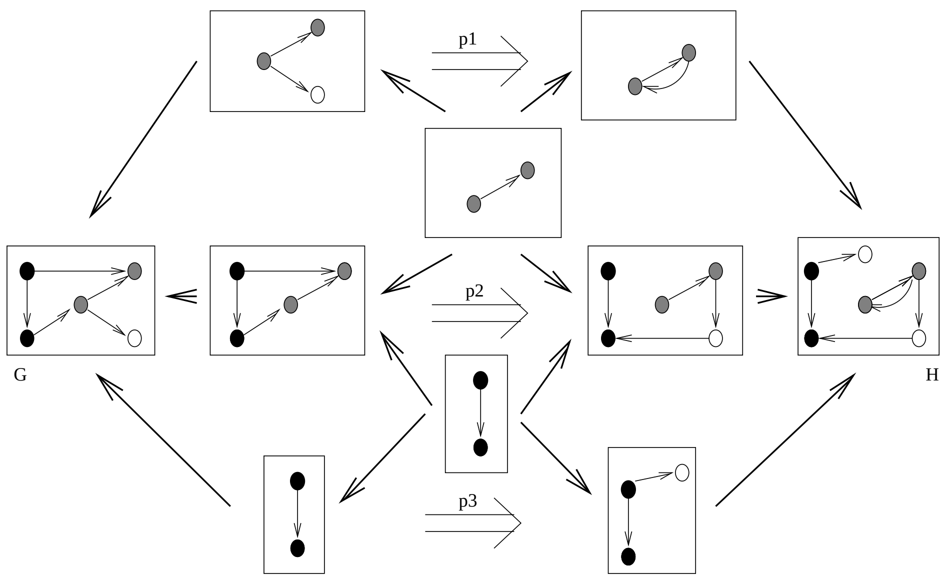

Figure 2.9: Covering approach to parallel replacement

dependent transformations to be executed in parallel. This is possible if some subtransformation can be determined. Applying two productions in parallel they have to be related by some common *subproduction*. If a *subproduction embedding* of some production *sub* into two productions $p1$ and $p2$ can be given they can be amalgamated to a new production, the *amalgamated production*, where those nodes and edges of the left and the right-hand side which are related by a node or an edge of the subproduction are identified and, therefore, occur only once in the amalgamated production. Roughly speaking, the concept of amalgamation of productions can be described as union where the subproduction describes the common part.

The amalgamation models a kind of synchronization where actions are allowed to be performed in parallel if they do the same on common parts. For example, they are allowed to delete the same item. They are not allowed to execute different actions on common parts, i.e. a node must not be deleted as well as preserved.

Independent graph transformations where no subtransformation is needed to synchronize them can be regarded as a special case where the subtransformation consists of the application of an empty production.

Example 2.2.5 (Synchronized actions) *In figure 2.10 we show a subproduction sub which describes the replacement of a grey node by a black one related to a preserved grey node by an edge. This action is reflected in its superproductions* $p1$ *and* $p2$*. As additional actions a new node is created in* $p1$ *and* $p2$ *describes additionally the reallocation of a node by deleting its adjacent edge and reestabilishing a new edge pointing to a new target. Production* sync *shows the parallel actions described by* $p1$ *and* $p2$ *synchronized at the subaction modeled by* sub*. Applying* sync *at graph* G *the*

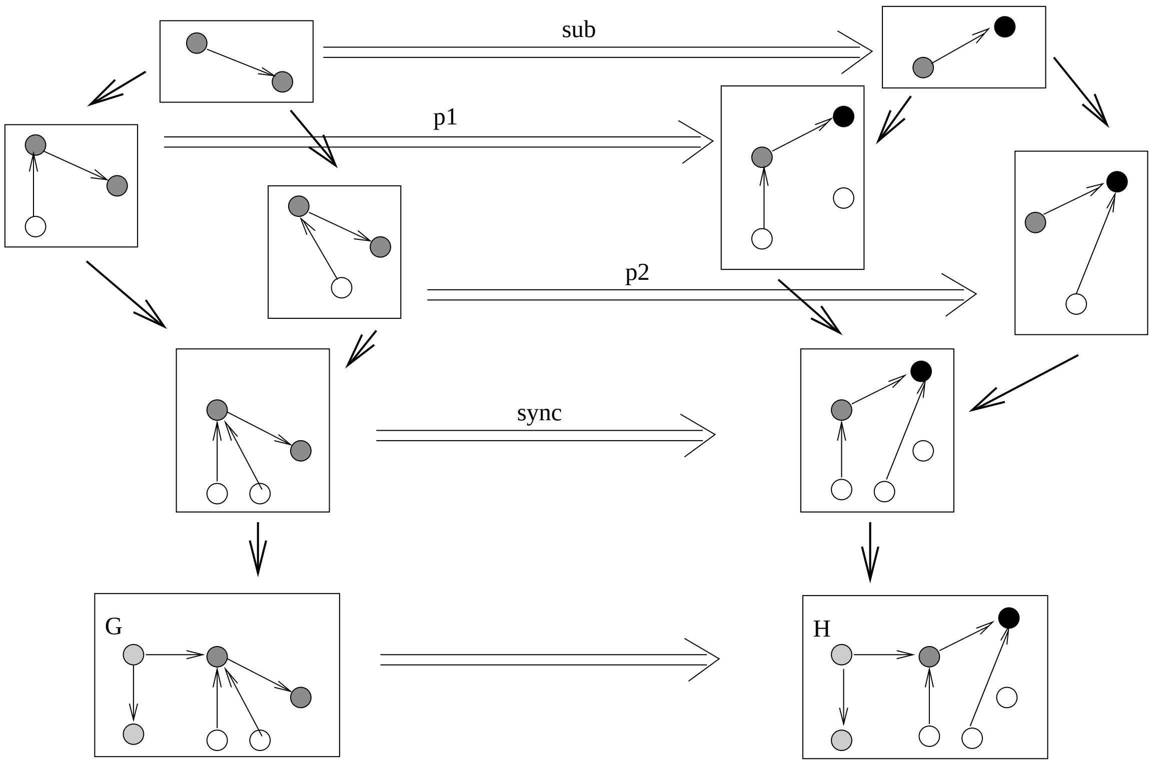

Figure 2.10: Synchronized actions

right grey node is deleted. Considering the resulting graph H only one black node is created.

Amalgamated graph transformations are equivalent with transformations where first the subproduction is applied and then the *remainders* of the two parallel productions are applied in either order. This relation is stated in the *amalgamation theorem*.

In [Löw93] Löwe presents amalgamation of graph transformation in the single-pushout approach. The concept of subproduction is slightly more general as in the double-pushout approach and allows e.g. a node to be deleted by the subproduction but preserved by its superproduction.

In both approaches the amalgamation is restricted to two productions.

2.2.5 Comparison

In this subsection we compare the approaches presented above covering different concepts which will be combined in parallel graph transformation as presented in chapter 3.

Except of amalgamation all approaches to parallel replacement on graphs above are motivated by L-systems and their applications. The approaches differ in the kind of elementary productions allowed. These are node replacement, hyperedge replacement and general graph replacement productions. The first two kinds of replacement belong to the *context-free* graph replacement whereas general replacement is not context-free. For node replacement overlappings in the mother graph are not allowed and for hyperedge replacement they are allowed on nodes only. General replacement offers the possibility to overlap also on entire subgraphs.

The way of parallel replacement follows the way of L-systems as already discussed in the introduction of this section. All approaches except of amalgamated graph transformation use this kind of replacement and offer nearly *unbounded parallelism*. Several restrictions are formulated but none of the approaches uses these restrictions to define different kinds of parallel replacement.

Synchronization of parallel transformations is more or less restricted. Connection rules and coverings mainly guarantee that the connections between the daughter graphs are inherited from the mother graph. The rendezvous mechanism and the amalgamation concept allow synchronization by performing common subactions.

All approaches except of parallel graph grammars and amalgamated graph transformations use the concept of tables to distinguish between different parallel operations.

It is the aim of introducing *parallel graph transformation* (in chapters 3 and 4) to combine most of the different concepts presented here and moreover, to offer flexible and powerful synchronization possibilities for parallel actions as well as restriction possibilities for unbounded parallel execution of actions to obtain a solid basis for parallel system specification.

2.3 Modeling Distributed Systems by Graph Transformation: State of the Art

Describing *distributed systems* several aspects have to be taken into account. Concerning the system's parallelism we have to ask the following questions: What is the topological structure of the distributed system? This can be clearly described by a graphical representation. Using graph transformation to model this aspect leads to some kind of dynamic network structure. The next aspect is what are the actions that should be performed in the local systems and, thus, in parallel? These actions are modeled very differently in the following approaches. If this level is included in the specification the actions are described by graph transformation as done in nondistributed systems. Another common way is to code them into attributes of nodes or to describe them by other specification techniques. The question of what to put into the graph and what into attributes is often decided along the paradigm: Specify all parallel aspects graphically and describe sequential executions textually.

In the following, we distinguish the graph transformation approaches whether graph transformation is used to describe dynamic network structures or local actions.

2.3.1 Modeling Dynamic Network Structures by Graph Transformation

A Model for Distributed Systems Based on Graph Rewriting In [DM87] Degano and Montanari introduce a model where a graph shows topological and temporal aspects of a distributed system through the relations adjacency and causality. It describes a net of parallel processes communicating through ports and the computational history of the system consisting of a partial ordering of events. Infinite graphs model systems that run forever, e.g. infinite computations.

Stand-alone evolution of processes is specified by context-free productions. If several processes evolve in parallel a production is derived by synchronization of productions. The synchronization was the motivation to develop the amalgamation concept in subsection 2.2.4.

For this model of distributed systems several properties of computations are discussed such as termination, evolution of conflicting processes and fairness.

Modeling the operational semantics of a distributed language In [BS94] Barthelmann and Schied present a new small programming language, called *DHOP*, for distributed systems. Its operational semantics is described by attributed graph transformation in the double-pushout approach.

DHOP distinguishes processes and channels which are used for message passing between processes. As usual states of process networks are modeled by graphs. The process evolution is captured by graph transformation. The node attributes contain program fragments which are executed in following transformation steps.

The execution model may support distributed evolution. A graph production handles exactly one process. Independent parallel executions can be described by just applying several productions in parallel shown in [Sch94]. Communication between two processes can be modeled then by amalgamating two graph productions at channel nodes as common gluing points.

Modeling Actor Systems In [JR91] Janssens and Rozenberg use graph transformation to model *actor systems*. A configuration of an actor system which models some state is described by a *configuration graph*. The behavior of a system is consequently modeled by transformations of graphs.

An *actor grammar* consists of a set of configuration graphs and a set of *primitive event transformations* which model the processing of a message by an actor. *Primitive structured transformations* describing the handling of one event cannot be generated from others by using one of the composition operators: concurrent composition, sequential composition and augmentation which is a kind of embedding of a transformation into a bigger context.

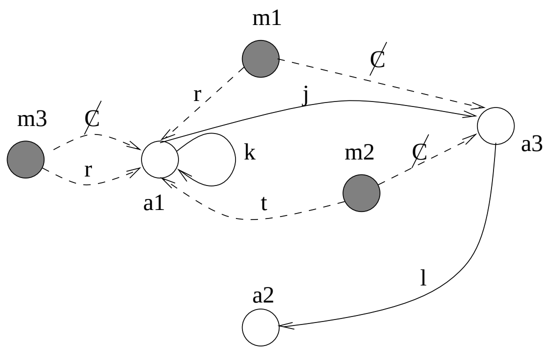

Figure 2.11: A configuration graph modeling a state of an actor system

Example 2.3.1 (Configuration graph) *In figure 2.11 a configuration graph is given which consists of three actors (white nodes) and three messages (black nodes). The edges correspond to acquaintance relations between actors and messages (dashed arrows) and actors themselves (solid arrows). Each actor and each message is labeled which can be used to model the local states of actors and the values of messages. The edge labels indicate acquaintance names. The special label ¢ indicates the receiver of a message.*

The derivation process in an *actor grammar* can be represented by a *computation graph*. Similar to the model of Degano and Montanari in [DM87] this graph shows the computation history for actor grammars. Computation graphs resemble occurrence nets, a special Petri net class.

For a given actor grammar the class of all structured transformations and the class of all computations basically described by computation graphs correspond to each other.

Actor graph grammars are generalized to *ESM systems* by Janssens [Jan93] where a transformation step corresponds to a (single) pushout construction as in the single-pushout approach [Löw93]. Transformation processes in ESM-systems are respresented by computation structures and communication corresponds to gluing of computation structures.

Correspondingly, Korff presents *algebraic actor graph grammars* in [Kor93, Kor94]. Very similarly, computational states are modeled by graphs and events by graph transformations. Algebraic actor graph grammars are related formally to the actor paradigm of Clinger [Cli81].

The ideas of algebraic actor graph grammars are generalized to *generalized graph structure grammars* in [Kor96] to model *concurrent object-based systems*. They are equipped with a fair concurrency semantics developed in [Kor96].

Modeling actor systems by graph transformation concentrates on modeling the communication topology and its computational changes. Communication is modeled by creating a message node in one graph transformation and processing (reading and

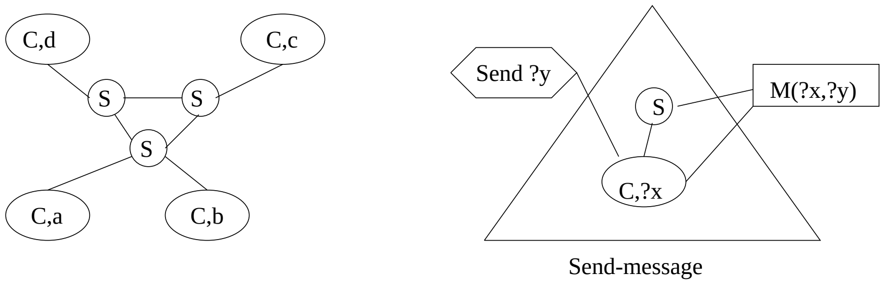

Figure 2.12: A client/server system graph with send-message production

deleting) it in another transformation.

Specifying distributed systems by Δ-grammars In [KLG91] Kaplan, Loyall and Goering use Δ-*grammars* for modeling distributed systems. They concentrate on the *network level* where, for example, clients and servers interact by propagating messages and establishing conncetions. As usual configurations are described by graphs and the behaviour of a system by productions.

For a *modular system design* a large specification can be divided into smaller parts, called *platforms*, realized by different sets of productions. The interface between such platforms is provided by *triggers*. An *input trigger* occurs as part of the preconditon of a production. *Output triggers* can occur in the right-hand side of a production calling a production of another platform. Internal states are basically described by Δ-grammars consisting of productions without triggers.

Example 2.3.2 (Clients and Server) *Figure 2.12 shows some system configuration described by an attributed graph. If a client of one server, for example (C,a), wants to send a message to some remote client, say (C,d), this is modeled by creating a message node attributed by the sender and receiver. This action is described by the Δ-production on the right of figure 2.12. The Send node on the left of the Δ functions as an input trigger which is inserted into a configuration graph from outside world. If such a trigger occurs in a graph and belongs to some client connected to some server, a message is created by the client and propagated to its server. The variables $?x$ and $?y$ occuring in the production are needed to specify the sender and the receiver of the message. They are instantiated when applying the production. In our example $?x = a$ and $?y = d$.*

Δ-grammars have also been applied to specify actor systems similarly to the actor models above. Moreover, they describe additional features of actor systems such as the actor's internal concurrency concerning processing of messages.

2.3.2 Modeling Local Actions by Graph Transformation

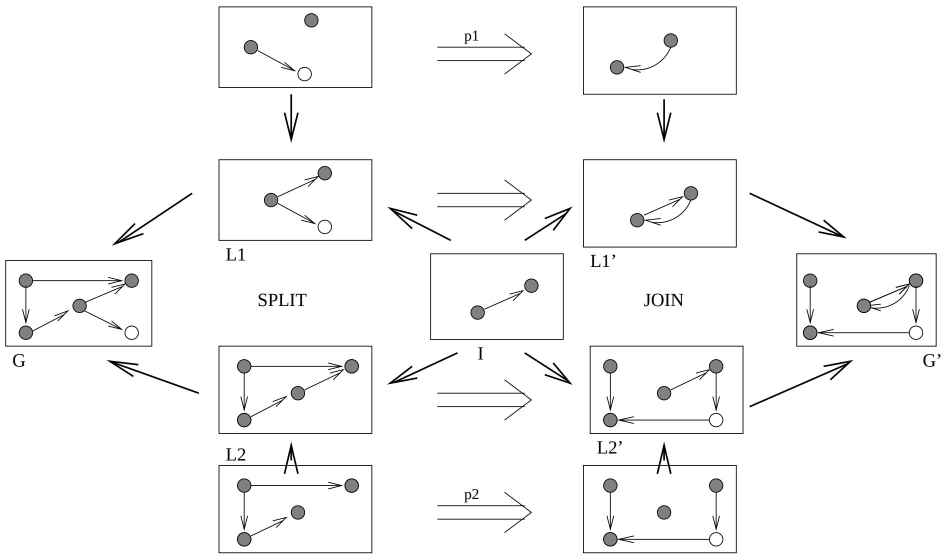

Figure 2.13: A distributed transformation

Towards Distributed Graph Transformation In [BEHL87] and [EBHL88] Ehrig et.al. present two main distribution operations on graph states: *split* and *join*. Split allows to split a *global state* graph into two *local state* graphs together with a corresponding *interface graph*. Join as the inverse operation of split performs the joining of two local state graphs along their interface graph such that a global state graph is obtained.

Distributed actions are modeled by graph transformations on local state graphs, i.e. *local graph transformations*. The interface graphs have to be preserved during transformation. The equivalence to the parallel application of the corresponding productions at their adjoint global state graph is stated in the *distributed parallelism theorem*.

Example 2.3.3 (A distributed transformation) *In figure 2.13 a distributed transformation is shown consisting of a splitting of global graph G into local graphs $L1$ and $L2$ sharing interface graph I. There is a transformation step on $L1$ leading to $L1'$ and on $L2$ leading to $L2'$. Local graph $L1'$ and $L2'$ are joined along graph I to the resulting global graph G' afterwards. Using the embeddings of the local graphs into global graph G the productions may also be applied directly at the global graph leading to the same result. Note that the productions may also be differently applied at the global graph G leading to different results, i.e. splitting into local graphs restricts the application area.*

The paper also contains a proposal for the definition of *distributed graph grammars* which consist of *global* and *local graph grammars* in the sense that the global and the two local graph grammars should correspond with each other. All possible interfaces are given by a set. Others are not allowed during all possible transformations. Additional productions in the global graph grammar have to be *interface consistent* which means that each global state graph that is constructed can be split.

In [EL93] Ehrig and Löwe introduce distributed graph transformations in the single-pushout approach. They distinguish *total* and *partial splittings*. A splitting is partial if at least one local graph is not completely contained in the global graph. As far as the interface embeddings remain total local actions seem to have local effects only. This property can be used as synchronization mechanism leading to different expressive power of distributed transformations. This idea is further developed in [Tae96b] defining synchronous and asynchronous graph transformation.

Furthermore, in [EL93] *dynamic interfaces* are introduced. The interface graph being preserved during local transformations is replaced by an interface transformation relating the interface of the given graph splitting to that of the resulting graph splitting. Thus, the interface is allowed to be updated consistently. Such dynamic distributed transformations are closely related to amalgamated transformations assuming that the productions are applied at *compatible matches*.

2.3.3 Transforming network and local graph structures

Categorical graph grammars for modeling distributed systems In [Sch93] Schneider introduces *categorical graph grammars* which extend the concept of labels of nodes and edges. In this approach labels may be arbitrary objects, e.g. strings, terms or graphs again. During a transformation step labels are allowed to be changed by special label transformations. Thus, a transformation consists of a usual graph transformation and of a set of label transformation on each label of a node or edge transformed. Transformations on different labels are not related somehow.

Example 2.3.4 (Generalized Petri nets) *In figure 2.14 a graph transformation is presented showing the firing of a transition in a Petri net where the places do not contain discrete tokens but graphs modeling some token structures. The graph inside the leftmost place models a simple FIFO-queue where* h *points to the head and* t *to the tail of the queue consisting of elements* a *and* c*. The transformation describes the consumption of element a such that the queue has to be updated. The second input place of the modeled transition contains an* m *if the next element shall be consumed.*

Parallel independent transformations may be executed in either order or in parallel, i.e. in a concurrent way. This graph transformation approach is well suited to model dynamic systems of processes which can grow or shrink.

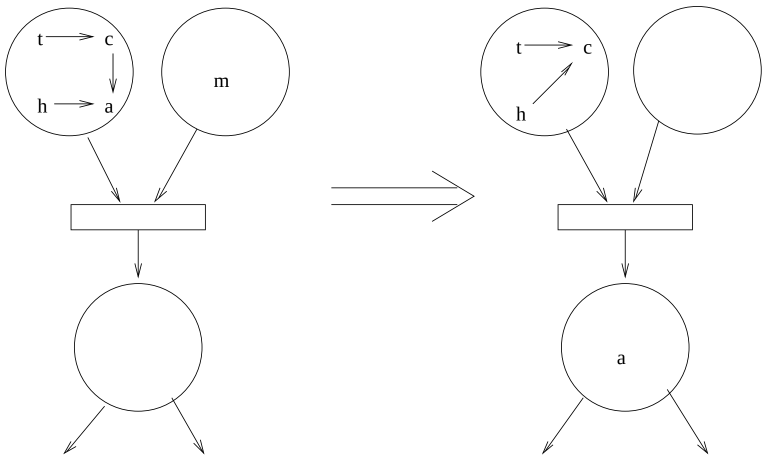

Figure 2.14: A generalized Petri net transformation

2.3.4 Comparison

In this subsection we compare the different ways how distributed systems have been described by graph transformation to get an overview of what is state of the art of modeling distributed systems by graph transformation. A general discussion how graph transformation can be used as specification technique for communication-based systems is given in section 7.4. The comparison mainly follows the requirements for modeling support presented in chapter 1. The different distributed graph transformation models use graph transformation for the specification of different system aspects. This leads to different views of the system which will be combined in distributed graph transformation presented in section 5 and 6.

Parallelism All the graph transformation models presented in this section model some kind of *large-grain parallelism*. Processes, actors, messages, clients and servers, local and label transformations are units of parallelism. These units may be regarded as local systems running in parallel.

The categorical and the algebraic approaches also model *fine-grain parallelism* and use graph transformation for the description of object structures and their changes. Parallel actions are not described by one production but several ones.

Communication All the approaches in subsection 2.3.1 model *message passing* as communication and synchronization paradigm. The message passing is modeled by common nodes and edges which are shared between different sequentially dependent production applications. Modeling the sending of a message to one unique receiver, additional edges, node labels or attributes are used to determine the receiver, for example, in the actor or the Δ-approach.

The algebraic and the categorical approach model data sharing. Nevertheless, there are differences: Whereas the algebraic approach uses interface graphs which have references in local graphs, i.e. they occur in the local graphs, the sharing in the categorical approach is based on unique labels of local nodes and edges.

Distribution and dynamic network structures In graph transformation models presented in subsection 2.3.1 internal states of actors, clients, etc. can be regarded as hidden, because they are not modeled. Such a full data abstraction might be adequate to model distributed system states in some sense. A local view is not possible. The Δ-approach introduces basic ideas for modular structuring of graph specifications using platforms.

The categorical and the algebraic approaches model local graphs separated from each others. Regarding the graphs without label graphs in the categorical approach allows a global view on a systems network state. Network graphs are also presented in the algebraic approach, but they are not explicitly mentioned there. The network is also reflected by introducing distributed graph grammars which allow modular structuring of a global system into subsystems running in parallel.

The approaches in subsection 2.3.1 have the full description power of their graph transformation approaches at their disposal to model dynamic networks. But modeling of actor systems, for example, restricts graph transformation for the description of event handling.

In the algebraic approach the network structures are rather restricted and split and join are the only operations on network structures. The categorical approach allows arbitrary graph structures but a restricted version of graph transformation not allowing split and join.

The aim of developing *distributed graph transformation* as presented in chapters 5 and 6 is to combine modeling by graph transformation on the network as well as on the local level. Such an approach will allow flexible description of dynamic networks by graph transformation, graphical representation of the local system states changable by graph transformation and synchronization via subgraphs. Textual attributes are not needed to describe communication. Thus, all important features of distributed systems are modeled within the graph transformation paradigm which seems to be very suitable for this task considering, for example, the case study of a repository for distributed software engineering in section 7.2.

Chapter 3

The Concepts of Parallel Graph Transformation

This chapter presents the main features of parallel graph transformation in an informal way. They are introduced in sections 3.1 and 3.2. Their formal description follows in chapter 4. Their expressive power is shown by several examples in section 3.3 and further case studies in sections 7.1 and 7.3. In section 3.4 the parallel replacement concepts presented here are compared with those parallel graph replacement approaches discussed in 2.2. Furthermore, the concepts are related to the requirements for communication-based systems in section 1.2. In chapter 7 parallel graph transformation is applied to larger examples showing their capabilities and restrictions for parallel specifications.

Parallel graph transformation models parallel state transition where a state is described by a graph which can be changed by several actions executed in parallel. Since the graph transformation is rule-based without restrictive execution prescription it offers the possibility for massively parallel execution. We begin with the case where all transformations are parallel independent of each other. More expressive power can be gained if also dependent but synchronized transformations are allowed to be executed in parallel. Keeping the rule-based character the synchronization is described by common subproductions.

For more convenience, further notions are proposed for parallel graph transformation which allow the efficient description of complex parallel operations. A well-known representative are *forall*-operations, i.e. actions which have to be done for all objects in a given set. In section 3.3 a first step towards language constructs for these complex parallel operations on graphs is described.

Readers of this chapter can get an operational understanding of the concepts for parallel graph replacement sufficient for their application although they might skip the formal description in chapter 4.

3.1 Synchronization of Parallel Actions

Parallel graph transformation models parallel actions on non-distributed states. A graph is a nice means to describe objects and their relations. Objects can be represented by nodes and their interrelations by edges. All parallel transformations operate on one graph without additional structure such as encapsulations. Object sharing with hiding concepts is presented in chapter 5 where distributed graphs are introduced.

In the graph grammar field actions are usually described by *graph productions* and modeled by *graph transformation*. In this thesis we use the double-pushout approach to graph transformation which characterizes some kind of cautious rewriting of graphs. This means that for a production and its matching part in some graph the *gluing condition* (see section 2.1.2) has to be satisfied. If the gluing condition is satisfied for a production and a *match* of its left hand side in the current graph a new graph is derived by deleting this occurrence and adding the right hand side of the production. (More details to this kind of graph transformation can be found in section 2.1 or, e.g. in [Ehr79, CMR$^+$96b].)

The basic actions on object structures are deletion, preservation, gluing and creation of objects and relations. All basic actions which are modeled within one production form some kind of transaction where either all actions are performed or none. If the production is applicable the transaction succeeds, otherwise it fails. Since we consider abstract object structures and do not model data attributes here, a reading action just means preservation of an object.

3.1.1 Independent Actions

The simplest type of parallel actions is that of *independent actions*. If they operate on different objects they can clearly be executed in parallel. If they overlap just in reading actions on common objects the situation does not change essentially. This kind of safe parallel execution is performed in various parallel specification and programming languages. It is also reflected by graph transformation where a *parallel production* which is a disjoint union of productions is applied to a graph. In this case the overlapping part, i.e. those objects which occur in the match of more than one production, is handled implicitly by the match of the parallel production.

The parallel execution of non-independent actions cannot be modeled since the gluing condition does not allow the identification of nodes or edges which are not preserved, i.e. one and the same object cannot be preserved by one action and deleted by the other.

Example 3.1.1 (Independent actions) *In figure 3.1 a parallel action is modeled by the application of a parallel production. The parallel production consists of two productions indicated by the upper two dashed ellipses forming one production and the lower two ones forming the second production. The upper production deletes the*

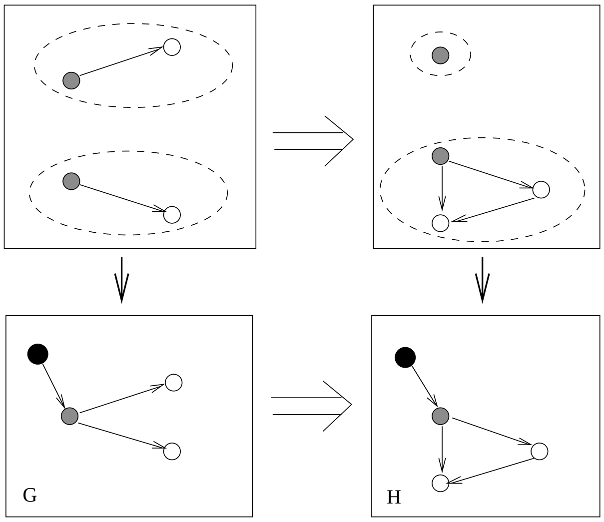

Figure 3.1: Independent actions

white object node and its adjacent relation edge. The lower production creates a new white object node with two adjacent relation edges. In both cases the grey object node is preserved.

Both grey object nodes of the production are mapped to the grey object node in graph G. Also their matches which are put together to one match (of the parallel production) overlap in the grey object node. They are independent because it is preserved.

The upper production is applied to the upper white object node and the lower production to the lower white object node in graph G. The context which consists of the black object node and its adjacent relation edge is left unchanged. The resulting graph is H.

Since the application of a parallel production can model the parallel execution of independent actions only it is equivalent to the application of the original productions in either order. This correlation is stated formally in the *parallelism theorem* 4.1.19.

3.1.2 Synchronized Actions

If actions are not independent of each others they may still be applied in parallel, if they can be synchronized by subactions. If, for example, two actions contain the deletion or creation of the same node this operation can be encapsulated in an extra action which is a common subaction of the original ones. Since actions are described by the application of graph productions a common subaction is modeled by the application of a *subproduction* of both.

Concept 3.1.2 *The application of productions synchronized by subproductions is performed by gluing the original productions at their subproductions which leads to the corresponding* amalgamated production. *The application of this production, called* amalgamated graph transformation.

Amalgamated transformation models the synchronized effect of all original actions. An example of an amalgamated graph transformation is given within subsection 2.2.4.

If a subproduction does not describe the whole subaction the part not described by the subproduction has to deal with preserved items only and is handled implicitly as for independent actions. In this case the actions are not *fully synchronized* by the subaction. If a subproduction models just the preservation of nodes and edges the corresponding actions are again independent but their overlapping part is explicitly described. Constructing the corresponding parallel production its application is equivalent with the application of the amalgamated production of the same original productions at empty subproductions.

Each application of an amalgamated production at a preserving subproduction (as described above) leads to the application of the corresponding parallel production using the same original productions. In this case the explicitly given overlappings are handled implicitly. This correlation is formally stated in the *parallel replacement theorem I* in section 4.4.

3.2 Massively Parallel Actions

Graph transformation, especially in the double-pushout approach, supports the idea of locality in changing the graph structure. This property is a very good basis for the massively parallel execution of actions. Since actions are described by the application of productions this means the *massively parallel application* of productions to a given graph.

The architecture which describes how productions are applied in parallel is naturally represented as graph again. The nodes correspond to productions and the edges to subproduction relations (as described above). As a very flexible structure which offers all synchronization possibilities we use the so-called star graphs. Some small members of the family of star graphs are shown in figure 3.2.

Star graphs contain two types of nodes: *elementary nodes* (white circles) and *interface nodes* (black circles). If we consider an interface node with its two adjacent edges as one connection edge star graphs become complete graphs. One may argue that this architecture is expensive. In this case some connections have to be cut which means that the corresponding elementary productions are not additionally synchronized. But it is the only architecture which allows problem-oriented parallelism in each case.

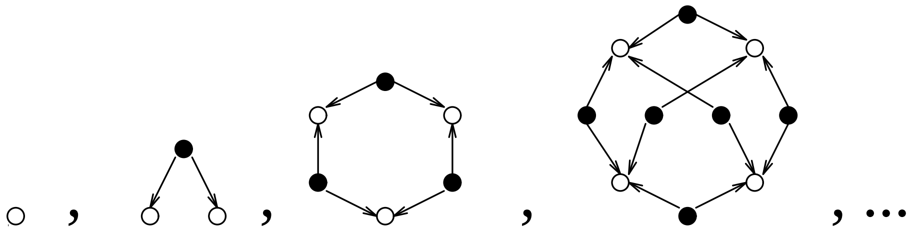

Figure 3.2: Star graphs

The massively parallel application of productions to a given graph is modeled by a star graph as follows. The nodes of the star graph are equipped with *elementary productions* such that each interface node is equipped with a subproduction of its adjacent elementary productions. To apply all the productions the union of their *elementary matches* has to cover the graph to which they are applied.

How is the gluing at such a complex structure like a star graph performed? All nodes (edges) of elementary parts which are related by nodes (edges) of interface parts are glued. Notice that this relation includes the transitive closure. In section 4.2 an example to this point is given.

Concept 3.2.1 *A* star-parallel graph transformation *consists of the gluing of all left-hand sides of elementary productions along the left-hand sides of their subproductions which yields the mother graph and the gluing of all elementary right-hand sides along the right-hand sides of the subproductions yielding the daughter graph.*

If the matches of elementary productions do not cover the given graph *completely* there is a context part which should not be transformed. The concept of context does not fit very well to that of massively parallel execution. The help in this situation is the idea to consider identical context productions. Taking a star graph of the next higher degree the additional elementary node is equipped with the identical context production. The productions at the adjacent interface nodes handle the interaction between the context and the corresponding elementary production. This construction is called *covering completion* of a *partial covering.*

Concept 3.2.2 *Star-parallel graph transformation with a preceding covering completion is called* contextual parallel graph transformation*.*

Given a partial covering its completion is not always possible. If a node is deleted by any of the elementary productions we have to make sure that there is not a context edge that dangles. If two nodes (edges) of the left-hand sides of the elementary productions are identified in the covered graph both have to be either preserved or there are nodes (edges) in the interface parts which relate them such that deletion is allowed. This pair of conditions is the pendant to the gluing condition in the sequential case and, therefore, it is called *parallel gluing condition.*

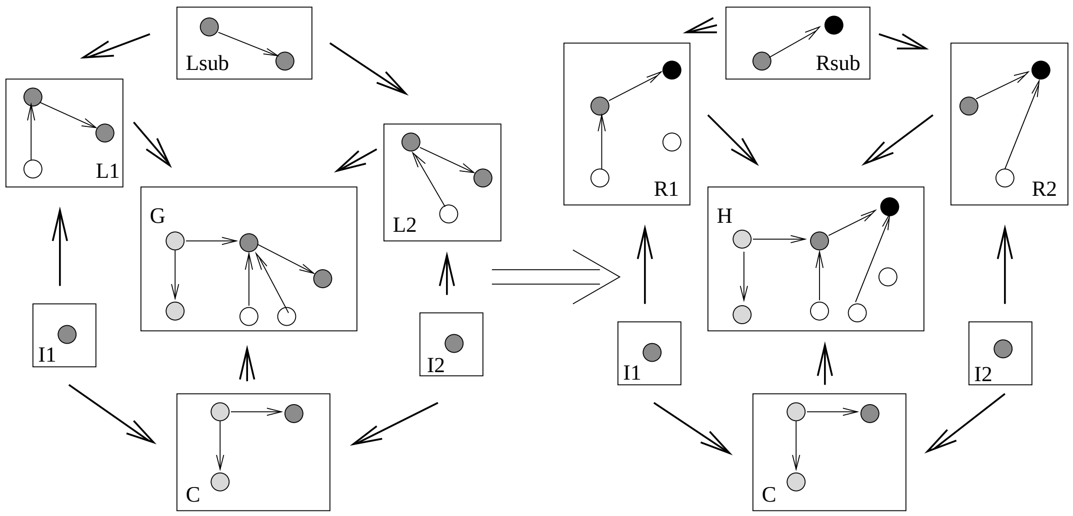

Figure 3.3: Contextual parallel actions

Example 3.2.3 (Contextual parallel actions) *An example of contextual parallel actions is shown in figure 3.3 where elementary productions* $p1$ *and* $p2$ *and subproduction* sub *of example 2.2.5 are given. Additionally we have the identical production* c *over context* C *and further identical interface productions* $i1$ *over graph* $I1$ *and* $i2$ *over* $I2$*. The left star graph in figure 3.3 covers graph* G *completely. The resulting graph* H *is constructed by gluing all right-hand sides of the elementary productions along those of the interface productions. Comparing example 2.2.5 the same graph* H *is constructed there.*

There is not that big difference between amalgamated graph transformation and contextual parallel graph transformation. The difference may be the degree of parallelism. Whereas the application of the amalgamated production models the synchronization of actions before execution contextual parallel actions are synchronized during execution. The effect is the same as stated formally in the *parallel replacement theorem II* in 4.4.7. Amalgamated productions have the advantage in their frequent reuse in different contexts. Contextual parallel actions fit into the given architecture of star graphs and allow massively parallel execution.

3.3 Production Schemes

Massive parallelism is mostly achieved by composing similar actions synchronized along some regularly structured network such as a star graph. Therefore, it is often very convenient to specify the complex parallel operations by using some language constructs.

3.3.1 Specifying Complex Parallel Operations

The most popular representative of this kind of operation are so-called *forall-operations*. They occur, for example, as map-operations in functional languages or as a parallel for-loop (compare section 1.2). The basic scheme is always the same: Do some action for a given set of objects. In the case of independent actions this concept is realized in various parallel programming languages and specification calculi.

For graph transformation we want to extend the forall-concept for synchronized actions. Given a set of elementary productions we have to decide how and how often the productions should be applied. According to the forall-idea a production should be applied at *all different* matches it has in a given graph. Such an operation may exhibit a massively parallel execution of actions which is not always wanted. To restrict the degree of parallelism two control features are introduced: the *interaction scheme* and the *covering construction*.

The interaction scheme restricts the *synchronization possibilities* of actions.

Concept 3.3.1 *An* interaction scheme *consists of a set of* elementary productions*, a set of* subproductions *and a set of their embedding into the given elementary productions.*

The covering construction restricts the *matching possibilities* for production of the interaction scheme. What do we mean if we want to consider *all* matches? There is much unaccuracy with the notion "all". "All" matches can mean as many as there can be found, infinitely many, or all which are different. One possibility is to require that all occuring images in the given graph have to be different. Or do we allow images which are equal but covered by different productions? For this kind of covering the so-called *different-match-coverings* are defined that do not allow to contain identical covered parts for the same elementary productions.

The algorithmic way of constructing *all-match-coverings* for a given interaction scheme $\mathcal{IS}$ and a graph G is the following:

Look first for all matches of all elementary productions of $\mathcal{IS}$ in G. For each two distinct matches of elementary productions p_1 and p_2 find then a common subproduction s in $\mathcal{IS}$ with subproduction embeddings into p_1 and p_2 such that the left hand sides of p_1 and p_2 overlap in the mappings of the left-hand side of s in G which has to be covered. The covering of G consists of as many copies of all elementary productions as different matches have been found together with their matches in G. Furthermore, for each two copies of them a copy of the adequate subproduction is joined. All elementary productions with matches in G that do not have common subproductions do not belong to the covering of G.

If we are not interested in covering the whole graph but only the surroundings of a special part the all-match-coverings can be further restricted such that there is only one subproduction with its match. All matches of the elementary productions have to overlap then in this special match. This covering construction is called *local-all* (, shortly *locall*). The construction is presented in detail in [Bet92].

To handle interaction schemes and covering constructions independently of each other they are combined in the so-called amalgamation scheme.

Concept 3.3.2 *An* amalgamation scheme *consists of an interaction scheme and a covering construction. The covering construction is described by all properties which the coverings should have. The properties are listed in table 3.1.*

basic	only one elementary match
comp	the given graph is completely covered
sync	two elementary matches overlap completely in their common submatch
fast	each two elementary actions are independent of each other
diff	equal elementary productions match different parts
all	all different elementary matches
locdiff	all different elementary matches that overlap in one submatch
locall	all elementary matches overlap in the match of one subproduction
arb	without additional properties

Table 3.1: Special covering constructions

A *parallel system* can be described then by a parallel graph grammar.

Concept 3.3.3 *A* parallel graph grammar *consists of a start graph, a set of amalgamation schemes and a pair of node and edge label sets where subsets of them, the* terminal labels*, functioning as a termination condition.*

A graph has to be transformed as long as the whole graph is not terminally colored. This termination condition is especially useful if the parallel system should describe the parallel generation of objects.

3.3.2 Examples of Parallel Operations

After the discussion of the concepts for extended parallel graph replacement this section gives an introduction to their possible applications. The following rather small examples are mainly taken from the areas of fractal generation

Example 3.3.4 (Sierpinski triangles) The generation of Sierpinski triangles has already been described by parallel hyperedge replacement in example 2.2.2 where some examples of Sierpinski triangles have been shown. The following parallel graph grammar also generates the set of all Sierpinski triangles. Let

$$SIERP = (S_{SIERP}, \{A_{SIERP,STEP}, A_{SIERP,TERMINATE}\}, C_{SIERP}, T_{SIERP})$$

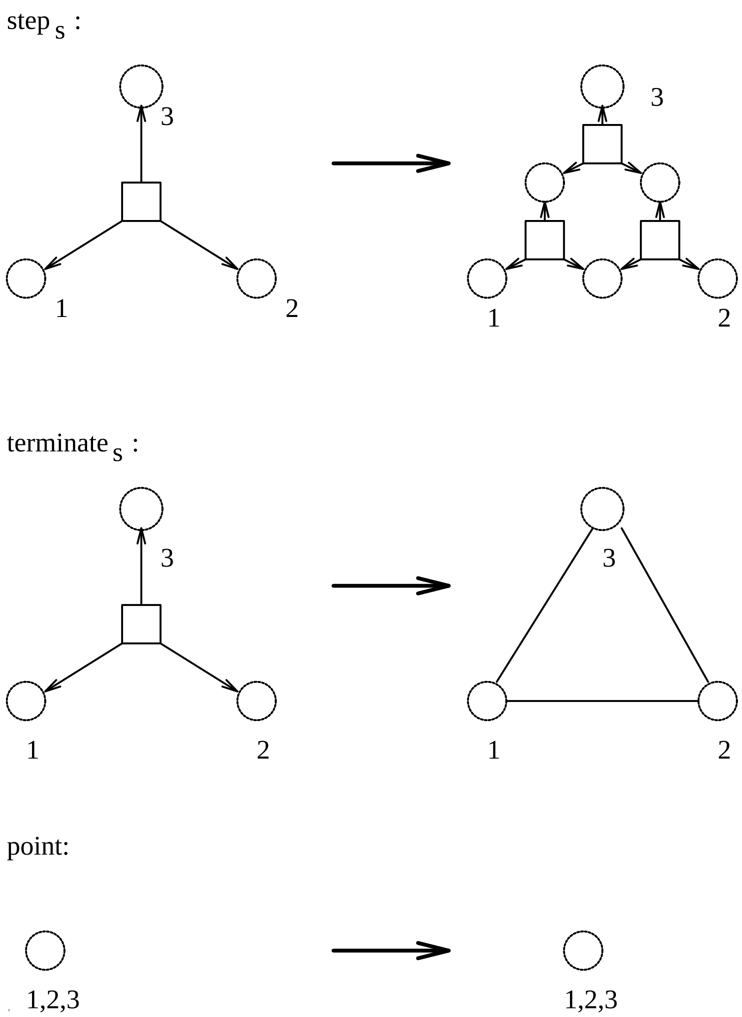

Figure 3.4: Example: Generation of Sierpinski triangles

be a parallel graph grammar[1] with the interaction schemes of $A_{SIERP,STEP} = (IS_{SIERP,STEP}, \{all\})$ and $A_{SIERP,TERMINATE} = (IS_{SIERP,TERMINATE}, \{all\})$ given in figure 3.5. Productions $step_s$, $terminate_s$ and *point* are given in figure 3.4. The different possibilities of embedding *point* into $step_s$ and $terminate_s$ are indicated by different numbers separated by commas. The start graph S_{SIERP} is isomorphic to one of the left-hand sides of $step_s$ or $terminate_s$. Labeling set $T_{SIERP} = \{\bigcirc, -\}$ of terminal labels is a subset of $C_{SIERP} = T_{SIERP} \cup \{\square\}$.

All coverings which can be constructed are complete, fast and fully synchronized. Using subproduction *point* the overlappings of matches in certain nodes are explicitly shown. Since these nodes are preserved we do not really need the synchronization

[1]The undirected edges in the following productions are considered as a short notation of two edges in opposite direction.

Figure 3.5: Interaction schemes of $A_{SIERP,STEP}$ and $A_{SIERP,TERMINATE}$

at this subproduction. But usually, fully synchronized coverings have a more clear understanding. The replacement mechanism in this example is very similar to that in example 2.2.2.

$SIERP$ generates Sierpinski triangles as depicted in figure 2.4.

Example 3.3.5 (Square grids) In this example we review the generation of square grids by parallel graph grammars already done with hyperedge replacement in example 2.2.3. The elementary productions are very similar but their interaction is described more clearly in a graphical way. Like in the previous example the productions are applied at all occurrences in the given working graph. Unlike the previous example the interaction schemes used are not constant any more. Let

$$GRID = (S_{GRID}, \{A_{GRID,STEP}, A_{GRID,TERMINATE}\}, C_{GRID}, T_{GRID})$$

be a parallel graph grammar with the interaction schemes of $A_{GRID,STEP} = (IS_{GRID,STEP}, \{sync, all\})$ and $A_{GRID,TERMINATE} = (IS_{GRID,TERMINATE}, \{sync, all\})$ given in figure 3.7 where $corner_s, edge_s$, $corner_t$, $edge_t$ and $glue$ are given in figure 3.6. Start graph S_{GRID} is also shown in figure 3.7. Labeling set $T_{GRID} = \{\bigcirc, -\}$ of terminal labels is a subset of $C_{GRID} = T_{SIERP} \cup \{\square, \blacksquare\}$.

$GRID$ generates square grids as depicted in figure 2.8. All coverings which can be constructed are fast.

Example 3.3.6 (Generation of Petri nets) Petri nets can be considered as a special kind of graphs which consist mainly of transitions and places described by nodes, and connections between transitions and places indicated by edges. Places are usually considered as some passive parts where objects modeled by tokens can be stored. Transitions are considered to be the active parts that can fire, i.e. they can take tokens from input places and put tokens to output places. Input and output places are connected with their transitions by edges going from a place to a transition and from a transition to a place, resp. It is not allowed to have more than one edge between the same place and transition in the same direction. In the following we consider condition/event nets where at most one token can lay on each place. The graph transformation description of Petri nets follows mainly that in [Kre81]. (For more details to Petri nets see section 1.3.)

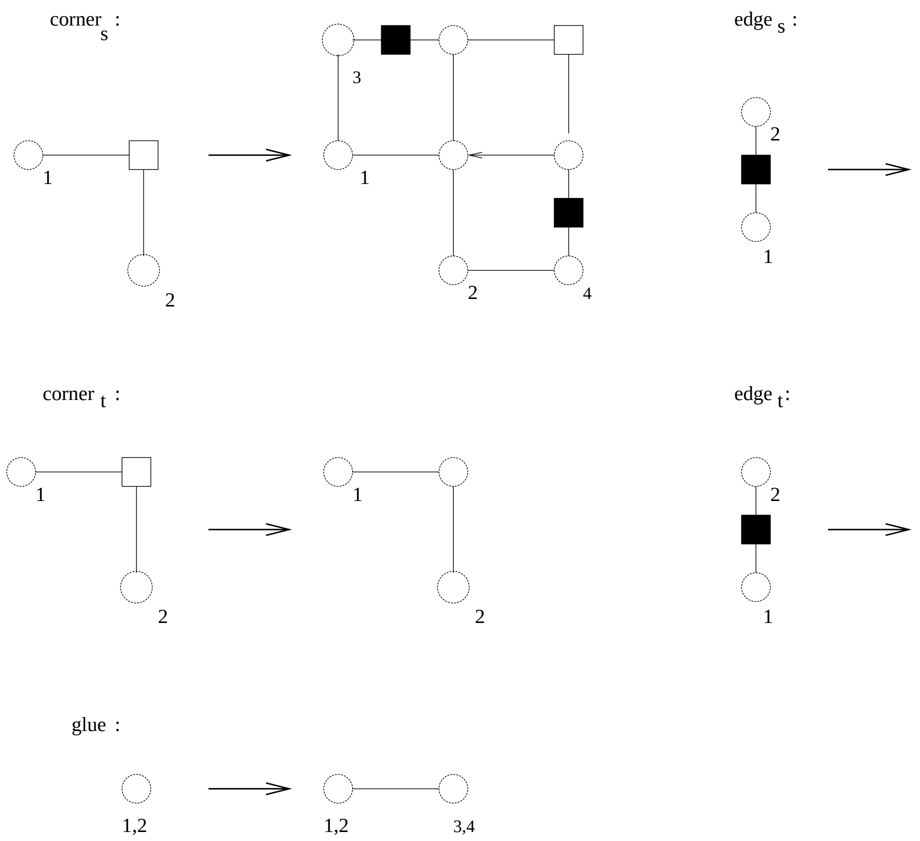

Figure 3.6: Example: Generation of square grids

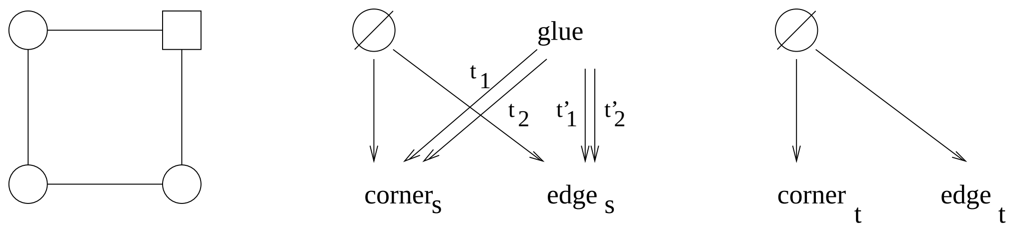

Figure 3.7: Start graph S_{GRID} and interaction schemes of $A_{GRID,STEP}$ and $A_{GRID,TERMINATE}$

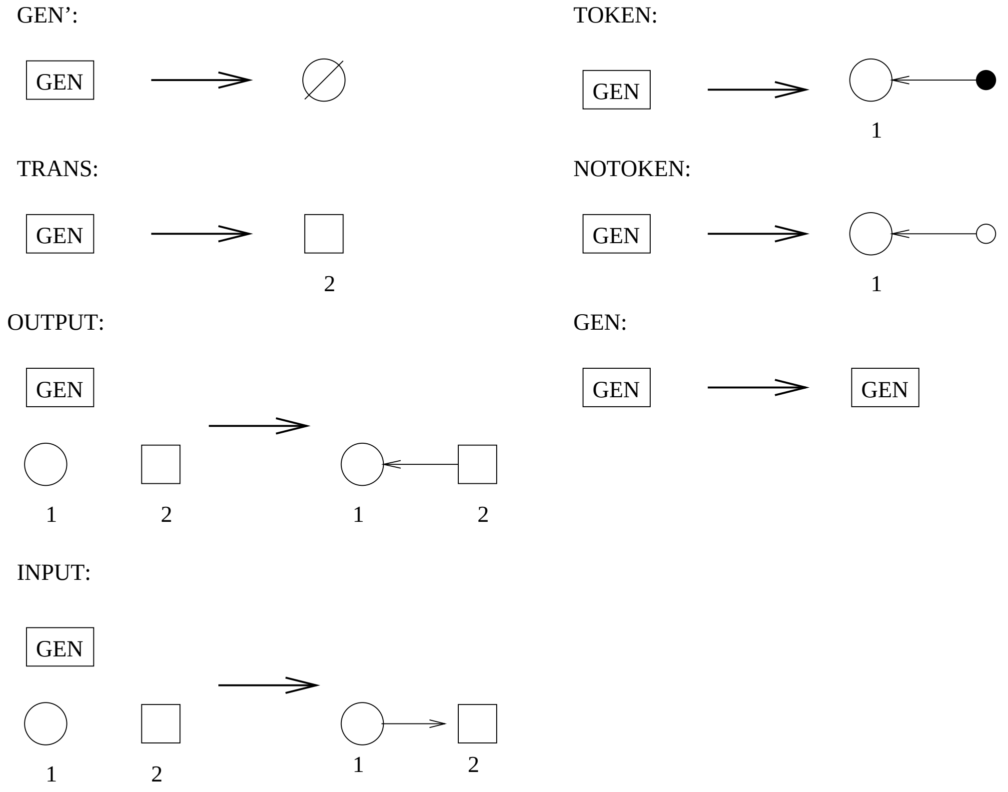

Figure 3.8: Example: Generation of condition/event-nets

In [CEL+94] Corradini et.al. has shown that graph grammars are able to express place/transition nets. Such a graph grammar contains a set of productions over discrete graphs with empty gluing graphs. Tokens are represented as nodes. The places and transitions are not graphically modeled. The relation between places and their tokens is expressed by node labels. The explicit firing order of transitions is transferred into an implicit causality order of production applications.

Usually places are indicated by big circles ($\bigcirc$), tokens are drawn by small black circles ($\bullet$) and transitions by big boxes ($\square$). Tokens are assigned to places by edges. If there is no token on a place a small white circle ($\circ$) is equipped to it. Let

$$PETRI_{GEN} = (\boxed{\text{GEN}}, \{A_{GEN}, A_{GEN'}\}, C_{PETRI}, T_{PETRI})$$

be a parallel graph grammar that generates condition/event nets without loops. The interaction schemes of $A_{GEN} = (IS_{GEN}, \{sync\})$ $A_{GEN'} = (IS_{GEN'}, \{sync\})$ are shown in figure 3.9. Productions GEN, GEN', $TOKEN$, $NOTOKEN$, $TRANS$, $INPUT$ and $OUTPUT$ are depicted in figure 3.8. Labeling set $T_{PETRI} = \{\circ, \bullet, \bigcirc, \square, \rightarrow\}$ is a subset of $C_{PETRI} = T_{PETRI} \cup \{\boxed{\text{GEN}}\}$.

The whole generation of a particular net is done in two steps. With $TRANS$, $TOKEN$ and $NOTOKEN$ arbitrary many transitions and places with or without

Figure 3.9: The interaction schemes of A_{GEN} and A_{GEN}

token can be created. This is done in the first step. In the second step the network between transitions and places is generated using productions $INPUT$ and $OUTPUT$. Several applications of $INPUT$ ($OUPUT$) have to fully synchronize in themselves. Thus, between each pair of place and transition at most one edge in each direction can be generated. The purpose of node $\boxed{\text{GEN}}$ in the start graph and elementary productions is to allow the application of $A_{GEN'}$ at most once. Otherwise graphs can be generated which do not have the properties of condition/event nets.

Example 3.3.7 (Firing of transitions in Petri nets) In this example we want to describe the firing of transitions by parallel graph grammars. As start graph P_0 an arbitrary condition/event net is allowed. The following amalgamation scheme $A_{SEQUENTIAL}$ describes sequential firing of arbitrary transitions.

$$PETRI_{FIRE} = (P_0, \{A_{SEQUENTIAL}\}, T_{PETRI}, C_{PETRI})$$

with the interaction scheme of $A_{SEQUENTIAL} = (IS_{SEQUENTIAL}, \{locall\})$ is shown in figure 3.14. Productions $TAKE$, PUT and $TRANS_s$ are depicted in figure 3.10. Label sets T_{PETRI} and C_{PETRI} are those of the previous example.

Usually a transition can fire if all input places have a token and all output places are free. Modeling the firing by the application of an amalgamated production the parallel gluing condition, especially the dangling condition, is satisfied if the transition can fire. Since the elementary productions delete the transition and reconstruct it again the dangling conditon is not satisfied if not all input and output places of the firing transition are covered. Not all coverings which can be constructed are fast. More precisely all coverings which are not basic are not fast. If there is more than one transition in the working net the coverings constructed are not complete either.

Given an example net as in figure 3.11 we need for example a production that describes the firing of a transition with one input and two output places to fire transition $t2$. Such a production can be generated by one copy of $TAKE$ and two copies of PUT which are amalgamated at three copies of $TRANS_s$. The result is given in figure 3.12.

All the nets $PETRI_{FIRE}$ generated of the starting net in figure 3.11 are shown in figure 3.13. Since the structure of Petri nets does not change by firing transitions only the tokens of each net are given oriented in the same way as they are in figure 3.11.

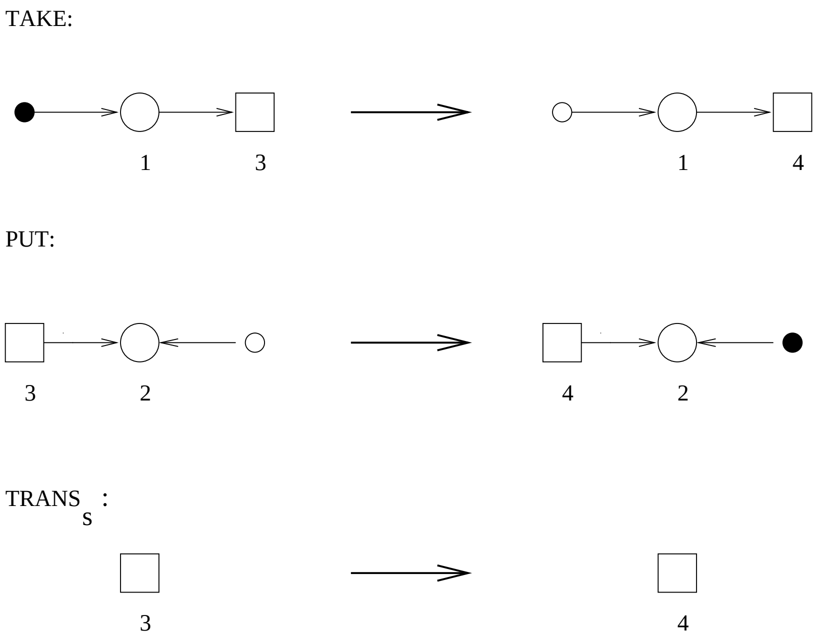

Figure 3.10: Example: Firing of transitions in condition/event-nets

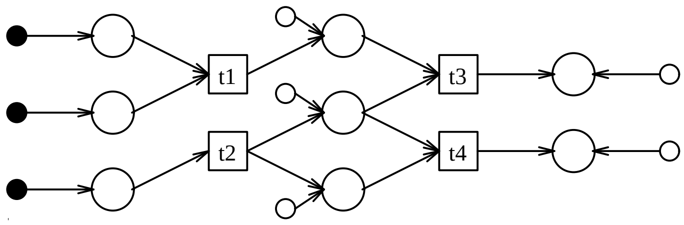

Figure 3.11: An example condition/event net P_0

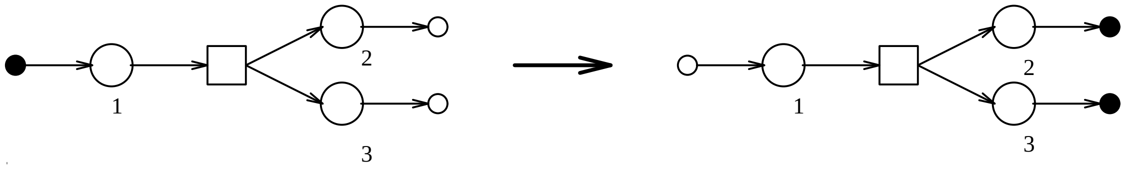

Figure 3.12: Amalgamated production for firing of transitions with one input and two output places

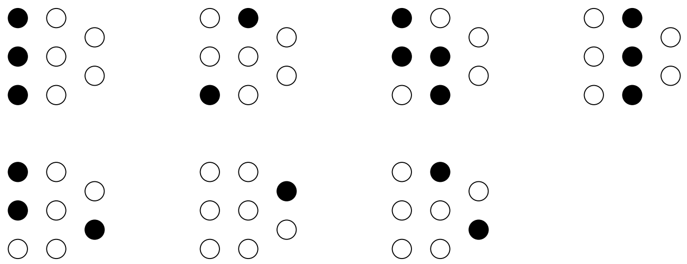

Figure 3.13: Possible markings of the net in figure 3.11

Figure 3.14: Interaction schemes of $A_{SEQUENTIAL}$ and $A_{PARALLEL}$

Moreover, parallel firing of transitions can be modeled without changing the productions. Only the amalgamation scheme has to be adapted.

$$PETRI'_{FIRE} = (P_0, \{A_{PARALLEL}\}, C_{PETRI}, T_{PETRI})$$

with the interaction scheme of $A_{PARALLEL} = (IS_{PARALLEL}, \{all, sync\})$ shown in figure 3.14. Label sets T_{PETRI} and C_{PETRI} are those of the previous example.

If we consider the example net in figure 3.11 both transitions $t1$ and $t2$ can fire in parallel. Additional parts of the net cannot be covered because the coverings have to be fully synchronized (no overlapping at places) and the dangling condition has to be satisfied (all intput and output places of a transition have to be covered). Conflicts between different transitions are forbidden because all coverings have to be fully synchronized. Thus, three copies of $TAKE$ and three copies of PUT are applied to the net in figure 3.11. All productions which are applied to the same transition have $TRANS_s$ as their common subproduction. The occurrences of all other copies do not overlap, thus, their common subproduction is the empty one.

Considering the net after firing $t1$ and $t2$ transitions $t3$ and $t4$ cannot fire in parallel because they would be in conflict. This would be the case because they have an input place in common. Thus, the occurrences of $TAKE$ would overlap not only in transitions but also in a place which is not covered by their subproduction $TRANS_s$. This is not allowed because the coverings should be fully synchronized.

Generation and firing of condition/event nets can be combined by the following parallel graph grammar:

$$PETRI = (\boxed{\text{GEN}}, \{A_{GEN}, A_{SEQUENTIAL}, A_{PARALLEL}\}, T_{PETRI})$$

$PETRI$ does not generate more graphs than $PETRI_{GEN}$ but can additionally simulate the token game of Petri nets.

Amalgamation schemes $A_{SEQUENTIAL}$ and $A_{PARALLEL}$ model the firing of any transition in any condition/event net. In comparison with these amalgamation schemes the graph transformation model of Petri nets in [CEL+94] relates Petri nets and graph grammars one-to-one, i.e. one graph grammar models the firing of one Petri net.

3.3.3 Embedding into Variable Context

If an object shall be deleted it always has to be cleared what to do with the adjacent context edges. Modeling an object deletion by the double-pushout approach to graph transformation all its adjacent relations to other objects have to be deleted first. Or they have to be mentioned in the corresponding production. But if the set of attached relations in the object structure is variable an infinite set of similar productions is needed to describe the object deletion (with all adjacent relations). Such a production scheme is not needed if the NLC, single-pushout and algorithmic approaches because dangling edges are automatically deleted in these approaches.

Moreover, it might be convenient to establish additional relations between a new object and already existing ones, for example, dependent on the type of a context object. This kind of embedding operation also depends on the concrete number of suitable context objects and, therefore, is not capable by one production. (This feature is not available in the single-pushout approach.)

But this problem can be solved using amalgamation schemes. The interaction scheme consists of the original production as the only subproduction. All elementary productions of the interaction scheme describe possible embedding transformations which may be deletion, copying, creation and many more actions on the matching part and its surrounding context. For each elementary production there is exactly one possible subproduction embedding.

For such an interaction scheme only local-all-matches-coverings are allowed. This covering construction fits very well because we want to look first for a match of the subproduction (modeling the original action) and then for all different elementary matches overlapping with the submatch (modeling embedding transformation).

Example 3.3.8 (Embedding of actions) The following example which is adapted from [ET92] shows a context dependent graph transformation where an action is embedded into a variable context.

A node with label x shall be replaced by a handle (which is an edge with a source and a target node) labeled by y and z, written (y,z), in an arbitrary graph.

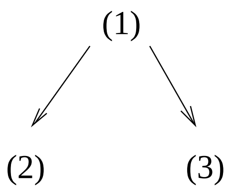

Figure 3.15: The interaction scheme of A_{EMBED}

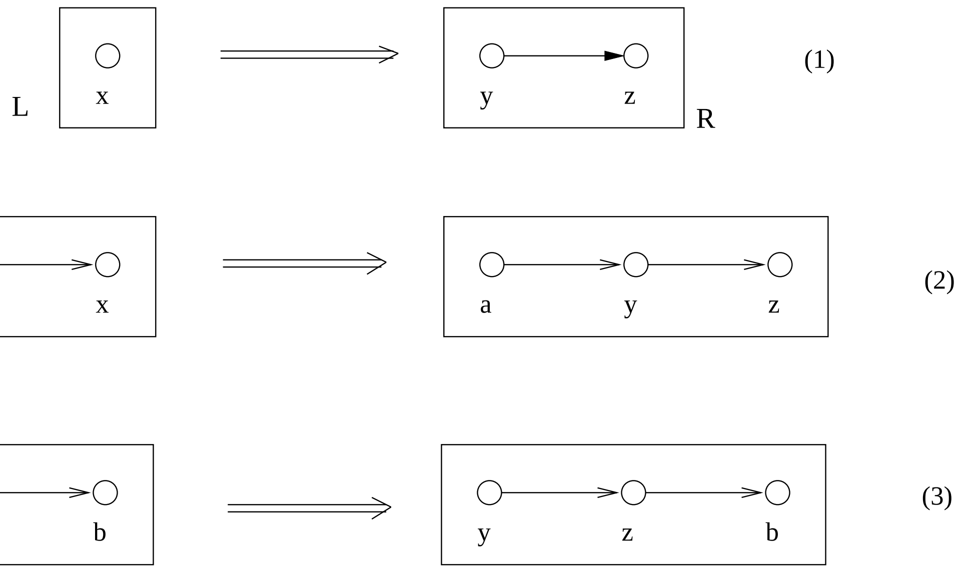

Figure 3.16: Embedding specification

For adding the handle (y,z) to a graph an embedding specification is given regulating the relations between the handle and the context:

1. If there is an edge from a node with label a to a node with label x then there is an edge from the a-node to the y-node.
2. If there is an edge from a node with label x to a node with label b then there is an edge from the z-node to the b-node.

This context dependent graph transformation can be described by a production as introduced in section 2.1: $p = (L, R, \emptyset, \{(x, y), (x, z)\}, \emptyset)$ where L and R are the graphs as depicted in figure 3.16. The original action is described by production (1) and the embedding specification by productions (2) and (3) in figure 3.16. The corresponding amalgamation scheme is defined by $A_{EMBED} = (IS_{EMBED}, \{locall\})$ with IS_{EMBED} depicted in figure 3.15.

For explicit graphs the parallel replacement at amalgamation scheme A_{EMBED} would yield the derivations in figure 3.17.

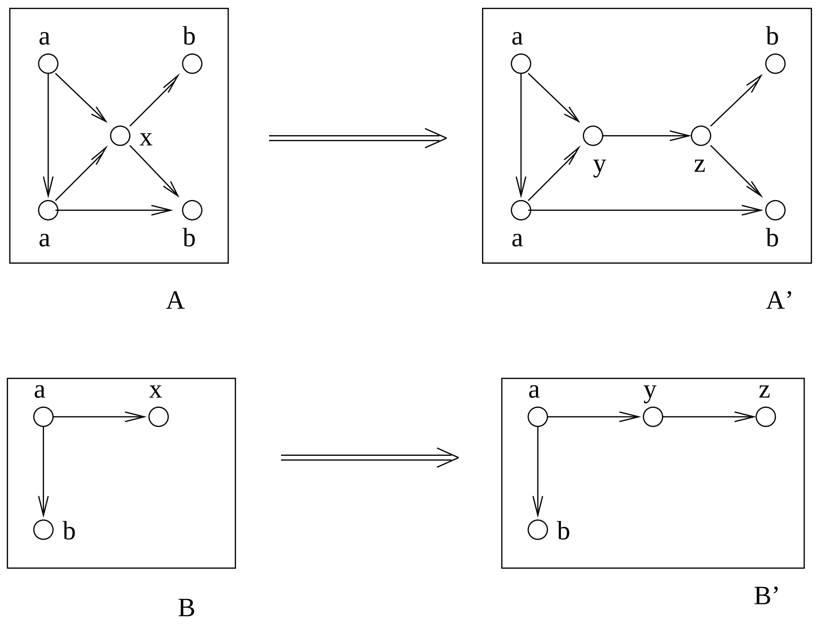

Figure 3.17: Example embeddings

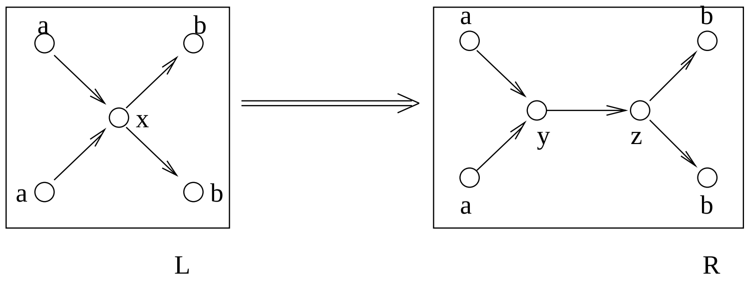

Figure 3.18: Amalgamated production for embedding into graph A

If we use the embedding productions (2) and (3) as elementary productions describing the relation of node x to the context, node x and its particular context in graph A can be covered by matches of these elementary productions. All four matches overlap in node x, so this node can be viewed as an interface. The actual part of the graph which is transformed is shown in L in figure 3.18.

Figure 3.18 shows the amalgamated production fitting to graph A. For graph B we get a similar but different production. Thus, using amalgamation scheme A_{EMBED} we describe a production scheme each element of which shows one concrete embedding.

3.4 Comparison with Related Graph Transformation Concepts

In the following, the concepts of parallel graph transformations introduced in this chapter are compared with the parallel graph rewriting concepts existing in other graph transformation approaches presented in section 2.2. It turns out that all these approaches can be covered by parallel graph transformation in the algebraic approach. Moreover, additional features are introduced. Their additional expressive power is shown within chapter 7 where several examples are presented. A general discussion how parallel graph transformation can be used for specification of communication-based systems is given in section 7.4.

Parallel graph transformation as presented in this chapter is based on the covering approach to parallel graph grammars in the algebraic approach. The covering approach is a special parallel graph transformation where all subproductions used are identical, i.e. the overlapping parts are not changed.

Parallel graph transformation integrates the amalgamation concepts introduced in [BFH87]. The synchronization of two actions by a subaction is extended to an arbitrary, but finite number of actions where each pair may be synchronized by a subaction.

Amalgamated graph transformation as well as contextual parallel graph transformation are kinds of parallel replacement executed in some context. Amalgamated graph transformation corresponds exactly to the idea of mixing parallel and sequential graph rewriting as presented in [Nag79] (see also section 2.2). Contextual parallel graph transformation is essentially a pure parallel replacement on graphs.

All-match-coverings realize the L-system way of parallel graph replacement. Local-all-match-coverings can reflect the embedding mechanisms presented in section 2.1 and inherent to the NLC- and algorithmic approaches as shortly discussed in subsection 3.3.3. Moreover, local-all-match-coverings allow the description of context embeddings dependent of the existence of arbitrary subgraphs which extends the embedding possibilities of the NLC- and algorithmic approaches considerably. The usefulness of such a flexible embedding concept is shown in example 7.1.

Variable paths (with variable length) as presented in PROGRES [Sch91a] may be described by production schemes again. The idea is to cover a path by handles overlapping in common nodes.

The embedding ideas can be extended to the concept of connection rules in the parallel node replacement approaches. This means that the specification of connections between daughter graphs is not restricted to stencils but allow arbitrary subgraphs as connections between daughter graphs dependent of arbitrary connections between the corresponding mother graphs. Such a flexible connection would be specified by an interaction scheme where subproductions describe those production applied in parallel. The elementary productions specify possible connections if several subproductions are embedded into one elementary production. Furthermore, they may describe the context embedding as for sequential replacement.

The idea of tables well-known from L-systems (see subsection 1.3.1) is very much extended by interaction schemes since the interaction schemes allow the specification of interaction between any two elementary productions by subproductions. This idea captures also the concepts of rendezvous specifications in [DDK93] which can be used for the description of node fusions. Such a fusion can be expressed by interaction schemes if gluing subproductions are defined whenever node fusion is desired. Compare examples 2.2.3 and 3.3.5 for different descriptions.

Altogether, the way covering constructions are introduced is a very general one allowing a flexibility in specifying synchronization of parallel replacements not present in other parallel graph transformation approaches.

Chapter 4

Formal Description of Parallel Graph Transformation

After the informal introduction of parallel graph transformation in the previous chapter this chapter gives their formal description based on the double-pushout approach to graph transformation. A short version can be found in [TB94].

The algebraic double-pushout approach is reviewed in section 4.1. This approach is based on category theory as presented in, e.g. [Mac71] or [AHS90]. The underlying category for all further investigations is the category **GRAPH** of graphs and total graph morphisms describing graph relations. The gluing constructions for graphs, basic for algebraic graph transformation, are characterized by *pushouts*, a special kind of *colimits* known from category theory. Parallel rewriting is based on a more general kind of colimits, the *pushout-stars*.The formal description of all the concepts of parallel replacement follows in sections 4.2 and 4.3.

The *main results* achieved within this chapter concern the relations between sequential and parallel graph transformation on one hand and different parallel graph transformation concepts on the other hand. (The graph transformation concepts investigated here are introduced informally in section 2.1 and chapter 3.) Amalgamated graph transformation at identical subproductions is compared with sequential graph transformation at parallel productions in *Parallel Replacement Theorem I* (see 4.4.1). *Parallel Replacement Theorem II* (see 4.4.7) shows the equivalence of amalgamated graph transformation and contextual parallel graph transformation. The proof of this theorem is based on the commutativity property for colimits presented, for example, in [Mac71]. All these results are stated in section 4.4.

The parallel transformation concepts as well as both results listed above have also been generalized for high-level replacement systems [EHKP91b] which are an axiomatic framework based on double-pushouts in order to unify replacement systems like grammars for different kinds of graphs and relational structures or other types of structures like algebraic specifications. The generalization of parallel replacement is done in [ET92, Tae92].

Readers of this chapter are assumed to be familiar with the basic notions of category theory, including *colimits* and their *communtativity*.

4.1 The Double-Pushout Approach to Graph Transformation

In this section the formal description of the basic ideas of graph transformation in the double-pushout (DPO) approach is given. This approach has been introduced in [EPS73]. A comprehensive introduction is given in [Ehr79, CMR+96b]. Some shorter introductions with tutorial character are [Ehr87, EKL91].

The name "double-pushout" points to the main concept of this graph transformation approach which can be described by two pushouts. A pushout is an essential concept of category theory. But nevertheless, all concepts presented in this chapter are first described in a set-theoretical way and then, if possible, they are characterized by categorical concepts.

In the following, we want to deal with graphs and different kinds of their transformation. To keep the presentation as simple as possible we take directed, node and edge labeled graphs as introduced informally in the introduction to graph transformation in section 2.1. Some remarks how the following concepts can be generalized to more complex types of graphs as hypergraphs or attributed graphs are given in section 2.1, too.

To describe the relation between two graphs the notion of a graph morphism is presented. It consists of two mappings for nodes and edges which have to respect the structures of the graphs. The colors of nodes and edges have to be preserved by the mappings.

Definition 4.1.1 (Graphs and graph morphisms)

1. *Let $C = (C^E, C^N)$ be a pair of color alphabets for nodes and edges. A* graph *$G = (G^E, G^N, s_G, t_G, m_{GE}, m_{GN})$ consists of the sets of edges and nodes, G^E and G^N, mappings $s_G, t_G : G^E \to G^N$, called source resp. target map, and the coloring maps for arcs and nodes, $m_{GE} : G^E \to C^E$ and $m_{GN} : G^N \to C^N$. G is said to be* colored according to C.

2. *Given two graphs G and H a* graph morphism *$f : G \to H$ is a pair of mappings $(f^E : G^E \to H^E, f^N : G^N \to H^N)$ such that f^E and f^N are compatible with source and target mappings and label preserving, i.e. $f^E(s_G(e)) = s_H(f^N(e))$, $f^E(t_G(e)) = t_H(f^N(e))$, $f^E(m_{GE}(e)) = m_{HE}(e)$ and $f^N(m_{GN}(n)) = m_{HN}(n)$, $\forall e \in G^E$ and $\forall n \in G^N$.*

3. *A graph morphism $f = (f^E, f^N)$ is called* injective (surjective, bijective) *if f^E and f^N are injective (surjective, bijective).*

Remark: If we do not want to distinguish between nodes and edges the addition "N" or "E" is left out, i.e. "G" instead of "G^N". Most of the following operations on graphs are componentwise defined such as the disjoint union $G + H$ of graphs G and H consisting of the disjoint union of node and edge sets $G^N + H^N$ and $G^E + H^E$. If the constructions are completely analogous we do not distinguish the node and the edge case.

Graph isomorphisms are given by two bijective mappings which relate the nodes and edges "one-to-one". Thus, the structure of two isomorphic graphs is the same and so are the colors of corresponding nodes and edges. The only difference can be in the graphical layout of the two graphs. Since we usually want to describe structures by graphs two graphs are considered to be equal if they differ only in the representation. Thus, isomorphic graphs are considered to be equal.

In the following examples, graphs and graph morphisms are given in a graphical way where the nodes and edges are not explicitly named. A set-theoretical description can be uniquely derived from the drawings. The mappings of nodes and edges are either given by dashed edges between corresponding items, corresponding fill styles and positions of nodes or small numbers where those items which are related by the mapping are numbered alike. Often some of the numbers are left out if they can be added in a unique way.

All following constructions on graphs are given "up to isomorphism" which means that an isomorphic graph[1] would do the job, too. This applies, for example, to the following gluing of graphs. Roughly speaking, the union D of two graphs B and C, which overlap at a common subgraph A, is constructed. In this union the subgraph occurs just once. More precisely, A need not to be a subgraph of B and C but can be an arbitrary graph which is the domain of a graph morphism to B and a morphism to C. If these morphisms are not injective the situation is slightly more difficult since nodes and edges can be identified.

Definition 4.1.2 (Gluing of graphs) *Given three graphs A, B and C and two graph morphisms $f : A \to B$ and $g : A \to C$ graph D and two graph morphisms $h : B \to D$ and $k : C \to D$ are constructed in the following way:*

Let $\bar{I}_A \subseteq B \times C$ be the equivalence relation generated by the relation $I_A = \{(f(x), g(x)) \mid x \in A\}$. The glued graph *$D$ is then defined by $D = (B + C)_{/\bar{I}_A}$, the quotient of the disjoint union of B and C. Functions $h : B \to D$ and $k : C \to D$ send each element of B and C to its equivalence class in D.*

Mappings s_D, t_D, m_{DN} and m_{GE} are uniquely determined by definition 4.1.2.

The following fact shows that all the essential notions of graph, graph morphism and gluing of graphs can be characterized by concepts from category theory. This

[1] Construction "up to isomorphism" might not be adequate if we consider equivalence on graph transformation in connection with sequential composition. Work on this topic has led to the notion of *standard isomorphism* first introduced in [CEL+96].

fact is especially useful if we want to show further properties of graph transformation since they can be based on properties for pushouts.

Fact 4.1.3 (Category GRAPH)

1. *All graphs and graph morphisms as defined above form a category* **GRAPH**.

2. *Bijective graph morphisms are isomorphisms in* **GRAPH**.

3. *The empty graph $\emptyset$, i.e. a graph with empty node and edge sets, is the initial object in* **GRAPH**.

4. *The disjoint union $\sum_1^n G_i$ of graphs G_i, $\forall\ 1 \leq i \leq n$, with the injections $in_i : G_i \to \sum_1^n G_i$ is the coproduct of all G_i's in* **GRAPH**.

5. *The gluing construction in 4.1.2 can be characterized as a pushout in the category* **GRAPH**.

Proof: Compare [EPS73, Ehr79, ELM+].

□

Now, we start to describe sequential graph transformation which is performed by applying one production to the given graph. A production describes which nodes and edges shall be deleted, preserved and newly created. This is expressed by two graphs L and R, the left-hand side and the right-hand side. Moreover, we have an intermediate graph I, also called the gluing graph. This graph shows the preserved items. Graph I corresponds to a subgraph of L. The rest of graph L specifies that part which shall be deleted. Moreover, preserved items can be glued which is handled by the morphism to the right-hand side that may be non-injective.

Definition 4.1.4 (Production) *A* production $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ *consists of graphs L, I and R, called* left-hand side, intermediate *or* gluing graph *and* right-hand side *of p, respectively, and two morphisms $I \xrightarrow{l} L$ and $I \xrightarrow{r} R$.*

- *p is called* l-injective *if morphism l is injective.*

- *p is called* injective *if morphisms l and r are injective.*

- *A production $p = (I \xleftarrow{id_I} I \xrightarrow{id_I} I)$ of two identities is called* identical production.

- *The production $p^{-1} = (R \xleftarrow{r} I \xrightarrow{l} L)$ is called the* inverse production *of p.*

- *The production $\emptyset = (\emptyset \longleftarrow \emptyset \longrightarrow \emptyset)$ is called* empty.

Remark: In examples the intermediate graph is usually left out and can be uniquely determined by small numbers at nodes and edges occuring in the left-hand and the right-hand side of the production, by the nodes which are connected by dashed arrows, or all nodes which are filled and positioned alike.

Given a production an occurrence of its left-hand side has to be searched where the production shall be applied. If nodes shall be deleted we have to make sure that there is no context edge which would dangle after deletion, i.e. which would not have a source or a target node afterwards. Furthermore, this occurrence need not to be an isomorphic image of the left-hand side. But we have to ensure that there are not two items identified where one is deleted and the other is preserved. Moreover, we must be careful with deletion which means isomorphic images of the deleting part are demanded. All these restriction are combined in the following gluing condition. If the occurrence satisfies the gluing condition it is called *match*.

Definition 4.1.5 (Gluing condition) *Given an l-injective production* $p = (L \xleftarrow{l} I \xrightarrow{r} R)$*, a graph morphism* $m : L \to G$*, called* occurrence*, satisfies the* gluing condition *with respect to* p *if*

1. *Identification-Condition:*

$$\{x \in L \mid m(x) = m(y) \text{ for some } y \in L, y \neq x\} \subseteq l(I)$$

2. *Dangling Condition:*

$$\{x \in L^N \mid m^N(x) = s_G(e) \text{ or } m^N(x) = t_G(e) \text{ for some } e \in G^E - m^E(L^E)\} \subseteq l(I)$$

Occurrence m *is called* match *of the left-hand side* L *of production* p *if* m *satisfies the gluing condition with respect to* p*.* m *is also called match of production* p*.*

If the gluing condition is satisfied a production can be applied at the corresponding match. In the first step all those items of working graph G are deleted which have preimages in the left-hand side but not in the interface graph. The resulting graph is called context graph. This graph shows some kind of intermediate state of graph transformation where the deletion step is executed but the creation step still has to be done. The second step is then the creation step where the resulting graph is constructed.

Definition 4.1.6 (Context graph) *Given an l-injective production* $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ *and a match* $m : L \to G$ *graph* $C = G - m(L) + m(l(I)))$ *is called* context graph *of* l *and* m*.*

The deletion step can be described as inverse gluing of graphs. Taking the left-hand side L and the context graph C which overlap in the interface graph I the gluing of L and C along I yields graph G again.

$$
\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & I \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "c"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "q"] \\
G & C \arrow[l, "c_g"] \arrow[r, "c_h"'] & H
\end{tikzcd}
$$

Figure 4.1: Direct transformation

Fact 4.1.7 (Applicability of productions) *Given an l-injective production $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ and a match $m : L \to G$ there are a context graph C and two graph morphisms $c_g : C \to G$ as well as $c : I \to C$ such that (G, m, c_g) is the pushout of l and c in* **GRAPH**. *(C, c, c_g) is also called pushout-complement of (G, m, l).*

Proof: Compare [Ehr79, ELM+]. □

Definition 4.1.8 (Direct transformation) *Given an l-injective production $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ and a match $m : L \to G$, a* direct transformation *$G \Longrightarrow_d H$ using p and m, short $G \overset{p,m}{\Longrightarrow}_d H$, from a graph G to a graph H is given by the two pushout diagrams (1) and (2) in the category* **GRAPH** *shown in figure 4.1.*

If the production is l-injective context graph C can be chosen uniquely wrt. graph G , i.e. C is the unique pushout complement (up to isomorphism). Otherwise a graph $C' = G - m(L) + I$ which is non-isomorphic to C would fit as well. Choosing such a C' as context graph copying of nodes and edges could be described. But C' may not be unique.

Fact 4.1.9 (Uniqueness of direct transformations) *Given an l-injective production $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ and a match $m : L \to G$. Graph morphism l is injective if and only if the direct transformation $G \overset{p,m}{\Longrightarrow}_d H$ is unique up to isomorphism.*

Proof: Compare [Ehr79, ELM+]. □

Definition 4.1.10 (Graph transformation (sequence)) *A* graph transformation sequence*, short* graph transformation, transformation sequence *or* transformation, *$G \overset{*}{\Longrightarrow}_d H$ is a sequence $G = G_0 \overset{p_1,m_1}{\Longrightarrow}_d G_1 \overset{p_2,m_2}{\Longrightarrow}_d \ldots G_n = H$ of n direct transformations $G_{n-1} \overset{p_n,m_n}{\Longrightarrow}_d G_n$ using productions p_n at matches m_n.*

Definition 4.1.11 (Graph grammar, language and semantics) *A* graph grammar *$GG = (S, P, C, T)$ is given by a start graph S, a production set P, a pair C of node and edge colors (or labels) and a pair T of terminal node and edge colors such that $T \subseteq C$. Start graph S and all graphs of the productions of P have to be colored according to C.*

$$
\begin{tikzcd}
L \arrow[d, "m"'] & I \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & R \arrow[dr, "n"] \arrow[drrr, dotted, "e"] & & L' \arrow[dl, "m'"'] \arrow[dlll, dotted, "d"'] & I' \arrow[l, "l'"'] \arrow[r, "r'"] \arrow[d] & R' \arrow[d] \\
G & C \arrow[l, "c_g"'] \arrow[rr, "c_h"] & & H & & C' \arrow[ll, "c'_h"'] \arrow[r] & X
\end{tikzcd}
$$

Figure 4.2: Sequential independence

The language $\mathcal{L}(GG)$ *of* GG *is defined by the class of all terminal colored graphs derivable from* S *by a sequence of direct transformations via productions of* P, *i.e.* $\mathcal{L}(GG) = \{G | S \overset{*}{\Longrightarrow}_d G\}$.

The operational semantics $Sem(GG)$ *of* GG *is given by the class of all possible transformations starting at* S *using production set* P, *i.e.* $Sem(GG) = \{S \overset{*}{\Longrightarrow}_d G\}$.

Given a graph G and a production p colored according to a color alphabet C, the resulting graph H is automatically colored according to C due to the construction of a graph transformation.

In the last part of this section we review those concepts of parallelism which have been developed for sequential graph transformation. The starting point for all the concepts in this part is the question: In which case can two productions be applied in either order such that equal graphs are obtained? The answer is: If the applications of the productions are independent of each other. But what does independence mean? Clearly, two transformations are independent if the first one does not create items that are used by the second one, and vice versa. In this case the two productions can also be applied in opposite order although they may use common items that are preserved. This kind of independence is reflected by the following notion of sequential independence.

Definition 4.1.12 (Sequential Independence) *Given two productions* $p = (L \overset{l}{\longleftarrow} I \overset{r}{\longrightarrow} R)$ *and* $p' = (L' \overset{l'}{\longleftarrow} I' \overset{r'}{\longrightarrow} R')$, *a transformation sequence* $G \overset{p,m}{\Longrightarrow}_d H \overset{p',m'}{\Longrightarrow}_d X$ *is called* sequentially independent *if we have for morphism* $n : R \to H$ *induced by the first transformation:*

$$m'(L') \cap n(R) \subseteq m'(l'(I')) \cap n(r(I))$$

Fact 4.1.13 (Characterization of the sequential independence) *A transformation sequence* $G \overset{p,m}{\Longrightarrow}_d H \overset{p',m'}{\Longrightarrow}_d X$ *as shown in figure 4.2 is sequentially independent iff there are two graph morphisms* $L' \overset{d}{\longrightarrow} C$ *and* $R \overset{e}{\longrightarrow} C'$ *(dotted arrows) such that* $c_h \circ d = m'$ *and* $c'_h \circ e = n$.

Proof: Compare [Ehr79, ELM+]. □

If two productions can be applied to a graph G their applications are considered to be parallel independent if their matches overlap at most on preserved items. This

$$\begin{tikzcd}
R' \arrow[d] & I' \arrow[l, "r'"'] \arrow[r, "l'"] \arrow[d] & L' \arrow[dr, "m'"'] \arrow[drrr, dotted, "e"] & & L \arrow[dl, "m"] \arrow[dlll, dotted, "\bar{d}"'] & I \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & R \arrow[d] \\
H' & C' \arrow[l, "\bar{c}_{h'}"'] \arrow[rr, "\bar{c}_{g'}"] & & G & & C \arrow[ll, "c_g"'] \arrow[r, "c_h"] & H
\end{tikzcd}$$

Figure 4.3: Parallel independence

is legal because they are not changed. They are not allowed to overlap in deleting parts.

Definition 4.1.14 (Parallel Independence) *Given productions* $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ *and* $p' = (L' \xleftarrow{l'} I' \xrightarrow{r'} R')$, *two transformations* $G \overset{p,m}{\Longrightarrow}_d H$ *and* $G \overset{p',m'}{\Longrightarrow}_d H'$ *are called* parallel independent *if we have:*

$$m(L) \cap m'(L') \subseteq m(l(I)) \cap m'(l'(I'))$$

Fact 4.1.15 (Characterization of the parallel independence) *Two direct transformations* $G \overset{p,m}{\Longrightarrow}_d H$ *and* $G \overset{p',m'}{\Longrightarrow}_d H'$ *as shown in figure 4.3 are parallel independent iff there are two graph morphisms* $L \xrightarrow{d} C'$ *and* $L' \xrightarrow{e} C$ *(dotted arrows) such that* $c_{g'} \circ d = m$ *and* $c_g \circ e = m'$.

Proof: Compare [Ehr79, ELM+]. □

Theorem 4.1.16 (Local Church-Rosser Theorem I) *Given parallel independent direct transformations* $G \overset{p,m}{\Longrightarrow}_d H$ *and* $G \overset{p',m'}{\Longrightarrow}_d H'$ *as in figure 4.3 there are a graph* X *and direct transformations* $H \overset{p',c_h \circ e}{\Longrightarrow}_d X$ *and* $H' \overset{p,\bar{c}_{h'} \circ \bar{d}}{\Longrightarrow}_d X$, *such that the transformation sequences* $G \overset{p,m}{\Longrightarrow}_d H \overset{p',c_h \circ e}{\Longrightarrow}_d X$ *and* $G \overset{p',m'}{\Longrightarrow}_d H' \overset{p,\bar{c}_{h'} \circ \bar{d}}{\Longrightarrow}_d X$ *are sequentially independent.*

Proof: Compare [Ehr79, ELM+]. □

Theorem 4.1.17 (Local Church-Rosser Theorem II) *Given a sequentially independent transformation* $G \overset{p,m}{\Longrightarrow}_d H \overset{p',m'}{\Longrightarrow}_d X$ *as in figure 4.2 there is a transformation* $G \overset{p',c_g \circ d}{\Longrightarrow}_d H'$ *with context graph* $\bar{C}'$ *parallel independent of* $G \overset{p,m}{\Longrightarrow}_d H$, *i.e. there is a graph morphism* $\bar{d} : L \to \bar{C}'$ *such that* $\bar{c}_g \circ \bar{d} = m$. *Moreover, there is a sequentially independent transformation* $G \overset{p',c_g \circ d}{\Longrightarrow}_d H' \overset{p,\bar{c}_{h'} \circ \bar{d}}{\Longrightarrow}_d X$. *(Compare figure 4.4.)*

Proof: Compare [Ehr79, ELM+]. □

One step further we try to combine the effects of the sequential application of two productions by constructing a parallel production of them. This production is a disjoint union of the given ones and its application simulates their simultaneous

$$\begin{tikzcd}
R' \arrow[d] & I' \arrow[l, "r'"'] \arrow[r, "l'"] \arrow[d] & L' \arrow[d, "c_g \circ d"'] \arrow[rr, dotted, "d", pos=0.8, shift={(0,-1)}, to path={-- ([yshift=-2ex]\tikztotarget)}] & L \arrow[dl, "m"] \arrow[dll, dotted, "\bar{d}"'] & I \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & R \arrow[d, "n"] \\
H' & \bar{C}' \arrow[l, "\bar{c}_{h'}"] \arrow[r, "\bar{c}_g"'] & G & & C \arrow[ll, "c_g"] \arrow[r, "c_h"'] & H
\end{tikzcd}$$

Figure 4.4: Parallel independent transformations $G \overset{p,m}{\Longrightarrow}_d H$ and $G \overset{p',c_g \circ d}{\Longrightarrow}_d H'$

application. The productions applied in parallel may overlap in items which are preserved by all of them. These overlappings are handled implicitly by non-injective matches of the corresponding parallel productions.

Definition 4.1.18 (Parallel production) *Given productions* $p = (L \overset{l}{\longleftarrow} I \overset{r}{\longrightarrow} R)$ *and* $p' = (L' \overset{l'}{\longleftarrow} I' \overset{r'}{\longrightarrow} R')$ *the production* $p+p' = (L+L' \overset{l+l'}{\longleftarrow} I+I' \overset{r+r'}{\longrightarrow} R+R')$ *is called the* parallel production *of* p *and* p' *where* $L+L$, $I+I'$ *and* $R+R'$ *are coproduct graphs and* $l+l'$ *as well as* $r+r'$ *the induced morphisms.*

The construction of parallel productions can be iterated[2] leading to productions $p_1 + p_2 + \ldots + p_n$ with $n \geq 1$ which are also called parallel productions, written $\sum_1^n p_i = (\sum_1^n L_i \overset{\sum l_i}{\longleftarrow} \sum_1^n I_i \overset{\sum r_i}{\longrightarrow} \sum_1^n R_i)$.

Theorem 4.1.19 (Parallelism theorem) *Given injective productions* p *and* p' *the following statements are equivalent:*

1. *There is a direct transformation* $t : G \overset{\hat{p},\hat{m}}{\Longrightarrow}_d X$ *with* $\hat{p} = p + p'$.
2. *There is a sequentially independent transformation* $s : G \overset{p,m}{\Longrightarrow}_d H \overset{p',m'}{\Longrightarrow}_d X$ *as in figure 4.2.*

Proof: Compare [Ehr79, CMR+96b]. □

Corollary 4.1.20 (Parallelism closure of graph languages) *Given a graph grammar* $GG = (S, P, C, T)$ *where all productions of* P *are injective, let* P^+ *be the smallest extension of* P *including all parallel productions* $p_1 + p_2$ *for* $p_1, p_2 \in P^+$ *and* $GG^+ = (S, P^+, C, T)$*, then the generated languages are equal:*

$$L(GG) = L(GG^+)$$

[2]Please note that the construction of a parallel production is associative up to isomorphism ([CMR+96b]) since it is based on the construction of coproducts. In the following, we denote a parallel production $((...(p_1 + p_2) + ...) + p_n)$ by $\sum_1^n p_i$ which simplifies the syntax but does not imply that coproducts are unique.

Proof: Direct consequence of the parallelism theorem 4.1.19. □

In this section we reviewed the simplest form of parallel replacement on graphs where productions can be applied in parallel if they are independent of each others. Their matches can overlap in preserved items. These overlapping parts are not handled explicitly in the parallel production but implicitly in its match. This match always gets non-injective in the case of overlappings.

In the next section extended parallel replacement on graphs is considered where overlappings can be handled explicitly in connection with the productions. Thus, we have the possibility to specify which overlappings should be allowed. Moreover, matches can overlap also in parts which are deleted. This means that applications of productions do not need to be independent any longer.

4.2 Extended Parallel Replacement on Graphs

In [EK76b] Ehrig and Kreowski presented a kind of parallel replacement on graphs which was motivated by L-systems. As a generalization of L-systems a kind of transformation is introduced in [EK76b] where several productions can be applied in parallel if their matches cover the whole graph. Overlappings of the matches can be handled explicitly by interface graphs. Similarly to the transformations in the previous section matches are not allowed to overlap in parts that should be deleted.

In this section we want to go a step further in parallel replacement on graphs and give up several restrictions made so far. As a basis for this work the amalgamation concept for productions is used. This concept is described in section 2.2.4 and in [BFH87] by Böhm, Fonio and Habel.

Using amalgamation of productions overlappings of matches can be allowed also on deleting parts, i.e. productions can be applied in parallel although their applications are dependent of each others. This is possible if they do all the same with an item, i.e. conflicts between deletion and preservation are prohibited.

Furthermore, interfaces between different productions can be dissolved or extended during the transformations. In this case, an interface is not modeled by only one graph but by a production itself which has to be a subproduction of all productions that use it as interface. These interface productions are used to amalgamate parts of the component productions in order to avoid deletion or addition of subgraphs inside the whole graph several times.

Moreover, we allow that a graph is not covered completely by matches as this is the case for transformations with parallel productions. Hence, the context has to be determined and joined unchanged to the derived part.

The main contribution in [EK76b] is the concept of star gluing of graphs. The essentials of the parallel replacement in [EK76b] are described by two star gluings. Gluing all left-hand sides of the productions along the interface graphs yields the mother graph, gluing all right-hand sides along the interface graphs we obtain the

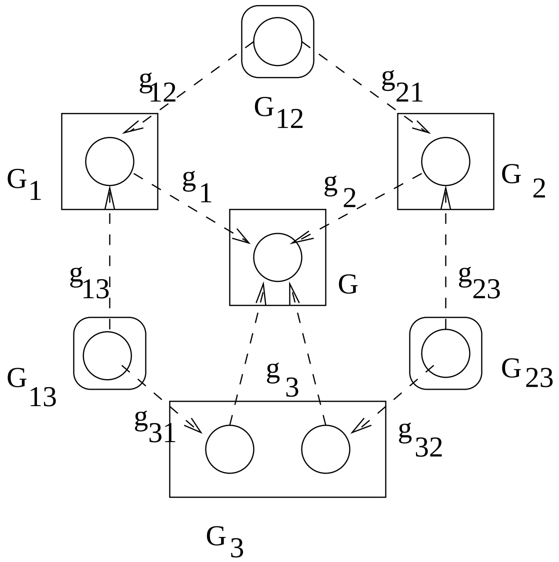

Figure 4.5: Example of star gluing of graphs

derived graph. The star gluing generalizes the concept of gluing of graphs to more than two. Each two of the graphs are combined by an interface graph and corresponding morphisms.

Definition 4.2.1 (Star gluing of graphs)

1. *Given a* graph morphism star $MS = (G_i \xleftarrow{g_{ij}} G_{ij} \xrightarrow{g_{ji}} G_j)_{1 \leq i < j \leq n}$ *a* gluing relation $\bar{\mathcal{I}}_{MS}$ *is the equivalence relation generated by the relation* $\mathcal{I}_{MS} = \{(g_{ij}(x), g_{ji}(x)) \mid x \in G_{ij},\ 1 \leq i < j \leq n\}$.

2. *Given a graph morphism star* $MS = (G_i \xleftarrow{g_{ij}} G_{ij} \xrightarrow{g_{ji}} G_j)_{1 \leq i < j \leq n}$ *with its gluing relation* $\bar{\mathcal{I}}_{MS}$ *the* star gluing graph G *is defined by* $G = (\sum_{1 \leq i \leq n} G_i)_{/\bar{\mathcal{I}}_{MS}}$, *the quotient set of the disjoint union of all* G_i. *Functions* $g_i : G_i \longrightarrow G$ *for all* $1 \leq i \leq n$ *send each element of* G_i *to its equivalence class in* G. *The pair* $(G, \{g_i | 1 \leq i \leq n\})$ *is called* star gluing *of* MS.

Example 4.2.2 (Star gluing of graphs) *The most complex part of the star gluing is the gluing relation. Since it is a real equivalence relation transitivity is very important. In contrast to the gluing of two graphs we have the following phenomena:*

Given a morphism star of size $n = 3$ *where all morphisms are injective it can be the case that not all* g_i*'s are injective. (See figure 4.5.) This is not possible in case of* $n = 2$. *In figure 4.5 the two nodes of* G_3 *are identified indirectly by* G_{12}. *That is why morphism* g_3 *becomes non-injective.*

$$\begin{tikzcd}
L_s \arrow[d, "e"'] \arrow[dr, phantom, "(=)"] & I_s \arrow[l, "l_s"'] \arrow[r, "r_s"] \arrow[d, "f"] \arrow[dr, phantom, "(=)"] & R_s \arrow[d, "g"] \\
L & I \arrow[l, "l"] \arrow[r, "r"'] & R
\end{tikzcd}$$

Figure 4.6: Subproduction

Fact 4.2.3 (Star gluing is colimit) *Given a graph morphism star* $MS = (G_i \xleftarrow{g_{ij}} G_{ij} \xrightarrow{g_{ji}} G_j)_{1 \leq i < j \leq n}$*, a graph* G *together with graph morphisms* $g_i : G_i \to G$ *for all* $1 \leq i \leq n$ *as they are constructed in 4.2.1 are a colimit in* **GRAPH***, also called pushout-star.* G *is called* pushout-star graph*, short POS-graph.*

Proof: Compare [EK76b]. □

In the following, different morphism stars are related with each others to describe parallel graph replacement. Such relations consist of a set of morphisms relating elementary and interface parts.

Definition 4.2.4 (Relations between morphism stars) *Given two morphism stars* $GS = (G_i \xleftarrow{g_{ij}} G_{ij} \xrightarrow{g_{ji}} G_j)_{1 \leq i < j \leq n}$ *and* $HS = (H_i \xleftarrow{h_{ij}} H_{ij} \xrightarrow{h_{ji}} H_j)_{1 \leq i < j \leq n}$ *and a set* $\hat{f}$ *of morphisms* $\{f_i : G_i \to H_i | 1 \leq i \leq n\} \cup \{f_{ij} : G_{ij} \to H_{ij} | 1 \leq i < j \leq n\}$ *such that* $f_i \circ g_{ij} = h_{ij} \circ f_{ij}$ *and* $f_i \circ g_{ji} = h_{ji} \circ f_{ij}$*,* $\forall\, 1 \leq i < j \leq n$*.* $\hat{f}$ *is called* injective *if for all* $x \in G_i$ *and* $y \in G_j$*:* $[f_i(x)] = [f_j(y)]$ *with respect to* $\bar{\mathcal{I}}_{HS}$ *implies* $[x] = [y]$ *with respect to* $\bar{\mathcal{I}}_{GS}$*.*

For the description of interface actions we use the concept of subproductions. Since we are not interested in sequentializing the parallel replacement a simplified notion of subproductions is used here compared to that in [BFH87].

Definition 4.2.5 (Subproductions) *Given a production* $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ *a production* $s = (L_s \xleftarrow{l_s} I_s \xrightarrow{r_s} R_s)$ *is called* subproduction *of* p *if there are graph morphisms* $L_s \xrightarrow{e} L$*,* $I_s \xrightarrow{f} I$ *and* $R_s \xrightarrow{g} R$ *such that* $e \circ l_s = l \circ f$ *and* $g \circ r_s = r \circ f$*, i.e. the diagram in figure 4.6 commutes. The triple* $t = (e, f, g)$ *from* s *to* p *(short* $s \xrightarrow{t} p$*) is called* subproduction embedding*.*

Remark: If e, f and g are isomorphisms t is called isomorphic subproduction embedding. Given two subproduction embeddings $t = (e, f, g)$ of s into p and $t' = (e', f', g')$ of p into q $t' \circ t = (e' \circ e, f' \circ f, g' \circ g)$ is again a subproduction embedding of s into q.

Given some productions which interact via common subproductions a graph G is covered, at least partially, if there is an occurrence for each production compatible

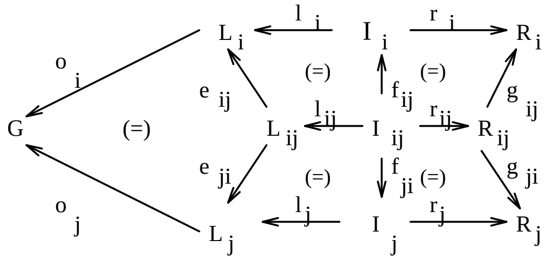

Figure 4.7: Partial covering

with its subproductions. This means that the occurrence of a subproduction of productions p_1 and p_2 has to be inside of both occurrences of p_1 as well as p_2. Similarly to parallel productions each production has to be applied exactly once. Thus, the normal case is that the graph G is not covered completely.

Definition 4.2.6 (Partial coverings) *Let* $T = (\mathcal{P}, \mathcal{SP}, E)$ *be a triple consisting of sets of productions* $\mathcal{P} = \{p_i | p_i = (L_i \xleftarrow{l_i} I_i \xrightarrow{r_i} R_i),\ 1 \leq i \leq n\}$, $\mathcal{SP} = \{s_{ij} | s_{ij} = (L_{ij} \xleftarrow{l_{ij}} I_{ij} \xrightarrow{r_{ij}} R_{ij}),\ 1 \leq i < j \leq n\}$ *and subproduction embeddings* $E = \{t_{ij} = (e_{ij}, f_{ij}, g_{ij}) | 1 \leq i \neq j \leq n\}$ *where* t_{ij} *are embeddings of* s_{ij} *into* p_i *and* t_{ji} *are embeddings of* s_{ij} *into* p_j *for all* $1 \leq i < j \leq n$. *T is called* component production set.

Given T *and a graph* G, *a set* $O = \{o_i | 1 \leq i \leq n\}$ *of graph morphisms* $L_i \xrightarrow{o_i} G$ *is called* parallel occurrence *if* $o_i \circ e_{ij} = o_j \circ e_{ji},\ \forall\, 1 \leq i < j \leq n$.

T and O together are called partial covering $COV = (T, O)$ *of* G. $LS = (L_i \xleftarrow{e_{ij}} L_{ij} \xrightarrow{e_{ji}} L_j)_{1 \leq i < j \leq n}$ *is called the* left-hand side star, $IS = (I_i \xleftarrow{f_{ij}} I_{ij} \xrightarrow{f_{ji}} I_j)_{1 \leq i < j \leq n}$ *is called the* interface star *and* $RS = (R_i \xleftarrow{g_{ij}} R_{ij} \xrightarrow{g_{ji}} R_j)_{1 \leq i < j \leq n}$ *is called the* right-hand side star.

- *If* $n = 1$ COV *is called* basic.
- *A partial covering* COV *of* G *is called* complete covering *if* G *together with all* $o_i \in O$ *is the star gluing of* LS.
- *If* $o_i(L_i) \cap o_j(L_j) \subseteq o_i \circ l_{ij}(L_{ij}),\ \forall\, 1 \leq i < j \leq n$, COV *is called* fully synchronized.
- *If* $o_i(L_i) \cap o_j(L_j) \subseteq o_i(l_i(I_i)) \cap o_j(l_j(I_j)),\ \forall\, 1 \leq i < j \leq n$, COV *is called* fast.
- *In the case that all subproductions* s_{ij} *are identical productions,* COV *is called* partial covering with constant interfaces.
- COV *is called* left injective *if* $\hat{l} = \{l_i | 1 \leq i \leq n\} \cup \{l_{ij} | 1 \leq i < j \leq n\}$ *is injective (compare definition 4.2.4).*

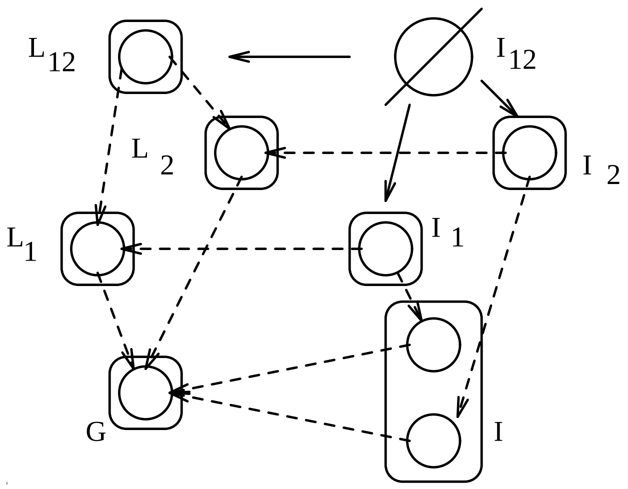

Figure 4.8: Example of a non-injective covering

- *Moreover, COV is called* injective *if it is left injective and right injective, i.e.* $\hat{r} = \{r_i | 1 \leq i \leq n\} \cup \{r_{ij} | 1 \leq i < j \leq n\}$ *is injective, too.*

A basic partial covering can be considered as the parallel version of a production and an occurrence. Thus, the notion of a partial covering "covers" the notion of production and occurrence as special case if the three morphism stars would be glued.

If G is the star gluing graph of LS, COV covers G completely, i.e. is a real covering. Fully synchronized partial coverings allow overlappings of occurrences only in occurrences of interfaces. COV is called fast if each two occurrences overlap only in preserved items, i.e. are independent of each others.

If the component production set of a partial covering consists of injective productions only the covering itself is not automatically injective. (See the following example.)

Example 4.2.7 (Covering) *Given two productions p_1 and p_2 which both preserve one node and a common subproduction s_{12} that deletes a node. The matches of p_1 and p_2 overlap in the only node of G. Although all productions are injective the corresponding partial covering is not injective as you can see in figure 4.8.*

In this figure only the left half of the covering is depicted where I is the (star) gluing graph of the interface star IS. The covering is complete, therefore, G is isomorphic to the (star) gluing graph L of LS. We can see that the resulting morphism from I to G is not injective.

The first kind of extended parallel replacement which is shown here is the so-called star-parallel transformation where the whole mother graph has to be covered. This means that the corresponding covering has to be complete. Similarly to the

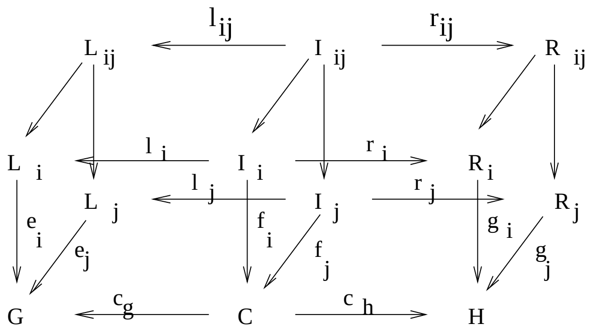

Figure 4.9: Star-parallel transformation

sequential replacement, the star-parallel transformation is performed in two steps: first the deletion step and afterwards the creation step. The gluing of all interfaces yields the context graph which shows the intermediate state after deletion. By gluing of all right-hand sides the resulting graph is constructed.

Definition 4.2.8 (Star-parallel transformation) *Given a complete covering COV of a graph G as in definition 4.2.6 a* star-parallel transformation $G \overset{COV}{\Longrightarrow}_{sp} H$ *over COV is given by the diagram in figure 4.9 where the pushout–star construction of IS yields C with morphisms $I_i \xrightarrow{f_i} C$ and that of RS yields graph H with morphisms $R_i \xrightarrow{g_i} H$, $1 \leq i \leq n$. The graph morphisms $C \xrightarrow{c_g} G$ and $C \xrightarrow{c_h} H$ are the induced morphisms out of POS-graph C such that $c_g \circ f_i = e_i \circ l_i$ ($c_h \circ f_i = g_i \circ r_i$), $\forall\, 1 \leq i \leq n$.*

The component production set T of COV provides also a complete covering for H since H is the gluing graph of RS. Thus, there is also star-parallel transformation $H \overset{COV^{-1}}{\Longrightarrow}_{sp} G$ via $COV^{-1} = (T^{-1}, \{g_i |\ 1 \leq i \leq n\})$ where T^{-1} contains all inverse productions with the same embedding morphisms of T.

The star-parallel transformation over a covering with constant interfaces was first defined as parallel replacement on graphs in [EK76b].

Corollary 4.2.9 (Uniqueness of star-parallel transformation) *Given a complete covering COV of graph G a star-parallel transformation $G \overset{COV}{\Longrightarrow}_{sp} G'$ is unique up to isomorphism.*

Proof: Due to properties of colimits and fact 4.2.3. □

A partial covering which covers a graph G really partially can be completed if the context is determined and explicitly mentioned in the covering. Remember for sequential graph transformations it is possible to determine the context graph if the gluing condition is satisfied. How can this condition be adapted to the parallel case? If a node is deleted by any of the productions we have to make sure that there is not a context edge that dangles. If two items of the left-hand side of the productions are identified in G both are preserved, i.e. they are in the intermediate graphs of their own productions. Otherwise both items have to be identified by the gluing relation, i.e. they are interface items which are allowed to be deleted. The following parallel gluing condition formally describes these ideas.

Definition 4.2.10 (Parallel Gluing Condition) *Given a partial covering COV as in definition 4.2.6 parallel occurrences $o_i : L_i \longrightarrow G, \;\; \forall\, 1 \leq i \leq n$, satisfy the* parallel gluing condition *if the following conditions are valid:*

1. *Parallel identification condition:*

   For all $x \in L_i$ and $y \in L_j$ with $x \neq y$ and $o_i(x) = o_j(y)$ there is at least one of the following conditions satisfied:

   (a) $[x] = [y]$ with respect to $\bar{\mathcal{I}}_{LS}$

   (b) $x \in l_i(I_i)$ and $y \in l_j(I_j)$ with $1 \leq i, j \leq n$

2. *Parallel dangling condition:*

   For all edges $e \in G^E - \bigcup_{1 \leq i \leq n} o_i^E(L_i^E)$ with $s_G(e) = o_i^N(x)$ or $t_G(e) = o_i^N(x)$ for some $x \in L_i^N$ we have

$$o_i^N(x) \in \bigcup_{1 \leq j \leq n} o_j^N(l_j^N(I_j^N)) \qquad \forall\, 1 \leq i \leq n$$

Remark: For special partial coverings the parallel gluing condition can be formulated easier:

1. For basic coverings the parallel gluing condition is satisfied if the gluing condition holds for the only production and the only occurrence in COV.

2. For complete coverings the dangling condition is always fulfilled.

3. For fast coverings the following holds: If the gluing condition holds for each occurrence o_i, $\forall 1 \leq i \leq n$, the parallel gluing condition is satisfied.

   This is due to the fact that the occurrences of two productions overlap exclusively on preserved items. Thus, satisfaction of the parallel identification condition is not based on condition 1.(a). The parallel dangling condition is less restrictive than the dangling condition for all occurrences since the union of all preserved items can be used for satisfaction.

Definition 4.2.11 (Matching partial covering) *A* matching partial covering *is a partial covering $COV = (T, O)$ which satisfies the parallel gluing condition. All $o_i \in O$ are called matches then.*

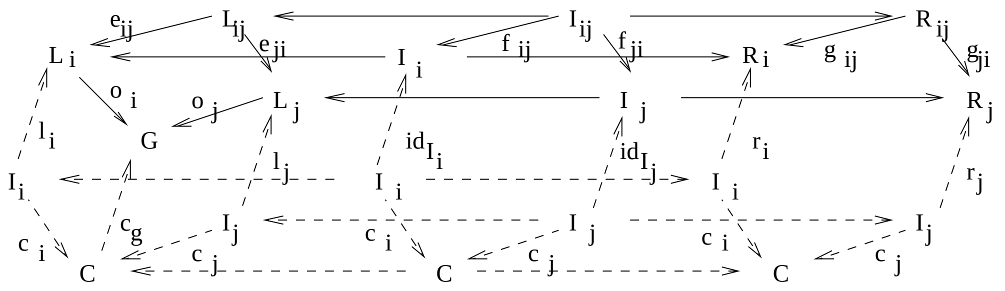

Figure 4.10: Completed covering

Similarly as in the sequential case the context graph can be determined.

Definition 4.2.12 (Context graph of a partial covering) *Given a matching partial covering COV of graph G as in definition 4.2.6, $C = G - \bigcup_{1\leq i\leq n} o_i(L_i) + \bigcup_{1\leq i\leq n} o_i(l_i(I_i))$ is called* context graph *of COV.*

In the following, we show that a partial covering can be completed if it satisfies the parallel gluing conditon.

Proposition 4.2.13 (Completability of coverings)

1. *Given a matching partial covering $COV = (T, O)$ of graph G as in definition 4.2.6, there are a context graph C and morphisms $I_i \xrightarrow{c_i} C$, $\forall\, 1 \leq i \leq n$, and $C \xrightarrow{c_g} G$ such that the star gluing of $LSE = (L_i \xleftarrow{e_{ij}} L_{ij} \xrightarrow{e_{ji}} L_j)_{1\leq i<j\leq n+1}$ with $L_{i(n+1)} = I_i$, $L_{n+1} = C$, $e_{i(n+1)} = l_i$ and $e_{(n+1)i} = c_i$, $\forall\, 1 \leq i \leq n$, yields graph G and morphisms $O \cup \{c_g\}$. Additionally, we have $c_i \circ f_{ij} = c_j \circ f_{ji}$, $\forall 1 \leq i < j \leq n$. (Compare figure 4.10.)*

2. *There is a complete covering $COV' = (T', O')$ of G consisting of $T' = (\mathcal{P}', \mathcal{SP}', E')$ with $\mathcal{P}' = \mathcal{P} \cup \{p_{n+1} \mid p_{n+1} = (C \xleftarrow{id_C} C \xrightarrow{id_C} C)\}$,*

$$\mathcal{SP}' = \mathcal{SP} \cup \{s_{i(n+1)} \mid s_{i(n+1)} = (I_i \xleftarrow{id_{I_i}} I_i \xrightarrow{id_{I_i}} I_i),\ 1 \leq i \leq n\} \text{ and}$$

$$E' = E \cup \{t_{i(n+1)} | t_{i(n+1)} = (l_i, id_{I_i}, r_i),\ 1 \leq i \leq n\} \cup \{t_{(n+1)i} | t_{(n+1)i} = (c_i, c_i, c_i),\ 1 \leq i \leq n\}$$

and $O' = O \cup \{c_g\}$.

3. *COV' is unique iff COV is left injective.*

Proof: Let $(L, (e_i : L_i \to L)_{1\leq i\leq n})$ $(I, (f_i : I_i \to I)_{1\leq i\leq n})$ be the star gluing of $LS = (L_i \xleftarrow{e_{ij}} L_{ij} \xrightarrow{e_{ji}} L_j)_{1\leq i<j\leq n}$ $(IS = (I_i \xleftarrow{f_{ij}} I_{ij} \xrightarrow{f_{ji}} I_j)_{1\leq i<j\leq n})$ and $I \xrightarrow{l} L$ the induced morphism according to pushout-star properties.

1. First we have to show that if all occurrences $o_i : L_i \longrightarrow G$, $\forall\, 1 \leq i \leq n$, satisfy the parallel gluing condition, the induced morphism $o : L \longrightarrow G$ satisfies the gluing condition which is possible since each element in L has a preimage in some L_i. This is valid according to the colimes construction where all colimes morphisms are jointly surjective.

   - Identification Condition:
     Assume that there are two items $x \in L_i$ and $y \in L_j$ with $x \neq y$ and $o_i(x) = o_j(y)$, $1 \leq i < j \leq n$. Then they are either in the same equivalence class with respect to $\bar{\mathcal{I}}_{LS}$, i.e. $[x] = [y] \in L$, or $x \in l_i(I_i)$ and $y \in l_j(I_j)$, i.e. $[x], [y] \in l(I)$.
   - Dangling Condition:
     Assume that there is an edge $e \in G^E - \bigcup_{1 \leq i \leq n} o_i^E(L_i^E)$ with $s_G(e) = o_i^N(x)$ or $t_G(e) = o_i^N(x)$ for some $x \in L_i^N$, $1 \leq i \leq n$. Then we have

     $$o_i^N(x) \in \bigcup_{1 \leq j \leq n} o_j^N(l_j^N(I_j^N)) \qquad \forall\, 1 \leq i, j \leq n$$

     which is equivalent to

     $$o^N([x]) \in o^N(l^N(I^N))$$

     since the universal properties of I and L hold for G.

   So there exists a context graph $C' = G - o(L) + o(l(I))$ of l and o and morphisms $c' : I \longrightarrow C'$ and $c'_g : C' \longrightarrow G$ such that o and c'_g are the pushout of l and c' according to fact 4.1.7.

   Since $o(L) = \bigcup_{1 \leq i \leq n} o_i(L_i)$ and $o(l(I)) = \bigcup_{1 \leq i \leq n} o_i(l_i(I_i))$ we have $C' = C$. It remains to show that all o_i and c'_g are the star gluing of LSE.

   Let $c_i = c' \circ f_i$, $\forall 1 \leq i \leq n$. Since $o_i \circ e_{ij} = o_j \circ e_{ji}$ and $o_i \circ l_i = o \circ e_i \circ l_i = o \circ l \circ f_i = c'_g \circ c' \circ f_i = c'_g \circ c_i$, $\forall\, 1 \leq i < j \leq n$, commutativity is shown.

   Assuming there is a graph X with $L_i \xrightarrow{x_i} X$, $C \xrightarrow{c_x} X$, $x_i \circ e_{ij} = x_j \circ e_{ji}$ and $x_i \circ l_i = c_x \circ c_i$, $\forall\, 1 \leq i < j \leq n$, then there is a unique morphism $L \xrightarrow{l_x} X$ with $l_x \circ e_i = x_i$ since L is a star gluing graph. Using pushout-star properties and $x_i \circ l_i = c_i \circ c_x$ we obtain $l_x \circ l = c_x \circ c'$. Because of pushout properties there is a unique morphism $G \xrightarrow{x} X$ with $x \circ o = l_x$ and $x \circ c'_g = c_x$. Hence, we have $x \circ o_i = x \circ o \circ e_i = l_x \circ e_i = x_i$, $\forall 1 \leq i \leq n$. Uniqueness of x follows from that of l_x. Thus, G is the star gluing graph of LSE.

   Moreover, we have $c_i \circ f_{ij} = c' \circ f_i \circ f_{ij} = c' \circ f_j \circ f_{ji} = c_j \circ f_{ji}$, $\forall\, 1 \leq i < j \leq n$, because of pushout-star properties concerning I.

2. (a) Each production $s_{ij} \in \mathcal{SP}$ is subproduction of p_i and $p_j \in \mathcal{P}$, $\forall\, 1 \leq i < j \leq n$, because COV is a covering. Each production $s_{i(n+1)} = (I_i \xleftarrow{id_{I_i}} I_i \xrightarrow{id_{I_i}} I_i) \in \mathcal{SP}'$ is subproduction of $p_i = (L_i \xleftarrow{l_i} I_i \xrightarrow{r_i} R_i) \in \mathcal{P}'$, $\forall\, 1 \leq i \leq n$. Each production $s_{i(n+1)}$ is subproduction of $p_{n+1} = (C \xleftarrow{id_C} C \xrightarrow{id_C} C) \in \mathcal{P}'$, $\forall\, 1 \leq i \leq n$, since we have $c_i \circ id_{I_i} = c_i = c_i \circ id_C$, $\forall\, 1 \leq i \leq n$.

   (b) Due to the construction of C, all c_i's and c_g G, all o_i's and $c_g = o_{n+1}$ is the star gluing of LSE, $\forall\, 1 \leq i \leq n$.

3. According to fact 4.1.9 context graph C is unique iff $l : I \longrightarrow L$ is injective. This means that for all $x \in I_i$ and $y \in I_j$ with $x \neq y$, $[l_i(x)] = [l_j(y)]$ with respect to $\bar{\mathcal{I}}_{LS}$ implies $[x] = [y]$ with respect to $\bar{\mathcal{I}}_{IS}$. Hence COV has to be l-injective.

   Otherwise if COV is l-injective, l is injective and therefore C is unique. Uniqueness of COV' directly depends on that of C, since the colimit construction is unique up to isomorphisms.

$\square$

Now, we are able to define a parallel replacement based on a partial covering which is not complete. This is done by completing the given covering according to proposition 4.2.13 and doing a star-parallel transformation then.

Definition 4.2.14 (Contextual parallel transformation) *Given a matching partial covering COV of a graph G, COV is completed to a covering COV' as in proposition 4.2.13.* A contextual parallel transformation $G \overset{COV}{\Longrightarrow}_{con} H$ *is defined by the star-parallel transformation* $G \overset{COV'}{\Longrightarrow}_{sp} H$.

Remark: By definition each contextual parallel transformation $G \overset{COV}{\Longrightarrow}_{con} H$ is a star-parallel transformation. The other way around, for each star-parallel transformation $G \overset{COV}{\Longrightarrow}_{sp} H$ there is also a contextual parallel transformation $G \overset{COV'}{\Longrightarrow}_{con} H$ via completed covering COV' of COV with C being the POS-graph of $(I_i \xleftarrow{f_{ij}} I_{ij} \xrightarrow{f_{ji}} I_j)_{1 \leq i < j \leq n}$ and all c_i's being POS-morphisms, c_g is obtained then as induced morphism from C to G. It is easy to proof that COV' is a completed covering of COV. The completion of COV seems to be contra-intuitive since COV is already complete. But strictly considered a covering has to be completed for a contextual parallel transformation as defined in 4.2.14 if it is complete or not. This has been done for COV.

A special case of a contextual parallel transformation is the direct transformation. For $n = 1$ the covering consists of only one graph L_1 and the context completion is the context graph C together with morphisms $I_1 \xrightarrow{c_1} C$ and $C \xrightarrow{c_g} G$ such that the pushout construction over $(L_1 \xleftarrow{l_1} I_1 \xrightarrow{c_1} C)$ yields G. The sets of productions $\mathcal{P}$ and $\mathcal{S}$ are extended by the identical productions $(C \xleftarrow{id_C} C \xrightarrow{id_C} C)$ and $(I_1 \xleftarrow{id_{I_1}} I_1 \xrightarrow{id_{I_1}} I_1)$ and result H of the transformation is obtained by the pushout construction over $(R_1 \xleftarrow{r_1} I_1 \xrightarrow{c_1} C)$.

Corollary 4.2.15 (Uniqueness of contextual parallel transformation) *Given a matching partial covering COV of graph G which is l-injective, the contextual parallel transformation $G \overset{COV}{\Longrightarrow}_{con} H$ is unique up to isomorphism.*

Proof: According to proposition 4.2.13 and corollary 4.2.9. □

Another way to derive a graph from a partial covering is to allow parallel replacement only on that part which is covered. The effect of the replacement is combined into a new production at a new match. To obtain this production the amalgamation concept is used. Afterwards the newly created production is applied at its match in the sequential way. These so-called amalgamated transformations which combine parallel and sequential replacement are especially useful if the production generated can be reused frequently, i.e. the covered part occurs in different contexts.

Definition 4.2.16 (Amalgamated transformation) *Given a matching partial covering $COV = (T, O)$ of a graph G as in definition 4.2.6, an* amalgamated transformation *$G \overset{COV}{\Longrightarrow}_a H$ consists of the following star-parallel transformation $L \overset{COV'}{\Longrightarrow}_{sp} R$ and direct transformation $G \overset{p,o}{\Longrightarrow}_d H$:*

1. *Step: Construction of the* amalgamated production *$p = (L \xleftarrow{l} I \xrightarrow{r} R)$ by the star-parallel transformation $L \overset{COV'}{\Longrightarrow}_{sp} R$ with $COV' = (T, \{L_i \xrightarrow{e_i} L \mid 1 \leq i \leq n\})$ where L and all e_i are the pushout-star of $(L_i \xleftarrow{e_{ij}} L_{ij} \xrightarrow{e_{ji}} L_j)_{1 \leq i < j \leq n}$. Graphs I, R and graph morphisms l and r are uniquely determined by $L \overset{COV'}{\Longrightarrow}_{sp} R$. The morphism $o : L \to G$ which is the induced morphism such that $o \circ e_i = o_i$, $\forall\, 1 \leq i \leq n$, is called* amalgamated match.

2. *Step: Application of the amalgamated production p to the graph G at the amalgamated match o by the direct transformation $G \overset{p,o}{\Longrightarrow}_d H$.*

Obviously, an amalgamated production is a production in the sense of 4.1.4.

Remark: A transformation at a parallel production is a special amalgamated transformation at a partial covering where all subproductions are empty.

While the first step of an amalgamated transformation can always be performed the second step is only possible if the amalgamated match is a match of the amalgamated production in the sense of 4.1.5. Since COV is a matching partial covering it satisfies the parallel gluing condition. In the following corollary it is shown that the amalgamated match satisfies the gluing condition then.

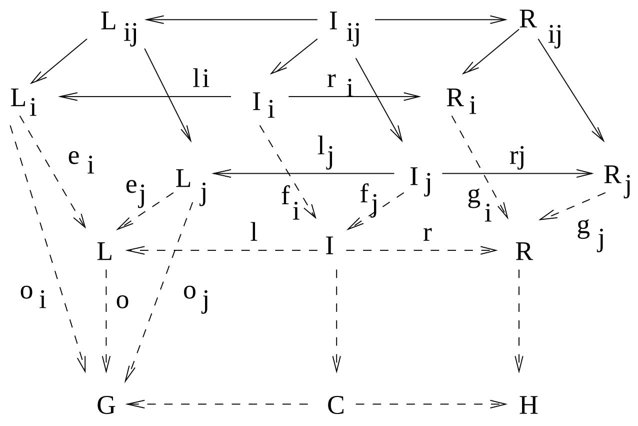

Figure 4.11: Amalgamated transformation

Corollary 4.2.17 (Applicability of amalgamated production) *Given an amalgamated production p and the graph morphism o as in definition 4.2.16 o is a match for p, i.e. o satisfies the gluing condition wrt. p.*

Proof: Compare the first part of the proof of proposition 4.2.13. □

Corollary 4.2.18 (Uniqueness of amalgamated transformation) *Given a matching partial covering COV of a graph G which is l-injective, the amalgamated transformation* $G \overset{COV}{\Longrightarrow}_a H$ *is unique up to isomorphism.*

Proof: The first step of an amalgamated transformation is unique according to corollary 4.2.9 and the second step is unique due to corollary 4.2.17, fact 4.1.7 and fact 4.1.9. □

The different kinds of parallel replacement just defined are compared with each other in section 4.4.

4.3 Parallel Graph Grammars

Grammars where graphs can be derived via parallel replacement are called parallel graph grammars. Until now the effects of parallel replacement are specified very concretely by component production sets where each production is applied exactly once. Comparing this with the L-system's way of parallel replacement productions can be applied more than once, once or not at all, resp., in one step. Generalizing this principle for parallel graph grammars we just have to give a set of productions which can be applied in parallel in one step.

Such a set is called table in the context of L-systems which are then called TOL-systems (compare section 1.3). Since our productions are not applied independently of each other, but they are related by subproductions it should be possible to specify these relations, too. This is done in the following definition by interaction schemes.

For the description of a complex operation, the elementary productions which can be applied in principle are summarized in interaction schemes. Moreover, all interaction possibilities with other elementary productions are described by specifying the subproduction embeddings allowed. One of the interaction possibilities has to be used for synchronizing the elementary productions. If interaction is not required the empty production has to be added to the set of subproductions of the corresponding productions.

Given an interaction scheme and a graph we have to construct a suitable partial covering with a component production set containing each production of the interaction scheme as often as it should be aplied to the given graph.

Definition 4.3.1 (Interaction scheme) *Let $\mathcal{E}$ be a set of productions, called* elementary productions *and $\mathcal{S}$ a set of productions, called* subproductions. *Let SE be a set of subproduction embeddings of the form $se : s \to e$ where $s \in \mathcal{S}$ and $e \in \mathcal{E}$. An* interaction scheme *$\mathcal{IS}$ is then a triple $\mathcal{IS} = (\mathcal{E}, \mathcal{S}, SE)$. For each $e \in \mathcal{E} : I(e) = \{se \in SE | se : s \to e$ for some $s \in \mathcal{S}\}$ describes its* interface.

- *If forall $e \in \mathcal{E}$ the cardinality of $I(e)$, $|I(e)|$, is less or equal to 1, $\mathcal{IS}$ is called* simple.
- *If $\mathcal{S}$ contains identical productions only $\mathcal{IS}$ is called* constant.
- *An interaction scheme $(\{e\}, \emptyset, \emptyset)$ is called* basic.

Remark: An interaction scheme can also be seen as a graph where the nodes are labeled by elementary productions and subproductions. The edges are labeled by subproduction embeddings. Thus, an interaction scheme can be described by a functor $G \to$ **PROD(GRAPH)** where G is a graph and interpreted as a scheme category and **PROD(GRAPH)** is the category of productions in the category **GRAPH**. Morphisms in this category are subproduction embeddings as defined in 4.2.5. Composition and identities are defined componentwise via that of morphisms in **GRAPH**.

Examples of interaction schemes are given in examples 3.3.4, 3.3.5, etc.

Given such an interaction scheme and an arbitrary graph the question arises which coverings exist using the productions of the interaction scheme? Here, we do not restrict the kind of coverings to a special class. So usually, the following covering set is infinite. A covering set describes all possible situations for transformations of a given graph at a given interaction scheme.

Definition 4.3.2 (Covering set) *Given an interaction scheme $\mathcal{IS} = (\mathcal{E}, \mathcal{S}, SE)$ and a graph G, the* covering set *$\mathcal{COV}(\mathcal{IS}, G)$ of G by $\mathcal{IS}$ consists of all matching* *partial coverings* $COV = ((\mathcal{P}, \mathcal{SP}, E), O)$ *as in definition 4.2.6 such that there are mappings*[3]

- $m_p : \mathcal{P} \to \mathcal{E}$ *with* $m_p(p_i) \cong p_i$, $1 \leq i \leq n$,
- $m_{sp} : \mathcal{SP} \to \mathcal{S}$ *with* $m_{sp}(s_{ij}) \cong s_{ij}$, $1 \leq i < j \leq n$ *and*
- $m_e : E \to SE$ *with* $m_e(t_{ij}) : m_{sp}(s_{ij}) \to m_p(p_i)$, $1 \leq i \neq j \leq n$

The L-system way of parallel replacement replaces the whole graph completely where all productions are applied simultaneously. Although this kind of replacement is very common (see f. ex. [NA83], [BCF91] or [Kre92]) it is not the only one. Others which differ in the kind of coverings of the mother graph are imaginable. Some of them are described within this thesis which are motivated by theoretical results or different examples in the following.

Essentially, the choice of special covering sets is adapted to the different types of coverings such as "basic", "complete", "fully-synchronized", "fast", etc. and those coverings where all matches have to be found, the so-called "all-match"-coverings. A construction looking for all matches in a certain part produces "local-all-match"-coverings.

Definition 4.3.3 (Special covering constructions) *We distinguish the following* special sets of coverings *which are subsets of* $\mathcal{COV}(\mathcal{IS}, G)$ *defined as in 4.3.2:*

1. Basic coverings: *All* $COV \in \mathcal{COV}^{basic}(\mathcal{IS}, G)$ *are basic.*
2. Complete coverings*: All* $COV \in \mathcal{COV}^{comp}(\mathcal{IS}, G)$ *are complete.*
3. Fully-synchronized coverings*: All* $COV \in \mathcal{COV}^{sync}(\mathcal{IS}, G)$ *are fully synchronized.*
4. Fast coverings*: All* $COV \in \mathcal{COV}^{fast}(\mathcal{IS}, G)$ *are fast.*
5. Different-match-coverings*: All* $COV \in \mathcal{COV}^{diff}(\mathcal{IS}, G)$ *satisfy the condition:* $\forall p_i, p_j \in \mathcal{P} : p_i \ncong p_j \Rightarrow o_i(L_i) \neq o_j(L_j)$.
6. All-match-coverings*: Let* $\mathcal{COV}^{all}(\mathcal{IS}, G) \subseteq \mathcal{COV}^{diff}(\mathcal{IS}, G)$ *such that* $\forall COV = (T, O) \in \mathcal{COV}^{all}(\mathcal{IS}, G) : \neg\exists COV' = (T', O') \in \mathcal{COV}^{diff}(\mathcal{IS}, G) : O \overset{\subseteq}{\neq} O'$.
7. Local-different-match-coverings*: Let* $\mathcal{COV}^{locdiff}(\mathcal{IS}, G) \subseteq \mathcal{COV}^{diff}(\mathcal{IS}, G)$ *such that* $\forall COV \in \mathcal{COV}^{diff}(\mathcal{IS}, G)$ $\forall 1 \leq i < j_k, j_l \leq n : s_{ij_k} = s_{ij_l}$.

[3] where $p \cong q$ denotes that p is isomorphic to q

8. Local-all-match-coverings*: Let $\mathcal{COV}^{locall}(\mathcal{IS},G) \subseteq \mathcal{COV}^{locdiff}(\mathcal{IS},G)$ such that $\forall COV = (T,O) \in \mathcal{COV}^{locall}(\mathcal{SR},S) : \neg\exists COV' = (T',O') \in \mathcal{COV}^{locdiff}(\mathcal{SR},S) : O \underset{\neq}{\subseteq} O'$.*

Examples for special covering constructions *sync*, *all* and *locall* have been shown in subsection 3.3.2. Note that in example 3.3.6 neither all-match-coverings nor local-all-match-coverings are used. Instead not even the difference of occurrences is required, but only full synchronization.

Considering somehow simplified interaction schemes, uniqueness can be stated for some covering constructions. If $\mathcal{IS}$ is basic all the different kinds of coverings coincide because no subproduction is given in the scheme. Thus, the covering constructed cannot have a size greater than 1.

Furthermore, $\mathcal{COV}^{all}(\mathcal{IS},G)$ usually has cardinality greater than 1. If we assume that we have two different matches o_1 and o_2 of the elementary production e and the set $I(e)$ has cardinality greater than 1. Then we can choose each subproduction embedding $se : s \to e \in I(e)$ if matches o_1 and o_2 overlap in the image of s in G. This is not possible if the interaction scheme is simple.

Since "locall"-coverings cover just the surroundings of an match of the only subproduction, the number of "locall"-coverings depends directly on the number of matches if this subproduction is uniquely embedded in all elementary productions.

Lemma 4.3.4 (Unique covering constructions) *Let $\mathcal{IS} = (\mathcal{E},\mathcal{S},SE)$ be an interaction scheme and G a graph.*

1. *If $\mathcal{IS}$ is basic then $\mathcal{COV}(\mathcal{IS},G) = \mathcal{COV}^{sync}(\mathcal{IS},G) = \mathcal{COV}^{fast}(\mathcal{IS},G) = \mathcal{COV}^{diff}(\mathcal{IS},G) = \mathcal{COV}^{all}(\mathcal{IS},G) = \mathcal{COV}^{locdiff}(\mathcal{IS},G) = \mathcal{COV}^{locall}(\mathcal{IS},G) = \mathcal{COV}^{basic}(\mathcal{IS},G)$ and $|\mathcal{COV}^{basic}(\mathcal{IS},G)| = 1$.*

2. *If $\mathcal{IS}$ is simple then $|\mathcal{COV}^{all}(\mathcal{IS},G)| = 1$.*

3. *If $\mathcal{IS} = (\mathcal{E},\{s\},SE)$ is simple then $|\mathcal{COV}^{locall}(\mathcal{IS},G)| = |Hom(L_s,G)|$ where $Hom(L_s,G)$ is the set of all possible graph morphisms from the left-hand side L_s of s to G.*

Proof: These results follow directly from the construction chosen above. □

Note that the coverings constructed with one of the constructions given above are empty if $n = 0$, i.e. no matches of productions of $\mathcal{E}$ can be found.

Formulating an interaction scheme we have to decide which set of coverings would be appropriate. Since this decision is very much dependent on the interaction scheme it would be convenient to specify for each scheme the class of coverings. We have seen in the examples in subsection 3.3.2 that the kind of covering is very much connected with the complex action that has to be specified. This connection is formulated in a *amalgamation scheme* defined below.

Definition 4.3.5 (Amalgamation scheme) *Given an interaction scheme $\mathcal{IS}$ and a covering construction $cc \subseteq \{basic, comp, sync, fast, diff, all, locdiff, locall, arb\}$ where arb allows all possible coverings, the pair $a = (\mathcal{IS}, cc)$ is called* amalgamation scheme.

Given a graph G, $\mathcal{COV}(a, G) = \bigcap_{cc_i \in cc} \mathcal{COV}^{cc_i}(\mathcal{IS}, G)$ with $arb \notin cc$. Otherwise the set $\mathcal{COV}(a, G) = \mathcal{COV}(\mathcal{IS}, G)$.

Let $\mathcal{A}^{cc} = \{(IS_i, cc) | IS_i$ is an interaction scheme $, i \in \mathcal{N}\}$ where[4] all amalgamation schemes in A^{cc} use the same covering construction.

Examples for amalgamation schemes using different covering constructions are given in section 3.3. In examples 3.3.5 and 3.3.7 covering constructions are used where cc contains more than one restriction.

In the following we define a *parallel graph grammar* which is in some sense parameterized over the types of coverings which are used. It contains a set of amalgamation schemes which describe the set of partial coverings that may be used. In each transformation step another amalgamation scheme may be applied. Since Parallel Replacement Theorem II in 4.4.7 shows the equivalence of amalgamated graph transformation and contextual parallel graph transformation both kinds are allowed to determine the language of a parallel graph grammar.

Definition 4.3.6 (Parallel graph grammars and languages)

1. *A* parallel graph grammar *$PGG = (S, \mathcal{A}, C, T)$ is given by a start graph S, a set $\mathcal{A}$ of amalgamation schemes, a pair C of node and edge colors and a pair T of terminal node and edge color sets such that $T \subseteq C$. Start graph S and all graphs of the productions of P have to be colored according to C.*

2. *A graph H is* parallel derivable *from G via $\mathcal{A}$ if there is a sequence of transformations $G_{i-1} \Longrightarrow_{d_i} G_i$ via COV_i and $COV_i \in \mathcal{COV}(a_i, G_{i-1})$ with $a_i \in \mathcal{A}$ for $1 \leq i \leq m$ such that $G = G_0$, $H = G_m$ and $d_i = a$ or $d_i = con$.*

3. *The* language *$\mathcal{L}(PGG)$ of a parallel graph grammar $PGG = (S, \mathcal{A}, C, T)$ is given by the set of all terminal labeled graphs parallel derivable from S.*

4. *If $\mathcal{A} = \mathcal{A}^{cc}$ the parallel graph grammar $PGG = (S, \mathcal{A}^{cc}, C, T)$ is also written PGG^{cc}.*

5. *If all graphs in $\mathcal{L}(PGG)$are parallel derived from S by amalgamated transformations (contextual parallel transformations) language $\mathcal{L}(PGG)$ is also denoted by $\mathcal{L}^a(PGG)$ ($\mathcal{L}^{con}(PGG)$).*

[4] $\mathcal{N}$ denotes the set of natural numbers.

4.4 Relations Between Different Algebraic Graph Transformation Concepts

In this section we want to relate the different kinds of parallel replacement given in section 4.2. If all interfaces are constant then amalgamated transformations can be compared with direct transformations via parallel productions. In theorem 4.4.1 we show that there is a correspondence between these two kinds of transformations and it turns out that the explicitly given interfaces in amalgamated transformations are handled implicitly in direct transformations via parallel productions.

This theorem gives us the information when an amalgamated transformation can be decomposed using Parallelism Theorem 4.1.19. This means that for each amalgamated transformation with constant interfaces there is a direct transformation via a parallel production which can be sequentialized by the 'analysis'-construction of the Parallelism Theorem.

Amalgamated transformations and contextual parallel transformations correspond to each other bijectively. This is shown in theorem 4.4.7.

Theorem 4.4.1 (Parallel Replacement Theorem I) *Let* $COV = ((\mathcal{P}, \mathcal{SP}, E), O)$ *be a partial covering with constant interfaces as in definition 4.2.6.*

1. Implicate*: Given an amalgamated transformation* $G \overset{COV}{\Longrightarrow}_a H$ *there is a direct transformation* $G \overset{\sum_1^n p_i}{\Longrightarrow}_d H$ *with* $\sum_1^n p_i = (\sum_1^n L_i \overset{\sum l_i}{\longleftarrow} \sum_1^n I_i \overset{\sum r_i}{\longrightarrow} \sum_1^n R_i)$ *and* $p_i \in \mathcal{P}$, $\forall 1 \leq i \leq n$.

2. Explicate*: Given a direct transformation* $G \overset{\sum_1^n p_i}{\Longrightarrow}_d H$ *there is an amalgamated transformation* $G \overset{COV}{\Longrightarrow}_a H$.

3. E/I-Identity*: Let* $G \overset{\sum_1^n p_i}{\Longrightarrow}_d H$ *be a direct transformation. Doing first 'explicate' and afterwards 'implicate' the transformation* $G \overset{\sum_1^n p_i}{\Longrightarrow}_d H$ *is constructed again.*

We cannot state I/E-Identity because for the construction 'explicate' there are different possibilities of choosing interfaces I_{ij} between the I_i's. Among others there are two distinguished ways to choose interfaces I_{ij}:

1. All I_{ij} are equal to the empty graph $\emptyset$ which means purely implicit handling of overlapping of I_i and I_j.

2. All I_{ij} are the intersection graphs of $(I_i \overset{c_i}{\longrightarrow} C \overset{c_j}{\longleftarrow} I_j)$ for $1 \leq i < j \leq n$ which means explicit declaration of the overlapping of I_i and I_j.

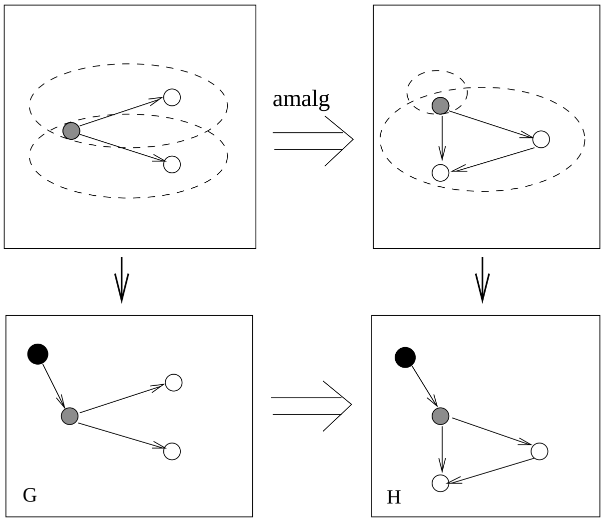

Figure 4.12: Application of an amalgamated production

Example 4.4.2 (Parallel and amalgamated production application) *Consider figure 3.1 showing the application of a parallel production and figure 4.12 where an amalgamated production is applied. The left and the right-hand sides of the elementary productions are indicated by dashed ellipses. Their subproduction is an identical production over one grey node not explicitly shown in the figure. The occurrences of the elementary productions overlap in one node. In the parallel production application this overlapping is handled implicitly whereas the overlapping is shown explicitly in the amalgamated production* amalg in 4.12.

Proof of theorem 4.4.1:

1. *Explicate*:

   Given a direct transformation $G \overset{\sum_1^n p_i}{\Longrightarrow}_d H$ (see figure 4.13).

   Since finite coproducts are pushout-stars over the initial object (compare for example [Mac71]) the coproducts over graph families (L_i), (I_i) and (R_i), $\forall\ 1 \leq i \leq n$, are equal (up to isomorphism) to pushout-stars of morphism stars $(L_i \longleftarrow \emptyset \longrightarrow L_j)_{1 \leq i < j \leq n}$, $(I_i \longleftarrow \emptyset \longrightarrow I_j)_{1 \leq i < j \leq n}$ and $(R_i \longleftarrow \emptyset \longrightarrow R_j)_{1 \leq i < j \leq n}$. All newly created diagram squares commute because of uniqueness of empty graph morphisms from $\emptyset$. Thus, there exists an amalgamated transformation $G \overset{COV'}{\Longrightarrow}_a H$. COV' contains $T' = (\mathcal{P}, \mathcal{SP}', E')$ with constant interfaces $\mathcal{SP}' = \{s_{ij} \mid s_{ij} = \emptyset,\ \forall\ 1 \leq i < j \leq n\}$ where $I \xrightarrow{l} L = \sum l$ and $I \xrightarrow{r} R = \sum r$ consisting of the diagram in figure 4.11 where all $L_{ij} = I_{ij} = R_{ij} = \emptyset$.

2. *Implicate*:

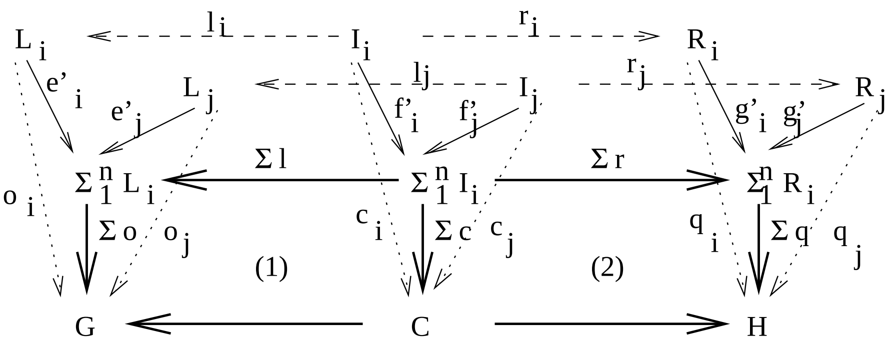

Figure 4.13: Direct transformation over parallel production $\sum_1^n p_i$

Let $G \overset{COV}{\Longrightarrow}_a H$ be an amalgamated transformation. We can construct first the coproducts containing $\sum_1^n L_i$, $\sum_1^n I_i$ and $\sum_1^n R_i$.

The morphism $\sum l$ ($\sum r$) is the induced morphism such that

$$I_i \xrightarrow{l_i} L_i \xrightarrow{e'_i} \sum_1^n L_i = I_i \xrightarrow{f'_i} \sum_1^n I_i \xrightarrow{\sum l} \sum_1^n L_i$$

$$(I_i \xrightarrow{r_i} R_i \xrightarrow{g'_i} \sum_1^n R_i = I_i \xrightarrow{f'_i} \sum_1^n I_i \xrightarrow{\sum r} \sum_1^n R_i)$$

$\forall 1 \leq i < j \leq n$. Moreover, there are induced morphisms $\sum_1^n L_i \xrightarrow{\sum o} G$, $\sum_1^n I_i \xrightarrow{\sum c} C$ and $\sum_1^n R_i \xrightarrow{\sum q} H$ such that

- $L_i \xrightarrow{e'_i} \sum_1^n L_i \xrightarrow{\sum o} G = L_i \xrightarrow{o_i} G = L_i \xrightarrow{e_i} L \xrightarrow{o} G$
- $I_i \xrightarrow{f'_i} \sum_1^n I_i \xrightarrow{\sum c} C = I_i \xrightarrow{c_i} C = I_i \xrightarrow{f_i} I \xrightarrow{c} C$ and
- $R_i \xrightarrow{g'_i} \sum_1^n R_i \xrightarrow{\sum q} H = R_i \xrightarrow{q_i} H = R_i \xrightarrow{g_i} R \xrightarrow{q} H, \ \forall\, 1 \leq i \leq n.$

If both squares (1) and (2) in figure 4.13 are pushouts the direct transformation $G \overset{\sum_1^n p_i}{\Longrightarrow}_{dd} H$ with $\sum_1^n p_i = (\sum_1^n L_i \xleftarrow{\sum l} \sum_1^n I_i \xrightarrow{\sum r} \sum_1^n R_i)$ is constructed.

Commutativity can be stated for (1) and (2) since the universal property of $\sum_1^n I_i$ holds for G and H.

Proving the universal property of (1) we can state the following:

Given a graph X with morphisms $\sum_1^n L_i \xrightarrow{ox'} X$ and $C \xrightarrow{cx} X$ satisfying $ox' \circ \sum l = cx \circ \sum c$ and define $ox_i = ox' \circ e'_i$ we can state $ox_i \circ e_{ij} = ox_j \circ e_{ji}$, $\forall 1 \leq i < j \leq n$.

$ox_i \circ e_{ij} = ox_i \circ e_{ij} \circ l_{ij} = ox_i \circ l_i \circ f_{ij} = ox' \circ e_i' \circ l_i \circ f_{ij} = ox' \circ \sum l \circ f_i' \circ f_{ij} = cx \circ \sum c \circ f_i' \circ f_{ij} = cx \circ c \circ f_i \circ f_{ij} = cx \circ c \circ f_j \circ f_{ji} = cx \circ \sum c \circ f_j' \circ f_{ji} = ox' \circ \sum l \circ f_j' \circ f_{ji} = ox' \circ e_j' \circ l_j \circ f_{ji} = ox_j \circ l_j \circ f_{ji} = ox_j \circ e_{ji} \circ l_{ij} = ox_j \circ e_{ji}$.

Remember that l_{ij} is an identity morphism since T has a constant interface.

Using pushout-star properties and $ox_i \circ l_i \circ f_{ij} = cx \circ c \circ c \circ f_i \circ f_{ij}$ we obtain $ox \circ l = cx \circ c$. Now, there exists a unique morphism $L \xrightarrow{ox} X$ with

$$L_i \xrightarrow{e_i} L \xrightarrow{ox} X = L_i \xrightarrow{ox_i} X = L_i \xrightarrow{e_i'} \sum_1^n L_i \xrightarrow{ox'} X,\ \forall\, 1 \leq i \leq n.$$

Since G is the gluing graph of $(L \xleftarrow{l} I \xrightarrow{c} C)$ there is a unique morphism $G \xrightarrow{gx} X$ with $L \xrightarrow{o} G \xrightarrow{gx} X = L \xrightarrow{ox} X$ and $C \xrightarrow{c_g} G \xrightarrow{gx} X = C \xrightarrow{cx} X$. Morphism $\sum ll : \sum_1^n L_i \to L$ is obtained as induced morphism from pushout-star $(L, (e_i')_{1 \leq i \leq n})$. Consequently, it follows

$\sum_1^n L_i \xrightarrow{ox'} X = \sum_1^n L_i \xrightarrow{\sum ll} L \xrightarrow{ox} X = \sum_1^n L_i \xrightarrow{\sum ll} L \xrightarrow{o} G \xrightarrow{gx} X =$

$\sum_1^n L_i \xrightarrow{\sum o} G \xrightarrow{gx} X$

The universal property of (2) can be proven analogously.

3. All colimit constructions are unique up to isomorphism. Applying the construction 'explicate' does not result into any graph or morphism newly constructed. By using construction 'implicate' the coproducts containing $\sum_1^n L_i$, $\sum_1^n I_i$ and $\sum_1^n R_i$ are reconstructed.

□

Corollary 4.4.3 (Analysis of special amalgamated transformations) *Given an amalgamated transformation $G \overset{COV}{\Longrightarrow}_a H$ with $COV = (T, O)$ being an injective partial covering where $T = (\mathcal{P}, \mathcal{SP}, E)$ is a component production set with constant interfaces. Then there is a sequentially independent transformation $G \overset{*}{\Longrightarrow}_d H$ of length n using the productions of $\mathcal{P}$.*

Proof: Direct consequence of theorems 4.4.1 and 4.1.19.

□

The 'explicate'-construction is not possible if the corresponding covering has to fulfill special properties. The following example shows a situation where 'explicate' is not possible if it has to yield a special covering.

Example 4.4.4 (Subsumption of graph languages) *1. Consider graph grammar $GG = (S, P, T, T)$ where $S = L$ and $P = \{p = (L \xleftarrow{l} \emptyset \xrightarrow{r} R)\}$ such that there are no matches $m : L \to R$ and only one possible match $id_L : L \to L$. Then the language $\mathcal{L}(GG^+) = \{L\} \cup \{\sum_1^n R | n \in N\}$ is mainly the set of all* *disjoint unions over R. This result is due to the fact that matches of L can overlap in L arbitrarily often.*

On the other side, let $PGG^{cc} = (S, \mathcal{A}^{cc}, T, T)$ be a parallel graph grammar with $\mathcal{A}^{cc} = ((P, \{\emptyset\}, \{\emptyset \to p\}), cc)$. All following languages are subsets of $\mathcal{L}(GG^{+})$.

$\mathcal{L}^a(PGG^{basic}) = \mathcal{L}^a(PGG^{sync}) = \mathcal{L}^a(PGG^{fast}) = \mathcal{L}^a(PGG^{diff}) =$

$\mathcal{L}^a(PGG^{all}) = \mathcal{L}^a(PGG^{locdiff}) = \mathcal{L}^a(PGG^{locall}) = \{L, R\}$. In all restricted coverings except the complete ones exactly one occurrence of L in S can be found because otherwise it is not

- *basic,*
- *fully synchronized: all occurrences have to overlap in L but the only possible interface is the empty production,*
- *fast: all occurrences have to overlap in L but there is not any preserved item*
- *a different-occurrence- or an all-occurrence-covering: there is only one possible occurrence $L \xrightarrow{id_L} L$,*
- *a local-different-occurrence- or a local-all-occurrence-covering with the same reason as above.*

2. *Let $P' = \{p' = (\emptyset \xleftarrow{l} \emptyset \xrightarrow{r} R)\}$ we have $\mathcal{L}((S, P'^{+}, T, T)) = \mathcal{L}(GG^{+})$. But $\mathcal{L}^a((S, \mathcal{A}', T, T)^{comp}) = \{L\}$ with $L \neq \emptyset$ and $\mathcal{A}' = ((P', \{\emptyset\}, \{\emptyset \to p'\}), \{comp\})$ since even a parallel application of p' would not yield a complete covering.*

Corollary 4.4.5 (Subsumption of graph languages) *Let all $\mathcal{IS}_u = (\mathcal{E}_u, \mathcal{S}_u, SE_u)$ with $u \in U$ be constant, $\mathcal{A}^{cc} = \{(\mathcal{IS}_u, cc) | u \in U\}$ and $P = \bigcup_{u \in U} \mathcal{E}_u$:*

1. *The generated languages of $PGG^{arb} = (S, \mathcal{A}^{arb}, T, T)$ and $GG^{+} = (S, P+, T, T)$ are equal, i.e. $\mathcal{L}^a(PGG^{arb}) = \mathcal{L}(GG^{+})$*

2. *Restricting the generated language of PGG^{cc} by using special coverings only, it is a proper sublanguage of $\mathcal{L}(GG+)$, i.e.*

   (a) *$\mathcal{L}^a((S, \mathcal{A}^{cc}, T, T)^{cc}) \subseteq \mathcal{L}((S, P^{+}, T, T))$ for $cc \in \{basic, comp, sync, fast, diff, all, locdiff, locall\}$*

   (b) *$\mathcal{L}^a((S, \mathcal{A}^{cc}, T, T)^{cc}) \neq \mathcal{L}((S, P^{+}, T))$ for some parallel graph grammars*

Proof: 1. Direct consequence of parallel replacement theorem I. 2.(a) is obvious because of 1. 2.(b) is shown by example 4.4.4. □

The following Parallel Replacement Theorem II shows the very close relation between amalgamated transformations and contextual parallel transformations. First, we give a small example for this relation.

Example 4.4.6 (Relation between different parallel transformation types) *Consider figures 2.10 for an example of an amalgamated transformation and 3.3 for an example of a contextual parallel transformation. Both transformations use the same partial covering consisting of two elementary productions* p1 *and* p2, *their common subproduction and and the parallel occurrence to graph* G.

Theorem 4.4.7 (Parallel Replacement Theorem II) *Let* COV *be a partial covering.*

1. Decapsulate: *Given an amalgamated transformation* $G\overset{COV}{\Longrightarrow}_a H$ *there is a contextual parallel transformation* $G\overset{COV}{\Longrightarrow}_{con} H$.
2. Encapsulate: *Given a contextual parallel transformation* $G\overset{COV}{\Longrightarrow}_{con} H$ *there is an amalgamated transformation* $G\overset{COV}{\Longrightarrow}_a H$.
3. Bijective Correspondence: *The constructions 'decapsulate' and 'encapsulate' are inverse to each other which means that there is a bijective correspondence between amalgamated transformations* $G\overset{COV}{\Longrightarrow}_a H$ *and contextual parallel transformations* $G\overset{COV}{\Longrightarrow}_{con} H$.

The proof of theorem 4.4.7 makes use of commutativity properties of colimits ([Mac71]) which are also used, for example, in the framework of module specifications in [EM90]. In our case we have to apply it to pushouts and pushout–stars which are special colimits.

Proof:

1. *Decapsulate*:

   Let $G\overset{COV}{\Longrightarrow}_a H$ be an amalgamated transformation as depicted in figure 4.11. Morphism stars $(L_i \xleftarrow{e_{ij}} L_{ij} \xrightarrow{e_{ji}} L_j)_{1\leq i<j\leq n}$,$(I_i \xleftarrow{f_{ij}} I_{ij} \xrightarrow{f_{ji}} I_j)_{1\leq i<j\leq n}$ and $(R_i \xleftarrow{g_{ij}} R_{ij} \xrightarrow{g_{ji}} R_j)_{1\leq i<j\leq n}$ can be extended to size $n+1$ such that their pushout-star graphs and the morphisms to these graphs are the same if $L_{i(n+1)} = L_{n+1} = I_{i(n+1)} = I_{n+1} = R_{i(n+1)} = R_{n+1} = \emptyset,\ \forall\, 1 \leq i \leq n$. (See lemma 4.4.9.) Thus, also

   - L is the POS-object of $LS = (L_i \xleftarrow{e_{ij}} L_{ij} \xrightarrow{e_{ji}} L_j)_{1\leq i<j\leq n+1}$,
   - I is the POS-object of $IS = (I_i \xleftarrow{f_{ij}} I_{ij} \xrightarrow{f_{ji}} I_j)_{1\leq i<j\leq n+1}$ and
   - R is the POS-object of $RS = (R_i \xleftarrow{g_{ij}} R_{ij} \xrightarrow{g_{ji}} R_j)_{1\leq i<j\leq n+1}$.

   Morphism $\emptyset \longrightarrow \emptyset$ is unique. Moreover, we have $\emptyset \longrightarrow \emptyset \longrightarrow L_i = \emptyset \longrightarrow L_i = \emptyset \longrightarrow I_i \xrightarrow{l_i} L_i$ because $\emptyset \longrightarrow L_i$ is unique and $\emptyset \longrightarrow \emptyset \longrightarrow R_i = \emptyset \longrightarrow R_i = \emptyset \longrightarrow I_i \xrightarrow{r_i} R_i$ because $\emptyset \longrightarrow R_i$ is unique.

There is a morphism star $CS = (I_i \xleftarrow{f_{ij}} I_{ij} \xrightarrow{f_{ji}} I_j)_{1\leq i<j\leq n+1}$ with $I_{i(n+1)} \xrightarrow{f_{i(n+1)}} I_i = I_i \xrightarrow{id_{I_i}} I_i$ and $I_{i(n+1)} \xrightarrow{f_{(n+1)i}} I_{n+1} = I_i \xrightarrow{c_i} C = I_i \xrightarrow{f_i} I \xrightarrow{c} C,\ \forall\, 1\leq i\leq n$, such that C is its POS-graph. (See lemma 4.4.10.)

All squares between IS and CS commute because of the uniqueness of morphisms $\emptyset \longrightarrow I_i,\ \forall\, 1\leq i\leq n$, and $\emptyset \longrightarrow C$ or identities.

Now the left part of the diagram just constructed consists of the solid part and the pushout over $L \xleftarrow{l} I \xrightarrow{c} C$ which is drawn dotted in figure 4.14. The right part of the diagram looks similarly.

Considering morphism stars LS, IS and CS with all morphisms running in between it is easy to see that this diagram is a functor from diagram scheme $PO \times POS_{n+1}$ to **GRAPH**. This applies in the same way for RS, IS and CS with corresponding morphisms. Thus, the property of commutativity of colimits implies first that the pushout of diagram $(L \xleftarrow{l} I \xrightarrow{c} C)$ and the pushout-star of diagram $ELS = (L_i \xleftarrow{e_{ij}} L_{ij} \xrightarrow{e_{ji}} L_j)_{1\leq i<j\leq n+1}$ with $L_{i(n+1)} \xrightarrow{e_{i(n+1)}} L_i = I_i \xrightarrow{l_i} L_i$ and $L_{i(n+1)} \xrightarrow{e_{(n+1)i}} L_{n+1} = I_i \xrightarrow{c_i} C,\ \forall\, 1\leq i\leq n$, are isomorphic and secondly that the pushout of diagram $(R \xleftarrow{r} I \xrightarrow{c} C)$ and the pushout-star of diagram $ERS = (R_i \xleftarrow{g_{ij}} R_{ij} \xrightarrow{g_{ji}} R_j)_{1\leq i<j\leq n+1}$ with $R_{i(n+1)} \xrightarrow{g_{i(n+1)}} R_i = I_i \xrightarrow{r_i} R_i$ and $R_{i(n+1)} \xrightarrow{g_{(n+1)i}} R_{n+1} = I_i \xrightarrow{c_i} C,\ \forall\, 1\leq i\leq n$, are isomorphic.

So there exists a contextual parallel transformation $G \overset{COV}{\Longrightarrow}_{con} H$ consisting of ELS, CS, ERS, the rule morphisms in between and pushout-stars of ELS and ERS.

2. *Encapsulate:*

   Given a contextual parallel transformation $G \overset{COV}{\Longrightarrow}_{con} H$ over $COV = (T, O)$ as in definition 4.2.14.

   Restricting the morphism stars of G, C and H and the interface-star to size n yields the morphism stars $LS = (L_i \xleftarrow{e_{ij}} L_{ij} \xrightarrow{e_{ji}} L_j)_{1\leq i<j\leq n}$, $IS = (I_i \xleftarrow{f_{ij}} I_{ij} \xrightarrow{f_{ji}} I_j)_{1\leq i<j\leq n}$ and $RS = (R_i \xleftarrow{g_{ij}} R_{ij} \xrightarrow{g_{ji}} R_j)_{1\leq i<j\leq n}$ with L being the POS-object of LS, I being the POS-object of IS and R being the POS-object of RS.

   Using lemma 4.4.9 we extend the stars again to size $n+1$ such that the pushout-star objects remain the same if we have $L_{i(n+1)} = L_{n+1} = I_{i(n+1)} = I_{n+1} = R_{i(n+1)} = R_{n+1} = \emptyset,\ \forall\, 1\leq i\leq n$.

   All morphisms between the stars of L and I and R and I are either given by assumption, identities or initial morphisms. Thus, all squares between the

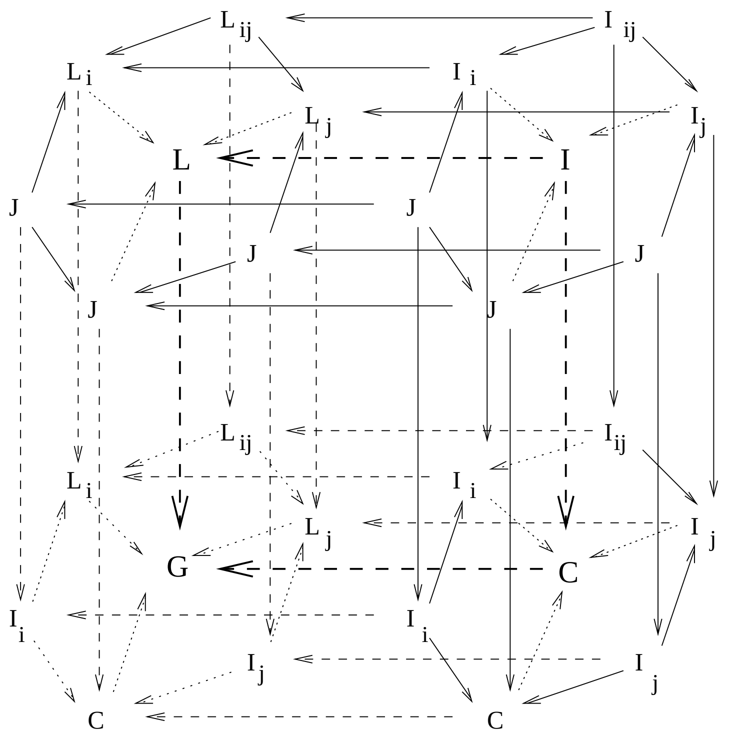

Figure 4.14: Commutativity of colimits

stars commute by assumption or because of the uniqueness of the corresponding morphisms. (This construction is very similar to its correspondent in the "decapsulate" step.)

Since the POS-graph of IS is I and due to the fact that we have

$$I_{ij} \xrightarrow{f_{ij}} I_i \xrightarrow{c_i} C = I_{ij} \xrightarrow{f_{ji}} I_j \xrightarrow{c_j} C, \ \forall\, 1 \leq i < j \leq n$$

because a contextual parallel transformation is a star-parallel transformation over a completed covering there is an induced morphism $I \xrightarrow{c} C$. According to the Context-Extending Lemma 4.4.10, C is the POS-graph of $(I_i \xleftarrow{f_{ij}} I_{ij} \xrightarrow{f_{ji}} I_j)_{1 \leq i < j \leq n+1}$ with $I_{i(n+1)} \xrightarrow{f_{i(n+1)}} I_i = I_i \xrightarrow{id_{I_i}} I_i$ and $I_{i(n+1)} \xrightarrow{f_{(n+1)i}} I_{n+1} = I_i \xrightarrow{c_i} C, \ \forall\, 1 \leq i \leq n$.

Now the left part of the diagram just constructed consists of the whole diagram in the figure 4.14 except of morphisms $L \xrightarrow{o} G$ and $C \xrightarrow{c_g} G$. The right side of the diagram looks similarly.

Similarly to the "decapsulate"-step we have the morphism stars of L, I, R and C and all morphisms in between which lead to diagrams over diagram scheme $PO \times POS_{n+1}$. Thus, the property of communtativity of colimits implies first that the pushout of diagram $(L \xleftarrow{l} I \xrightarrow{c} C)$ and the pushout-star of diagram$(L_i \xleftarrow{e_{ij}} L_{ij} \xrightarrow{e_{ji}} L_j)_{1\leq i<j\leq n+1}$ with $L_{i(n+1)} \xrightarrow{e_{i(n+1)}} L_i = I_i \xrightarrow{l_i} L_i$ and $L_{i(n+1)} \xrightarrow{e_{(n+1)i}} L_{n+1} = I_i \xrightarrow{c_i} C,\ \forall\, 1 \leq i \leq n$, are equal and secondly that the pushout of diagram $(R \xleftarrow{r} I \xrightarrow{c} C)$ and the pushout-star of diagram $(R_i \xleftarrow{g_{ij}} R_{ij} \xrightarrow{g_{ji}} R_j)_{1\leq i<j\leq n+1}$ with $R_{i(n+1)} \xrightarrow{g_{i(n+1)}} R_i = I_i \xrightarrow{r_i} R_i$ and $R_{i(n+1)} \xrightarrow{g_{(n+1)i}} R_{n+1} = I_i \xrightarrow{c_i} C,\ \ \forall\, 1 \leq i \leq n$, are equal. Hence, there is an amalgamated transformation $G \overset{COV}{\Longrightarrow}_a H$ as depicted in figure 4.11.

3. (Bijective Correspondence)

   All pushout-star constructions are unique up to isomorphism. Applying construction 'decapsulate' there is not any object or morphism that is newly constructed. By doing construction 'encapsulate' the pushout-stars of LS, IS, RS and CS with POS-graphs L, I, R and C are constructed and induce unique morphisms $I \xrightarrow{l} L$, $I \xrightarrow{r} R$ and $I \xrightarrow{c} C$.

$\square$

Corollary 4.4.8 (Extended parallel closure of graph languages) *Let* $cc \subseteq \{arb, basic, comp, sync, fast, diff, all, locdiff, locall\}$. *The generated languages of parallel graph grammars using amalgamated or contextual parallel transformations only are equal, i.e.*

$$\mathcal{L}^a(PGG^{cc}) = \mathcal{L}^{con}(PGG^{cc})$$

Proof: Direct consequence of parallel replacement theorem II. $\square$

For proving the theorems we need the following lemmata:

Lemma 4.4.9 (Initial-Extending-Lemma) *There is a pushout-star* $(G_i \xleftarrow{g_{ij}} G_{ij} \xrightarrow{g_{ji}} G_j)_{1\leq i<j\leq n+1}$ *with* $G_{i(n+1)} = G_{n+1} = \emptyset,\ \forall\ 1 \leq i \leq n$, *pushout-star graph* G *and morphisms* $G_i \xrightarrow{g_i} G,\ \ \forall\ 1 \leq i \leq n+1$, *if and only if there is a pushout-star* $(G_i \xleftarrow{g_{ij}} G_{ij} \xrightarrow{g_{ji}} G_j)_{1\leq i<j\leq n}$ *with the same pushout-star graph* G *and morphisms* g_i $\forall\, 1 \leq i \leq n$.

Proof:

1. Given a graph G being the POS-graph of the morphism star $(G_i \xleftarrow{g_{ij}} G_{ij} \xrightarrow{g_{ji}} G_j)_{1\leq i<j\leq n+1}$, we have $g_i \circ g_{ij} = g_j \circ g_{ji},\ \forall\, 1 \leq i < j \leq n$. There exists a unique morphism $G \xrightarrow{u} H$ for all graphs H with $G_{ij} \xrightarrow{g_{ij}} G_i \xrightarrow{g'_i} H = G_{ij} \xrightarrow{g_{ji}} G_j \xrightarrow{g'_j} H,\ \forall\, 1 \leq i < j \leq n+1$, and we have $u \circ g_i = g'_i,\ \forall\, 1 \leq i \leq n$. Morphism u is unique because $u \circ g_{n+1} = \emptyset \to H = g'_{n+1}$. This is exactly each H with $g'_i \circ g_{ij} = g'_j \circ g_{ji}$, $\forall\, 1 \leq i < j \leq n$, because $g'_i \circ g_{i(n+1)} = \emptyset \longrightarrow H = g'_{n+1} \circ g_{(n+1)i},\ \ \forall\, 1 \leq i \leq n$, is always satisfied. Hence G is POS-graph of $(G_i \xleftarrow{g_{ij}} G_{ij} \xrightarrow{g_{ji}} G_j)_{1\leq i<j\leq n}$.

2. Given an graph G being the POS-graph of the morphism star $(G_i \xleftarrow{g_{ij}} G_{ij} \xrightarrow{g_{ji}} G_j)_{1\leq i<j\leq n}$, we have $g_i \circ g_{ij} = g_j \circ g_{ji},\ \forall\, 1 \leq i < j \leq n$, and $\emptyset \longrightarrow G_i \xrightarrow{g_i} G = \emptyset \longrightarrow G = \emptyset \longrightarrow \emptyset \longrightarrow G,\ \forall\, 1 \leq i \leq n$, because morphism $\emptyset \longrightarrow G$ is unique.

   For all graphs H with $g'_i \circ g_{ij} = g'_j \circ g_{ji},\ \forall\, 1 \leq i < j \leq n$, and unique u as above we have $u \circ g_i = g'_i,\ \forall\, 1 \leq i \leq n+1$ because we have $\emptyset \longrightarrow G \xrightarrow{u} H = \emptyset \longrightarrow H$ because of the uniqueness of initial morphisms. Hence G is also POS-graph of $(G_i \xleftarrow{g_{ij}} G_{ij} \xrightarrow{g_{ji}} G_j)_{1\leq i<j\leq n+1}$.

□

Lemma 4.4.10 (Context-Extending-Lemma) *Given a graph I as POS-graph of a morphism star $(I_i \xleftarrow{f_{ij}} I_{ij} \xrightarrow{f_{ji}} I_j)_{1\leq i<j\leq n}$ and a morphism $I \xrightarrow{c} C$, graph C with morphisms $I_i \xrightarrow{c_i} C = I_i \xrightarrow{f_i} I \xrightarrow{c} C,\ \forall\, 1 \leq i \leq n$, and id_C, then C is POS-graph of morphism star $(I_i \xleftarrow{f_{ij}} I_{ij} \xrightarrow{f_{ji}} I_j)_{1\leq i<j\leq n+1}$ with morphisms $I_{i(n+1)} \xrightarrow{f_{i(n+1)}} I_i = I_i \xrightarrow{id_{I_i}} I_i$ and $I_{i(n+1)} \xrightarrow{f_{(n+1)i}} C = I_i \xrightarrow{c_i} C,\ \forall\, 1 \leq i \leq n$.*

Proof: We can state $c_i \circ f_{ij} = c_j \circ f_{ji},\ \forall\, 1 \leq i < j \leq n+1$, since

1. $c_i \circ f_{ij} = c \circ f_i \circ f_{ij} = c \circ f_j \circ f_{ji} = c_j \circ f_{ji},\ \forall\, 1 \leq i < j \leq n$ and
2. $c_i \circ id_{I_i} = c_i = id_C \circ c_i\ \ \forall\, 1 \leq i \leq n$

For all graphs C' with $I_{ij} \xrightarrow{f_{ij}} I_i \xrightarrow{c'_i} C' = I_{ij} \xrightarrow{f_{ji}} I_j \xrightarrow{c'_j} C',\ \forall\, 1 \leq i < j \leq n+1$, exists a morphism $C \xrightarrow{c'_{n+1}} C'$ by assumption with $c'_{n+1} \circ c_i = c'_{n+1} \circ c_i = c'_i \circ id_{I_i} = c'_i$. Assume that there are morphisms $\alpha, \beta : C \longrightarrow C'$ with $I_i \xrightarrow{c_i} C \xrightarrow{\alpha,\beta} C' = I_i \xrightarrow{c'_i} C'$, $\forall\, 1 \leq i \leq n$, it follows $\alpha = \alpha \circ id_C = x = \beta \circ id_C = \beta$.

□

Chapter 5

The Concepts of Distributed Graph Transformation

This chapter presents the main concepts of distributed graph transformation in an informal way. They will be formalized in chapter 6. Distributed states and actions are modeled by structured graphs and graph transformations in section 5.1. We consider three different kinds of distributed actions: local actions, network administration and communication as well as synchronization. An entire distributed system can be described by a distributed graph transformation system specifying all possible distributed actions.

In section 5.2 distributed graph transformation as presented in this chapter is compared with the graph transformation models for distributed systems presented in section 2.3.

Distributed graph transformation is introduced formally in chapter 6. Even readers not much familiar with graph transformation can get an operational understanding of distributed graph transformation after finishing this chapter.

5.1 Modeling Distributed States and Actions

In this section the main features for the modeling of distributed systems by algebraic graph transformations are presented. Distributed states are modeled by graphs and state transitions, i.e. combinations of local actions, network activities or communication are modeled by graph transformation.

5.1.1 Distributed States

A state of a system can be described by a graph where the nodes represent objects and the edges relations between them. If the state is a distributed one it can be reflected by several graphs where each of them shows a local state. In the following we call graphs describing a local state *local graphs*. The objects and relations in such a local state are called *local*, too.

Different local systems are usually connected by some kind of network. They interact with each other by some interfaces which are regarded as some special parts of the local systems with their own states and behavior. The interfaces as well as the local systems are allowed to change their states. This means that we have not only local states but also interface states and, moreover, network states. In the network states the current distribution structure of the whole system is stored. It changes if, for example, new local systems are added or connections are changed.

If we model the network structure of local systems by a *network graph* its nodes represent the local systems. These nodes are called *network nodes*. The edges of the network graph describe relations between local systems. These *network edges* can be understood as some kind of synchronization links that have to be held consistent. The whole distributed state is described by a so-called distributed graph.

Concept 5.1.1 *A* distributed graph*, short D-graph, consists of a network graph where each network node is equipped with a local graph and each network edge is equipped with a total graph morphism.*

In the following, the network nodes of a distributed graph are depicted by big ellipses and the network edges by thick arrows between them. Local graphs are drawn inside the ellipses. Local graph morphisms are represented by colors and layout positions of nodes where equally colored positioned alike in their local graphs are mapped to each others. Examples for distributed graphs are given in the next example 5.1.2.

The local graphs can represent current states of local systems whereas local graph morphisms describe the synchronization relation between two local states. These total graph morphisms are the essential basis for interaction of local systems. They are used to describe which local objects and relations correspond to each other in different local parts and have to be used for synchronization. Local graphs which are targets of such graph morphisms are called *target graphs*. Analogously, the sources of graph morphisms are called *source graphs* or *interface graphs*.

If two local parts have access to objects synchronized with each other communication can take place by exchanging information via these objects.

When modeling *object synchronization* by a D-graph, the set of all synchronized object nodes together with interrelation edges functions as interface graph. A structure equivalent copy of this interface graph can be found in all local graphs which are connected to it.

Example 5.1.2 (Object Synchronization) *This example shows how parts of a distributed system can base their communication on objects. To model object sharing in a distributed environment we simply place all objects which should be shared into a common interface and the local parts have replications of them. This can be described*

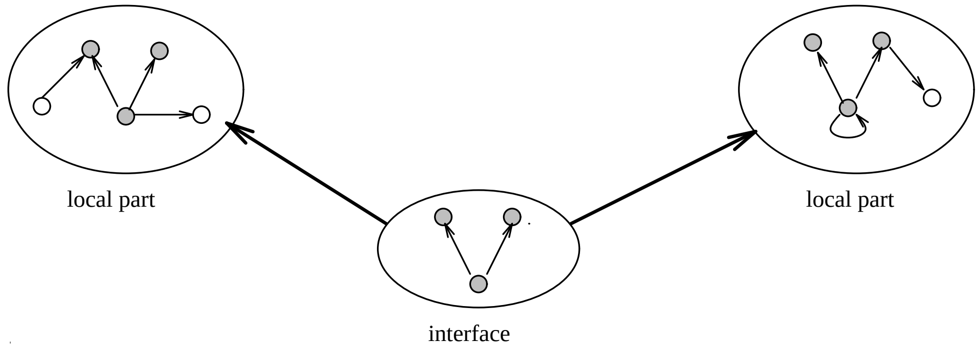

Figure 5.1: A communication interface for object sharing

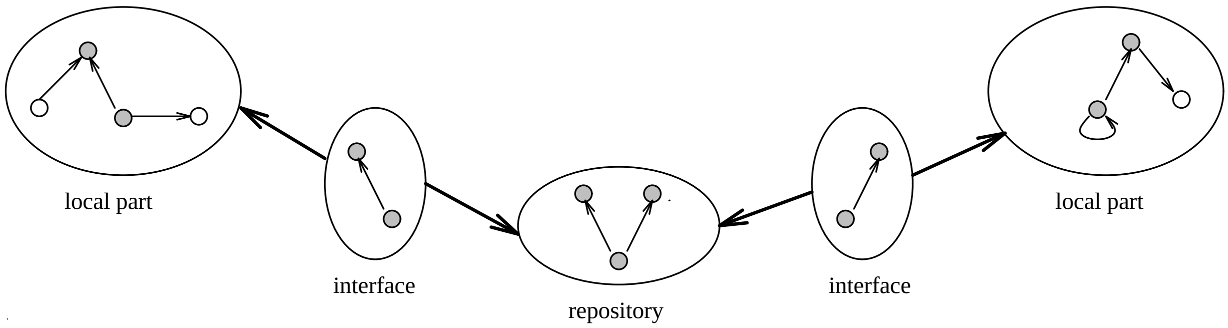

Figure 5.2: A repository for shared objects

by object synchronization as discussed above. Figure 5.1 illustrates this kind of object replication. All object nodes and relation edges which should be replicated are located in the interface graph. They are colored grey. The grey colored nodes and all edges in between in the local states describe replicated objects and relations. This is expressed by the graph morphisms going from the interface graph to the local graphs. The distribution structure is shown by the network graph depicted by the three big ellipses indicating network nodes and thick arrows in between.

By modeling object replication in this way it cannot be expressed that there may be objects which are allowed to be replicated but are currently not used by local parts. This idea can be kept by a kind of repository for shared objects. Regarding figure 5.2 the repository is modeled as a local part, too, and other local parts are connected to it by interface graphs. The left local part just uses the left two objects of the repository and the right local part the right two objects. Thus, in the interface graphs we can see which objects and relations are really used in the local parts. Only one object is really shared by the local parts in figure 5.2. This is the lower grey object. We can get this somewhat implicit information of objects really shared by the intersection of the two interface graphs in the repository graph.

In [EBHL88] and [EL93] distributed graphs with exactly two local graphs and one interface graph, i.e. a source graph which describes common parts are considered.

This notion is extended in [Tae96b] to an arbitrary number of local graphs where an interface graph has to be established between each two. In both approaches it is not possible to define *arbitrary distribution structures*, i.e. they are restricted to star graphs (see section 3.2). This means that e.g. interfaces between interfaces cannot be modeled.

5.1.2 Local Actions

Conceptually distributed actions are described by D-graph productions.

Concept 5.1.3 *A* distributed production*, short D-graph production, consists of a* network production *and* local productions *for all network nodes preserved by the network production. Local productions have to be compatible with local graph morphisms.*

A distributed production modeling a local action consists of a *local production* which describes the local action and a network production which is an identical production here, since the network graph is actually not changed.

Such a D-graph production can be applied to a D-graph if the local action is compatible with the context where it takes place. An action on a local graph which is a target graph is not permitted if it destroys the synchronization links into other local graphs. I.e. a source graph cannot be mapped totally to its target graphs any more. Analogously, source graphs are not allowed to be extended, because otherwise the additional structure is not properly connected to the context. These conditions are summarized in the *connection conditon* rephrased in table 5.1.

connection condition	
deletion	do not delete in interfaces without reflecting this in connected local graphs do not delete local nodes and edges which have preimages in interfaces
insertion	do not extend interfaces without reflecting this in connected local graphs

Table 5.1: The connection conditon

Applying a D-graph production with an identical network production, first the network production is applied to the current network graph. This just means that the matching of the only network node is replaced identically. The local productions are applied to that local graph which belongs to the matched network node. The local production can be applied if the gluing condition is satisfied for this local graph transformation. After the application the matched network node is equipped with a new graph, the transformed one.

Since local actions do not affect other local parts they can be executed simultaneously or in either order, i.e. in a true concurrent way.

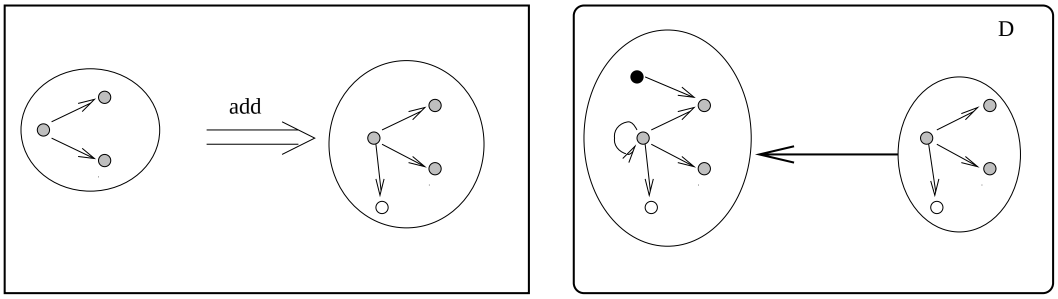

Figure 5.3: Local actions

Example 5.1.4 (Local actions) *A local action can be regarded as a simple transaction on object structures allowing an encapsulated set of atomic actions. Atomic actions are the creation, preservation, gluing and deletion of objects or relations. In this framework the gluing condition means that objects are not allowed to be deleted if there is some relation to it. And moreover, objects are only allowed to be glued if they are preserved.*

In figure 5.3 on the left-hand side a local action is modeled where an object is added. On the right hand side of the figure distributed graph D is shown where production "add" can be applied in different ways. Matches are indicated by using equal colors for object nodes. Production "add" can be applied to the left local graph of D since it does not model the deletion of a replicated object. This would affect the connection to the interface, too. Production "add" cannot be applied to the right local graph of D because in this case an object would be created in the interface which does not have a replication in the local part.

The inverse of production "add" which models the deletion of an object, cannot be applied to the right local graph of D either because its replication in the local part would loose the link to its origin. Moreover, the inverse of production "add" cannot be applied to the left local graph of D since then we get the situation that not all objects and relations of the interface have a replication in the local part. This kind of inconsistency is not allowed.

5.1.3 Network Administration

Modeling distributed systems by graph transformations includes mainly the description of network administration, i.e. changes of networks. They can be described by transformation of the network graph. The deletion and creation of network nodes and edges has to be done very cautious to avoid inconsistencies. Deletions must not destroy the synchronization reference structure. Creations have to fit into this reference structure. For example, a node in a new interface graph has to have a correspondent in all the local graphs it is an interface of. These conditons, summarized in the *network conditon*, are rephrased in more detail in table 5.2.

network condition	
deletion	a network node and its local graph have to be deleted as a whole do not delete a network edge if the local graph of its source node is not completely known
insertion	do not extend a local graph which becomes an interface

Table 5.2: The network conditon

If the network condition is satisfied the network actions can be performed concurrently, i.e. connections can be changed, new local parts can be inserted and other local parts (possibly with connections) can be deleted. Moreover, connections can be deleted and created, too.

Altogether a network administrating activity is also described by a D-graph production where its network production describes the changes of the network topology. For each network node which is preserved an identical production has to be applied to its local state graph. Each network node (edge) which is deleted or created is equipped with a local graph (morphism) on the left- or right-hand side of the D-graph production, respectively. The D-graph production can be applied if the network condition is satisfied. Moreover, the gluing condition has to be satisfied for the network part of the application.

Example 5.1.5 (Network administration) *Typical network administration is the creation or deletion of local parts or interfaces.*

In figure 5.4 different network administrating activities are described by productions. Production "delete" just describes the deletion of a local part. But it cannot be applied to distributed graph D in the upper right corner of figure 5.4 although an occurrence of the left hand side can be found twice. Deletion is cautious in the sense that everything which will be deleted has to be mentioned in the production. Thus, it cannot be applied to the local part of D. Furthermore, production "delete" cannot be applied to the interface graph of D since it has some reference to the other local graph. Thus, to delete the interface graph of D we have to look for an occurrence of production "delete in context" which mentions all the relations to replicated objects, too. The result is a distributed graph which contains the local part of D only.

The inverse production of "delete in context" is adding an interface depicted in the lower right corner of figure 5.4. Its application leads to a distributed graph with an additional interface graph at the left local graph of D. An object modeled in the left hand side of such a production indicates one which will be free for sharing afterwards.

5.1.4 Communication and Synchronization

Synchronization of different local actions is modeled by actions on network nodes connected with each other. In this case an action on a source graph has to be a

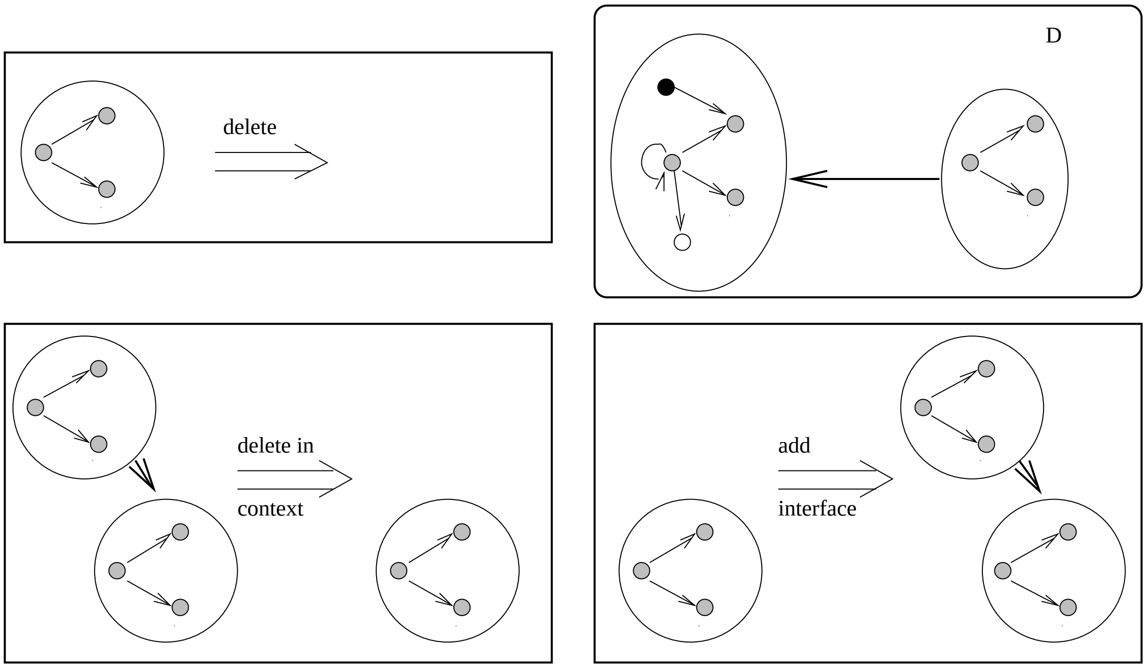

Figure 5.4: Network administration

subaction in some sense of those on all connected target graphs. This means roughly skeaking that the same actions on local items are allowed as in their adjoined local graphs. But in the source graph items may be deleted whereas some of their images in the corresponding target graph are preserved. Furthermore, a local item may be inserted in some source graph if its correspondents are preserved or inserted in all connected target graphs.

Synchronization is modeled by a D-graph production where the network production is an identical production, similarly to local actions. But here the whole network part which consists of more than one node is identically replaced. All nodes where a part of the synchronized action shall be performed have to occur in the network production. Moreover, all network edges which are used have to occur there. For each local part a local production, which models the action part of the synchronized action that has to be performed there, is be applied. As before, the application of a D-graph production must not destroy the synchronizing reference structure, i.e. the *connection* and the *network conditons* have to be satisfied. For each local production application the gluing condition has to be satisfied.

Asynchronous communication consists of at least two concurrent actions, i.e. producing and consuming some subgraph. They can be performed using some repository graphs (as in example 5.1.2). First one local part puts something into the repository and then the other one gets it. Considering the repository as an independent local part the other local parts have to synchronize themselves with the

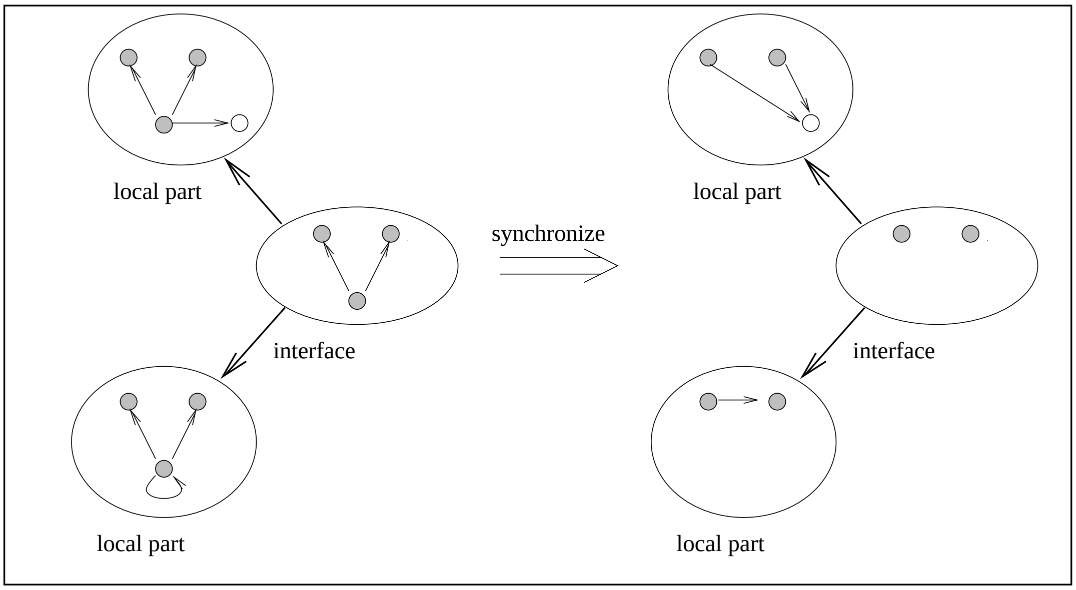

Figure 5.5: Synchronous communication

interface. Thus, D-graph productions model asynchronous communication on the basis of synchronous communication with interfaces.

Example 5.1.6 (Communication and Synchronization) *Changes in an interface automatically cause changes in the connected local parts. In figure 5.5 a D-graph production is depicted which describes the synchronization of two local parts via a common interface. Actions which are performed for the interface have to be executed for the local parts, too. Thus, after the synchronization a consistent state is reached again. This production can be applied for example, to the distributed graph in figure 5.1.*

Asynchronous communication between two local parts can be expressed by synchronization with their common communication medium. Looking at figure 5.2 both local parts have to synchronize themselves with the repository, one for sending and one for receiving.

In figure 5.6 two productions are given performing sending and receiving between two local parts. First local objects are copied to the repository and then they are further copied to the local part. Production "send" can be applied independently of production "receive" but not vice versa. Thus, there is the natural dependency of "receive" from "send" but "receive" can be performed some time later.

5.1.5 Distributed Systems

Starting with some network topology and local initializations the initial state of a distributed system can be described by a D-graph. Local actions, network activities,

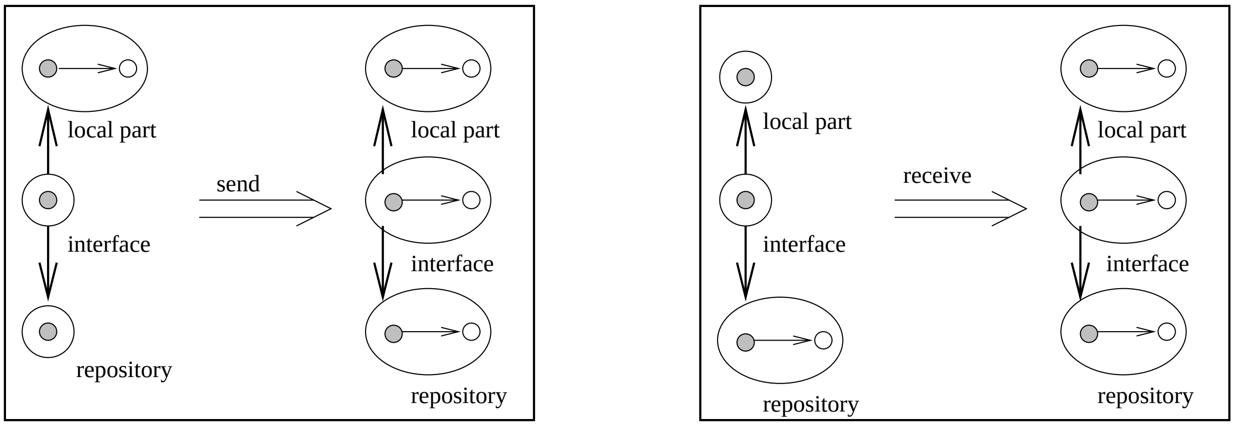

Figure 5.6: Asynchronous communication

some kind of communicating actions and, moreover, also mixtures of these actions are allowed as state transitions. The whole distributed system is described by a distributed graph grammar.

Concept 5.1.7 *A* distributed graph grammar*, short D-graph grammar, consists of a D-graph, the* start graph*, and a set of D-graph productions.*

The start graph models the intial state of a distributed system whereas the D-graph productions model distributed actions in the system. Distributed actions are described by the application of distributed productions satisfying the distributed gluing conditon.

Concept 5.1.8 *The* distributed gluing condition *consists of the gluing conditons for the network and all local transformations, the connection and the network condition.*

Each local system executes actions sequentially. Distributed actions are allowed to be performed simultanously if at most one action takes place in each local part.

Example 5.1.9 (Distributed actions) *In a distributed system it may happen that a "send" takes place somewhere whereas an interface is deleted somewhere else. This concurrent action can be modeled by applying D-graph productions "send" and "delete in context" in parallel. The distributed action is depicted in figures 5.6 and 5.4.*

Applying this combination to the left distributed graph in figure 5.7 we get the right distributed graph in this figure where an object (indicated by a white node) is sent to the repository and a useless interface is deleted.

All local states and connections which are not used for these actions are not affected like the right interface and its relations to the local part and the repository.

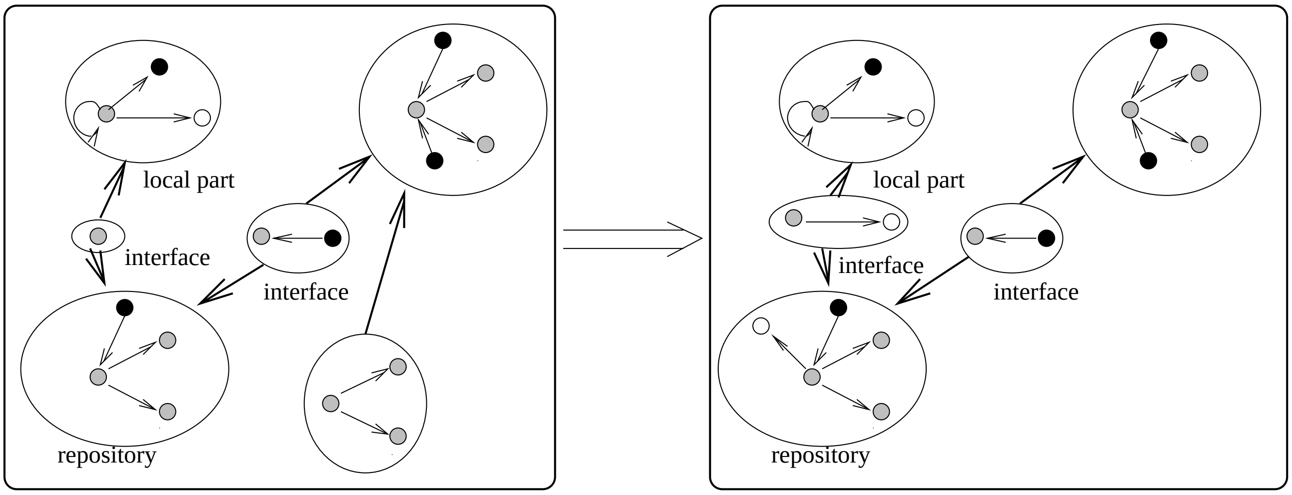

Figure 5.7: Distributed actions

5.2 Comparison with Related Graph Transformation Concepts

The concepts of distributed graph transformation introduced in this chapter are compared with the graph transformation models for distributed systems reviewed in section 2.3 along the requirements for modeling support in chapter 1. Distributed graph transformation combines most of the features offered by the other graph transformation models presented in chapter 2.3. Moreover, it offers additional ones essential for distributed system design which are applied in section 7.2 where the main basics for distributed software engineering are modeled. Distributed graph transformation is compared with other design techniques for distributed systems in section 7.4.

Parallelism Distributed graph transformation allows graph transformation on two levels: the network level to describe dynamic network structures and the local level to model transformation of object structures. Thus, parallelism can be modeled on both levels by allowing the parallel application of productions. On the network level large-grain parallelism is supported by the application of several distributed productions in parallel. Fine-grain parallelism may occur on the local level allowing parallel productions for the description of local actions executed in parallel. Both kinds of parallelism are problem-oriented since arbitrary structured distributed productions may be applied.

These two kinds of parallelism are also incorporated in the categorical approach (see subsection 2.3.3) whereas all other graph transformation models in section 2.3 support mostly the network level parallelism. The algebraic approaches (see subsection 2.3.2) offers large-grain parallelism in a very restricted form due to the restricted type of distribution structures and operations. But on the local level they offer the parallel execution power of graph transformation.

Communication and synchronization Distributed graph transformation supports communication and synchronization by data/object sharing. It is modeled by different interface graphs containing the shared structure. Describing data/object sharing by a distributed graph, its graph morphisms model a generalized "contains" relation between local graphs which determines the shared parts. The use of graph morphisms for the description of sharing is derived from the algebraic approach as presented in subsection 2.3.2. This idea is not incorporated in the categorical approach where local graphs cannot be directly related by graph morphisms.

The properties of communication by sharing are also inherent to the distributed graph transformation model. It supports anonymous communication with broadcasting facilities. Communication by sharing is usually asynchronous, i.e. one action creates an object which is consumed by another one. This feature can easily cause inconsistencies. In this model, inconsistencies cannot occur since the distributed gluing condition has to obeyed. Distributed graph transformation can model synchronous as well as asynchronous communication by amalgamation and sharing (compare example 5.1.6.)

Distribution and dynamic network structures Similarly to the categorical approach, nearly the full description power of graph transformation is available for modeling dynamic network structures. It is restricted to non-gluing and non-splitting operations on the network graph, as the categorical approach. Considering the algebraic approach splitting and gluing (or joining) of network nodes are the only operations on network structures, i.e. these network operations are complementary to those allowed in the categorical approach or distributed graph transformation as presented here.

Altogether, distributed graph transformation as presented in this chapter provides graph transformation on two levels of abstraction. Thus, all distributed system models without abstraction concepts (as presented in subsection 2.3.1) are capable by this new formalism. The algebraic approach presented in subsection 2.3.2 is somewhat incomparable because it relies on special distribution operations "split" and "join" instead of graph transformation, but graph transformation provides a much higher flexibility and natural approach for the formulation of network dynamics. The categorical approach also provides graph transformation on two levels of abstraction but do not offer the concept of interface graphs. Thus, local graph transformations are executed independently of each others and may not be used for synchronization. Since this feature is provided by the algebraic approach, distributed graph transformation may be regarded as a combination of the algebraic and the categorical approaches.

Chapter 6

Formal Description of Distributed Graph Transformation

Distributed graph transformation informally introduced in the previous chapter is formally defined in this chapter based on the double-pushout approach to graph transformation. A short version is presented in [Tae96a][1].

The double-pushout approach has been reviewed in section 4.1. This approach to graph transformation seems to be appropriate to be extended by distribution concepts since it provides some kind of cautious graph rewriting. This means that the structure of distributed graphs is not destroyed during transformation and side-effects are not performed.

Distributed graph transformation where a distributed structure of graphs is transformed by local graph transformations can be characterized by a double-pushout over distributed graphs and graph morphisms, i.e. in the category **DISTR(GRAPH)**. The formal definition of distributed graph transformation can be found in section 6.2. Section 6.1 provides the necessary operations on distributed graphs which are mainly the pushout construction. A pushout over distributed graph morphisms cannot always be constructed componentwise. But especially the componentwise construction of a distributed graph transformation seems to reflect distributed computations best. Thus, we present the necessary conditons for componentwise construction of pushouts summarized in the *distributed gluing condition*.

Distributed graph transformation is compared with parallel graph transformation in the *Distribution Theorem* 6.3.3 in section 6.3. It turns out that both kinds of graph transformation are closely related if parallel graph transformation is extended by additional distribution structure compatible with the application structure of parallel productions.

Readers of this chapter are assumed to be familiar with basic notions of functor categories as presented in e.g. [Mac71] or [HS73].

[1] "Hierarchically distributed graph transformation" presented in [Tae96a] is called "distributed graph transformation" in this thesis.

6.1 Distributed Graphs and Graph Operations

Distributed states as presented in section 5.1 can be formally described by distributed graphs introduced below. To be able to describe distributed graph transformation formally we introduce a suitable kind of morphisms, the distributed graph morphisms. Furthermore, we need the construction of a pushout of distributed graph morphisms to characterize gluing facilities for distributed graphs. In this section some conditions are given to ensure the existence of a componentwise construction of such a pushout. These conditions are sufficient and necessary.

Definition 6.1.1 (D-graph) *Let* **GRAPH** *be the category of labeled graphs and (total) graph morphisms which are label preserving (as presented in fact 4.1.3. Given a graph G of* **GRAPH** *which is called* network graph *a functor $\hat{G} : G \to$* **GRAPH** *is called* distributed graph (D-graph)[2]*. D-graph $\hat{G}$ is said to* be of shape G.

A local graph is denoted $\hat{G}(i)$ for a network node $i \in G^N$. Given a network edge $e \in G^E$ the local graph $\hat{G}(s(e))$[3] is an interface graph or source graph, $\hat{G}(t(e))$ is called target graph and $\hat{G}(e)$ is a total graph morphism.

Example 6.1.2 (D-graph) *In figure 5.2 a D-graph $\hat{G} : G \to$* **GRAPH** *is shown. Graph G consists of all big ellipses with solidly drawn edges in between. Let i_1 be the left "interface"-node, its local graph which is drawn inside the ellipse is called $\hat{G}(i_1)$. Correspondingly the local graph of node r labeled by "repository" is called $\hat{G}(r)$. The network edge e with source node i_1 and target node r is equipped with a graph morphism $\hat{G}(e) : \hat{G}(i_1) \to \hat{G}(r)$ coded into the fill styles and positions of local nodes, i.e. nodes which are mapped to each other are filled and positioned alike. Local edges are mapped in a structure compatible way.*

Definition 6.1.3 (D-graph morphism) *Given two D-graphs $\hat{G}$ and $\hat{H}$ a distributed graph morphism, short* D-graph morphism *$\hat{f} = (\eta, f) : \hat{G} \to \hat{H}$ is a natural transformation $\eta = (\hat{f}_i)_{i \in G^N} : \hat{G} \to \hat{H} \circ f$ in* **GRAPH** *with $f : G \to H$ being a graph morphism of* **GRAPH***, called* network morphism.

If f is injective $\hat{f}$ is called n-injective*. If moreover, all $\hat{f}_i$ with $i \in G^N$ are injective, $\hat{f}$ is called* injective*, too.*

A D-graph morphism $\hat{f}$ assigns to each node $i \in G^N$ a graph morphism $\hat{f}_i : \hat{G}(i) \to \hat{H}(f(i))$, called **local graph morphism** such that $\forall e \in G^E$ where $s(e) = i$ and $t(e) = j$ the diagram in figure 6.1 commutes. More concretely, given two local graphs $\hat{G}(i)$ and $\hat{G}(j)$ which are connected by $\hat{G}(e)$ and these graphs are mapped by local graph morphisms $\hat{f}_i$ and $\hat{f}_j$ to graphs $\hat{H}(f(i))$ and $\hat{H}(f(j))$ the connection $\hat{G}(e)$ has to be adapted leading to $\hat{H}(f(e))$.

[2] Graph G and its induced small category are identified.

[3] In the following, it is always clear which source or target mapping is meant. Thus, we do not write $\hat{G}(s_G(e))$ $(\hat{G}(t_G(e)))$ but simpler $\hat{G}(s(e))$ $(\hat{G}(t(e)))$.

$$\begin{tikzcd} i \arrow[d, "e"'] & \hat{G}(i) \arrow[r, "\hat{f}_i"] \arrow[d, "\hat{G}(e)"'] \arrow[dr, phantom, "(=)"] & \hat{H}(f(i)) \arrow[d, "\hat{H}(f(e))"] \\ j & \hat{G}(j) \arrow[r, "\hat{f}_j"'] & \hat{H}(f(j)) \end{tikzcd}$$

Figure 6.1: D-graph morphism

$$\begin{tikzcd} i \arrow[d, "e"'] & \hat{G}(i) \arrow[r, "\hat{g}_i"] \arrow[d, "\hat{G}(e)"'] & \hat{K}(g(i)) \arrow[r, "\hat{h}_{g(i)}"] \arrow[d, "\hat{K}(g(e))"'] & \hat{H}(h(g(i))) \arrow[d, "\hat{H}(h(g(e)))"] \\ j & \hat{G}(j) \arrow[r, "\hat{g}_j"] & \hat{K}(g(j)) \arrow[r, "\hat{h}_{g(j)}"] & \hat{H}(h(g(j))) \end{tikzcd}$$

Figure 6.2: Composition of D-graph morphisms

Example 6.1.4 (D-graph morphism) *D-graph morphisms are shown in figures 5.3 on the left, 5.4 in the lower right corner and 5.6. In all figures the network morphisms are not explicitly shown but given by the layout of the source and target graphs. Nodes and edges at corresponding places are mapped to each others.*

Definition 6.1.5 (Composition of D-graph morphisms) *The* composition $\hat{f} = (\eta_f, f) : \hat{G} \to \hat{H}$ *of two D-graph morphisms* $\hat{g} = (\eta_g, g) : \hat{G} \to \hat{K}$ *and* $\hat{h} = (\eta_h, h) : \hat{K} \to \hat{H}$ *is defined by* $f = h \circ g$ *and* $\eta_f = (\eta_h \circ g) \circ \eta_g : \hat{G} \to \hat{K} \circ g \to \hat{H} \circ h \circ g$.

Proposition 6.1.6 (Category DISTR(GRAPH)) *All D-graphs and D-graph morphisms as defined above form a category* **DISTR(GRAPH)**.

Proof: The composition $\hat{f} : \hat{G} \to \hat{H}$ of two D-graph morphisms $\hat{g} : \hat{G} \to \hat{K}$ and $\hat{h} : \hat{K} \to \hat{H}$ is again a D-graph morphism. According to definition 6.1.5 we have $f_i = h_{g(i)} \circ g_i$ for all $i \in G^N$, and forall $e \in G^E$ with $s_G(e) = i$ and $t_G(e) = j$ the diagrams in figure 6.2 commute. Since the composition of D-graph morphisms is obtained by that of graph morphisms it is associative.

The identities $\hat{id}_{\hat{G}} = (id_G, \eta_{id_G}) : \hat{G} \to \hat{G}$ for each D-graph $\hat{G}$ are provided by the identities $\hat{id}_{\hat{G}(i)} : \hat{G}(i) \to \hat{G}(i)$ for all $i \in G^N$. $\hat{id}_{\hat{G}} \circ \hat{g} = \hat{g}$ and $\hat{f} \circ \hat{id}_{\hat{G}} = \hat{f}$ is satisfied for all D-graph morphisms $\hat{f} = (f, \eta_f) : \hat{G} \to \hat{H}$ and $\hat{g} = (g, \eta_g) : \hat{H} \to \hat{G}$, since $id_G \circ g = g$ and $\eta_g = (\eta_{id_G} \circ g) \circ \eta_g : \hat{H} \to \hat{G} \circ g \to \hat{G} \circ g$. Moreover, $f \circ id_G = f$ and $\eta_f = \eta_f \circ \eta_{id_G} : \hat{G} \to \hat{G} \to \hat{H} \circ f$.

□

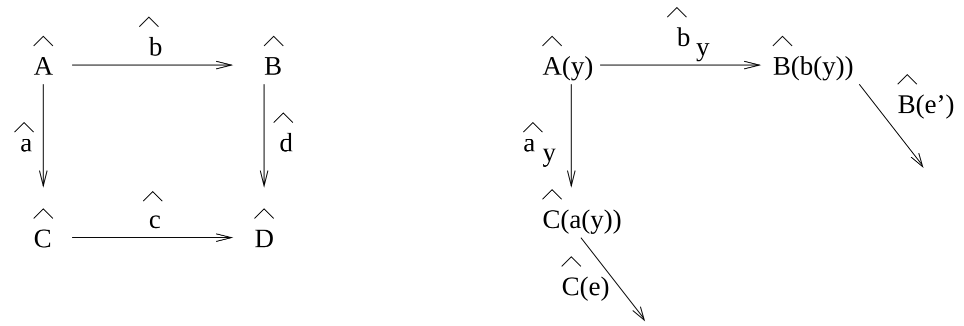

Figure 6.3: Illustration of PO-conditions

In the following we do not consider the existence of pushouts in the category **DISTR(GRAPH)** in general, but concentrate on those over n-injective morphisms which can be constructed componentwise. We will show that a componentwise construction of such a pushout exists if the following pushout conditions are satisfied. These conditions require for each network edge which has a source network node that is glued with another, that the local graph of this network node is isomorphically mapped into the pushout graph. The general pushout situation with given morphisms $\hat{a}$ and $\hat{b}$ is depicted in figure 6.3.

Definition 6.1.7 (Pushout Conditions (1) and (2)) *Two n-injective D-graph morphisms $\hat{a} : \hat{A} \to \hat{C}$ and $\hat{b} : \hat{A} \to \hat{B}$ satisfy the* pushout conditions *(1) and (2) if*
(1) $\forall e \in C^E - a(A^E) \forall y \in A^N : a(y) = s(e) \Longrightarrow \hat{b}_y$ is bijective and
(2) $\forall e \in B^E - b(A^E) \forall y \in A^N : b(y) = s(e) \Longrightarrow \hat{a}_y$ is bijective.
(Compare the right diagram in figure 6.3.)

Example 6.1.8 (Pushout Conditions (1) and (2)) *In figure 6.4 a situation is shown where pushout condition (1) is not satisfied. All local graph morphisms are depicted by dashed arrows between corresponding nodes. The mapping of edges can be uniquely derived. The network morphisms are not explicitly shown.*

Morphism $\hat{b}_y$ is not bijective. The node with a loop and its adjacent edges in $\hat{B}(b(y))$ which are copied into $\hat{D}(c(a(y)))$ cannot be mapped by $\hat{D}(c(e))$.

Proposition 6.1.9 (Existence of Pushouts) *Given two n-injective D-graph morphisms $\hat{a} : \hat{A} \to \hat{C}$ and $\hat{b} : \hat{A} \to \hat{B}$ which satisfy pushout conditions (1) and (2) the pushout of $\hat{a}$ and $\hat{b}$ in* **DISTR(GRAPH)** *exists and can be constructed componentwise.*

Proof: Let D with $c : C \to D$ and $d : B \to D$ be the pushout of a and b in **GRAPH**. (For the construction compare definition 4.1.2.) Pushout graph $\hat{D} : D \to$ **GRAPH** is constructed in the following way:

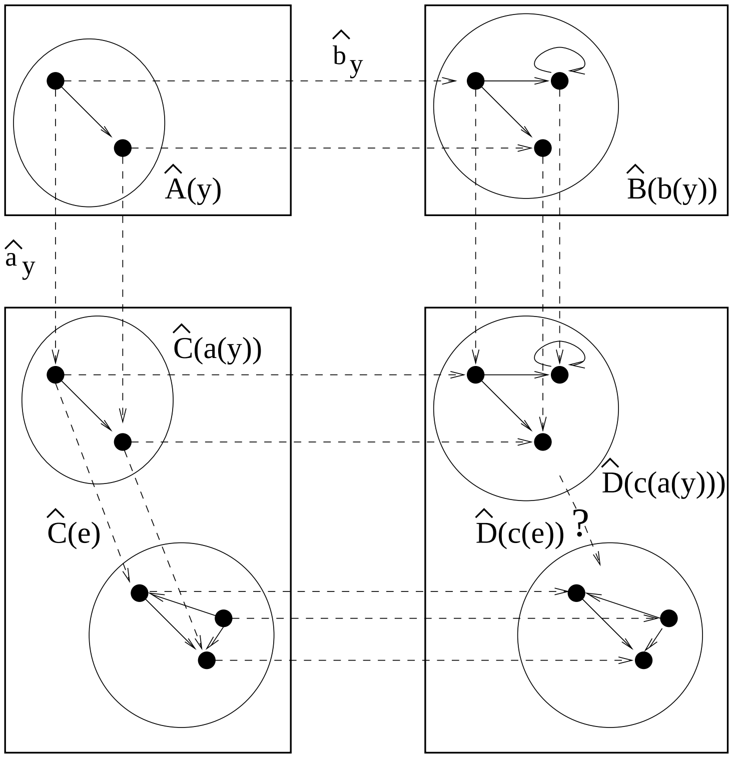

Figure 6.4: A gluing situation not allowed

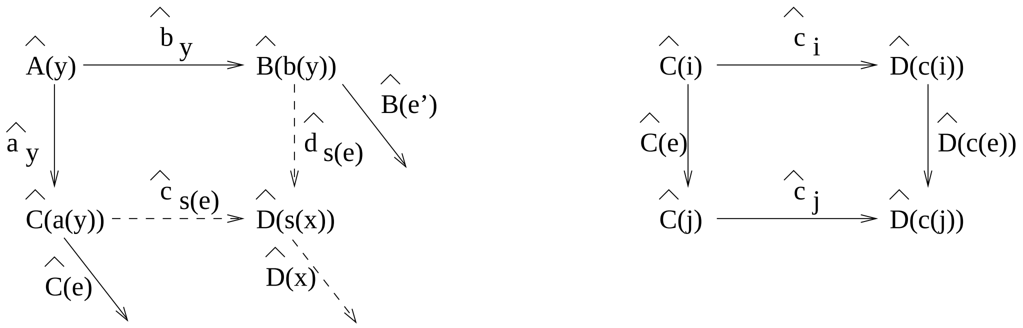

Figure 6.5: Illustrations for the proof of 6.1.9

$\forall x \in D^N$ there is at most one $y \in A^N$ with $c \circ a(y) = x$ because a and b are injective. Thus we can uniquely name $PO(\hat{a}_y, \hat{b}_y)$[4] by PO_x.

$$\hat{D}(x) := \begin{cases} PO_x & \text{, if } \exists y \in A^N \text{ with } c \circ a(y) = x \\ \hat{C}(z) & \text{, if } \exists z \in C^N - a(A^N) \text{ with } c(z) = x \\ \hat{B}(v) & \text{, if } \exists v \in B^N - b(A^N) \text{ with } d(v) = x \\ IND(PO_{s(x)}, PO_{t(x)}) & \text{, if } \exists e \in A^E \text{ with } c \circ a(e) = x \\ \hat{c}_{t(x)} \circ \hat{C}(e) \circ \hat{c}^{-1}_{s(x)} & \text{, if } \exists e \in C^E - a(A^E) \text{ with } c(e) = x \\ \hat{d}_{t(x)} \circ \hat{B}(e) \circ \hat{d}^{-1}_{s(x)} & \text{, if } \exists e \in B^E - b(A^E) \text{ with } d(e) = x \end{cases}$$

where $IND(PO_{s(x)}, PO_{t(x)})$ is the induced morphism from pushout graph $\hat{D}(s(x))$ to pushout graph $\hat{D}(t(x))$ with $\hat{a}_{t(e)} \circ \hat{A}(e) = \hat{C}(a(e)) \circ \hat{a}_{s(e)}$ and $\hat{b}_{t(e)} \circ \hat{A}(e) = \hat{B}(b(e)) \circ \hat{b}_{s(e)}$. Pushout morphisms $\hat{c} : \hat{C} \to \hat{D}$ and $\hat{d} : \hat{B} \to \hat{D}$ are defined as follows:

$$\hat{c}_x := \begin{cases} id_{\hat{C}(x)} & \text{, if } x \in C^N - a(A^N) \\ \text{PO-morphism of } PO_{c(x)} & \text{, otherwise} \end{cases}$$

$$\hat{d}_x := \begin{cases} id_{\hat{B}(x)} & \text{, if } x \in B^N - b(A^N) \\ \text{PO-morphism of } PO_{d(x)} & \text{, otherwise} \end{cases}$$

It is straightforward to show that $\hat{D}$ is a D-graph since for each node x of D^N there is a graph $\hat{D}(x)$ and for each edge e of D^E there is a graph morphism from the source graph $\hat{D}(s(e))$ to the target graph $\hat{D}(t(e))$. For all $e \in C^E - a(A^E)$ with $c(e) = x$ (for all $e \in B^E - b(A^E)$ with $d(e) = x$) morphisms $\hat{c}^{-1}_{s(x)}$ ($\hat{d}^{-1}_{s(x)}$) are isomorphisms either because of PO-properties and the pushout conditions or since $\hat{c}_{s(x)}$ ($\hat{d}_{s(x)}$) is an identity according to the definition of $\hat{c}$ ($\hat{d}$).

It is next to show that $\hat{c}$ as well as $\hat{d}$ are D-graph morphisms. First of all there is a unique graph morphism $\hat{c}_x$ for all $x \in C^N$ and $\hat{d}_y$ for all $y \in B^N$. Assuming $s(e) = i$ and $t(e) = j$, we have $\hat{c}_j \circ \hat{C}(e) = \hat{D}(c(e)) \circ \hat{c}_i$ since $\hat{D}(c(e)) = IND(PO_{c(i)}, PO_{c(j)})$ holds for all $e \in a(A^E)$. For all $e \in C^E - a(A^E)$, $c_j \circ \hat{C}(e) = \hat{D}(c(e))$ holds because

[4] $PO(\hat{a}_y, \hat{b}_y)$ is the pushout graph of the pushout over $\hat{a}_y$ and $\hat{b}_y$

$\hat{c}_i$ is an isomorphism. Analogously, we get $\hat{d}_j \circ \hat{B}(e) = \hat{D}(d(e)) \circ \hat{d}_i$. (For this part of the proof compare the right diagram in figure 6.5.)

Now, the pushout properties have to be shown. The commutativity $\hat{c} \circ \hat{a} = \hat{d} \circ \hat{b}$ follows directly from the construction, i.e. for all $x \in A^N$ there are pushout diagrams such that we have $\hat{c}_{a(x)} \circ \hat{a}_x = \hat{d}_{b(x)} \circ \hat{b}_x$.

Given a D-graph $\hat{X}$ as well as two D-graph morphisms $\hat{f} : \hat{B} \to \hat{X}$ and $\hat{g} : \hat{C} \to \hat{X}$ with $\hat{f} \circ \hat{b} = \hat{g} \circ \hat{a}$ the induced morphism $\hat{u}$ for the universal property consists of the induced morphisms for the underlying pushouts in **GRAPH** where they exist, i.e. for all $y \in A^N$ with $c \circ a(y) = x$, $\hat{u}_x$ is the induced morphism wrt. PO_x. For all $z \in C^N - a(A^N)$ with $c(z) = x$ ($v \in B^N - b(A^N)$ with $d(v) = x$), $\hat{u}_x = \hat{g}_z$ ($\hat{u}_x = \hat{f}_v$).

Well-definedness of $\hat{u}$ follows from the universal property of local pushouts and the pushout conditions. We have to show that $\hat{u}$ is totally defined which is the case since c and d are jointly surjective. Thus, there is a morphism u_x for all $x \in D^N$. Furthermore, $\hat{u}_j \circ \hat{D}(e) = \hat{X}(u(e)) \circ \hat{u}_i$ with $s(e) = i$ and $t(e) = j$ has to be shown. For the proof the following three cases have to be distinguished. (For this part of the proof compare the diagram in figure 6.6.)

1. $\forall e \in c \circ a(A^E)$: $\hat{u}_i$ and $\hat{u}_j$ are induced morphisms. Let $v, w \in B^N$ with $d(v) = i$ and $d(w) = j$. Thus, $\hat{u}_i \circ \hat{d}_v = \hat{f}_v$ and $\hat{u}_j \circ \hat{d}_w = \hat{f}_w$. On the other side let $y, z \in C^N$ with $c(y) = i$ and $c(z) = j$. Thus, $\hat{u}_i \circ \hat{c}_y = \hat{g}_y$ and $\hat{u}_j \circ \hat{c}_z = \hat{g}_z$. Let $e' \in C^E$ with $c(e') = e$ and $e'' \in B^E$ with $d(e'') = e$. Then, we have $\hat{g}_z \circ \hat{C}(e') \circ \hat{a}_x = \hat{u}_j \circ \hat{c}_z \circ \hat{C}(e') \circ \hat{a}_x = \hat{u}_j \circ \hat{D}(e) \circ \hat{c}_y \circ \hat{a}_x = \hat{u}_j \circ \hat{D}(e) \circ \hat{d}_v \circ \hat{b}_x = \hat{u}_j \circ \hat{d}_w \circ \hat{B}(e'') \circ \hat{b}_x = \hat{f}_w \circ \hat{B}(e'') \circ \hat{b}_x$. Hence, there is an induced morphism $m : \hat{D}(i) \to \hat{X}(u(j))$ which is unique according to $\hat{g}_z \circ \hat{C}(e')$ and $\hat{f}_w \circ \hat{B}(e'')$. Thus, we obtain $\hat{u}_j \circ \hat{D}(e) = m = \hat{X}(u(e)) \circ \hat{u}_i$.

2. $\forall e \in c(C^E - a(A^E))$: Let $e' \in C^E$ with $c(e') = e$ then $\hat{u}_j \circ \hat{D}(e) = \hat{u}_j \circ \hat{c}_{t(e')} \circ \hat{C}(e') \circ \hat{c}^{-1}_{s(e')} = \hat{g}_{t(e')} \circ \hat{C}(e') \circ \hat{c}^{-1}_{s(e')} = \hat{X}(g(e')) \circ \hat{g}_{s(e')} \circ \hat{c}^{-1}_{s(e')} = \hat{X}(g(e')) \circ \hat{u}_i \circ \hat{c}_{s(e')} \circ \hat{c}^{-1}_{s(e')} = \hat{X}(u(e)) \circ \hat{u}_i$ because of the definition of $\hat{D}(e)$ where $\hat{u}_j$ is either induced or equal to $\hat{g}_{t(e')}$ per definition. Moreover, $\hat{c}_{s(e')}$ is an isomorphism according to the pushout condition.

3. $\forall e \in d(B^E - b(A^E))$: $\hat{u}_j \circ \hat{D}(e) = \hat{X}(u(e)) \circ \hat{u}_i$ follows analogously to the previous case.

Next we have to show that $\hat{u} \circ \hat{c} = \hat{g}$ holds. $u \circ c = g$ holds because of pushout properties of D. For all $x \in C^N$ we have $\hat{u}_{c(x)} \circ \hat{c}_x = \hat{g}_x$ according to the definition of $\hat{u}$. $\hat{u} \circ \hat{d} = \hat{f}$ can be proven analogously.

The uniqueness of $\hat{u}$ which means that for all $\hat{u}' : \hat{D} \to \hat{X}$ with $\hat{u}' \circ \hat{c} = \hat{g}$ and $\hat{u}' \circ \hat{d} = \hat{f}$ $\hat{u} = \hat{u}'$ holds, can be obtained directly from the definition of $\hat{u}$. Assuming there is a $y \in D^N$ such that $\hat{u}_y \neq \hat{u}'_y$ and there is an $x \in C^N$ with $c(x) = y$. Then, we have $\hat{u}_y \circ \hat{c}_x = \hat{g}_x = \hat{u}'_y \circ \hat{c}_x$. Thus, $\hat{u}$ is unique. □

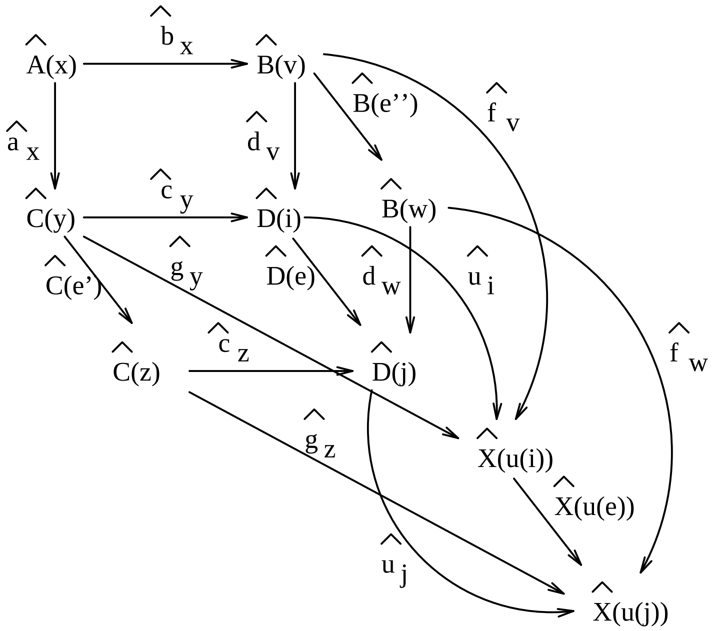

Figure 6.6: Illustration for the proof of proposition 6.1.9

A pushout as described above can be constructed componentwise in the following steps. First the pushout on network morphisms is created. For all network nodes in A the pushout on their local graph morphisms into $\hat{B}$ and $\hat{C}$ is constructed. All other local graphs of $\hat{B}$ and $\hat{C}$ are carried over to $\hat{D}$ unchanged. A network edge in D which has a preimage in A is equipped with the induced morphism between its source and target pushout graph. All other local graph morphisms of $\hat{B}$ and $\hat{C}$ are adapted to their new target graphs. The source graphs have to be mapped structure equivalent according to the pushout conditions.

One might argue that a gluing of distributed graphs where first the gluing of the corresponding network graphs is computed is not distributed. But this is just a compact description of what has to be done. The gluing of graphs especially with injective morphisms in between can be done elementwise. That means for example, a node in B and one in C which have a common origin in A are glued. All others are just copied to D. These actions can be done in parallel.

Example 6.1.10 (Pushout in DISTR(GRAPH)) *Figure 6.7 shows an example of a pushout over two n-injective D-graph morphisms. The pushout conditions are satisfied since $\hat{a}_j : \hat{A}(j) \to \hat{C}(l)$ is bijective. Note that all other local graph morphisms need not to be injective or surjective. The local graph morphism between $\hat{D}(c(k))$ and $\hat{D}(c(l))$ is constructed as the induced morphism between both pushout graphs. Since $\hat{a}_j$ is bijective it is clear how the local graph morphism $\hat{D}(d(e'))$ has to be constructed. This local graph morphism is defined by $\hat{D}(d(e')) = \hat{d}_w^{-1} \circ \hat{B}(e') \circ \hat{d}_x$. If for example $\hat{C}(l)$ would contain an additional node, morphism $\hat{d}_w : \hat{B}(w) \to \hat{D}(c(l))$*

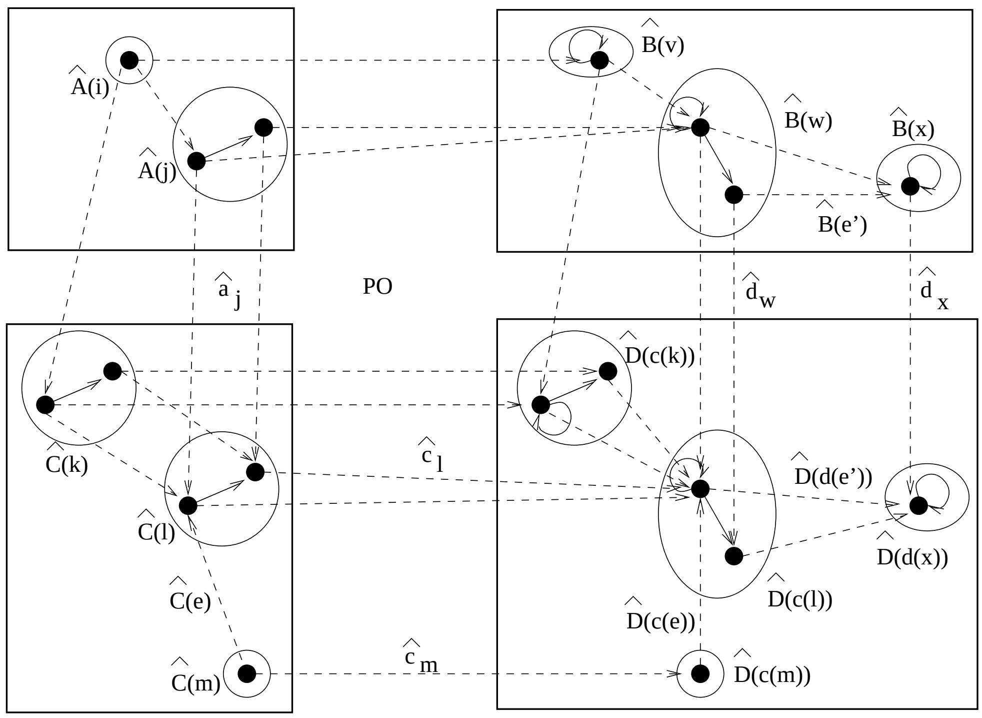

Figure 6.7: Example for a pushout in **DISTR(GRAPH)**

would not be bijective and could not be inverted. Local graph morphism $\hat{D}(c(e))$ *is constructed by* $\hat{D}(c(e')) = \hat{c}_m^{-1} \circ \hat{C}(e) \circ \hat{c}_l$.

The componentwise construction of pushouts in **DISTR(GRAPH)** can be characterized by the pushout conditions in definition 6.1.7. An example where the pushout cannot be constructed componentwise has been shown in example 6.1.8. The following proposition states that the pushout conditions are not only sufficient but also necessary for componentwise pushout construction in category **DISTR(GRAPH)**.

Proposition 6.1.11 (Componentwise PO-construction in DISTR(GRAPH)) *Given two n-injective D-graph morphisms* $\hat{a} : \hat{A} \to \hat{C}$ *and* $\hat{b} : \hat{A} \to \hat{B}$*, the pushout of* $\hat{a}$ *and* $\hat{b}$ *in* **DISTR(GRAPH)** *can be constructed componentwise if and only if pushout conditions (1) and (2) are satisfied.*

Proof: Proposition 6.1.9 states that pushout conditons (1) and (2) are sufficient for the componentwise construction of the pushout of $\hat{a}$ and $\hat{b}$. It remains to show that the pushout conditions are also necessary for the componentwise pushout construction. Assuming that pushout condition (1) is not satisfied there is a network edge $e \in C^E - a(A^E)$ such that there is a $y \in A^N$ with $a(y) = s(e)$ and $\hat{b}_y$ is not bijective. If $\hat{b}_y$ is not surjective there is a node $x \in \hat{B}(b(y))$ which does not have a preimage in

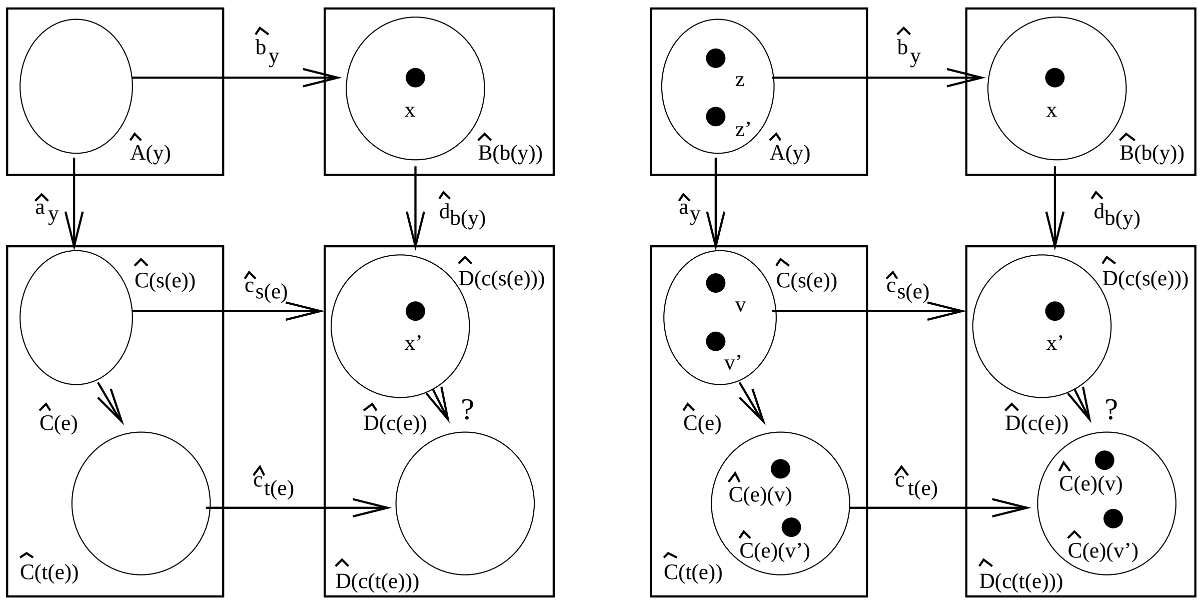

Figure 6.8: Illustrations for the proof of 6.1.11

$\hat{A}(y)$. (Compare the left diagram in figure 6.8 for this part of the proof.) Since $\hat{d}_{b(y)}$ has to be total there is also a node $x' \in \hat{D}(d(b(y))) = \hat{D}(c(s(e))$. Since $\hat{D}(c(s(e))$ is constructed as pushout graph of morphisms $\hat{a}_y$ and $\hat{b}_y$, local graph $\hat{C}(s(e))$ does not contain a preimage of x'. According to componentwise construction local graphs like $\hat{C}(t(e))$ which do not have a preimage in $\hat{A}$, are not changed, i.e. $\hat{D}(c(t(e))) = \hat{C}(t(e))$. Thus, it is possible to choose a local graph $\hat{C}(t(e)) = \hat{D}(c(t(e)))$ such that there is not a local graph morphism $\hat{D}(c(e)) : \hat{D}(c(s(e))) \to \hat{D}(c(t(e)))$.

If $\hat{b}_y$ is not injective there is a node $x \in \hat{B}(b(y))$ which has two preimages $z, z' \in \hat{A}(y)$ with $z \neq z'$. (Compare the right diagram in figure 6.8 for this part of the proof.) According to pushout construction x has to be mapped to a node $x' \in \hat{D}(c(s(e)))$. Let there are two nodes $v, v' \in \hat{C}(s(e))$ $\hat{a}_y(z) = v \neq v' = \hat{a}_y(z')$. If $\hat{C}(e)(v) \neq \hat{C}(e)(v')$, a local graph morphism $\hat{D}(c(e)) : \hat{D}(c(s(e))) \to \hat{D}(c(t(e)))$ can be chosen such that node $x' \in \hat{D}(c(s(e)))$ cannot be mapped to a node in $\hat{D}(c(t(e)))$.

□

Considering pushout situations in figure 6.8 distributed graphs $\hat{D}$ can become pushout graphs if local graphs $\hat{C}(t(e))$ are not identically mapped to $\hat{D}(c(t(e)))$ but $\hat{c}_{t(e)}$ has to reflect $\hat{c}_{s(e)}$. This observation leads to the conjecture that pushouts exist for each two n-injective D-graph morphisms $\hat{a} : \hat{A} \to \hat{C}$ and $\hat{b} : \hat{A} \to \hat{B}$ in category **DISTR(GRAPH)**, but if they do not satisfy the pushout conditions they cannot be constructed componentwise. Since the componentwise construction best reflects the distributed computation, pushouts for n-injective D-graph morphisms $\hat{a}$ and $\hat{b}$ in general are not investigated in this thesis.

6.2 Distributed Graph Transformation in the DPO-Approach

.

The distribution concepts presented in section 5.1 are formally described in the framework of the double-pushout approach to graph transformation. The double-pushout approach for the transformation of directed and colored graphs has been reviewed in section 4.1. The double-pushout approach has been generalized to so-called high-level replacement systems in [EHKP91a] where the main results of the graph grammar theory are abstracted to arbitrary objects and morphisms. In the following we show that distributed graph transformation fits into the framework of HLR-systems.

In section 5.1 we considered several kinds of distributed actions which can all be described by the general concept of D-graph productions.

Definition 6.2.1 (D-graph production) *A distributed graph production, short* D-graph production, $\hat{p} = (\hat{L} \xleftarrow{\hat{l}} \hat{I} \xrightarrow{\hat{r}} \hat{R})$ *consists of D-graphs* $\hat{L} : L \to \mathbf{GRAPH}$, $\hat{R} : R \to \mathbf{GRAPH}$ *and* $\hat{I} : I \to \mathbf{GRAPH}$, *called left- and right-hand side and intermediate D-graphs and two n-injective D-graph morphisms* $\hat{l}$ *and* $\hat{r}$.

- *If all* $\hat{p}_x = (\hat{L}(l(x)) \xleftarrow{\hat{l}_x} \hat{I}(x) \xrightarrow{\hat{r}_x} \hat{R}(r(x)))$ *for* $x \in I^N$ *are left-injective, i.e. all* $\hat{l}_x$ *are injective,* $\hat{p}$ *is called* left-injective, *too.*
- *If* $L \cong I \cong R$ *are network graphs consisting of exactly one node each,* $\hat{p}$ *is called* local.
- *If* $L \cong I \cong R$, $\hat{p}$ *is called* synchronized.
- *If* $\hat{p}_x$ *is identical (def. 4.1.4),* $\forall x \in I_N$, $\hat{p}$ *is called* network transforming production.

A local production describes a local action, communication and synchronization can be modeled by synchronized productions and network administration is expressed by network transforming productions.

Example 6.2.2 (D-graph production) *The D-graph productions in figures 5.3, 5.4, 5.5 and 5.6 are all left injective D-graph productions if they are interpreted in the following way. The D-graph productions "add", "add interface", "send" and "receive" are shown by the right morphism* $\hat{I} \xrightarrow{\hat{r}} \hat{R}$ *of a D-graph production. For each of these examples the left morphism* $\hat{I} \xrightarrow{\hat{l}} \hat{L}$ *is the identity* $id_{\hat{I}}$*. As usual in graph transformation this means that nothing is deleted. For all other D-graph productions mentioned in section 5.1, D-graphs* $\hat{L}$ *and* $\hat{R}$ *are given explicitly. D-graph* $\hat{I}$ *can be constructed by all nodes and edges drawn alike. All local and D-graph morphisms shown or constructed are injective.*

Definition 6.2.3 (Distributed gluing conditon and D-match) *Given a D-graph production* $\hat{p} = (\hat{L} \xleftarrow{\hat{l}} \hat{I} \xrightarrow{\hat{r}} \hat{R})$ *an n-injective D-graph morphism* $\hat{m} : \hat{L} \to \hat{G}$*, called* D-graph occurrence*, is a* D-match of $\hat{p}$ *if the following* distributed gluing condition *is satisfied:*

1. $m : L \to G$ *satisfies the gluing condition wrt.* $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ *(compare the gluing condition in definition 4.1.5).*
2. $\forall x \in I^N : \hat{m}_{l(x)}$ *satisfies the gluing condition wrt.* $\hat{p}_x = (\hat{L}(l(x)) \xleftarrow{\hat{l}_x} \hat{I}(x) \xrightarrow{\hat{r}_x} \hat{R}(r(x)))$
3. connection condition*: Let* $Context(q, n) = H - n(L') + n(l'(I'))$ *be the context graph of production* $q = (L' \xleftarrow{l'} I' \xrightarrow{r'} R')$ *and match* $n : L' \to H$*:*
   (a) $\forall x, y \in I^N \forall e : m(l(x)) \to m(l(y)) \in Context(p, m)^E$:[5]
   $$\hat{G}(e)(Context(\hat{p}_x, \hat{m}_{l(x)})) \subseteq Context(\hat{p}_y, \hat{m}_{l(y)})$$
   $\forall y \in I^N \forall e : z \to m(l(y)) \in G^E : \hat{G}(e)(\hat{G}(z)) \subseteq Context(\hat{p}_y, \hat{m}_{l(y)})$
   (b) $\forall x \in I^N \forall e : m(l(x)) \to z \in G^E - m(L^E) : \hat{l}_y$ *and* $\hat{r}_y$ *are bijective.*
4. network condition*: (a)* $\forall x \in L^N - l(I^N) : \hat{m}_x$ *is bijective*
   $\forall x \in I^N \forall e : l(x) \to y \in L^E - l(I^E) : \hat{m}_x$ *is bijective*
   (b) $\forall x \in I^N \forall e : r(x) \to y \in R^E - r(I^E) : \hat{m}_{l(y)}$ *is bijective.*

$\hat{m}$ *is said* to satisfy all local identification (dangling) conditons *if* $\hat{m}_{l(x)}$ *satisfies the identification (dangling) conditon wrt.* $\hat{p}_x$*,* $\forall x \in \hat{I}^N$*.*

The *connection condition* means the following: (a) An action on a target graph is not permitted to delete local items if some of their counterparts in connected source graphs are not deleted. (b) A local action on a source graph is not allowed to extend this graph if new items would not have images in connected target graphs. Furthermore, such an action is not allowed to delete local items without deleting references to items in target graphs. Lastly, local objects in a source graph are not allowed to be glued together if this is not reflected in the connected target graphs.

The *network condition* can be interpreted as follows: (a) The deletion of network nodes can be done if its local state graph is deleted as a whole in the same production, i.e. if the current local state corresponds with that in the production. If a network edge should be deleted, its source graph has to correspond bijectively with that of the production. (b) The other way around, if a new connection from an existing source graph should be established, this graph has to be structural equivalent with its correspondent given in the production.

The conditions above are shortly discussed within subsections 5.1.2 and 5.1.3.

[5] We write $e : a \to b$ for $s(e) = a$ and $t(e) = b$.

Definition 6.2.4 (D-context graph) *Given an l-injective D-graph production $\hat{p} = (\hat{L} \xleftarrow{\hat{l}} \hat{I} \xrightarrow{\hat{r}} \hat{R})$ and a D-match $\hat{m} : \hat{L} \to \hat{G}$, the following D-graph $\hat{C} : C \to$ **GRAPH** is called* D-context graph *of $\hat{p}$ and $\hat{m}$. Let $C = G - m(L - l(I))$ be the context graph of p and m in* **GRAPH**.

$$\hat{C}(x) := \begin{cases} \hat{G}(x) - \hat{m}_{l(y)}(\hat{L}(l(y)) - \hat{l}_y(\hat{I}(y))) & , \text{ if } \exists y \in I^N \text{ with } m \circ l(y) = x \\ \hat{G}(x) & , \text{ if } x \in G^N - m(L^N) \\ \hat{g}^{-1}_{t(x)} \circ \hat{G}(x) \circ \hat{g}_{s(x)} & , \text{ if } x \in C^E \end{cases}$$

where D-graph morphisms $\hat{g} : \hat{C} \to \hat{G}$ and $\hat{c} : \hat{I} \to \hat{C}$ are defined as follows. Let $c = m \circ l$ and $g = id_{G|_C}$, D-graph morphism $\hat{c}$ is defined by $\hat{c}_y = \hat{m}_{l(y)} \circ \hat{l}_y$, $\forall y \in I^N$, and D-graph morphism $\hat{g}_x = id_{\hat{G}(x)_{/\hat{C}(x)}}$, $\forall x \in C^N$.[6]

Remark: $\forall y \in I^N$ the PO-complement $(\hat{C}(c(y)), \hat{g}_{c(y)}, \hat{c}_y)$ of $\hat{m}_{l(y)}$ and $\hat{l}_y$ in **GRAPH** is constructed.

Proposition 6.2.5 (Applicability of D-graph productions) *Given an l-injective D-graph production $\hat{p} = (\hat{L} \xleftarrow{\hat{l}} \hat{I} \xrightarrow{\hat{r}} \hat{R})$, a D-match $\hat{m} : \hat{L} \to \hat{G}$ and the D-context graph $\hat{C}$ as well as D-graph morphisms $\hat{g}$ and $\hat{c}$ as defined in 6.2.4, then $(\hat{G}, \hat{m}, \hat{g})$ is the pushout of $\hat{c}$ and $\hat{l}$ in* **DISTR(GRAPH)**. *Furthermore, the pushout of $\hat{c}$ and $\hat{r}$ exists.*

Proof: $\hat{C}$ is a D-graph since the gluing condition for all local transformations and the connection condition (a) are satisfied. This means for each $x \in C^N$, $\hat{C}(x)$ is a graph, since the gluing condition is satisfied for the application of each local production $\hat{p}_x$ at match $\hat{m}_{l(x)}$. For each $e \in C^E$, $\hat{C}(e)$ is a total graph morphism, since the connection condition (a) is satisfied for e wrt. $\hat{p}_{s(e)}$, $\hat{p}_{t(e)}$, $\hat{m}_{s(e)}$ and $\hat{m}_{t(e)}$. Thus, $\forall x \in \hat{C}(s(e))$, there is a $y \in \hat{C}(t(e))$ with $\hat{G}(e) \circ \hat{g}_{s(e)}(x) = y$.

$\hat{g}$ is a D-graph morphisms since $\forall x \in C^N$ $\hat{g}_x$ is a total graph morphism. Furthermore, we have $\hat{g}_{t(e)} \circ \hat{C}(e) = \hat{G}(g(e)) \circ \hat{g}_{s(e)}$, $\forall e \in C^E$, since all g_x are identities. Since $\hat{m}$ and $\hat{l}$ are D-graph morphisms, their composition $\hat{c}$ is one, too.

Now we have to show the pushout properties for $(\hat{G}, \hat{m}, \hat{g})$. The commutativity $\hat{m} \circ \hat{l} = \hat{g} \circ \hat{c}$ follows from fact 4.1.7 stating that $\forall x \in I^N$ the PO-complement $(\hat{C}(c(x)), \hat{g}_{c(x)}, \hat{c}_x)$ can be built in category **GRAPH**. Thus, $\hat{m}_{l(x)} \circ \hat{l}_x = \hat{g}_{c(x)} \circ \hat{c}_x$ holds.

Next, we construct the pushout of $\hat{l}$ and $\hat{c}$ (which is possible since the distributed gluing conditions contain the pushout conditions which are part of network condition (a) and connection condition (b)) and obtain $(\hat{X}, \hat{f} : \hat{L} \to \hat{X}, \hat{h} : \hat{C} \to \hat{X})$. Then, we have to show that the resulting pushout graph $\hat{X}$ is isomorphic to $\hat{G}$. According to pushout properties there is a unique D-graph morphism $\hat{u} : \hat{X} \to \hat{G}$ with $\hat{u} \circ \hat{f} = \hat{m}$

[6] where $id_{\hat{G}(x)_{/\hat{G}(x)}} = id_{\hat{G}(x)}$

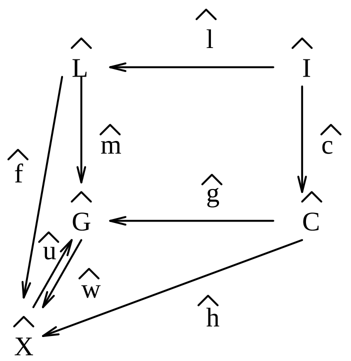

Figure 6.9: Illustration for the proof of proposition 6.2.5

and $\hat{u} \circ \hat{h} = \hat{g}$. Vice versa, a suitable D-graph morphism $\hat{w} : \hat{G} \to \hat{X}$ can be defined as follows. $\forall x \in m(l(I^N))$, $\hat{w}_x$ is the induced morphism wrt. $(\hat{G}(x), \hat{m}_x, \hat{g}_x)$. $\forall z \in C^N - c(I^N)$ with $g(z) = x$, $\hat{w}_x = \hat{h}_z$ and $\forall v \in L^N - l(I^N)$ with $m(v) = x$, $\hat{w}_x = \hat{f}_v \circ m_v^{-1}$. (Compare the diagram in figure 6.9.)

We show the well-definedness of $\hat{w}$ distinguishing the following cases:

1. $\forall x \in m(l(I^N)) : \hat{w}_x$ is well-defined according to PO-complement properties in **GRAPH**. (Compare fact 4.1.7.)

2. $\forall z \in C^N - c(I^N)$ with $g(z) = x : \hat{w}_x = \hat{h}_z$ is well-defined since $\hat{h}_z$ is a graph morphism.

3. $\forall v \in L^N - l(I^N)$ with $m(v) = x : \hat{m}_v$ is isomorphic according to network condition (a). Thus, $\hat{f}_v \circ \hat{m}_v^{-1}$ is well-defined, too.

4. $\forall e \in m \circ l(I^E) : \hat{w}_{t(e)} \circ \hat{G}(e) = \hat{X}(w(e)) \circ \hat{w}_{s(e)}$ can be shown analogously to case 1 in the proof of proposition 6.1.9.

5. $\forall e \in g(C^E - c(I^E))$ we have the following cases:

   (a) $s(e) \notin m(L^E)$: Then, $\hat{G}(s(e)) = \hat{C}(s(e))$. Thus, $\hat{w}_{t(e)} \circ \hat{G}(e) = \hat{w}_{t(e)} \circ \hat{g}_{t(e)} \circ \hat{C}(e) \circ \hat{g}_{s(e)}^{-1} = \hat{h}_{t(e)} \circ \hat{C}(e) \circ \hat{g}_{s(e)}^{-1} = \hat{X}(w(e)) \circ \hat{h}_{s(e)} \circ \hat{g}_{s(e)}^{-1} = \hat{X}(w(e)) \circ \hat{w}_{s(e))}$ according to the definition of $\hat{w}$, $\hat{g}$ and $\hat{c}$. (Compare the left diagram of figure 6.10.)

   (b) $s(e) \in m(L^E)$: Let $\hat{G}(s(e))$ be the corresponding pushout graph. According to pushout properties there is a morphism $q : \hat{G}(s(e)) \to \hat{X}(w(t(e)))$ with $q \circ \hat{m}_{s(e')} = \hat{X}(w(e)) \circ \hat{f}_{s(e')}$ for $e' \in L^E$ and $m(e') = e$. Furthermore, $q \circ \hat{g}_{s(e)} = \hat{X}(w(e)) \circ \hat{h}_{s(e)}$ holds. Using the morphism properties of $\hat{h}$, $\hat{g}$ and $\hat{f}$ and the knowledge that $\hat{m}_{s(e)}$ is isomorphic (network condition (a)) we get $\hat{w}_{t(e)} \circ \hat{G}(e) = \hat{X}(w(e)) \circ \hat{w}_{s(e)} = q$ according to the universal

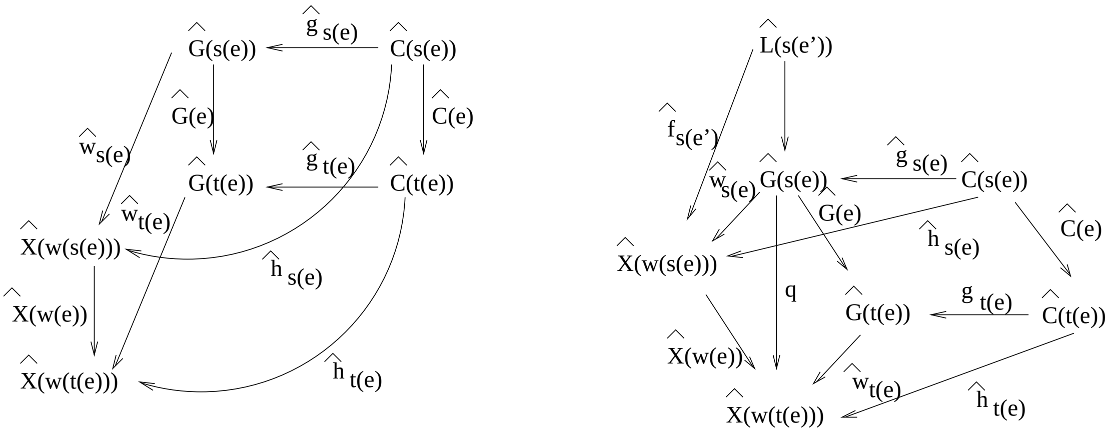

Figure 6.10: Illustrations for the proof of proposition 6.2.5

property of $(\hat{G}(s(e)), \hat{g}_{s(e)}, \hat{m}_{s(e)})$. (Compare the right diagram of figure 6.10.)

6. $\forall e \in m(L^E - l(I^E))$ we have again two cases:

   (a) $s(e) \in m \circ l(I^E)$: This case is analogous to the previous case.

   (b) Otherwise: Let $e' \in L^E$ with $m(e') = e$. Then we have $\hat{X}(w(e)) \circ \hat{w}_{s(e)} = \hat{X}(w(e)) \circ \hat{f}_{s(e')} \circ \hat{m}^{-1}_{s(e')} = \hat{f}_{t(e')} \circ \hat{L}(e') \circ \hat{m}^{-1}_{s(e')} = \hat{w}_{t(e)} \circ \hat{m}_{t(e')} \circ \hat{L}(e') \circ \hat{m}^{-1}_{s(e')} = \hat{w}_{t(e')} \circ \hat{G}(e)$ since $\hat{f}$ and $\hat{m}$ are D-graph morphisms and $\hat{f}_x = \hat{w}_x \circ \hat{m}_x$ $\forall x \in L^N$ according to pushout property or definition of $\hat{w}$.

It is easy to show that $\hat{w} \circ \hat{g} = \hat{h}$ and $\hat{w} \circ \hat{m} = \hat{f}$ hold. $\hat{w}_z \circ \hat{g}_z = \hat{h}_z$, $\forall z \in C^N$ and $\hat{w}_v \circ \hat{m}_v = \hat{f}_v$, $\forall v \in L^N$, follow directly from PO-properties or the definition of $\hat{w}$. It is straight forward to show $\hat{u} \circ \hat{w} = id_{\hat{G}}$ and $\hat{w} \circ \hat{u} = id_{\hat{X}}$. Thus, $\hat{G}$ and $\hat{X}$ are isomorphic.

The pushout of $\hat{c}$ and $\hat{r}$ exists according to proposition 6.1.9. Pushout condition (1) is part of connection condition (b) and pushout condition (2) is network condition (b).

□

Definition 6.2.6 (D-graph transformation (sequence)) *Given an l-injective D-graph production $\hat{p} = (\hat{L} \xleftarrow{\hat{l}} \hat{I} \xrightarrow{\hat{r}} \hat{R})$ and a D-graph match $\hat{m} : \hat{L} \to \hat{G}$, a distributed graph transformation, short* D-graph transformation, *$\hat{G} \Longrightarrow_{di} \hat{H}$ via $\hat{p}$ and $\hat{m}$, short $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow}_{di} \hat{H}$, from a D-graph $\hat{G}$ to a D-graph $\hat{H}$ is given by the two pushout diagrams (1) and (2) in the category* **DISTR(GRAPH)** *shown in figure 6.11. A D-graph transformation is of shape S if all its D-graphs are of shape S.*

$$
\begin{tikzcd}
\hat{L} \arrow[d, "\hat{m}"'] \arrow[dr, phantom, "(1)"] & \hat{I} \arrow[l, "\hat{l}"'] \arrow[r, "\hat{r}"] \arrow[d, "\hat{c}"'] \arrow[dr, phantom, "(2)"] & \hat{R} \arrow[d] \\
\hat{G} & \hat{C} \arrow[l, "\hat{g}"'] \arrow[r] & \hat{H}
\end{tikzcd}
$$

Figure 6.11: D-graph transformation

A D-graph transformation sequence $\hat{G} \overset{P}{\Longrightarrow}^*_{di} \hat{H}$ *is a sequence of* $n \geq 0$ *D-graph transformations* $\hat{G} = \hat{G}_0 \Longrightarrow_{di} \hat{G}_1 \Longrightarrow_{di} \ldots \Longrightarrow_{di} \hat{G}_n = \hat{H}$ *via D-graph productions of a set* P*.* $\hat{H}$ *is also called* D-derivable *from* $\hat{G}$ *by* P*.*

Proposition 6.2.7 (Uniqueness of D-graph transformation) *Given a left-injective D-graph production* $\hat{p} = (\hat{L} \xleftarrow{\hat{l}} \hat{I} \xrightarrow{\hat{r}} \hat{R})$ *and a D-match* $\hat{m} : \hat{L} \to \hat{G}$*, the D-graph transformation* $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow}_{di} \hat{H}$ *is unique up to isomorphism.*

Proof: First, we have to show that $\hat{C}$ together with $\hat{c}$ and $\hat{g}$ as defined in 6.2.4 are unique up to isomorphism. This is done similar to the proof of uniqueness of pushout complements in **GRAPH**: Assume that we have a D-graph $\hat{C}'$ with $\hat{g}'$ and $\hat{c}'$ such that $(\hat{G}, \hat{m}, \hat{g}')$ is a pushout in **DISTR(GRAPH)** we can show that there is an isomorphism between $\hat{C}$ and $\hat{C}'$. Graphs C and C' are isomorphic since morphism l is injective (compare the uniqueness of PO-complements in fact 4.1.7). Let $\hat{k} : \hat{C} \to \hat{C}'$ and $\hat{k}' : \hat{C}' \to \hat{C}$ be defined as follows: $\forall x \in C^N : \hat{k}_x = \hat{g}'^{-1}_{k(x)} \circ \hat{g}_x$ and $\forall x \in C'^N : \hat{k}'_x = \hat{g}^{-1}_{k'(x)} \circ \hat{g}'_x$. $\forall x \in c(I^N) : \hat{k}_x$ is well-defined according to PO-complement properties in **GRAPH**. $\forall x \in C^N - c(I^N) : \hat{g}_x = id_{\hat{G}(x)}$. Thus, $\hat{k}_x$ is well-defined according to pushout properties in **DISTR(GRAPH)**. Analogously, $\hat{k}'_x$ is well-defined. $\hat{k}$ and $\hat{k}'$ are D-graph morphisms since $\hat{g}$ and $\hat{g}'$ are so.

It remains to show $\hat{k} \circ \hat{k}' = id_{\hat{C}'}$ and $\hat{k}' \circ \hat{k} = id_{\hat{C}}$ which is easy to check.

Together with the fact that pushouts in arbitrary categories are unique up to isomorphisms we can state that $G \overset{\hat{p},\hat{m}}{\Longrightarrow}_{di} H$ is unique up to isomorphism. □

Specifying some distributed actions they may run simultaneously. Such a parallel execution of distributed actions is expressed by a parallel production on distributed graphs.

Definition 6.2.8 (Parallel D-graph production) *Given two D-graph productions* $\hat{p}_1 = (\hat{L}_1 \xleftarrow{\hat{l}_1} \hat{I}_1 \xrightarrow{\hat{r}_1} \hat{R}_1)$ *and* $\hat{p}_2 = (\hat{L}_2 \xleftarrow{\hat{l}_2} \hat{I}_2 \xrightarrow{\hat{r}_2} \hat{R}_2)$*, the D-graph production* $\hat{p}_1 + \hat{p}_2 = (\hat{L}_1 + \hat{L}_2 \xleftarrow{\hat{l}_1 + \hat{l}_2} \hat{I}_1 + \hat{I}_2 \xrightarrow{\hat{r}_1 + \hat{r}_2} \hat{R}_1 + \hat{R}_2)$ *is called* parallel D-graph production *of* $\hat{p}_1$ *and* $\hat{p}_2$ *where* $L_1 + L_2$*,* $I_1 + I_2$ *and* $R_1 + R_2$ *are coproduct graphs and* $l_1 + l_2$ *as well as* $r_1 + r_2$ *the induced morphisms.*

Definition 6.2.9 (Distributed graph grammar and its semantics) *A* distributed graph grammar $DGG = (\hat{S}, P)$ *is given by a D-graph* $\hat{S}$, *the* start graph *and a set* P *of D-graph productions.*[7]

Let P^+ *be the smallest extension of the set* P *including all parallel D-graph productions* $\hat{p}_1 + \hat{p}_2$ *for* $\hat{p}_1, \hat{p}_2 \in P^+$. *The* operational semantics $Sem(DGG)$ *of* DGG *is given by the class of all possible transformations starting at* $\hat{S}$ *using D-graph production set* P^+, *i.e.* $Sem(DGG) = \{\hat{S} \overset{P^+}{\Longrightarrow}{}^*_{di} \hat{G}\}$.

As operational semantics we take all possible distributed graph transformations where distributed productions may be applied in parallel.

6.3 Relations to Parallel Graph Transformation Concepts

In the following, we compare parallel and distributed graph transformation. While parallel graph replacement is performed as structured transformation on non-distributed graphs, distributed graph transformation describes structured transformation on structured graphs. Thus, this additional structure has to be added, if we want to compare parallel and distributed graph transformation. This structure is added by determining how the given graph is distributed. This distribution has to be done in a way such that the elementary matches become local matches, i.e. they have to fit to the given graph distribution. Not each elementary match may become a local match because the local gluing conditon may not be satisfied, although the parallel gluing conditon for all elementary matches is satisfied. If there are dangling edges in the local context of some node deletion which would not dangle in the parallel application of those productions, they are deleted by another production. The difference between the parallel and the distributed dangling conditon is defined by the following *difference dangling condition.*

Definition 6.3.1 (Difference dangling condition) *Let* $POS = (i \xleftarrow{ij'} ij \xrightarrow{ji'} j)_{1 \le i < j \le n}$ *be a graph (a star graph) and all following D-graphs of this shape, i.e.* $\hat{L} : POS \to \mathbf{GRAPH}$, $\hat{I} : POS \to \mathbf{GRAPH}$, $\hat{R} : POS \to \mathbf{GRAPH}$, *and* $\hat{G} : POS \to \mathbf{GRAPH}$ *with* $(G, (e_i)_{1 \le i \le n})$ *being the pushout-star of* $\hat{G}$ *and* $COV = (T, O)$ *with* $O = \{o_i : \hat{L}(i) \to G,\ 1 \le i \le n\}$ *a partial covering of* G.

Given a D-graph production $\hat{p} = (\hat{L} \xleftarrow{\hat{l}} \hat{I} \xrightarrow{\hat{r}} \hat{R})$, *a D-graph occurrence* $\hat{m} : \hat{L} \to \hat{G}$ *satisfies the* difference dangling conditon *wrt.* $\hat{p}$ *if*

$$\forall e \in (\hat{G}(i)^E \cap e_i^{-1E}(\bigcup_{1 \le j \le n} o_j^E(\hat{L}(j)^E))) - \hat{m}_i^E(\hat{L}(i)^E),\ \forall\, 1 \le i < j \le n$$

[7]Since we are not interested in classical language aspects here, we do not distinguish terminal and non-terminal graphs.

with $s_{\hat{G}(i)}(e) = \hat{m}_i^N(x)$ *or* $t_{\hat{G}(i)}(e) = \hat{m}_i^N(x)$ *we have* $\hat{m}_i^N(x) \in \hat{m}_i^N(\hat{l}_i^N(\hat{I}(i)))$. *Moreover,* $e_i \circ \hat{m}_i = o_i, \ \forall\, 1 \leq i \leq n$.

An interaction scheme is very similar to a distributed production. Both concepts contain the same components but differently composed. For a comparison of parallel graph transformation and distributed graph transformation we provide a mapping which transforms an interaction scheme into its corresponding distributed production.

Definition 6.3.2 (Interaction scheme mapping) *Given a set $\mathcal{IS}$ of interaction schemes and a set $\mathcal{DPROD}$ of D-graph productions. An* interaction scheme mapping *$is : \mathcal{IS} \to \mathcal{DPROD}$ is defined by $is(IS) = \hat{p}$, $\hat{p} \in \mathcal{DPROD}$, with $is(IS(x)(y)) = \hat{p}(y)(x)$ if $\hat{p}(y)(x) \cong IS(x)(y)$ for all $IS : G \to$ **PROD(GRAPH)** $\in \mathcal{IS}$ with $x \in G^N$ and $y \in \{L, I, R\}$ or $x \in G^E$ and $y \in \{l, r\}$.*

Theorem 6.3.3 (Distribution Theorem) *Let $\hat{L} = (\hat{L}(i) \xleftarrow{\hat{L}(ij')} \hat{L}(ij) \xrightarrow{\hat{L}(ji')} \hat{L}(j))_{1 \leq i < j \leq n}$ and $\hat{G} = (\hat{G}_(i) \xleftarrow{\hat{G}(ij')} \hat{G}(ij) \xrightarrow{\hat{G}(ji')} \hat{G}(j))_{1 \leq i < j \leq n}$ be distributed graphs where G is the pushout-star graph of $\hat{G}$, and $COV = (T, O)$ a partial covering of G. Then the following statements are equivalent:*

1. *There is a contextual parallel transformation $G \overset{COV}{\Longrightarrow}_{con} H$ and a distributed occurence $\hat{m} : \hat{L} \to \hat{G}$ satisfying all local identification conditions, the difference dangling condition and connection condition (a) wrt. $is(T)$. (Compare definition 6.3.2.)*

2. *There is an amalgamated transformation $G \overset{COV}{\Longrightarrow}_a H$ and a distributed occurence $\hat{m} : \hat{L} \to \hat{G}$ satisfying all local identification conditions, the difference dangling condition and connection condition (a) wrt. $is(T)$.*

3. *There is a distributed transformation $\hat{G} \overset{is(T), \hat{m}}{\Longrightarrow}_{di} \hat{H}$ with H being the pushout-star graph of D-graph $\hat{H}$.*

Proof: "1. $\iff$ 2.": Direct consequence of *Parallel Replacement Theorem II* 4.4.7.

"2. $\Longrightarrow$ 3.": Given an amalgamated transformation $G \overset{COV}{\Longrightarrow}_a H$ we first show that there are local transformations $\hat{G}(i) \overset{\hat{p}_i, \hat{m}_i}{\Longrightarrow}_d \hat{H}(i)$ for $1 \leq i \leq n$. The existence of such a local transformation depends on the existence of a pushout complement $(\hat{C}(i), \hat{c}_i, \hat{g}_i)$ of $\hat{l}_i$ and $\hat{m}_i$.

D-match $\hat{m}$ has to satisfy all local identification conditions by assumption. Since COV is a matching partial covering, the parallel dangling condition (4.2.10) is satisfied by O. Now, we have to show that if O satisfies the parallel dangling conditon and the difference dangling condition the local dangling conditon is satisfied for all $\hat{m}_i$, $1 \leq i \leq n$.

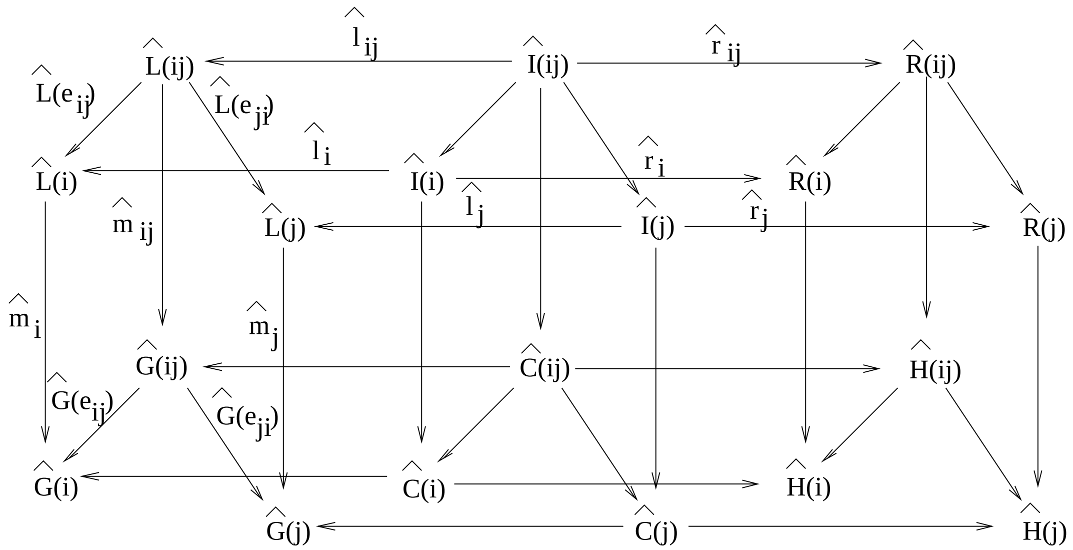

Figure 6.12: Distributed transformation $\hat{G} \overset{is(T),\hat{m}}{\Longrightarrow}_{di} \hat{H}$

To prove that all $\hat{m}_i$ satisfy the local dangling conditon wrt. $\hat{p}_i$, we have to distinguish two cases. For all context edges $e \in \hat{G}(i)^E$ with $s_{\hat{G}(i)}(e) = \hat{m}_i^N(x)$ or $t_{\hat{G}(i)}(e) = \hat{m}_i^N(x)$ and $e \notin \hat{m}_i(\hat{L}(i))$:

1. $e \in (\hat{G}(i)^E \cap e_i^{-1E}(\bigcup_{1\leq j\leq n} o_j^E(\hat{L}(j)^E)))$: Then $\hat{m}_i^N(x) \in \hat{m}_i^N(\hat{l}_i^N(\hat{I}(i)))$ according to the difference dangling condition.

2. $e \in (\hat{G}(i)^E - e_i^{-1E}(\bigcup_{1\leq j\leq n} o_j^E(\hat{L}(j)^E)))$; Then $\hat{m}_i^N(x) \in e_i^{-1N}(\bigcup_{1\leq j\leq n} o_j^N(\hat{l}_j^N(\hat{I}(j))))$ according to the parallel dangling condition. It follows directly that $\hat{m}_i^N(x) \in e_i^{-1N}(o_i^N(\hat{l}_i^N(\hat{I}(i)))) = \hat{m}_i(\hat{l}_i^N(\hat{I}(i)))$ according to the universal property of pushout-stars where we get $o_i = o \circ f_i = \hat{m}_i \circ e_i$, $\forall\ 1 \leq i < j \leq n$, where all $(f_i : \hat{L}_i \to L)_{1\leq i\leq n}$ are the pushout-star morphisms of $\hat{L}$ and $o : L \to G$ the amalgamated match.

Since all $\hat{m}_i$ satisfy the local gluing conditon wrt. $\hat{p}_i$, there are direct transformations $\hat{G}(i) \overset{\hat{p}_i,\hat{m}_i}{\Longrightarrow}_d \hat{H}(i)$, $\forall\ 1 \leq i \leq n$, as depicted in figure 6.12. There is a distributed transformation $\hat{G} \overset{is(T),\hat{m}}{\Longrightarrow}_{di} \hat{H}$ if $\hat{m}$ satisfies the distributed gluing conditon wrt. $is(T) = \hat{p}$.

D-graphs $\hat{L}$ and $\hat{G}$ are of shape POS as defined in 6.3.1. The interface graph and the right-hand side of $\hat{p}$ are also of shape POS. Thus, the network match is id_{POS} applied at identical production ($POS \overset{id_{POS}}{\longleftarrow} POS \overset{id_{POS}}{\longrightarrow} POS$). In this case the gluing conditon is trivally satisfied.

All $\hat{m}_i$ satisfy the gluing conditon wrt. $\hat{p}_i$ as shown above. Connection conditon (a) is satisfied by assumption. Since the entire graph POS is identically replaced by the above defined network transformation $POS \overset{id_{POS}}{\Longrightarrow}_d POS$, connection conditon (b) and network conditions (a) and (b) are trivally satisfied.

"3. $\Longrightarrow$ 2.": Vice versa, given a distributed transformation $\hat{G} \overset{\hat{is}(T),\hat{m}}{\Longrightarrow}_{di} \hat{H}$ as in figure 6.12, the amalgamated transformation $G \overset{COV}{\Longrightarrow}_a H$ with $COV = (T, O)$ is obtained as follows. Gluing the left-hand side star $\hat{L}$, interface star $\hat{I}$ and right-hand side star $\hat{R}$ we obtain pushout-stars $(L, (f_i)_{1 \le i < j \le n})$, $(I, (a_i)_{1 \le i < j \le n})$ and $(R, (b_i)_{1 \le i < j \le n})$. Graph morphisms l and r, forming the amalgamated production p, are the induced morphisms from pushout-star graph I. These constructions already describe the first step of amalgamated transformation $G \overset{COV}{\Longrightarrow}_a H$. Graphs G, C and H are the pushout-star graphs of $\hat{G}$, $\hat{C}$ and $\hat{H}$, respectively. Occurrence o is the induced morphism using all $\hat{m}_i$ and the pushout-star of $\hat{G}$. Parallel occurrence O consists of all $o_i = o \circ f_i$, $1 \le i \le n$. Moreover, graph morphisms $c : I \to C$, $g : C \to G$, $h : C \to H$ and $q : R \to H$ are obtained as induced morphisms in a similar way using the universal property of the pushout-stars of $\hat{I}$, $\hat{C}$ and $\hat{R}$.

Using the commutativity property of colimits ([Mac71], [HS73]) the two morphism squares (l, o, c, g) and (r, q, c, h) are pushouts representing the direct transformation $G \overset{p,o}{\Longrightarrow}_d H$ at amalgamated production p forming the second step of amalgamated transformation $G \overset{COV}{\Longrightarrow}_a H$.

Distributed occurence $\hat{m}$ exists by definiton of distributed transformations and satsifies all local identification conditons and connection conditon (a). It also satisfies the difference dangling conditon because of the construction of O and since all $\hat{m}_i$ and $\hat{m}_{ij}$, $1 \le i < j \le n$, satisfy their (local) dangling conditon wrt. $\hat{p}_i$ and $\hat{p}_{ij}$, resp.

□

The distribution theorem extends a similar result achieved for $n = 2$ presented in [CMR+96b]. The proof idea for the distribution theorem goes back to the proof of this result in a preliminary version of this contribution.

Chapter 7

Modeling Communication-Based Systems by Parallel and Distributed Graph Transformation

Parallel and distributed graph transformation as introduced in chapters 3 to 6 are now applied to communication-based systems coming from different application fields. The applications are described in different detail. Two examples which are specified quite precisely, are AGG– *an algebraic graph grammar system* using parallel graph transformation and a *data repository for distributed software engineering* modeled by distributed graph transformation. For a study in greater detail the reader is referred to further literature which is mentioned within the corresponding sections. The applications are shortly introduced and parts of the communication-based aspects modeled by graph transformation are shown. For each application, its description by graph transformation is discussed along the requirements for modeling support stated in chapter 1.

In section 7.3 two further applications of parallel graph transformation are shortly presented: *collision-free motions of objects in configuration spaces* and *feature recognition in the area of computer integrated manufacturing (CIM)*. These applications have also communication aspects which are the main part of their presentation in this thesis.

Although the applications of parallel and distributed graph transformation come from different application fields the underlying ideas how to use it show similarities. Graph transformation describes the transition of complex states. States are always described by graphs and state transition by graph transformation. The modeling ideas of communication aspects are summarized in section 7.4. Section 7.5 contains a short discussion of further features which would be useful for the development of a comprehensive specification technique for communication-based systems based on graph transformation. In this context, we shortly mention GRACE, a graph- and rule-centered language under development which has the aim to support structured specification based on graph transformation.

Due to their operational and very constructive character graph transformations can be directly implemented although there is still little tool support for graph transformation. In section 7.6 we show how parallel and distributed graph transformation concepts can be mapped to the AGG-graph model (introduced in section 7.1). Furthermore, we touch on the construction of parallel and distributed graph transformation based on their AGG-graph model.

7.1 Example: AGG – An Algebraic Graph Grammar System

Since graphs are a good means for modeling the structural aspects of a software system they can be found in a lot of CASE tools. Besides of e.g. Petri nets or state charts (compare section 1.3), especially graph transformation is useful to model the dynamic aspects. Moreover, graph transformation supports the combination of structure and flow modeling by showing evolving graphs. CASE functionality based on graph transformation is performed, for example, by the IPSEN environment developed at the RWTH Aachen using the programming language PROGRES. Another tool supporting graph transformation is the AGG-system.[1] It is an ongoing implementation effort in the graph grammar group of the TU Berlin. This prototype implementation of the single-pushout approach to graph transformation mainly consists of a flexible graph editor with a transformation component which performs direct transformation steps for user-selected productions and occurrences. Parts of the AGG-system have been specified by graph transformation themselves showing the power of the graph transformation concept and its intuitive description possibilities. This comprehensive work is presented in [CGHB+95]. Short versions of modeling the functionality of AGG by graph transformation have been presented in [Tae92, TB94, AEH+96]. Similar work is done in [Zün95a] where a development environment for PROGRES is specified in PROGRES itself.

7.1.1 The AGG-Editor

The editor of the AGG-system provides a comprehensive functionality for the input and modification of graphs with a mouse/menu-driven interface. It supports a graph model with an abstraction concept, i.e. hierarchical graphs as mentioned in section 2.1 and higher-order edges, i.e edges between edges. Such a graph model is useful to describe and manipulate graphs, productions and occurrences in a uniform way. It includes the usual definition of graphs as a special case. Graphs fitting into this model are called AGG-graphs in the following.

[1]Note that AGG stands for algebraic graph grammars and should not be mixed up with attributed graph grammars.

Higher-order edges are used to represent relations between graphs again. Common relations between graphs are productions and occurrences of graphs. Edges between objects of any order, e.g. between an edge and a node, are also possible. Graphs can be abstracted to nodes, bundles of edges can be abstracted to single edges. The formal definition of this concept which can be found in [CGHB+95] regards graphs as a set of objects with three partial functions: *source, target*, and *abstraction* . A node is an object for which both, the *source* and *target* function, are undefined. Some rather obvious consistency conditions that are enforced by the AGG-system, e.g. the *abstraction* function has to be compatible with the *source* and *target* function and it has to be acyclic.

The editor supports different views on graphs in different windows and allows the editing of several graphs concurrently, each on different abstraction levels . An abstraction level may be shown in several windows. All windows can be used in the same way for editing as well as transforming graphs. Any change in one window is immediately reflected in all windows showing that graph (view/controller concept).

Some edit operations are especially suited for graph transformations, e.g. a *gluing operation which merges two objects to a new object that inherits the incidence relations of the original objects. Another example are the* copy connect operations drawing edges between all original nodes and edges, and their copies. These operations facilitate the input of transformation productions and occurrences. The transformation component implements the single-pushout approach to graph transformation (see section 2.1).

Example 7.1.1 (Production and match on AGG**)** *A simple example of a production together with an occurrence is shown in figure 7.1 on the left side. The four nodes in the upper left corner form the left-hand side, the three nodes and all solidly drawn edges on the upper right part is the right-hand side, and all remaining nodes and solidly drawn edges in the lower left corner form the work graph. The left-hand side, the right-hand side and all dashed edges in between are compounded to the production which is applied at a match indicated by all vertically drawn dashed edges.*

Note that this example shows a graph transformation step in the single-pushout approach where matches do not have to satisfy the gluing condition. Moreover, productions are partial graph morphisms. The gluing part of the production is indicated by all the nodes and edges connected by arrows drawn dashed. Thus, higher-order edges are needed here (although they do not occur in example 7.1.1 but in example 7.1.2).

The abstraction concept can be used to present an interface to the transformation component: graphs and the left and right-hand side of productions are abstracted to single nodes, production morphisms and matches are abstracted to single edges. In this way, the higher abstraction level is used as a diagram level.

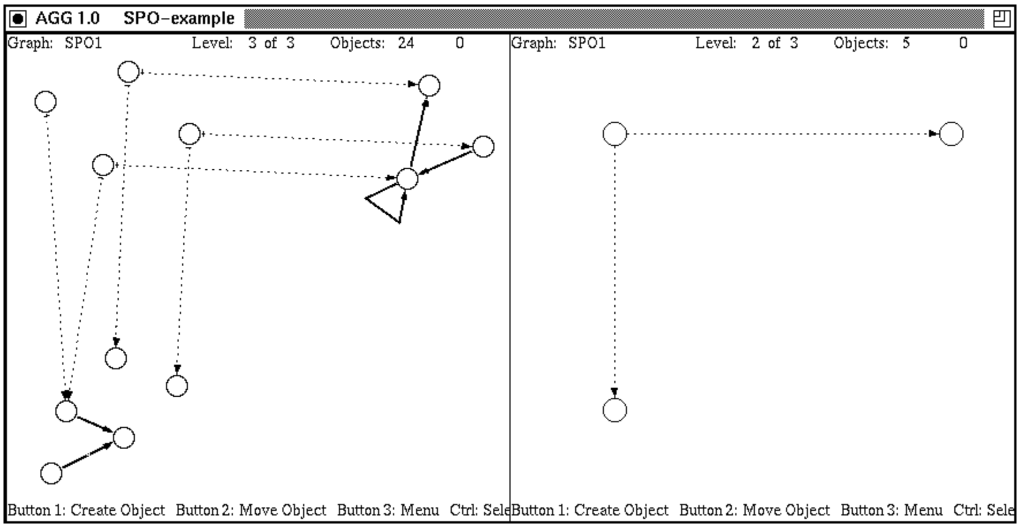

Figure 7.1: Production and occurrence: graph level and diagram level

After the execution of the transformation step, the diagram level contains a rectangle where the derived graph and its relations to the original one and the right-hand side of the production are indicated additionally, the graph level shows the transformed graph in the lower right corner of the left side as well as its relations in detail.

Example 7.1.2 (Graph transformation on AGG**)** *An example of the diagram level of production and match in the single-pushout approach is shown in figure 7.1 on the right. The upper edge with its source and target nodes represents a production with left-hand and right-hand side depicted on the left-hand part of the figure. The lower left node represents the graph to which the production has to be applied along the occurrence indicated by the left edge with adjacent nodes.*

The graph transformation in the single-pushout approach to graph transformation is illustrated in figure 7.2.

Further extensions of the AGG-system will also support the double-pushout approach to graph transformation.

AGG-graphs are not restricted to two abstraction levels. The abstraction concept can also be used to structure large graphs by hiding details in lower abstraction levels.

7.1.2 Modeling the AGG-Editor by Graph Transformation

In this section, an overview over the specification of the AGG-editor using graph transformation as presented in [CGHB$^+$95] is given where the parts using parallel graph transformation are shown in detail. The whole specification is presented in

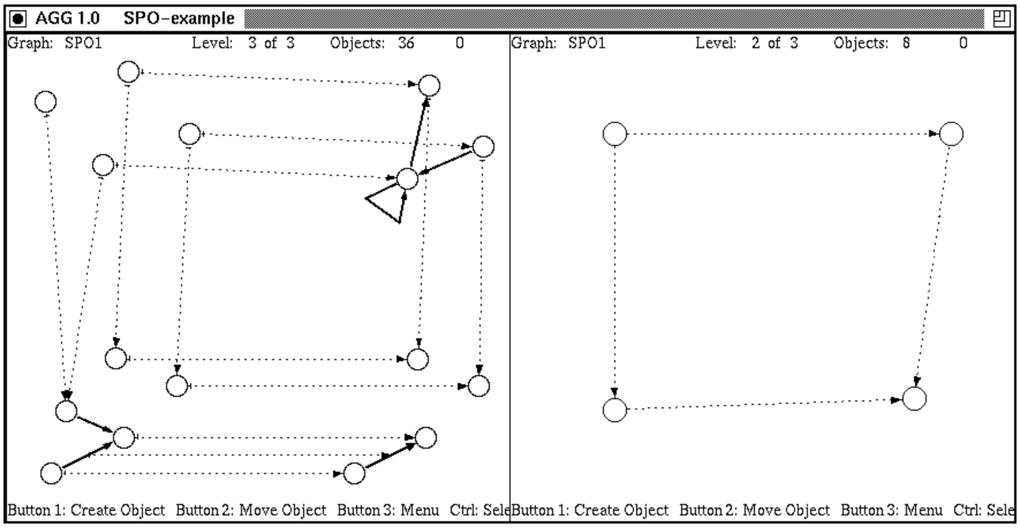

Figure 7.2: A graph transformation step: graph level and diagram level

detail in [CGHB+95]. The main idea is to describe a state of the AGG-system by an attributed graph and a state transition, i.e. an editor operation or a transformation step in the AGG-system by graph transformation. The specification of the AGG-system in [CGHB+95] follows the single-pushout approach as mentioned in section 2.1 together with application conditions, amalgamation schemes and attributes. Since we are mainly interested in the communication aspects the presentation of the specification in this thesis uses amalgamation schemes based on the double-pushout approach here, to be compatible with the formal descriptions in previous chapters. The additional expressive power of the single-pushout approach in contrast to the double-pushout approach is not needed in this example. The operational understanding of the structural and parallelism aspects is independent of the particular graph transformation approach.

The internal states of the AGG-system are modeled by graphs where AGG-objects like windows, AGG-graphs, abstraction levels, AGG-nodes, AGG-edges, etc. are represented by nodes and the relations between them by edges. Attributes of these AGG-objects like the position of nodes, their type, their state of selection, etc. are reflected by attributes of the corresponding nodes.

It is convenient to distinguish different levels of description for AGG-graphs. In the following we consider the modeling of the logical structure of AGG-graphs, the system states of AGGas its refinement and lastly, the full representation modeling also the layout of AGG-graphs as another refinement.

In a first step, we consider the logical structure of AGG-graphs and their modification operations like the insertion and deletion of AGG-nodes and -edges as well as abstraction levels, copying and gluing of AGG-nodes or -edges, etc. The AGG-objects are represented as nodes of type "object". They are distinguishable into

AGG-node and AGG-edge via the pointers to their source (s) and target (t). For AGG-nodes, source and target are not defined, thus, their adjacent "s" or "t"-edges point to a special "$\perp$"-object. Moreover, a chain of abstraction levels are introduced as nodes of type "abstr level". Edges of type "a" describe the abstraction relations between the AGG-objects. Every object points to the "abstr level"-node where it belongs to (by an untyped edge). Modeling AGG-nodes and -edges in this uniform way higher-order edges and the abstraction mechanism can be easily described. The initial state of the AGG-system is described by one abstraction level which is empty.

One step further the system states of AGGare specified. Now, windows and the cursor become further AGG-objects. Selection of AGG-objects and hiding of AGG-objects are indicated by special edges which are loops at "object"-nodes and "vis(ibility)"-edges to "window"-nodes.

The modeling of the layout of graphs is not described in this section but in [CGHB+95]. The position of AGG-nodes, their type, the size and position of windows, the bending of AGG-edges, the visibility of AGG-objects, etc. can be described by different types of attributes.

Example 7.1.3 (AGG-state) *In figure 7.3 the modeling of an arbitrary state of the AGG-system is shown, where the model of the initial state is bordered with dashed lines (including an additional edge from the initial abstraction level to $\top$ not shown). AGG-nodes and -edges are represented as objects belonging to an abstraction level. The cursor points to the current window. All abstraction levels belonging to one graph, form a list which is described by "a"-edges between different levels. The bottom and the top level do not have a refinement and an abstraction, resp. This is indicated by edges from and to $\perp$ and $\top$. In figure 7.3 an AGG-state is modeled consisting of one graph with two abstraction levels. The lower abstraction level contains a node with a loop and is shown in two windows. The upper level is empty and not shown.*

AGG-editor operations are modeled by parallel graph transformation. One example is the opening of a new window showing the current abstraction level. This operation defines the new window as the current one. In this window all objects of the previous current window are shown, too. Thus, opening a window produces a number of further actions which have to be performed. The amalgamation scheme in the following example may be interpreted as a kind of broadcasting to all objects of the current abstraction level to compute their graphical representation in the new window.

Example 7.1.4 (AGG-operation "new window") *In figure 7.4 the production "new window" is depicted. This production has to be applied to the current AGG-state graph which can contain arbitrary many objects. Thus, the production "show object" shown in figure 7.5 has to be applied as often as possible to show all objects in the new window. Those parts of the left and the right-hand sides of "show object"*

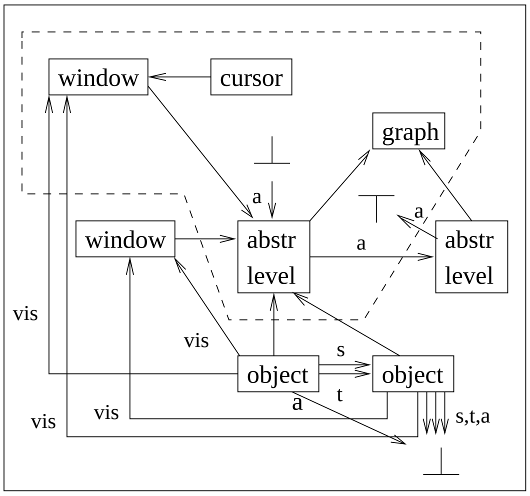

Figure 7.3: Section of an AGG-state modeled by a graph

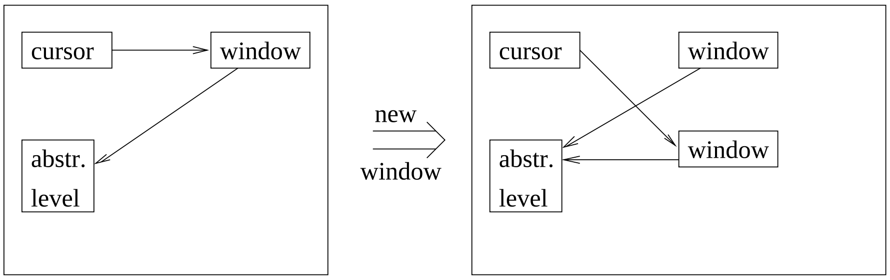

Figure 7.4: Production for opening a new window

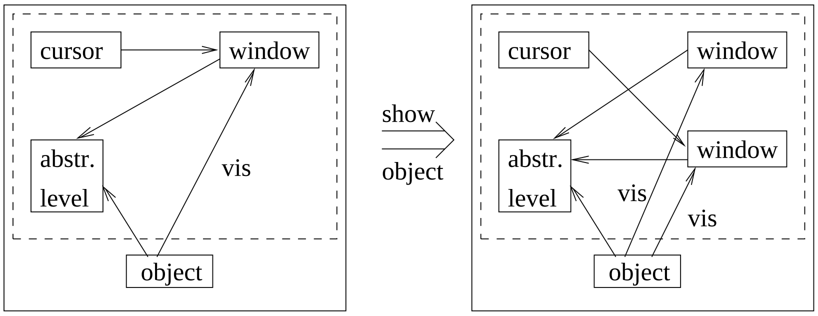

Figure 7.5: Productions for computing the visibility of nodes

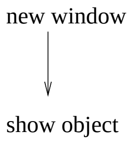

Figure 7.6: Interaction scheme of "new window" operation

which are completely inside the dashed boxes are copies of "new window". Thus, "new window" is a subproduction of "show object". All "show object" applications have to synchronize in that of "new window" for modeling the creation of a new window exactly ones. This operation can be modeled by an amalgamation scheme (see definition 4.3.5). $A_{NEW\_WINDOW} = (IS_{NEW\_WINDOW}, \{locall, sync\})$ *with its interaction scheme being depicted in figure 7.6. We model this operation by fully synchronized local-all-match coverings (definition 4.3.3) because only those objects belonging to the current window shall be shown and each object should have exactly one visibility edge to one window.*

The following descriptions of AGG-editor operations use the same conventions as explained for the "new window"-operation in example 7.1.4. As for this operation, the interaction schemes of the following amalgamation schemes consist of one production and a subproduction uniquely embedded. Thus, the interaction schemes are not shown in the following specifications of editor operation if they are of this kind.

Another very basic AGG-operation is the insertion of a node into the current graph. The corresponding amalgamation scheme can be regarded as an abstract model for some kind of broadcasting of this graph modification to all windows to adapt their contents.

Example 7.1.5 (Insertion of a node) *The insertion of a node into the current abstraction level is described by those parts of figure 7.7 which are framed dashed. This basic action has to be extended by showing the new node in each window belonging to the current abstraction level. Deletion of a node can be modeled by inverting*

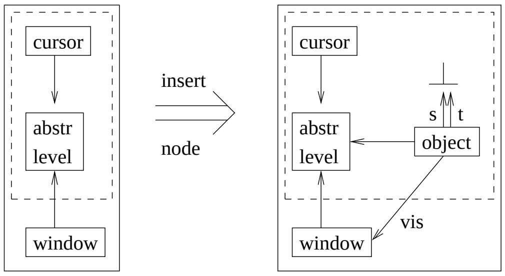

Figure 7.7: Productions for insertion of a node

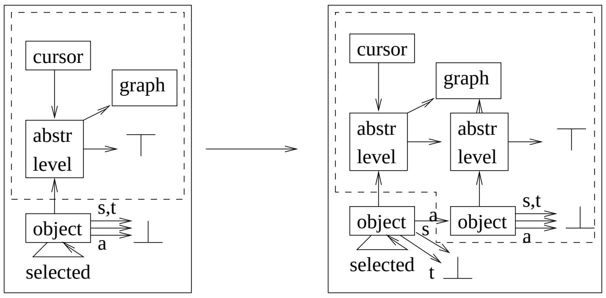

Figure 7.8: Create abstraction - case 1

the productions depicted in figure 7.7.

Amalgamation schemes can also describe the abstraction concepts of AGG.

Example 7.1.6 (Create Abstraction) *In section 7.1 we described the abstraction concept of* AGG*. The* AGG*-operation "create abstraction" allows to abstract graphs to nodes and bundles of edges in the same direction to single edges. Here we show only the abstraction of nodes. Specifying "create abstraction" we have to distinguish two main cases. The first one is that the current abstraction level is the top one. Then a new level has to be created which is the new top level. A new abstraction node abstracting all selected nodes has to be inserted into the new level. All the selected nodes are not yet abstracted. Such an information is described by an a-edge running from a selected node to* $\perp$ *instead of an abstraction node. Selected objects are indicated by a "selected"-labeled loop at the object.*

An abstraction node is created in a basic action. For each selected node an abstraction edge is inserted pointing to the abstraction node. (See figure 7.8).

The second case is that the abstraction level of the current level already exists. Thus, only a new abstraction node has to be inserted which is the basic action. Similar to the first case all selected nodes are joined to the abstraction node by an "a"-edge (figure 7.9).

The abstraction level of the current one can already be shown in any number of windows which is not possible in the case that the abstraction level is newly created.

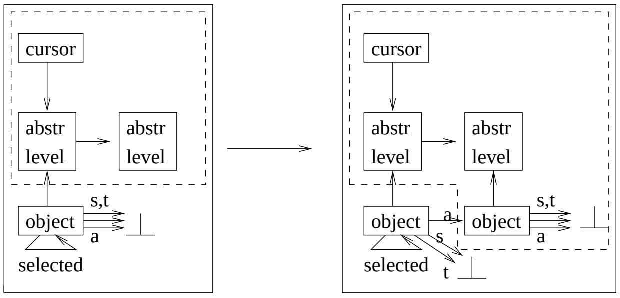

Figure 7.9: Create abstraction - case 2.1

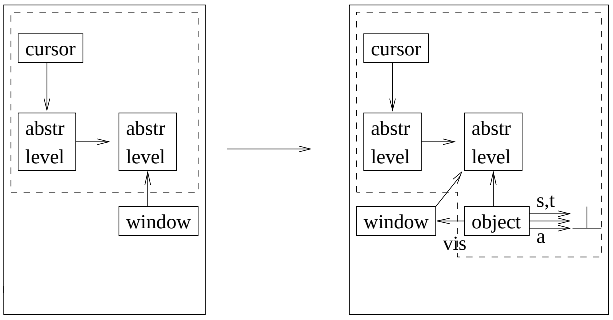

Figure 7.10: Create abstraction - case 2.2

For each window showing the abstraction level the abstraction node has to be shown (figure 7.10). Thus, for this type of AGG*-operation two different kinds of elementary productions are needed. But, although different kinds of elementary productions are used to describe the abstraction of nodes they all can be applied in parallel due to the fact that they overlap in the same subproduction.*

The interaction scheme for this AGG*-operation consists of the common subproduction and its embeddings in both elementary productions depicted in figures 7.9 and 7.10.*

Example 7.1.7 (Selection of objects) *Selection of an object is modeled by the only production shown in figure 7.11. The interaction scheme of this operation is a basic one.*

Incorporating also the layout representation of an AGG-graph we have to turn to attributed graphs. The position of AGG-nodes, their type, the size and position of windows, the bending of AGG-edges, the visibility of AGG-objects, etc. can be described by different types of node and edge attributes.

Example 7.1.8 (Layout representation) *In figure 7.12 the model of an* AGG*-state as given in example 7.1.3 is extended by some of the attributes needed for graph layout. Especially, the positions of windows,* AGG*-nodes and -edges are shown where*

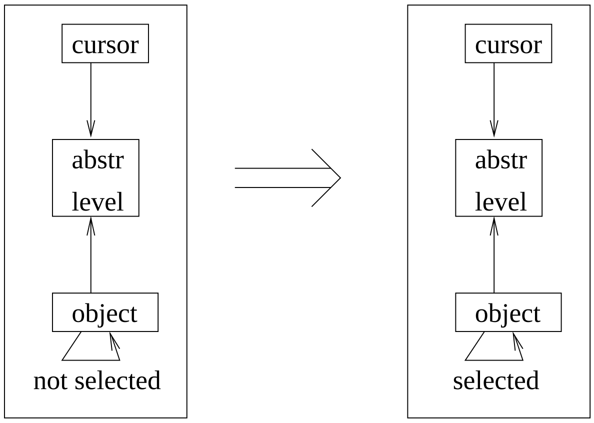

Figure 7.11: Selection of an object

the position of an AGG*-edge is described by the positions of source and target and possible bending points. Each "vis"-edhe is attributed by a Boolean value indicating the visibility. Source, target and abstraction mappings not defined are not shown in this state.(In contrast to the previous model where the mappings are totalized by* $\perp$ *and* $\top$*.)*

Also the AGG-operations have to be adapted to the layout representation. In the following, the operation "new window" presented in example 7.1.4 is reviewed.

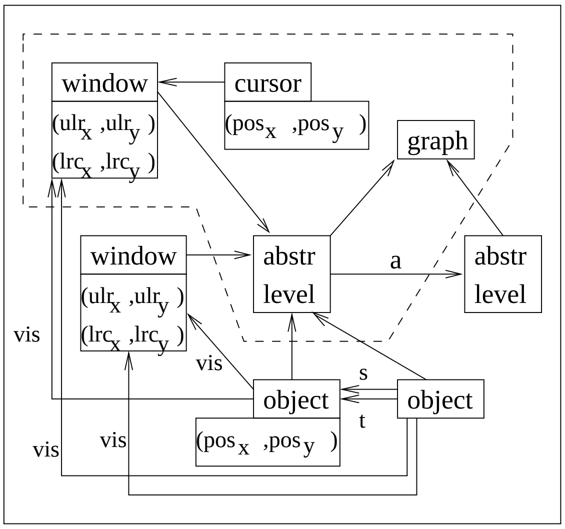

Figure 7.12: AGG-state with layout attributes

Example 7.1.9 (New window operation with layout representation)
AGG-*operation "open new window" opens a new window which is the current window then. To adjust its size the values for the upper left corner (ulc) and the lower right corner (lrc) have to be given. In this window all objects of the previous current window are shown, too. For each object in this new window it has to be decided whether it is visible according to the window size or not.*

In figure 7.13 the production "new window" is depicted which gets the size of the new window as input parameters. They become the size attributes of the new "window"-node if they are consistent which is tested by an application condition. Furthermore, the production "show node" shown in figure 7.5 has to be applied as often as possible. This rule models the checking of the visibility of a node belonging to the current abstraction level. Here, negative application conditions are used to distinguish an object as AGG-*node. Thus, $\perp$ and $\top$ are not needed any more. If the abstraction, source or target mapping of an object is not defined the* AGG-*state do not contain a corresponding edge. All nodes and edges belonging to the application conditions are crossed out. Similar to rule "show node" a rule "show edge" is needed that computes the visibility of* AGG-*edges dependent on that given by their source and target* AGG-*nodes.*

Amalgamated graph transformation applying attributed productions with application conditions which are needed here is performed according to its definition in the single-pushout approach to graph transformation in [HMTW95].

Other AGG-operations as described in the beginning of this section can be specified similarly by parallel graph transformation. Together with a start graph modeling the initial AGG-state and suitable labels where we do not distinguish between

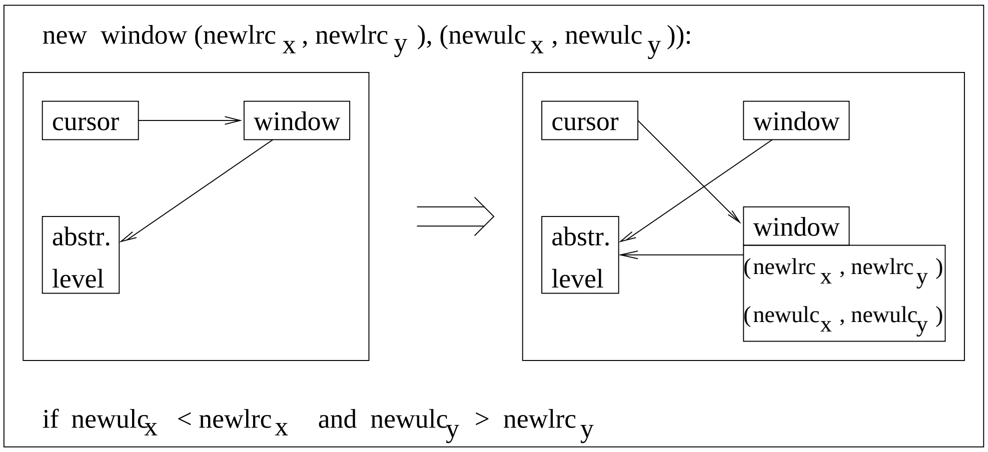

Figure 7.13: Opening a new window with size parameters

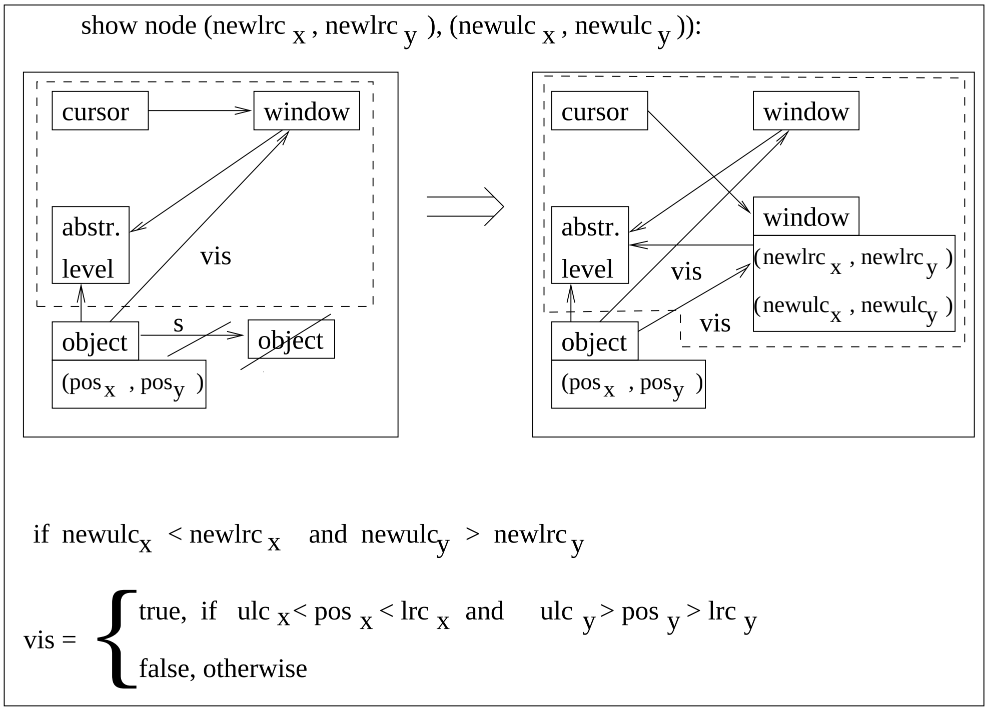

Figure 7.14: Computing the visibility of nodes

terminal and non-terminal labels a parallel graph grammar is obtained describing all reachable AGG-states.

7.1.3 Conclusion

The description of AGG-states by graphs and AGG-operations by graph transformations is an imperative one and rather close to object-based description methods. Its graphical presentation demonstrates that graph transformation leads to intuitive, constructive specifications.

The description of AGG-operations based on parallel graph transformation helps the designer to specify complex operations in parallel. Comparing the examples given in this section with one another the amalgamation schemes presented model some kind of view/controller concept. Every time a window is created or modified all the objects of the current abstraction level have to be informed. Every time an abstraction level is modified all the windows showing it have to be adapted. Especially amalgamation schemes using local-all-match-coverings seem to be well suited to model broadcasting facilities on such an abstract level. In the underlying object-oriented implementation of the AGG-system all the view/controller operations are executed sequentially.

The biggest problem in developing a graph specification of the AGG-system has been its complexity. AGG separates different kinds of graph representation and offers a bunch of editor operations with complex functionality. Parallel graph transformation do not support refinement possibilities for a stepwise incorporation of more (structural) information. But recently first attempts into this direction have been made in [CH95] based on sequential graph transformation.

For a specification of further parts of AGG e.g. the transformation component, modularization concepts have to be supported. The concepts of DIEGO systems [TS95] may help to specify a modular AGG-system where different parts act in parallel.

To support the development of correct specifications, consistency checking has to be offered. First results ensuring consistency of graph transformation specifications have been achieved in [HW95]. They have to be adapted to parallel and distributed graph transformation.

7.2 Example: A Data Repository for Distributed Software Engineering

Several people developing a software system in parallel have to cooperate with each other. As an support *project flow graphs* (presented in [KH95]) or *development graphs* (introduced in [PW94]) can be used which describe different states of software development. These kinds of graphs have been introduced in order to assure

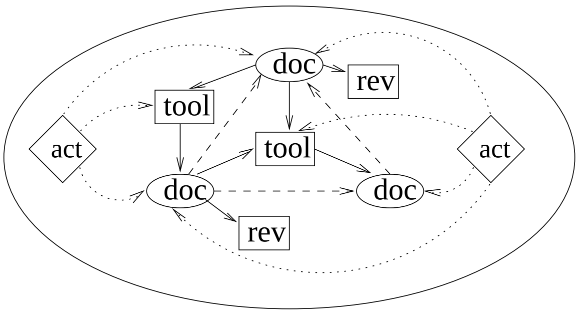

Figure 7.15: Project flow graph

software quality and possibly speed up the project. The developers are allowed to concurrently work on the software. Every *development step* is described by a graph transformation. In the following, project flow graphs and their developments are reviewed, simplified in some minor points. Afterwards they are transfered into distributed graphs reflecting the topology of distributed software development. The dynamics of these *distributed project flow graphs* is described by distributed graph transformation. In order to develop new versions cooperatively a *data repository* has to be installed to allow all developers a consistent view on the project documents. The communication with this repository as well as its internal behavior is described by distributed graph transformation.

This application has been used as running example in [Tae96a] and [TS95].

7.2.1 Project Flow Graphs

Project flow graphs show dependencies between different units or, more concrete, documents and their revisions as well as input and output relations between development tools and documents. Such a graph stores about the same information as a revision and configuration management system together with a tool like "make". Roles and access rights in a software engineering project are modeled by *(act)or* nodes linked to document nodes they may read or edit and tools they may use.

Example 7.2.1 (Project flow graph) *The project flow graph in figure 7.15 contains three nodes of type "doc" modeling documents, two "tool"-nodes modeling two different development tools, e.g. editors or compilers, and two "rev"-nodes standing for two revisions of documents. Furthermore, input and output relations are described by edges from "doc"-nodes to "tool"-nodes and vice versa. They are drawn as solid arrows. Edges drawn as dashed arrows model dependencies between different documents. A "doc"-node represents a document and connected "rev"-nodes their revisions. Furthermore, two actors (developers) are modeled by "act" nodes and their access to documents and tools by dotted edges.*

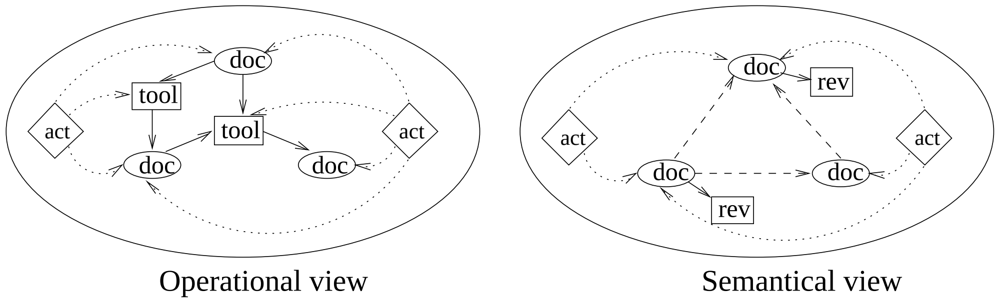

Figure 7.16: The operational and the semantical view

In [KH95] two different views on project flow graphs are defined: the *operational view* and the *semantical view*. The semantical view shows all documents as well as revisions and their interdependencies. The operational view is restricted to the connections between tools and documents.

Example 7.2.2 (Views on project flow graphs) *Given the project flow graph in figure 7.15 the operational and the semantical view on it are shown in figure 7.16.*

The dynamic aspects of project flow graphs concern modification of its structure and model the progress of software development. Typical situations that cause changes of a project flow graph include creation, splitting and melting of documents or revisions, the introduction of new tools and managing access of developers to documents by check in and checkout operations. The project flow graph model does not make specific assumptions about document contents and formats but addresses to project administration. The consistency of the processing structure between documents is achieved by satisfying certain consistency rules.

7.2.2 Describing Revision Management by Distributed Graph Transformation

The distributed development of software is heavily based on reliable revision management. This aspect is reflected by the semantical view on a project flow graph. In the following, we concentrate on this view and show a *data repository* managing revisions of documents based on project flow graphs. The behavior of a data repository is modeled by distributed productions.

A data repository manages the access rights to documents. In this example we distinguish between read and write access. The repository internally stores all revisions of a document but offers only the last revision of a document at its export interface.

Example 7.2.3 (A distributed project flow graph) *Figure 7.17 shows a distributed graph which models the semantical view of a possible distributed project flow*

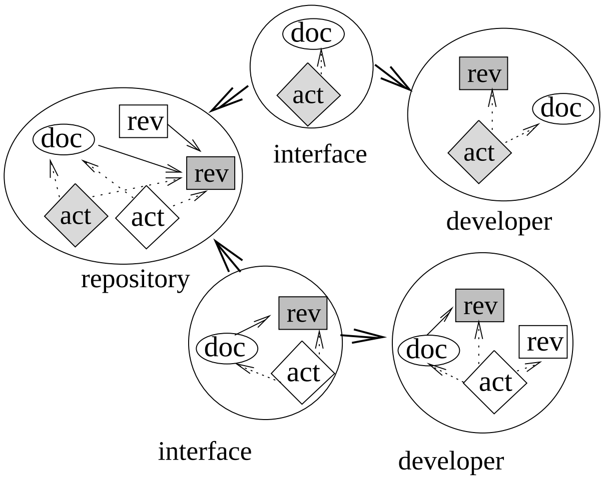

Figure 7.17: A distributed state

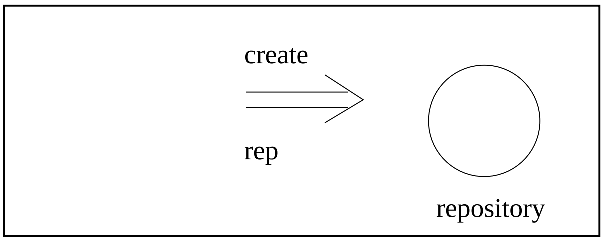

Figure 7.18: Distributed production *create rep*

graph. In this graph we have one repository and two developers. Both developers have checked out the same revision, one with read the other with write access. It is the second revision of the only document in the repository.

In the following, distributed productions show possible communicating actions and synchronizations between a data repository and a developer. These are check out and check in of revisions and adding a document. Additionally, there is a distributed production *create dev* for adding a developer's workspace. Moreover, developers who want to checkout revisions are registered in the data repository also modeled by *create dev*. Developers are allowed to perform also local actions, such as the local creation and modification of revisions.

The initial state of the whole distributed system is empty. The first action that may be performed is the creation of a repository. Initially, a data repository is empty. It can be created by distributed production *create rep* in figure 7.18.

A data repository may be accessed by an arbitrary number of developers working concurrently. If the workspace of a new developer is created by distributed production *create dev* in figure 7.19 a personal $act(or)$-node is created in the repository as well as in the developer's part. A *destroy* production may be the inverse of *create dev*

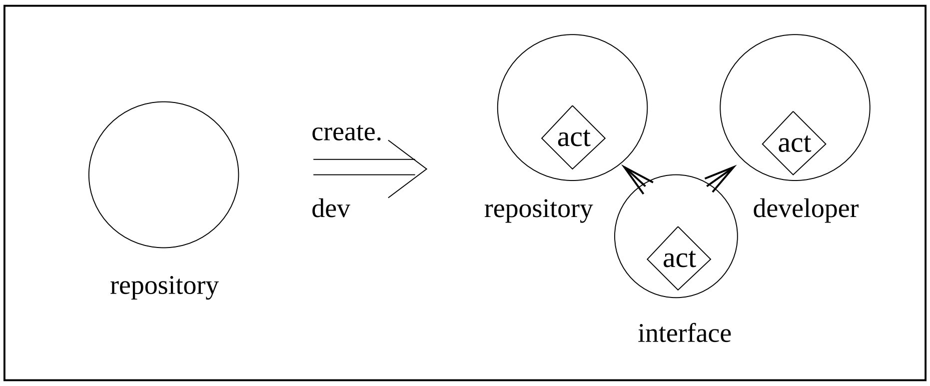

Figure 7.19: Distributed production *create dev*

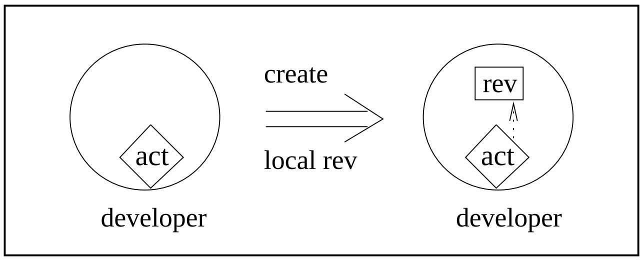

Figure 7.20: Distributed production *create local dev*

which means that a workspace of a developer has to be cleared before it is destroyed. Production *create local rev* in figure 7.20 creates a new local document with a first revision. Production *new local rev* in figure 7.21 is also a local action which reflects the main work a software developer does, changing revisions.

The check out of a revision is performed by two distributed productions which are *get rlock* and *get wcopy*. Distributed production *get rlock* in figure 7.22 provides a developer with the last version of a document by creating a new edge from an *act* node to the latest revision indicated by an edge from the *doc*(*ument*) node. The access is read-only modeled by the *rev*(*ision*) node in the interface.

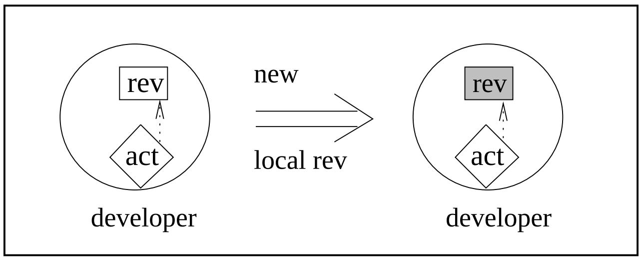

Figure 7.21: Distributed production *new local rev*

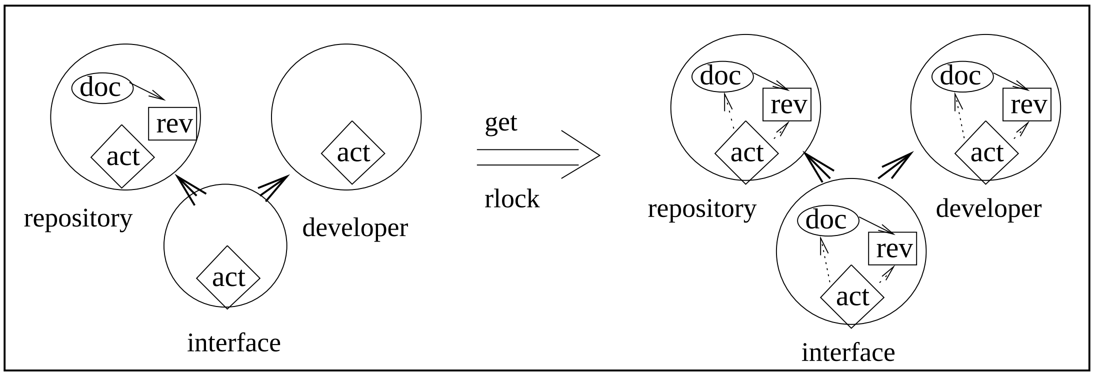

Figure 7.22: Distributed production *get rlock*

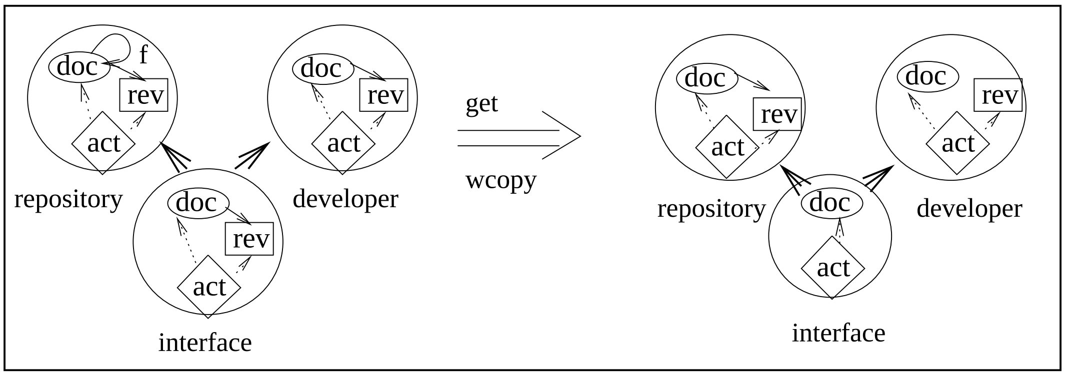

Figure 7.23: Distributed production *get wcopy*

The intention of *get wcopy* in figure 7.23 is to give a developer write access. This possible if there is not another developer who has already write access to this document. The small loop with label $f(ree)$ indicates that there is not a write access on this document. An application of distributed production *get wcopy* decouples the revisions in the repository and in the developer's workspace such that the developer may change the revision locally.

The check in of a revision is also performed by two distributed productions, *put wcopy* and *release rlock*. *put wcopy* in figure 7.24 stores a new revision and gives back the write access. Note that the document points always to the latest revision.

release rlock in figure 7.25 removes the read access for a revision. Distributed production *add doc* in figure 7.26 allows to add a new document with a revision to the data repository. The developer who create a new document has read access but no write access to its revision (indicated by the f-loop).

The operational semantics of distributed graph grammar $DATAREP = (\hat{S}, P)$ where $\hat{S}$ is the empty distributed graph and P consists of all the productions mentioned in this subsection describes all possible distributed actions executed in parallel

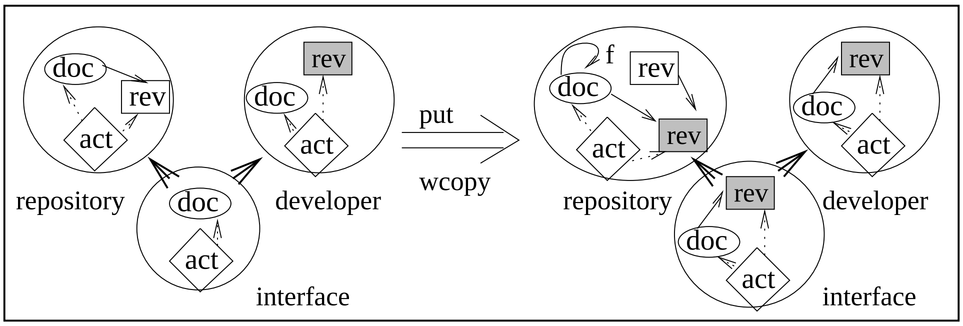

Figure 7.24: Distributed production *put wcopy*

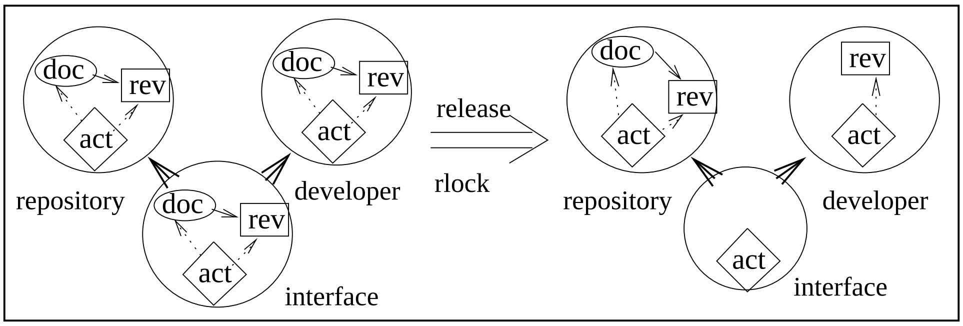

Figure 7.25: Distributed production *release rlock*

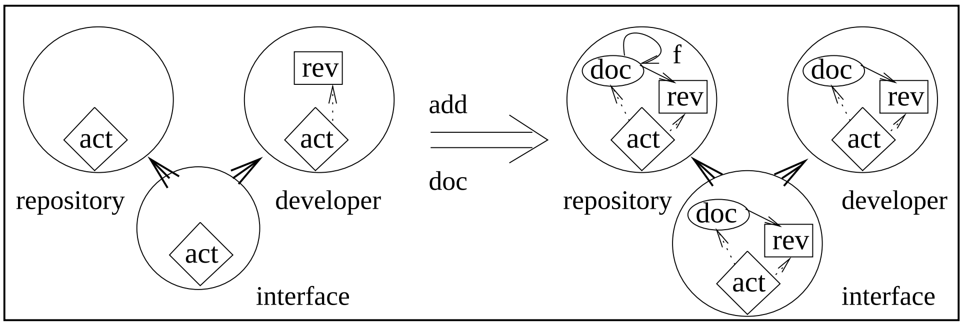

Figure 7.26: Distributed production *add doc*

which may happen in a distributed system of data repositories and developers.

7.2.3 Conclusion

Using distributed graphs for structuring of project flow graphs two abstraction levels are possible. On the network level the cooperating developers and their communication structure is modeled. On the local level the developers' local project flow graphs show the local parts of the distributed project. The access to documents is modeled by a data repository which is an extra part in the network managing the revisions. This model is much more elaborated and structured than the global project flow graphs presented in [KH95], where actor nodes together with access edges describe local parts.

Distributed graph grammar $DATAREP$ exhibits a high amount of parallelism since different developers may act concurrently. All distributed actions can be performed in parallel unless they are concerned with equal local system parts. The synchronization concepts seem to be clear and powerful enough to describe the communication between a repository and a developer. Nevertheless, higher-order communication constructs may be helpful to describe more complex communication types. Somewhat more complex is the communication by *broadcasting*, e.g. check in of a revision may cause a message to all other developers that a new revision is available.

$DATAREP$ describes an distributed system with component architecture in the sense that new developers may enter the system while others may leave. Moreover, big projects may use more than one data repository storing different kinds of documents.

Using distributed graph transformation a communicating action is described by one distributed production. Because of this atomicity the application of synchronized productions resembles the execution of simple transactions in some sense. Indeed, a certain kind of consistency can be derived. If, for example, no data repository is available there cannot be a workspace for a developer. Another example is the consistency of the check out operation. A developer cannot change a revision without write access because production *new local rev* cannot be applied in the *developer* part then.

7.3 Further Examples

In the following, we present two further applications of parallel graph transformation supporting parallel execution on different levels of description.

7.3.1 Collision-free motions of objects in configuration spaces

This application of parallel graph transformation is very shortly discussed in this thesis. The work is presented in more detail in [TS91] and comprehensively in [ST90] where it is based on amalgamated graph transformation.

In this example, configuration spaces are considered where arbitrary objects in a two-dimensional space can move by translations and rotations within this space. The space is modeled in a discrete way by laying a Cartesian grid in it and modeling by a grid graph. Objects in the space are modeled by coloring those parts of the grid graph which correspond to positions of object parts in the space.

If one or more objects can move simultaneously in this space the recognition of collisions between objects is essential. To tackle these problems, motion in configuration spaces is modeled by amalgamated graph transformation which is proven in [ST90, TS91] to be collision-free, thus, preserve the consistency of configuration spaces.

Application fields of object movement in configuration spaces may be computer animation, simulation in robotics, etc.

This application of parallel graph transformation uses a high degree of parallelism on different levels. Motion and collision detection of objects is described by fine-grain parallelism which can be modeled by amalgamation schemes where elementary productions model motion of object parts. Large-grain parallelism on the level of objects modeling parallel motions of objects is also supported by parallel application of amalgamated productions. To avoid collisions each object has to "look forward" to decide whether it can continue its way, has to change its way or to wait until the way is free again. If motions are collision-free and therefore independent, they are described by the application of parallel productions.

The underlying concepts are as general as possible such that translation and rotation of objects with arbitrary shape can be described. But large configuration spaces with small objects cause very large grid graphs such that the complexity of graph transformation does not seem to be adequate. Without dynamic abstraction concepts where different parts of the configuration space use differently fine grids, this example seems to be not adequate for "real world" applications.

7.3.2 Feature recognition in the area of computer integrated manufacturing (CIM)

Features as considered in the case study [Hof93] are domain- and company-specific description elements of workpieces based on geometrical and technological data. Features may be related, e.g. they are adjacent or overlapping. Feature recognition provides steps of abstraction where several features are composed to more complex features. A feature description of a workpiece is hierarchically structured in the sense that a feature is refined by its parts.

A workpiece described by atomic features and their neighborhood relation, is described by an attributed graph. Feature recognition steps can be performed in parallel where the computations of all the relations of the new recognized feature to the already existing ones is done in parallel. These computations can be modeled by parallel graph transformations where computations of related features have to be synchronized. In contrast to this fine-grain parallelism, large-grain parallelism occurs on the level of feature recognition. If two complex features are recognized from completely different sets of features, their recognition may be performed in parallel.

This application is based on amalgamated graph transformation in the single-pushout approach extended by attributes and application conditions [HMTW95]. Since each step of feature recognition adds information to the feature graph the modeling productions are non-deleting. Thus, the amalgamated graph transformation in the double-pushout approach can be used as well, but has to be extended by attributes and application conditions first.

7.4 Application of Parallel and Distributed Graph Transformation to Communication-Based Systems: Resume

In this section we want to summarize how parallel and distributed graph transformation is applied to communication-based systems. In chapter 1 we introduced the main requirements and aspects of modeling support for parallel and distributed systems. They are used as guideline for this summary.

Objects and Actions System states described by some object structures can be described by graphs. Node and edge colors may be used to distinguish different types of objects. Graphs modeling AGG-states or data repository states are shown in figures 7.3 and 7.17. Atomicity of nodes and edges refers to the level of structure description. The AGG-graph model does not contain graphical layout of AGG-graphs and the project flow graphs abstract from the contents of documents, revisions, tools, etc. These parts of system specification are referred to the attribute part. Graph productions can describe actions on these object structures. Their application, i.e. graph transformation, models the execution of actions.

Parallelism In general, parallelism is expressed by the parallel application of productions. Parallel execution is nicely supported since the production application has local effects only. Since production application has atomic character, i.e. a production is entirely applied or not applied, this is the unit of parallelism which is offered by graph transformation. All the different kinds of parallel units described in section 1.2.1 may be modeled to some extent in the graph transformation world.

Parallel actions can be simply described by parallel application of productions. The concept of parallel composition is offered in mainly two ways, with or without interaction. Parallel productions describe independent composition whereas amalgamated productions model parallel actions synchronized by common subactions. Parallel graph transformation supports unbounded, problem-oriented parallelism by allowing massive parallelism with flexible synchronization mechanisms using *interaction schemes*. The description of AGG-operations by amalgamation schemes allows to model the update of a graph representation of arbitrary many windows in parallel. They interact in the action on the pure AGG-graph model (without window representations). The firing of Petri nets in example 3.3.7 is modeled by parallel deletion and addition of tokens on all input and output places of a transition. All these parallel actions have to include the same transition.

Functional and logical parallelism resembles parallel graph transformation in some sense. Similar to the computation of a function, the application of a production is side-effect free. As functions production applications can be in causal dependency in the way that the applicability of one production depends on the application of another. The all-match-covering constructions which allow the parallel application of productions at all different matches seem to have a similar effect than higher-order function "map" which applies a function to all elements of e.g. a set.

Local graph transformations can be interpreted as *processes* since they support the property of processes to have separate states and data. Parallel processes would correspond to distributed graph transformation. Suitable network operations on distributed graphs may model the creation of processes. Applications of distributed productions *create rep* and *create dev* in section 7.2 can be considered as process creations. The local process behavior and communication can be determined by distributed productions as productions *create local doc* and *get rlock*.

Local graphs and graph transformation can model encapsulation and concurrency aspects of *active objects*. Application of local productions is restricted to that local part specified in the production like in local production *create local rev* in section 7.2. Thus, a certain kind of encapsulation can be modeled by distributed graph transformation. But hiding of local parts is not explicitly supported by the framework of distributed graph transformation. This observation has led to the development of DIEGO-systems [TS95] supporting hiding and classification aspects. The communication by message passing can be modeled by distributed graph transformation. If we consider distributed productions *get rlock* and *get wcopy*, their application can be interpreted as sending a revision from the repository to the developer. The interface graph can be considered as transfering medium like a channel. Creation and destruction of objects triggered by other objects can be modeled by synchronizing distributed productions such as *create dev* in section 7.2 where the creation of a developer's workspace is dependent of the existence of a data repository.

Communication and synchronization Communication and synchronization is described by graph sharing. Distributed shared graph parts are described by interface graphs and their relations to local graphs. Distributed graph transformation in the double-pushout approach allows synchronous updates only to preserve graph consistency. Nevertheless, asynchronous communication can be simulated, too, as we have seen in the example in section 7.2. Graph sharing has the properties data/object sharing usually has. It supports a transfer of information accessible by all involved parts. For example, if a revision is stored back in the data repository it is accessible by all developers. Communication modeled by distributed graph transformation has immediate effect, i.e. directly after finishing a graph transformation. Considering production *get rlock* the transfer of the revision is modeled as atomic action without any delay between sending and receiving the revision.

Synchronization can be achieved by related graph parts and productions. For example, read access to a revision stored in a data repository is modeled by a shared revision (replicated in several local parts). It must not be modified by a developer without synchronization. Synchronized distributed productions offer the possibility to simultaneously changing related objects in different local parts. Synchronization by common subactions is modeled by amalgamated graph transformation for the non-distributed systems in examples in sections 7.1, 7.3.1 and 7.3.2.

Communication networks Parallel and distributed graph transformation allow the flexible description of communication networks. Star graphs (section 3.2) can describe the connection between each two system parts. Regular network structures as square grids (examples 2.2.3 and 3.3.5) building the basis for massive parallelism can be described by parallel graph grammars.

Distributed graphs offer the possibility for the description of arbitrary distribution topologies and their flexible modification by graph transformation. Thus, the modeling of evolving object structures and systems with component architecture can be supported.

Handling of partial failures Handling of partial failures is usually not described on the level of design specifications. But for the design specification of fault-tolerant systems it may be useful to handle partial failure already on this level. In this case, the occurrence of a failure has to be coded into a graph transformation. The common concepts for handling of partial failures such as replication or transactions can be modeled within the graph transformation paradigm. Different transaction models with their typical consistency properties are modeled by graph transformation in [Koc96].

7.5 Towards a Specification Technique for Communication-Based Systems Based on Graph Transformation

Considering the examples in sections 7.1, 7.2 and 7.3 as well as the evaluation of parallel and distributed graph transformation in section 7.4 we can state that graph transformation has a good potential to serve for a specification technique of communication-based systems.

For a comprehensive modeling support of communication-based systems the following features should be offered additionally:

Local parallelism and distribution operations Parallel execution of actions should be allowed not only on different local graphs, but also on one local graph modeling parallelism inside a local system. The distribution operations "split" and "join" presented in [EBHL88] cannot be simulated in distributed graph transformation as presented in chapters 5 and 6. Both concepts can be covered by generalizing this approach to non-injective network morphisms. This extension of distributed graph transformation would contain contextual parallel and amalgamated graph transformation as special cases.

Distributed amalgamation schemes Interaction schemes and covering constructions on distributed graphs would support more complex distribution operations and synchronization mechanisms with broadcasting facilities. A combination of parallel and distributed graph transformation concepts would offer these additional features.

Application conditions If elementary productions and their subproductions are equipped with additional application conditions there are different possibilities to amalgamate them. One possibility is to not extend the subproduction embedding to the application conditions. In this case, the amalgamated production is equipped with the union of all application conditions of participating productions. This kind of combination has been worked out in [HMTW95]. Another possibility is to extend subproduction embedding and gluing construction to application conditions which is very much dependent on the relation between application conditions.

A comprehensive specification technique has to support the modeling of data part of a system also. To achieve this parallel and distributed graph transformation has to be equipped with *attribute handling*. A first combination of attributed graph transformation with application conditions and parallel graph transformation is proposed for the single-pushout approach in [HMTW95]. Furthermore, analysis techniques for e.g. *consistency* of production application are available for graph transformation useful for reasoning about system design.

Attributed graphs Attributed graphs are suitable to specify also the data part of a communication-based system. The extension of parallel and distributed graph transformation to attributed graphs seems to be problem-free since the main parts of a transformation can be done componentwise. Amalgamating productions variables at synchronizing parts are considered to be equal whereas all others should be handled to be different.

Consistency It is widely stated that consistency is a main subject for specification of communication-based systems. Recently, a construction has been proposed in [HW95] to transfer a graph transformation system into another graph transformation system satisfying a certain set of consistency conditions. It may be worthwhile to extend this construction to parallel and distributed graph transformation.

To specify large systems further *structuring concepts*, e.g. modularization concepts aspresented in [EE96], should be available.

Distributed encapsulated graph object systems (DIEGO) A first approach to describe distributed modular systems based on graph transformation is done by *DIEGO systems* which add encapsulation and classification to distributed graph transformation [TS95]. DIEGO systems model distributed systems consisting of concurrently interacting objects. They support *information hiding* of data and behavior as well as grouping similar objects together into *classes*. Providing a local system part with import and export interfaces, a clear distinction in hidden parts, public parts which may be shared and shared parts is possible. *Use relations* between export and import interfaces are provided to construct component architectures and to allow reuse of classes in different environments.

Refinement and hierarchical graph concepts In [ES95] and [CH95] *refinement concepts* for graphs and graph transformation are introduced to support stepwise system development. Different graph grammars which model a system specification in some stage are related by graph grammar morphisms describing refinement steps allowed. In [ES95] *hierarchical graphs* are introduced to support abstraction of graph parts. Abstraction is combined with encapsulation, classification and refinement concepts for graphs in [ES95] to manage the complexity of large system states.

Distributed graphs can be regarded as hierarchical graphs with two abstraction levels, the network and the local level. Further abstraction levels can be used to describe e.g. encapsulation concepts as presented for DIEGO.

It is the aim of R. Betschko in [Bet] to provide additional structuring concepts and a methodology for the specification of communication-based systems by graph transformation based on distributed graph transformation, refinement and additional abstraction levels.

Transformation units in GRACE Unfortunately the large number of graph transformation features are not all available in one graph transformation approach. It is the aim of GRACE, a new graph and rule centered language under development by researchers from Aachen, Berlin, Bremen, Hildesheim and Leiden [AEH+96], to integrate different graph transformation approaches such that the potentials of graph transformation become more visible. The main concepts of GRACE concern *approach independent graph transformation* and *additional structuring concepts*.

One important contribution of GRACE is the concept of *transformation units* [KK96] which are defined independently of a particular graph transformation approach. Transformation units allow the specification of complex operations on graphs not capable by finite sets of productions. A transformation unit over a certain graph transformation approach consists of input and output graph class specifications, a set of transformation units which are used, a set of productions and a control condition controling the transformation process. It is the idea that transformation units defined over different approaches can communicate. Transformation units are provided with an interleaving semantics which is a binary relation on graphs, i.e. this semantics abstracts from the transformation process. An integration of transformation units into the DIEGO concept, for example, may offer the possibility to describe more complex communications.

Tool support for the specification of communication-based systems Last but not least, *tool support* for editing, animating and analyzing system models based on parallel and distributed graph transformation and additional concepts are needed to really convince people using graph transformation for the specification of communication-based systems. Moreover, a support for generating implementation prototypes of parallel and distributed systems would give graph transformation a real chance to become a description paradigm for communication-based systems.

7.6 Towards Implementation of Parallel and Distributed Graph Transformation

In this section, we show how the concepts of parallel and distributed graph transformation can be mapped to the AGG-graph model (as presented in section 7.1). The main idea is to map the additional structure concerning parallel application and distribution of graphs and productions to different abstraction levels. Furthermore, we discuss shortly how attributed graph transformation can be supported, and which steps have to be done to construct parallel and distributed graph transformations based on their AGG-graph models.

7.6.1 Attributed Graph Transformation on the AGG-system

At the moment, the AGG-system (section 7.1) consists mainly of a powerful *graph editor*. It has been described in some detail in section 7.1. Besides of basic functionality for graph manipulation it offers complex graph operations which support graph transformation in the single-pushout approach. Recently, AGG has been extended by hybrid editing facilities, i.e. editing of textual parts belonging to nodes and edges which may serve as labels or attributes.

In an ongoing work, the editor will be accompanied by a separate *transformation component* to perform animated graph transformation. It will support application of productions at automatically as well as user-selected matches. Moreover, it will offer computation of transformation sequences as well as their animation in a user-defined style.

The graph transformation is supported by the abstraction concept of AGG using two levels: the *graph level* and the *diagram level*. Graphs can be abstracted to single nodes, and morphisms between two graphs can be abstracted to a handle where the source graph is abstracted to the source node, the target graph to the target node. The mapping between these graphs is described by a bunch of edges between corresponding nodes and edges. The mapping is abstracted by an edge between source and target node.

Example 7.6.1 (Graph morphisms in AGG) *Figure 7.27 shows how a graph morphism is described in the* AGG*-system. All circles depict nodes, all solidly and dashed drawn arrows depict edges. The circles and lines drawn dotted do not represent nodes or edges. They have been added to clarify the abstraction relation since in the* AGG*-system different levels are shown in different windows. The abstraction relation can be shown by selections of nodes and edges. The two abstraction levels we distinguish, i.e. the graph level and the diagram level, are shown by the two outer rectangles in figure 7.27. All nodes and edges inside the left (right) circle on the graph level are abstracted to the left (right) node on the diagram level. All dashed drawn edges on the graph level indicating a graph morphism between the two graphs are abstracted to the only edge on the diagram level.*

Direct graph transformation (in the double-pushout approach) can be represented by two squares on the diagram level indicating the double-pushout diagram. The application of a production corresponds to the completion of the two squares on the diagram level as shown in figure 7.28.

Altogether, the underlying graph model of the AGG-system seems to be well suited to describe arbitrary diagrams over graph morphisms. This graph model is supported by the AGG-editor as well as the transformation component.

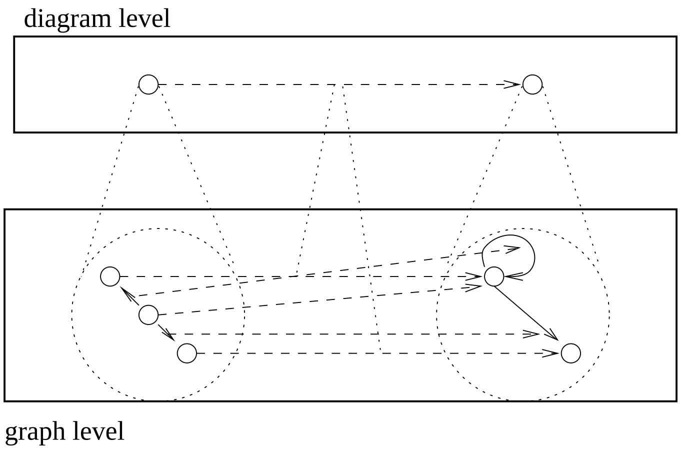

Figure 7.27: A graph morphism represented in the AGG-system

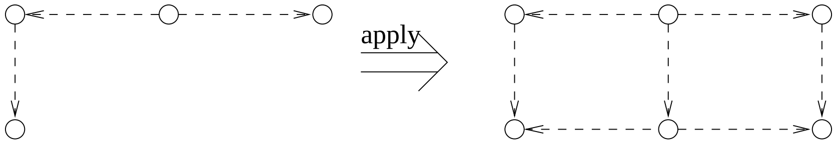

Figure 7.28: Application of a production on the diagram level

The AGG-system is designed as distributed modular system supporting reusability and easy incorporation of new software components. For example, the AGG-editor may be accompanied by special editors supporting graph editing in different application areas. The communication between the editor and the transformation component is two-fold: After editing a graph it can be send to the transformation component to be used for transformation. On the other side, transformed graphs as well as diagrams may be shown by the editor. Furthermore, the transformation component controls access to graphs such that they are not changed simultaneously by different system parts, e.g. the editor and the transformation component. This feature will become really important in the future when additional components such as an analyzing tool or additional editors are added. Compare connections "transform" and "show" in figure 7.29.

To perform *attributed graph transformation* we propose an component environment consisting of the *attribute manager*, and a number of *attribute handlers*. The idea is that each attribute handler manipulates its type of attributes which may be integers, strings, terms, processes or even graphs again. The attribute manager has to coordinate an arbitrary number of attribute handlers with the editor and the transformation component. These two parts of the AGG-system use the attribute manager to create, modify, compare or delete attributes of different types possibly handled by different attribute handlers.

The whole AGG-system will be realized as distributed application to support its

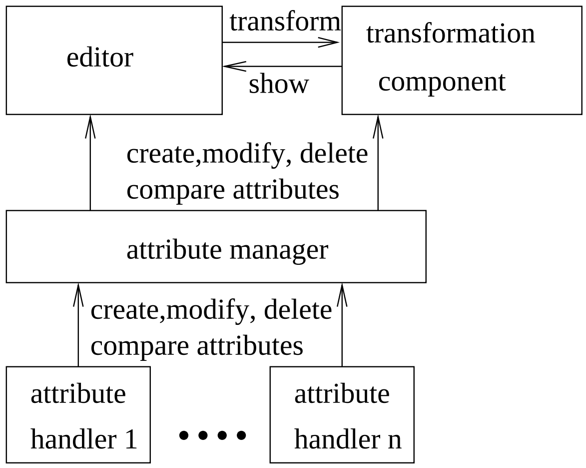

Figure 7.29: Extended AGG-system

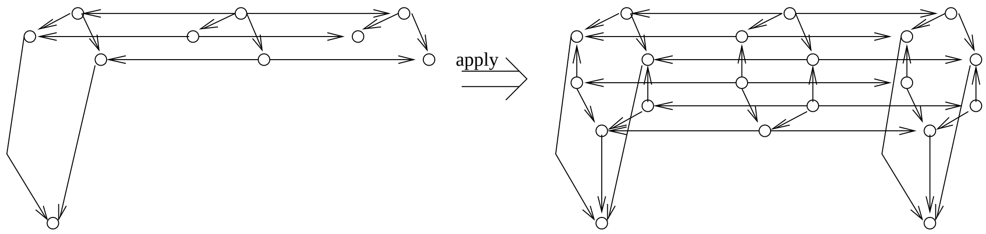

Figure 7.30: Diagram level of a contextual parallel transformation

openness concerning attributes. Providing attributed graph transformation in such a flexible way as described above, the AGG-system can be used to control parallel processes. For this task an attribute handler has to be provided which can create and perform processes and terminate themselves if they get the corresponding signal.

7.6.2 Parallel Graph Transformation on the AGG-system

Similarly to direct graph transformations, parallel graph transformations are performed on more than one abstraction level. On the diagram level we allow more complex diagrams than the double square now. Given a partial covering the contextual parallel and the amalgamated transformation are constructed by a transformation on the diagram as well as on the graph level.

Example 7.6.2 (Diagram level of parallel graph transformation) *The construction of a contextual parallel transformation on the diagram level is shown in figure 7.30. Given a partial covering of size $n = 2$ it has to be completed and the resulting graph has to be computed.*

In figure 7.31 the computation of an amalgamated transformation in shown for a partial covering of the same size as above. The amalgamated production and match

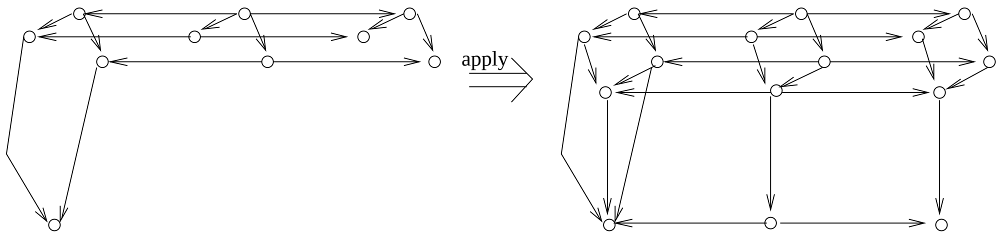

Figure 7.31: Diagram level of an amalgamated transformation

have to be constructed and a direct transformation is computed.

If a parallel action is specified by an amalgamation scheme, all component production sets can be constructed from the given interaction scheme. An interaction scheme is modeled by three abstraction levels where we have the *scheme level* on top of the *diagram level.* On the scheme level the nodes describe productions and the edges subproduction embeddings. A node on the scheme level is refined by a span, i.e. a graph as shown on the scheme level in figure 7.32. Each edge on the scheme level is refined by three edges on the diagram level describing a subproduction embedding.

Example 7.6.3 (Interaction schemes on AGG**)** *In figure 7.32 the* AGG*-model of an interaction scheme with two elementary productions and one subproduction uniquely embedded into the elementary productions is depicted. It shows the scheme level and the diagram level.*

Given an interaction scheme $\mathcal{IS}$ which has to be applied to a given graph G the covering construction determines how the partial covering has to look like. A partial covering consists of a component production set and a parallel occurrence. The component production set can be modeled by an AGG-graph in the same way as an interaction scheme. Its scheme level is restricted to star graphs. The constructed partial covering COV is in the covering set over $\mathcal{IS}$ and G if there is a graph morphism between the graphs depicted on the scheme levels.

Example 7.6.4 (Covering construction on AGG**)** *Figure 7.33 shows the scheme level of an component production set of size $n = 3$ (shown on the left) and of an interaction scheme (shown on the right). Moreover, the relation between the component production set and the interaction scheme is depicted. Note that the subproduction of the interaction scheme is used three times in the component production set. One of the elementary productions of the interaction scheme is used three times, too, whereas the other elementary production is not used at all. These correspondences are depicted by the dashed arrows in figure 7.33. All subproduction embeddings in the component production set are mapped to that between the used productions in the interaction scheme. For simplicity the corresponding edges are not shown in figure 7.33.*

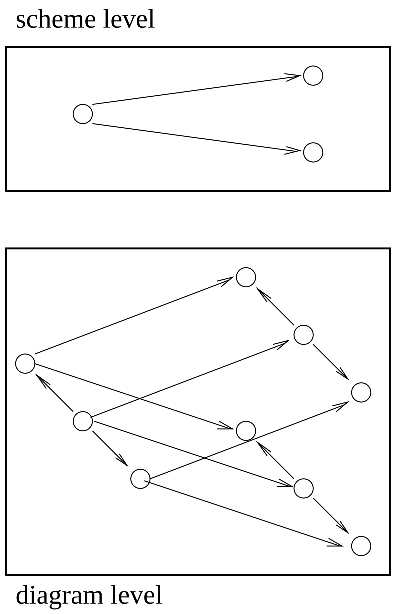

Figure 7.32: Scheme and diagram level of an interaction scheme

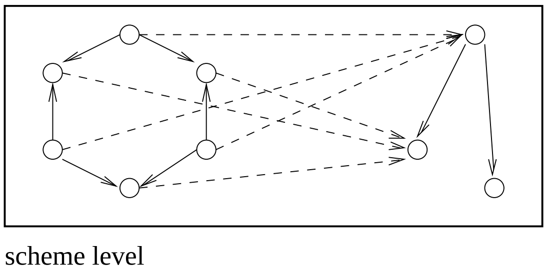

Figure 7.33: Relation of covering and interaction scheme on scheme level

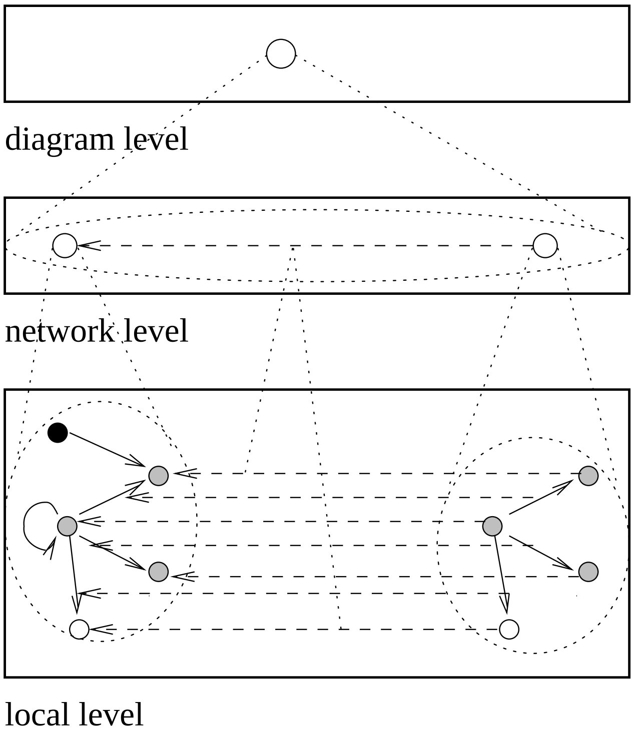

Figure 7.34: A distributed graph as AGG-graph

The algorithmic way for the construction of all-match-coverings and local-all-match-coverings is described in subsection 3.3.1. The construction of pushout-stars can be divided into two steps, as that of pushouts. First the disjoint union of all graphs of the star is computed and then they are identified according to the equivalence relation given by the star morphisms.

7.6.3 Distributed Graph Transformation on the AGG-system

Distributed graph transformation uses also three abstraction levels which function a little bit different as for parallel graph transformation. In the distributed case we distinguish the *local level*, the *network level* and the *diagram level*. Unlike the AGG-model for parallel graph transformation the diagram level has two refinement levels since distributed graphs are transformed.

Example 7.6.5 (Distributed graph on AGG) *In figure 7.34 distributed graph D shown in figure 5.3 is presented as AGG-graph. The local level shows two local graphs and a local graph morphisms in between. Each local graph is abstracted to one node on the network level and each local graph morphism to a network edge. The entire distributed graph is represented by one node on the diagram level. The abstraction is depicted as in figure 7.27.*

Distributed graph transformation can be modeled on the three abstraction levels as described above where the diagram level consists of two squares indicating a double pushout. For the simulation of a distributed graph transformation it has to be shown that the result of the transformation on corresponding AGG-graphs models the resulting distributed graph.

Conclusion

We first summarize the main results of this thesis and then give an outlook over future work.

Summary In this thesis we investigated parallel and distributed graph transformation and their application to communication-based systems. Distributed graph transformation can be regarded as structured transformation of structured graphs. Not considering interaction schemes and covering constructions, parallel graph transformation can be regarded to be a special case of distributed graph transformation where the transformed graphs are non-distributed, i.e. unstructured.

All concepts presented are motivated by communication-based systems. The concepts have first been introduced informally and then formalized on the basis of the double-pushout approach to graph transformation as well as additional notions from category theory. Since the main concepts can be characterized in a purely categorical way they may be generalized to high-level replacement systems [EHKP91a].

Modeling communication aspects of systems by parallel and distributed graph transformation includes modeling concepts common to all applications:

- *Parallelism* is described by simultaneous application of productions. The parallelism is problem-oriented because we allow star graphs or, even more liberally, arbitrary graphs as application structure. The grain of parallelism differs according to the application context. Using parallel graph transformation the parallelism is mostly fine-grained on the level of parallel statements and actions. Distributed graph transformation displays parallelism with a usually larger grain on the level of processes and objects.

- *Communication* and *synchronization* are modeled by graph sharing. Data/object sharing can thus be modeled very well and the properties of this communication type also characterize graph sharing.

- A powerful and flexible description of *dynamic networks* is obtained by applying graph transformation to network structures.

Although the main aspects of communication-based systems can be described by parallel and distributed graph transformation in a clear and flexible way, the concepts presented have to be further developed and combined with other concepts to serve as a comprehensive specification technique for communication-based systems.

Outlook For the development of a comprehensive specification technique for communication-based systems, parallel and distributed graph transformation have to be extended by at least the following features.

Parallel execution of actions should be allowed not only on different local graphs, but also on one local graph modeling parallelism inside a local system. The distribution operations "split" and "join" presented in [EBHL88] cannot be simulated in distributed graph transformation as presented in chapters 5 and 6. Both concepts can be formalized by generalizing the distributed graph transformation approach to non-injective network morphisms. This extension would contain different kinds of parallel graph transformation as special cases.

Interaction schemes and covering constructions on distributed graphs would support more complex distribution operations and synchronization mechanisms. A combination of parallel and distributed graph transformation concepts would offer these additional features.

Attribution of graphs is needed to extend the description to the data part of a system. Moreover, *additional application conditions* may be helpful to further restrict parallelism. A first combination of attributed graph transformation with application conditions and parallel graph transformation is proposed for the single-pushout approach in [HMTW95]. Furthermore, analysis techniques such as for *consistency* of production application have to be available for parallel and distributed graph transformation to be useful for reasoning about system design.

To specify large systems, further *structuring concepts* like modularization concepts [EE96] should be available. A first approach to describing distributed modular systems based on graph transformation was made by *DIEGO systems*, adding encapsulation and classification to distributed graph transformation [TS95]. For the specification of complex operations on graphs, *transformation units* [KK96, AEH$^+$96] have been developed. In [ES95] and [CH95] *refinement concepts* for graphs and graph transformation are introduced. Moreover, in [ES95] *hierarchical graphs* are introduced and combined with the concepts just mentioned to manage the complexity of larger systems.

All these concepts should be integrated into one calculus to provide a solid basis for a specification technique for communication-based systems which may be large and complex.

Since we have shown that distributed graph transformation fits into the *high-level replacement* framework [EHKP91a], it can be expected that results for high-level replacement systems concerning concepts such as parallelism, concurrency and embedding can be carried over to distributed graph transformation.

Last but not least, *tool support* for editing, animating and analyzing system models based on parallel and distributed graph transformation and additional concepts are needed to really convince people using graph transformation for the specification of communication-based systems. Moreover, support for generating implementation prototypes of parallel and distributed systems would give graph transformation a real chance of becoming a description paradigm for communication-based systems.

Bibliography

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, 19(8):26 – 34, Aug. 1986.

[AEH+96] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph transformation for specification and programming. to appear, 1996.

[Agh86] G. Agha. *ACTORS: A model of concurrent computation in distributed systems*. MIT Press, Cambrdge, Mass., 1986.

[AHS90] J. Adamek, H. Herrlich, and G. Strecker. *Abstract and Concrete Categories*. Series in Pure and Applied Mathematics. John Wiley and Sons, 1990.

[AN92] K. Aizawa and A. Nakamura. On relationships between graph L-systems and picture languages: Part II. In Rozenberg and Salomaa [RS92].

[BCF91] D. A. Bailey, J. E. Cuny, and C. D. Fisher. Programming with very large graphs. In Ehrig et al. [EKR91]. Lecture Notes in Computer Science 532.

[BD87] E. Best and R. Devillers. Sequential and concurrent behaviour in Petri net theory. *Theoretical Computer Science*, 55:87–136, 1987.

[BEHL87] P. Böhm, H. Ehrig, U. Hummert, and M. Löwe. Towards distributed graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *3rd Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 291*, pages 86–98, Berlin, 1987. Springer Verlag.

[Bet] R. Betschko. Konzepte und Methodik zur Spezifikation von kommunikationsbasierten Systemen mit Graphgrammatiken. Ph. D. thesis, in preparation.

[Bet92] R. Betschko. Parallele Graphgrammatiken mit Synchronisation. Studienarbeit, Technical University of Berlin, Dep. of Comp. Sci., 1992.

[BFH87] P. Böhm, H.-R. Fonio, and A. Habel. Amalgamation of graph transformations: a synchronization mechanism. *Journal of Computer and System Science*, 34:377–408, 1987.

[BGT91] Horst Bunke, T. Glauser, and T.-H. Tran. An efficient implementation of graph grammars based on the RETE-matching algorithm, in Ehrig et. al. [EKR91], 174 - 189, 1991.

[BK86] J.A. Bergstra and J.W. Klop. Algebra of communicating processes. In *Proc. of the CWIU Symp. Math. and Comp. Sci.*, pages 89 – 138. CWI Monographs I series, NH Amsterdam, 1986.

[BS94] K. Barthelmann and G. Schied. Graph-grammar semantics of a higher-order programming language for distributed systems. In Schneider and Ehrig [SE94].

[BST89] H. Bal, J. Steiner, and A. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261 – 322, (Sept.) 1989.

[Bun82] Horst Bunke. On the generative power of sequential and parallel programmed graph grammars. *Computing*, 29:89–112, 1982.

[CEER96] J. Cuny, H. Ehrig, G. Engels, and G. (eds.) Rozenberg. *Graph Grammars and Their Application to Computer Science*. Springer, LNCS1073, 1996.

[CEL+94] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. An event structure semantics for safe graph grammars. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*. North-Holland, 1994. IFIP Transactions A-56.

[CEL+96] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. An event structure semantics for graph grammars with parallel productions. In Proc. of 5. Int. Workshop on Graph Grammars and their Applications to Computer Science, LNCS 1073, Springer, Berlin, 1996.

[CGHB+95] M. Conrad, M. Gajewsky, R. Holl-Biniasz, M. Rudolf, J. Demuth, S. Weber, R. Heckel, J. Müller, G. Taentzer, and A. Wagner. Graphische Spezifikation ausgewählter Teile von AGG - einem algebraischen Graphgrammatiksystem. Technical Report 95-7, TU-Berlin, 1995.

[CH95] A. Corradini and R. Heckel. A compositional approach to structuring and refinement of typed graph grammars. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2, 1995. http://www.elsevier.nl/locate/entcs/volume2.html .

[Cli81] W.D. Clinger. *Foundations of Actor Semantics.* PhD thesis, MIT Artificial Intelligence Laboratory, AI-TR-633, May 1981.

[CMR96a] A. Corradini, U. Montanari, and F. Rossi. Graph processes. Accepted for special issue of Fundamenta Informaticae, 1996.

[CMR+96b] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *The Handbook of Graph Grammars, Volume 1: Foundations.* World Scientific, 1996. To appear. Preprint available as technical report TR-96-17, University of Pisa, Department of Computer Science.

[Day90] J. Dayhoff. *Neural Network Architectures.* Van Nostrand Reinhold, Melbourne, Australia, 1990.

[DDK93] G. David, F. Drewes, and H.-J. Kreowski. Hyperedge replacement with rendezvous. to appear in proc. of 12th conf. of FST and TCS'92, 1993.

[DFS75] E. Dehnert, R. Franck, and W. Streng. PLAN2D - towards a two-dimensional programming language. In *LNCS 26*, 1975.

[DM87] P. Degano and U. Montanari. A model of distributed systems based on graph rewriting. *Journal of the ACM*, 34(2):411–449, 1987.

[EBHL88] H. Ehrig, P. Böhm, U. Hummert, and M. Löwe. Distributed parallelism of graph transformation. In *13th Int. Workshop on Graph Theoretic Concepts in Computer Science, LNCS 314*, pages 1–19, Berlin, 1988. Springer Verlag.

[EE96] H. Ehrig and G. Engels. Pragmatic and semantic aspects of a module concept for graph transformation systems. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94, LNCS 1073*, 1996.

[EH85] H. Ehrig and A. Habel. Graph grammars with application conditions. In G. Rozenberg and A. Salomaa, editors, *The Book of L*, pages 87–100. Springer Verlag, 1985.

[EHKP91a] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. From graph grammars to High Level Replacement Systems. In *4th Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 532*, pages 269–291. Springer Verlag, 1991.

[EHKP91b] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. Parallelism and concurrency in High Level Replacement Systems. *Mathematical Structures in Comp. Sci.*, 1:361–404, 1991.

[Ehr79] H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *1st Graph Grammar Workshop, Lecture Notes in Computer Science 73*, pages 1–69. Springer Verlag, 1979.

[Ehr87] H. Ehrig. Tutorial introduction to the algebraic approach of graph grammars. In *3rd Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 291*, pages 3–14. Springer Verlag, 1987.

[EK76a] H. Ehrig and H.-J. Kreowski. Categorical theory of graphical systems and graph grammars. In *Conf. Report Algebraic System Theory, Udine 1975, Lecture Notes Econ. Math. Syst. 131 (1976)*, pages 323–351. Springer Verlag, 1976.

[EK76b] H. Ehrig and H.-J. Kreowski. Parallel graph grammars. In A. Lindenmayer and G. Rozenberg, editors, *Automata,Languages, Development*, pages 425–447. Amsterdam: North Holland, 1976.

[EKL91] H. Ehrig, M. Korff, and M. Löwe. Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In *4th Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 532*, pages 24–37. Springer Verlag, 1991.

[EKR91] H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *4th International Workshop on Graph Grammars and Their Application to Computer Science*. Springer Verlag, 1991. Lecture Notes in Computer Science 532.

[EL93] H. Ehrig and M. Löwe. Categorical principles, techniques and results for high-level replacement systems in computer science. *Applied Categorical Structures*, 1(1):21–50, 1993.

[ELM+] H. Ehrig, M. Löwe, J. Müller, J. Padberg, and G. Taentzer. Fundamental concepts and results of algebraic graph transformation systems and high-level replacement systems. in preparation.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, Berlin, 1990.

[ENRR87] H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors. *3rd Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 291*. Springer Verlag, 1987.

[EPR94] H. Ehrig, J. Padberg, and L. Ribeiro. Algebraic high-level nets: Petri nets revisited. In *Recent Trends in Data Type Specification*, pages 188–206, Caldes de Malavella, Spain, 1994. Springer Verlag. Lecture Notes in Computer Science 785.

[EPS73] H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.

[ER76] H. Ehrig and G. Rozenberg. Some definitional suggestions for parallel graph grammars. In A. Lindenmayer and G. Rozenberg, editors, *Formal Languages, Automata and Development*, pages 443 – 468. North Holland, 1976.

[ES95] G. Engels and A. Schürr. Hierarchical graphs, graph types and meta types. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2, 1995. http://www.elsevier.nl/locate/entcs/volume2.html .

[ET92] H. Ehrig and G. Taentzer. From parallel graph grammars to parallel high- level replacement systems. In *Lindenmayer Systems*, pages 283–303. Springer Verlag, 1992.

[GKS91] J.R.W. Glauert, J.R. Kenneway, and M.R. Sleep. Dactl: An experimental graph rewriting language. In Ehrig et al. [EKR91], pages 378–395. Lecture Notes in Computer Science 532.

[Göt88] Herbert Göttler. *Graphgrammatiken in der Softwaretechnik*, volume 178 of *Informatik-Fachberichte*. Springer-Verlag, Berlin, 1988.

[Hab92] A. Habel. *Hyperedge replacement: Grammars and Languages*, volume 643 of *LNCS*. Springer Verlag, Berlin, 1992.

[Har87] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HHT96] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. Accepted for special issue of Fundamenta Informaticae, 1996.

[Him91] M. Himsolt. GraphEd: An interactive tool for developing graph grammars. In Ehrig et al. [EKR91], pages 61–65. Lecture Notes in Computer Science 532.

[HK92] A. Habel and H.-J. Kreowski. Collage grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*. Springer LNCS 532, 1992.

[HMTW95] R. Heckel, J. Müller, G. Taentzer, and A. Wagner. Attributed graph transformations with controlled application of rules. In G. Valiente and F. Rossello Llompart, editors, *Proc. Colloquium on Graph Transformation and its Application in Computer Science*. Technical Report B - 19, Universitat de les Illes Balears, 1995.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hof93] L. Hofmann. Anwendung algebraischer Graphgrammatiken zur Werkstückerkennung im Rahmen von CIM. Studienarbeit, TU Berlin, 1993.

[HS73] H. Herrlich and G. Strecker. *Category Theory*. Allyn and Bacon, Rockleigh, New Jersey, 1973.

[HW95] R. Heckel and A. Wagner. Ensuring consistency of conditional graph grammars – a constructive approach. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2, 1995. http://www.elsevier.nl/locate/entcs/volume2.html.

[Jan93] D. Janssens. ESM systems and the composition of their computations. In *Proc. Graph Grammar Workshop Dagstuhl 93*. Springer Verlag, 1993. Lecture Notes in Computer Science 776.

[Jen92] K. Jensen. *Coloured Petri nets. Basic concepts, analysis methods and practical use*. Springer-Verlag, Berlin, 1992.

[JR91] D. Janssens and G. Rozenberg. Structured transformations and computation graphs for actor grammars. In Ehrig et al. [EKR91], pages 446–460. Lecture Notes in Computer Science 532.

[Kel76] R.M. Keller. Formal verification of parallel programs. *Comm. ACM*, 19(7):371 – 384, 1976.

[KH95] D. Kips and G. Heidenreich. Project flow graphs – a meta-model to support quality assurance in software-engineering. In *Proc. of 2. Int. Conf. on Industrial Engineering and Production Management'95*. Artiba, A. (ed.), Faculte's Universitaires Catholiques de Mons, Belgien, pages 347 - 357, 1995.

[KK96] H.-J. Kreowski and S. Kuske. On the interleaving semantics of transformation units - a step into GRACE. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94, LNCS 1073* [CEER96], pages 89 – 106.

[KLG91] S.M. Kaplan, J.P. Loyall, and S.K. Goering. Specifying concurrent languages and systems with Δ-grammars. In Ehrig et al. [EKR91], pages 475–489. Lecture Notes in Computer Science 532.

[Koc96] M. Koch. Modellierung und Nachweis der Konsistenz von verteilten Transaktionsmodellen für Dastenbanksysteme mit algebraischen Graphgrammatiken. Master's thesis, TU-Berlin, 1996. in preparation.

[Kor93] M. Korff. An algebraic graph grammar model for actor systems. Technical Report 93/21, Technical University of Berlin, 1993.

[Kor94] M. Korff. Single pushout transformations of equationally defined graph structures with applications to actor systems. In *Proc. Graph Grammar Workshop Dagstuhl 93*, pages 234–247. Springer Verlag, 1994. Lecture Notes in Computer Science 776.

[Kor96] M. Korff. *Generalized Graph Structure Grammars with Applications to Concurrent Object-Oriented Systems.* PhD thesis, TU Berlin, 1996.

[KR90] Hans-Jörg Kreowski and Grzegorz Rozenberg. On structured graph grammars, I and II. *Information Sciences*, 52:185–210 and 221–246, 1990.

[Kre81] H.-J. Kreowski. A comparison between Petri-nets and graph grammars. In *Lecture Notes in Computer Science 100*, pages 1–19. Springer Verlag, 1981.

[Kre92] H.-J. Kreowski. Parallel hyperedge replacement. In *Lindenmayer Systems*, pages 271–282. Springer Verlag, 1992.

[LB93] M. Löwe and M. Beyer. AGG — an implementation of algebraic graph rewriting. In *Proc. Fifth Int. Conf. Rewriting Techniques and Applications, '93, LNCS 690*, pages 451–456. Springer Verlag, 1993.

[Lin87] A. Lindenmayer. An introduction to parallel map generating systems. In Ehrig et al. [ENRR87].

[LJ92] A. Lindenmayer and H. Jürgensen. Grammars of development: discrete-state models for growth, differentiation and gene expression in modular organisms. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer Systems*, pages 3 – 21. Springer, 1992.

[LKW93] M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.

[LM93] Igor Litovsky and Yves Métivier. Computing with graph rewriting systems with priorities. *Theoretical Computer Science*, 115:191–224, 1993.

[Löw93] M. Löwe. Algebraic approach to single-pushout graph transformation. *TCS*, 109:181–224, 1993.

[Mac71] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, New York, 1971.

[Man82] B. Mandelbrot. *The Fractal Geometry of Nature*. W. Freeman and Company, New York, 1982.

[Mar89] A. Marsan. Stochastic petri nets: an elementary introduction. In *Advances in Petri Nets*, pages 1 – 29. Springer LNCS 424, 1989.

[Mil80] R. Milner, editor. *A calculus for communicationg systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[NA83] A. Nakamura and K. Aizawa. On a relationship between graph l-systems and picture languages. *TCS*, 24:161–177, 1983.

[Nag79] M. Nagl. *Graph-Grammatiken : Theorie, Implementierung, Anwendungen*. Vieweg, 1979.

[Nag87] M. Nagl. A software development environment based on graph technology. In Ehrig et al. [ENRR87], pages 458–478.

[Plu94] Detlef Plump. Critical pairs in term graph rewriting. In Igor Privara, Branislav Rovan and Peter Ruzicka, eds. Proc. *Mathematical Foundations of Computer Science*, LNCS 841, 556 - 566, 1994.

[PM91] P. Pepper and B. Möller. Programming with (finite) mappings. In M. Broy, editor, *Informatik und Mathematik*, pages 381 – 405. Springer Verlag, 1991.

[PW94] P. Pepper and Wirsing, M., et al. KORSO – A Methodology for the Development of Correct Software. In M. Broy and S. Jähnichen, editors, KORSO , *Correct Software by Formal Methods, LNCS*. Springer, 1994.

[Rei85] W. Reisig. *Petri nets*. Springer Verlag, 1985.

[Roz] G. Rozenberg. The handbook to graph grammars, volume 1: Foundations. to appear.

[Roz87] G. Rozenberg. An introduction to the NLC way of rewriting graphs. In *3rd Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 291*, pages 55–66. Springer Verlag, 1987.

[RS86] G. Rozenberg and A. Salomaa. *The Book of L*. Springer Verlag, Berlin, 1986.

[RS92] G. Rozenberg and A. Salomaa. *Lindenmayer Systems*. Springer Verlag, 1992.

[San93] C. Sander. Implementierung eines Ableitungssystems für algebraische Graphgrammatiken. Master's thesis, Technical University of Berlin, Dep. of Comp. Sci., 1993.

[Sch91a] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Deutscher Universitätsverlag GmbH, Wiesbaden, 1991.

[Sch91b] A. Schürr. PROGRESS–Editor: A text–oriented hybrid editor for PROgrammed graph REwriting SyStems. In Ehrig et al. [EKR91], page 67. Lecture Notes in Computer Science 532.

[Sch93] H.-J. Schneider. On categorical graph grammars integrating structural transformation and operations on labels. *TCS*, 109:257 – 274, 1993.

[Sch94] G. Schied. On relating rewriting systems and graph grammars to event structures. In Schneider and Ehrig [SE94].

[Sch95] Andy Schürr. Logic based programmed structure rewriting systems. *Fundamenta Informaticae*, 1995. To appear.

[SE94] H.-J. Schneider and H. Ehrig. *Graph Transformations in Computer Science*. LNCS 776. Springer, 1994.

[ST90] H. Schween and G. Taentzer. Anwendungen und Erweiterungen der Theorie von Graphgrammatiken auf Bewegungsprobleme in der Robotik. Master's thesis, TU Berlin, 1990.

[Tae92] G. Taentzer. Parallel high-level replacement systems. Technical Report 92/10, TU Berlin, 1992. Submitted to TCS.

[Tae96a] G. Taentzer. Hierarchically distributed graph transformation. In Cuny et.al. [CEER96], pages 304 – 320.

[Tae96b] G. Taentzer. Towards synchronous and asynchronous graph transformations. Accepted for special issue of Fundamenta Informaticae, 1996.

[TB94] G. Taentzer and M. Beyer. Amalgamated graph transformation systems and their use for specifying AGG – an algebraic graph grammar system. Springer Verlag, 1994.

[TS91] G. Taentzer and H. Schween. Movements of objects in configuration spaces modelled by graph grammars. In *4th International Workshop on Graph Grammars and Their Application to Computer Science*, 1991.

[TS95] G. Taentzer and A. Schürr. DIEGO, another step towards a module concept for graph transformation systems. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2, 1995. http://www.elsevier.nl/locate/entcs/volume2.html.

[Wan92] E. Wanke. PLEXUS: Tools for analyzing graph grammars. In *LNCS 532*, 1992.

[YT87] Y. Yokote and M. Tokoro. Concurrent programming in ConcurrentSmalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-oriented concurent programming*, pages 129 – 158, Cambridge, Mass., 1987. MIT Press.

[Zün95a] A. Zündorf. *Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungsSysteme*. PhD thesis, RWTH Aachen, Germany, 1995.

Index