Chapter 1

DISTRIBUTED GRAPH TRANSFORMATION WITH APPLICATION TO VISUAL DESIGN OF DISTRIBUTED SYSTEMS

Gabriele Taentzer, Manuel Koch, Ingrid Fischer, Victor Volle
TU Berlin, University of Erlangen-Nuremberg, Germany
{mlkoch,gabi}@cs.tu-berlin.de,{idfische,volle}@informatik.uni-erlangen.de

Distributed systems demand a number of requirements on specification techniques which do not have to be taken into account for the development of non-distributed software. Allocation of objects and tasks, object replication and migration, remote interactions, multiple threads of control as well as dynamic network topologies are important issues in distributed systems. Some of these distribution issues are already handled by common modeling techniques mostly based on graphical notations, but altogether distributed systems cannot be designed sufficiently with these techniques satisfying all the requirements. In this article, new concepts for visual design of distributed systems are presented which are based on distributed graph transformation as underlying formal specification technique. The kernel of this technique focuses on a structured graph with two abstraction levels, the network and the local level, and its transformation. On the network level, the (possibly) dynamic topological structure of a distributed system is specified. The local level contains the graphical description of local object and data structures where parts of them may be replicated in remote network nodes. Those structures may evolve independently or synchronized with remote structures using a interaction mechanism based on subactions. Network activities as well as local actions are described rule-based. The advantage of using rules in this context is to avoid purely sequential or serie-parallel control flow, since rules naturally support the design of multiple threads of control on the level of events without prescribing the order of execution. Distributed graph transformation is based on the double-pushout approach to graph transformation. Local graph transformations may be attributed and may use application conditions for rules. Using graph transformation as underlying formal framework, distributed behaviour is designed in a way that consistency of the network as well as of all object and data structures involved is ensured in a way that the result is again a distributed graph. This result is obtained from the formalization of distributed graph transformation as double-pushout in the category of distributed graphs and graph morphisms. For the illustration of the concepts presented we use a reference application modeling distributed version management for any kind of document. This subject becomes the more important the more remote software development becomes popular. In contrast to server-client solutions where all documents are held centrally, the documents are really distributed over the involved sites in our case.

1.1 Introduction

Graphical notations have been employed for a long time in specification and design of software systems. Prominent representatives are automata, data flows, message sequence charts, Petri nets and entity-relationship diagrams. Recently a number of object-oriented methods and languages have been introduced (e.g. OMT, OOD and UML) which highly rely on diagrammatic notations.

Traditionally, object-oriented methods concentrate on the design of object class structures, the main static part of an object-oriented system. They offer little help in understanding the interaction processes between objects. Methods which mainly support the modeling of operations and processes are based on classical control models such as those based on finite state-automata, state charts and message sequence charts. The crucial point is how the graphical methods for designing static structures on one side and dynamics one the other side work together to model dynamic object structures. There are modeling languages which combine operational process descriptions with object concepts mostly based on actor systems. But they do not address distribution issues.

Graph transformation is a good means to describe the dynamic aspects of various system structures, i.e. evolving class and object structures or reconfiguration of distributed networks. Graph transformation means the rule-based manipulation of graphical structures. For the visual design of distributed systems graph transformation can be employed twice: to describe the dynamic network topology of a possibly open distributed system and to model evolving data and object structures in the local systems. The usage of graph transformation on the local level can include the description of local as well as remote object interaction, such as object migration, replication, communication and synchronization. All these activities are described in a rule-based manner where a rule shows parts of the object structure before and after. Applying one or more graph rules, i.e. performing distributed graph transformation, results in a distributed scenario.

Common approaches to design distributed systems strongly follow the idea of strict data abstraction, i.e. local systems communicate exclusively by exported method or operation signatures. The design by distributed graph transformation allows to export parts of the internal structures as well as a certain set of actions on these exported structures. This information is extremely useful to find adequate services on remote systems, since names are often not expressable enough to find the right service. Another advantage is the dynamically changing interface which cares about the fact that not all services can be offered at any time. This property can be exploited since interfaces are again described by graphs and graph rules. A service is available if a certain interface rule is applicable to its interface graph.

Using graph transformation as underlying formal framework, consistency of the resulting distributed object structure is ensured. Local structures which are used (remotely) by other components must not be destroyed and components which are interconnected, must not be deleted without deleting also the connection. These features can be provided, since distributed graph transformation is formalized on the basis of the algebraic approach to graph transformation. A distributed graph is described as a diagram in the category of graphs and graph morphisms. A distributed graph transformation step is characterized by a double-pushout in the category **Distr(AGr)** which consists of distributed graphs as objects and distributed graph morphisms. Category **Distr(AGr)** is the union of all functor categories to all shapes defined by network graphs. This feature allows the modeling of dynamic network structures. To model the distributed behavior adequately, this pushout construction is componentwise. The sufficient conditions for componentwise pushout construction are given in this article.

As reference application for this new design technique we chose distributed configuration management for remote software development. In contrast to server-client solutions where all documents are hold centrally, the software is developed at distributed sites in a way that no central document archive exists. Big problems arise to ensure that a client can always get the most recent project documents and that all documents belonging together are not mixed up in different versions. Using distributed graph transformation it is possible to model this application as an open distributed system supporting a flexible number of document archives dynamically used by workspaces. Operations like checking a document in and out are modelled as distributed graph transformations on several local graphs and their interface graphs.

This article is structured as follows: The next section gives an overview on the state-of-the-art of specification and design techniques for distributed systems. Therafter a short review of the double-pushout approach, the graph transformation approach we use throughout this article, is given. Then, distributed graph transformation is informally introduced at a numberr of examples. In section 1.5, distributed graphs and graph transformation are presented formally. And last but not least, the reference application is presented.

1.2 Specification and Design of Distributed Systems: State-of-the-Art

1.2.1 Requirements of Distributed Systems

In contrast to usual software systems, distributed systems are a special challenge concerning analysis and design, languages, validation, tools and testing. Data and tasks are distributed over different processes at different sites and can only be accessed by using certain communication networks running certain protocols. For the description of cooperating distributed processes we need suitable concepts. Distributed scenarios in separate application components have to be modelled to manage the information exchange as well as the synchronization of accesses to resources commonly used. Communication can be synchronous or asynchronous. A strategy for fair solutions of conflicts concerning concurrent access of equal participants to communication channels commonly used have to be considered just as treatment of non-deterministic behaviour of communication media which may loose message or parts of their contents.
In praxis, there is not an infinite set of resources which may be used such that capacity limits have to obeyed and the exception handling for limit exceeding has to be determined. Common used communication media or a restricted number of user licenses for data base accesses are examples for such resources. In contrast to the usual input/output behaviour of sequential systems, communication services have to work continuously, even if some other components broke down.
Moreover, the specification of open distributed systems needs to include the system's architecture into the specification, because such systems require to dynamically reconfigure the system during run time. Such reconfiguration includes the dynamic integration and removal of single components as well as establishing and cutting off connections between distributed sites.
To manage the complexity of a distributed system the reuse of specification and implementation parts is important. Due to this point, design methods for distributed systems should offer abstraction- and refinement concepts and should support the distribution of a system into concurrent, communicating subsystems.

1.2.2 Formal Specification Techniques for Concurrent and Distributed Systems

There are a number of formal specification techniques such as temporal logics, state charts, process algebras, Petri nets, and actor systems to model the structure and behaviour of concurrent systems. In the following, each kind of the above mentioned specification techniques is discussed in little more detail. Moreover, the specification of distribution aspects is touched on for each technique.

Temporal logics There are various kinds of temporal logics [Sti92], e.g. modal logics, CTL and μ-calculus. They are used to formulate the system requirements in a very loose and abstract way. For the specification of concurrent systems temporal logics [MP92] become more and more important. These logics provide special operators to formulate statements on state and event sequences allowed. Temporal logics are better used to formulate requirements than for design. Moreover, there are no means to visualize complex structures.

Extended communicating finite state automata Finite state automata are classical models of computation. The communication between different cooperating automata is performed by asynchronous exchange of signals. To reduce the state space of communicating machines, state charts [Har87] have been developed which allow hierarchies of states. Moreover, parallel states were invented which model synchronous parallelism. The control flow can be visualized in finite state automata, but the data structures are usually textually described. Moreover, non-deterministic events can lead to an explosion of state transitions and, therefore, to unreadability of automata.

Process algebras Process calculi model the behaviour of concurrent processes in form of constraints on observable events. Events are able to describe synchronous interactions between processes. Processes are coupled by ports and complex processes can be composed from simpler ones by a number of composition operators. It is possible to model finite data structures by means of the calculus, but this approach is not usable in practice. Thus, e.g. formalism CCS [Mil89] was combined with algebraic specification concepts for abstract data types leading to language LOTOS [ISO89].
For the specification of dynamic reconfiguration, the π-calculus [?] has been developed. The difference between data and channel names existing in CCS, is removed in this calculus. It is usual to visualize the process structure by graphs . The textual description of such a reconfiguration becomes unreadable very quickly.
The correctness of a specification is proven by the equality of a process behaviour with an already known process.

Petri nets A formalism which is widely used to model and reason about the behavior of concurrent applications are Petri nets [Pet62,Rei87]. It is especially the graphical representation and the operational view of the behavior which supports modeling. The structure of a net reflects the observable control flow, whereas the marking can be used to model distributed system states. Communication is described by common transitions or places. Since the structure of a net is static, the description of dynamically reconfigurable systems is not possible in a direct way.
Elementary nets do not offer the possibility for data modelling. To meet this deficiency, high-level nets have been introduced which combine Petri nets with some kind of data specification techniques, such as algebraic specification or ML [Hum89,Rei91,Jen95]. Extending Petri nets in this way, there is no means to visualize complex data structures. Moreover, replication of data/objects is hardly to describe by Petri nets.
For Petri nets, a large amount of analysis techniques are available. But for the high-level kinds of nets the adaption of the analysis techniques is still not solved sufficiently.

Actor systems The actor model [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 transmission problems which can occur 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*. A *message* has a target actor, the receiver, it may have a value. An *event* is the reception 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 private state. Actor systems have been formally defined by providing a *partial ordering semantics* for actor systems. Several extensions of the actor model have been studied in the literature due to the incorporation of inheritance and modularization.

Conclusion None of the discussed specification techniques covers all aspects of distributed systems. Especially the aspects concerning distribution of data are hardly treated. Dynamic reconfiguration of a distributed system as well as distributed data handling cannot be addressed directly by most of the techniques. Considering all the techniques mentioned, actor systems are that technique which meets most of the features required. But also this technique does not support the handling of distributed data sufficiently, since e.g. replication cannot be addressed directly.

1.2.3 Object-Oriented Modeling Techniques for Distributed Systems

The object-oriented modeling methods and languages, like OMT [RBP+91], OOD [Boo94], and UML [Rat97], nicely support the development of static system structures, like class and object structures, as well as the description of sequential and concurrent behavior. For the description of allocation of objects and tasks, object replication and migration, remote interactions and dynamic network topologies which are important issues in distributed object systems these techniques promise support, but, however, seem to be insufficient. In the following, we consider the most promising language, UML.

UML UML is a recent approach to strengthen the efforts of designing an object-oriented modelling language where all main issues of system analysis and design are taken into account. UML provides deployment diagrams to show a system's network topology and the software components, processes and nodes that live on the network nodes. Such a node represents any kind of processing unit. For the description of network topologies nodes may be connected by associations indicating communication paths between nodes. It is possible to describe the kind of association more exactly by the use of stereotypes.
Instances of components as well as processes and objects that live on a node are located within this node. If one component uses the services of another component this is indicated by a dependency between those components. Objects which move during an interaction are modeled by two or more instances on several nodes connected by "becomes" dependencies.
The description of control flow in UML is supported by sequence and collaboration diagrams. Multi threads of control can be expressed by collaboration diagrams containing several active objects. There are different types of control flow containing synchronous and asynchronous communication between processes. Sequence and collaboration diagrams neglect the distributed object allocation, thus it not possible to explicitly address the distribution issues of object systems during the behaviour design.
The existing notation of deployment diagrams captures the description of migrating objects but does not provide notational elements to express replication of an object. This restriction seems to be easily removed by inventing a further stereotype which indicates this dependency. But there is no feature to describe the behaviour on replicated object structures. Dynamic networking is not addressed by UML. Maybe that is the case because there is no estab-

lished technique to describe an object-oriented system's behavior without any sequential ordering.

1.3 Describing Distributed Systems by Graph Transformation

After reviewing common specification techniques, discussing advantages and disadvantages, we concentrate now on graph transformation. In the following section, a short informal review of the algebraic approach to graph transformation is given. For a detailed presentation we refer to [CMR+97] and [EHK+96]. Afterwards, distributed graph transformation is introduced informally. Furthermore, we show how distributed graphs and rules can be used to design typical problems in distributed systems. This section is rounded off by a comparison of our graph transformation approach with other approaches and, moreover, with other specification techniques.

1.3.1 Informal Approach to Algebraic Graph Transformation

Firstly, the basic features of algebraic graph transformation are reviewed. The presentation follows an informal style and combines several concepts for algebraic graph transformation. The formal definition of the algebraic approaches can be found in [CMR+97] and [EHK+96]. In Section 1.5, the double-pushout approach on attributed graphs together with application conditions for rules is formally presented.
Graph transformation is defined by the application of graph rules which model the permitted actions on graphs representing system states. Graph transformation defines a relation on state graphs which can be iterated arbitrarily yielding the transformation process. In this way a graph grammar, consisting of a start graph and a set of rules, gets an operational semantics.
First, we present the kind of graphical structure suitable for algebraic graph transformation and afterwards the transformation process is explained. Finally, we have to mention AGG which is a tool for developing and interpreting algebraic graph grammars. Graph transformations may be accompanied by computations on attributes. These computations are described by expressions of the object-oriented programming language Java which has been developed by SUN. The AGG-system is presented in [TER98].

Graphs and Graph Morphisms The following graph transformation concepts are presented for *directed graphs* as basic structures where nodes and edges may be typed and attributed. The formalization in Section 1.5 is done for slightly more general attributed graphs. A graph consists of a set of nodes,

a set of edges and two mappings assigning a source and a target node to each edge. Moreover, nodes and edges are typed by a typing function where a type may be an element of an arbitrary set. Furthermore, nodes and edges may be equipped with attributes. An attribute can be a number, a text string, an expression, a list, etc. or even a graph again. Attributes are also typed. Attribute operations which have to be compatible with these types are available to manipulate the attributes. An attribute signature ([EM85]) tells us what sorts of attributes and what operations on them may be used.
Graphs may be related to each others by using *graph morphisms*. The simplest graph relation is the subgraph embedding where inclusions on node and edge sets have to exist such that source and target mappings as well as types and attributes are preserved. In general, a graph morphism is a family of node, edge and attribute mappings such that they are compatible with source, target and typing mappings as well as with attribute operations. The attribute mappings are usually simple ones, such as identities or variable instantiations.
Examples for attributed graphs and graph morphisms can be found in Figures 1.2 and 1.3. Graphs are depicted inside of rounded rectangles and graph morphisms are shown by arrows between these rectangles. Those nodes are mapped to each others which have corresponding types and attributes, resp. If the nodes are already mapped, the mapping of edges can be uniquely determined in a way that both mappings together form a morphism.

Graph Rules Graph rules describe graph transformation steps in an *if-then* style. *If* a certain graph, namely the left-hand side of the rule, does exist *then* it may be transformed into a new one, the right-hand side of the rule. Both sides may overlap in some nodes and edges. These nodes and edges remain unchanged during a transformation and function as central points where the transformation takes place. They form an intermediate graph which is part of the left and the right-hand side of the rule.
To transform also the attributes the left-hand sides may contain variables to be instantiated at application of the rule and in the right-hand sides arbitrary computations on these variables and constants may be stated using operations available for these kinds of attributes.
Moreover, a set of *application conditions* may be appended which restrict the application of the rule concerning additional constraints on the existence or non-existence of subgraphs as well as certain predicates on attributes. Special application conditions are the *dangling* and *identification conditions*. The dangling condition assures that no dangling edge will arise in a host graph by applying the rule. It requires that whenever the image of a left-hand node contacts some edge not being the image of a left-hand edge, then this node

has to be preserved (in the intermediate graph). The *identification condition* requires that the occurrence from the left-hand side into a host graph – which means nothing else than a graph morphism between these two graphs – may only identify nodes and edges which are preserved.

Graph rules with or without application conditions are depicted in Figures 1.9 and 1.10. Nodes which correspond to each others coincide in the same layout. Application conditions are presented by separate graphs equipped with logical operators. Although the application conditions are always preconditions they are depicted on the right of the rule or beneath, because we first want to show the rule and then all its application conditions. Two operators are used throughout this article. The non-existence of a graph structure is indicated by $\neg\exists$. The conditional graph existence is expressed by $\forall$ graphs $\exists$ graph. In Figure 1.9, a negative application condition is shown. Whereas the rule in this figure describes the creation of a new Account-node, the negative application condition prohibits the application of this rule, if there is already an Account-node with the same identity number as attribute. Please note that the Account-nodes are different ones.

In Figure 1.10, an attribute condition is depicted in the left-hand side of the rule. This condition has to be satisfied for the application of the rule.

Graph Transformation A graph rule is applied to a graph performing the following steps:

- *Choose* an occurrence of the left-hand side of the rule in the host graph.
- *Check* the application conditions of the rule. If they are fulfilled the occurrence found is called *match*.
- *Remove* the occurrence of the left-hand side (without the preserved intermediate graph) from the host graph.
- *Glue* the resulting graph and the right-hand side at the occurrence of the preserved intermediate graph, that is, add the right-hand side to the resulting graph such that the preserved part and its occurrence overlap. Moreover, execute the attribute computations stated in the right-hand side according to the variable instantiation of the occurrence.

We followed the double-pushout approach to graph transformation here, compare also Section 1.5.

Graph Grammars Given the notions of a graph rule and a graph transformation, graph grammars, and generated languages can be defined as follows. A *graph grammar* consists of a *start graph*, a set of rules and a type set for nodes and edges. Moreover, the attribute types and operations allowed have to be determined for the graph grammar.

The set of all graphs derived from the start graph by the graph grammar's rules which are typed accordingly to its type set is said to be the *language* of this graph grammar.

Note that this concepts allows non-determinism: Given a graph, any rule may be chosen from the set of rules applicable to this graph. Moreover, for a chosen rule, there may be several occurrences of its left-hand side in the host graph. Thus, the result of a graph transformation depends on the chosen rule and the chosen occurrence. Furthermore, there is no control structure on the set of rules in a graph grammar: A rule may be applied whenever there is an occurrence of the left-hand side and the application conditions are satisfied.

1.3.2 Concepts of Distributed Graph Transformation

This section presents the main concepts of distributed graph transformation in an informal way. Distributed graph transformation is structured graph transformation on two abstraction levels: the network and the local level. Distributed graphs can be nicely used to model the system's network and data structures which are distributed over the local systems. The graphical representation of these structures uses a UML-like notation ([Rat97]). All kinds of local and distributed object interactions as well as dynamic network changes can be described by distributed graph transformation. The concepts are presented formally in Section 1.5.

In the following, the concepts of distributed graph transformation are presented in paragraphs 'Distributed Graphs', 'Local Views', 'UML-like Notation', 'Distributed Graph Rules and Application Conditions', 'Distributed Graph Transformation' and 'Local View Grammars and Distributed System Grammars'. The design of distributed systems by distributed graph transformation is demonstrated at several examples in paragraphs 'Designing Distributed Data Structures', 'Designing Local Actions', 'Dynamic Networks', 'Synchronizing Distributed Actions' and 'Local View Rules and Their Application'.

To round off this section, distributed graph transformation is compared with other graph transformation approaches used to model distributed systems. Moreover, distributed graph transformation is comparde with the specification techniques mentioned in Section 1.2.

Distributed Graphs Distributed graphs are structured graphs with two abstraction levels: the network and the local level. The *network graph* consists of nodes which represent local systems. These nodes are called *network nodes*. The edges of the network graph describe connections between local system parts. A *distributed 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.

The local graphs are used to represent current states of local system parts whereas local graph morphisms describe the interrelation between two local states. Local graphs together with local graph morphisms form the local level. Local graph morphisms are the essential basis for interaction of local systems. They are used to describe which local graph objects and relations correspond to remote objects and relations. 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*. See 1.1 for an abstract and 1.2 as well as 1.3 for concrete examples of distributed graphs. In Figures 1.2 and 1.3 network nodes are represented by rounded rectangles and network edges by bold arrows in between.

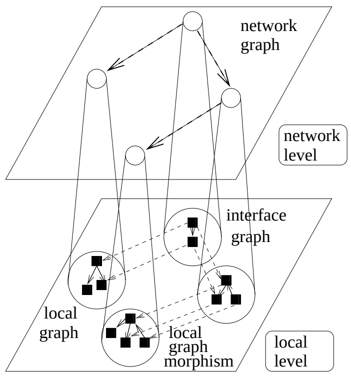

Figure 1.1: Example of a distributed graph

Local graphs may carry any number of attributes. Usually, the local graphs carry attributes to different signatures. This is especially useful for nodes and edges in interface graphs which do not have to carry all the attributes their correspondents in other local graphs do. See Figure 1.3 for an example where attribute *id* is not in the interface graphs.

Designing Distributed Data Structures Usually, the underlying network structure of a distributed system is described by a graph. This is done here by a network graph as introduced above. A local system's state can also be described by a graph where the nodes represent data objects and the edges showing any kind of relations inbetween.

Interfaces between local systems are special subsystems, i.e. interfaces contain not only action names and signatures, but show also parts of the local states which are exported and imported. A local system can have two different kinds of interfaces: import and export interfaces.

If two local systems have a network connection the import interface of one system and the export interface of the other system are connected. This connection is asymmetric, i.e. the import interface of one local system may be connected to the export interface of the other. Such a connection is modeled by a graph morphism from the import to the export interface. The graphs in these interfaces may carry any subset of attributes defined in local graphs. A typical situation of two local systems connected by an import and an export interface is shown in Figure 1.2. Generally, a local system can have any number of import and export interfaces.

A graph object in an import interface carries over those attributes available in the export interface it is connected with. Using these features the designer can decide very flexibly how much of the local states is exported. It is possible to export only special nodes which carry all needed information for communication. An example to this kind of data abstraction is shown in Figure 1.2. A next step to relax the data abstraction is to show graph parts, but without attributes. If we relax further, also attributes are exported. Consider Figure 1.3 as examples. If all attributes of a graph object are exported, these graph objects are fully replicated in the remote system.

Given two local systems where certain local objects should use remote objects, this situation is modeled by a distributed graph as follows:

The network graph consists of four network nodes. Two of them model the local systems and the other two model an import and an export interface. Network edges run between interfaces and local systems.

Local nodes and edges which shall be used remotely or replicated are put into the export interface graph. Those which are really used occur in the import interface. Furthermore, the graph in the import interface is mapped to its corresponding part in the export interface and in the local graphs. See Figure 1.2 for an example of remote usage of a simple request broker. In this example, absolute data abstraction is obtained by exporting exclusively the broker. It is responsible for the communication with the bank. Typical services such as the acceptance of a request are modeled below. To keep the graphs simple not all the attribute names and/or values occur in these and in all the following examples. Here, a bank manages a list of accounts ("listOfAcc") and an account only has an identity ("id") and a name as attributes. An account stub does not have these attributes directly, but holds a reference ("ref") to its origin.

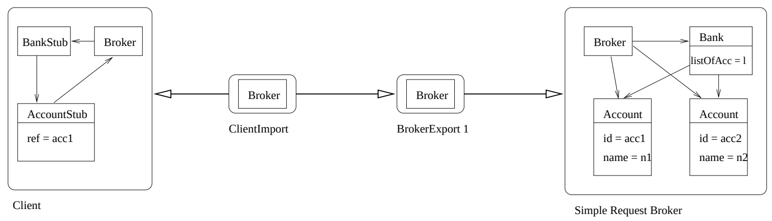

Figure 1.2: A sample distributed graph showing a state of a simple request broker

In Figure 1.3 an example for a replication is shown where a very simple state of a bank system is replicated for security reasons. The most usual reason for replication are security issues. Here, all accounts are exported for replication. Currently, just one of them is already replicated,i.e. the account with name "n2".

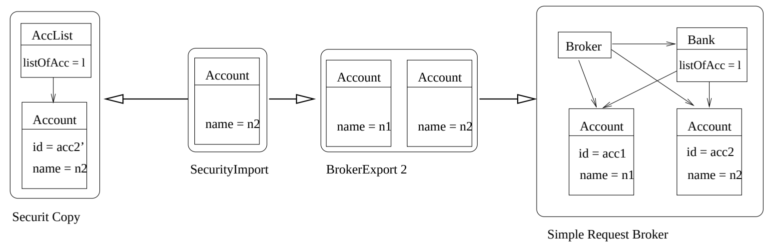

Figure 1.3: A sample distributed graph showing a bank with security copy in a certain state

Local Views Figures 1.2 and 1.3 show a somehow global view on their distributed graphs. Modeling a distributed system it is more adequate to take a local view on each of its local systems. A local view consists of a local system itself, a set of its import interfaces and a set of its export interface.Furthermore, a local view may contain a public import interface between each import and each export interface. A public import interface may contain local nodes and edges which are imported and also exported. Moreover, a local system knows

external export interfaces so that it can connect an own import interface to it. Furthermore, all external import interfaces that use local export interfaces are locally visible to know what is used in a remote import interface. If an import and an export interface of one and the same remote system is visible locally, the public import interface possibly existing in between is also visible.
The states of local systems and interfaces are modeled by local graphs whereas the local graph morphisms in between are used to describe the relations between these different parts. If the network graph has the structure described above, its corresponding distributed graph is called *local view graph*. In Figure 1.4, the local views of the states shown in Figures 1.2 and 1.3 are depicted. These examples do not show the use of public import interfaces. They are used in the reference application when modelling the check in of a document into the revision archive. (Compare Figures 1.33, 1.35 and 1.37.)

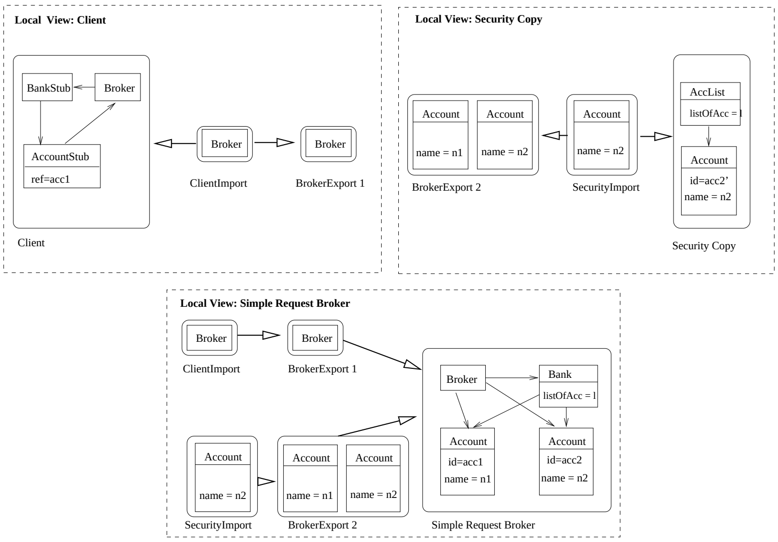

Figure 1.4: Local views on distributed graph in Figure 1.2

UML-like Graph Notation Figure 1.4 shows the direct visualization of the local views. For applying distributed graph transformation to design distributed systems, a UML-like notation represents the distributed object structure in a more compact, but restricted form. In this notation, network nodes are depicted as squares. Import interface nodes are not depicted, but interrelations between remote objects are directly modeled by dashed arrows between them. Different export interfaces are distinguished by named rectangles. (This layout is taken from the graphical notation for packages in UML.) It should be pointed out that one node may be part of different export interfaces. In this case the export areas overlap. Objects which are exported are filled grey. All their attributes which are allowed to be replicated have a grey background, too.

This compact notation of local views is possible, if all the local graph morphisms are injective which is the natural way to use them. Each interface graph with two injective graph morphisms to its target graphs it is connected with can be visualized by a set of arrows connecting the corresponding local objects directly. This is done for the import interfaces of different local systems. Moreover, an export interface is not depicted separately, but as a part of its local graph. Furthermore, all exported objects which have been imported are automatically in the public import. Such an objects is grey and it is the source of at least one dashed arrow indicating that it is imported. See Figure 1.5 for the UML-like notation of the sample local views presented in Figure 1.4. For an example of a public import in UML-like notation compare e.g. Figure1.35. In each local view that network node is drawn bold face which is the owner of the view. Taking the global view none of the network nodes is depicted with bold frame.

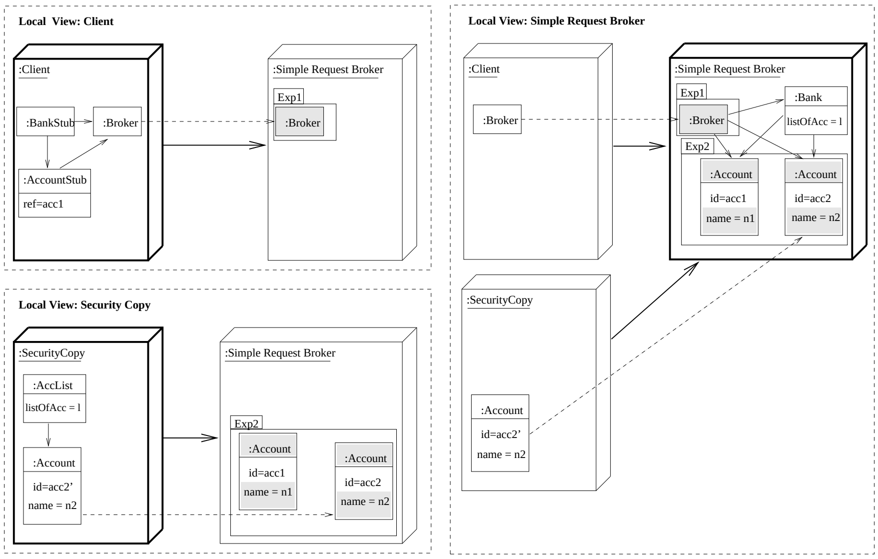

Figure 1.5: UML-like notation of the above local views

Distributed Graph Rules and Application Conditions A distributed graph rule consists of a network rule and local rules for all network nodes which are preserved by the network rule. All newly created network nodes as well as those which should be deleted are equipped with a local graph. Local rules have to be compatible with local graph morphisms. See Figure 1.6 for a schematic representation of a distributed rule. Distributed graph rules are useful to describe dynamic network structures by a network rule as well as dynamic local object structures in each component by a set of local rules.
As for distributed graphs, distributed rules can be notated in a global or in different local views. A distributed rule in a local view, called *local view rule*, comprises one local system rule. Moreover, it may contain local export and import rules as well as remote import and export rules. While local interfaces have to be connected to the local system, remote interfaces are also allowed to be disconnected from local ones. Since the network node can differ from the left to the right-hand side of a rule, also network actions can be described in a local view. (See paragraph 'Dynamic Networks' in the following.)

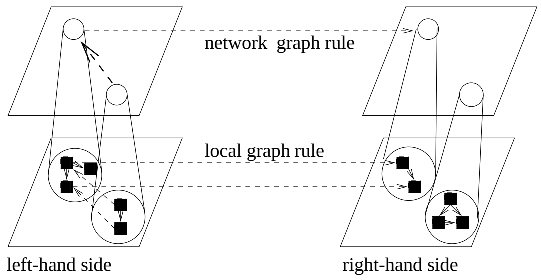

Figure 1.6: Schema of a distributed rule

To apply a distributed rule several application conditions have to be checked to ensure that the resulting structure is again a distributed graph, i.e. no dangling edges occur and no local morphism becomes partial after application. Moreover, the application of distributed rules can be performed componentwise and should not produce implicit side-effects on the graph structure.
Firstly, the application conditions for all local rules as well as for the network rule have to be satisfied. Furthermore, a local rule application on a target graph is not permitted if it destroys the mappings coming from source graphs, i.e. a source graph cannot be mapped totally to its target graphs anymore. Analogously, source graphs are not allowed to be extended without taking the target graphs they are connected to into account. Otherwise the additional structure is not properly connected to the context. Moreover, attributes of

nodes and edges in target graphs are not allowed to be manipulated if they have preimages in source graphs, because the attribute mapping would violated. These conditions are summarized in the *connection condition* rephrased in table 1.1.

connection condition	
deletion	do not delete graph objects of source graphs without deleting the equipped local mappings into target graphs; do not delete local nodes and edges which have preimages in interfaces;
insertion	do not extend source graphs without mapping the new graph objects to the connected target graphs
attributes	do not change attributes of graph objects which have preimages in interfaces;

Table 1.1: The connection condition

The deletion and creation of network nodes and edges has to be done very cautiously to avoid inconsistencies. Clearly, if a network node shall be deleted its local graph has to be deleted, too. Deletions must not destroy local graph morphisms. Thus, deletion of a network edge means deletion of its whole local graph morphism. This means especially that the source graph of this local morphism has to be completely known by the rule. Otherwise, there may be additional mappings which are not deleted. Creations have to fit, too. If a new network edge has to be inserted, the local graph of its source node has to be completely known by the rule. Otherwise, there may be additional graph objects in this source graph which are not mapped by the new local graph morphism. These conditions, summarized in the *network condition*, are rephrased in table 1.2.

network condition	
deletion	if a network node has to be deleted, its local graph has to be completely known by the rule; if a network edge has to be deleted, the local graph of its source node and the local morphism have to be completely known by the rule;
insertion	if a new network edge has to be inserted, the local graph of its source node has to be completely known by the rule;

Table 1.2: The network condition

Consider distributed graph D in Figure 1.7. Nodes with equal numbers are mapped to each others. Distributed graph rules “add” and “add interface” shown in Figure 1.8 can be applied to graph D in different ways. But not all occurrences are real matches, because the connection or the network condition are not fulfilled.

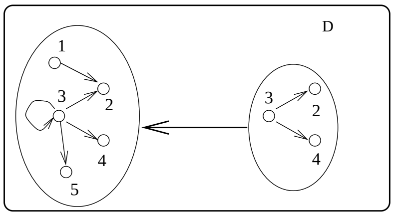

Figure 1.7: A sample distributed graph

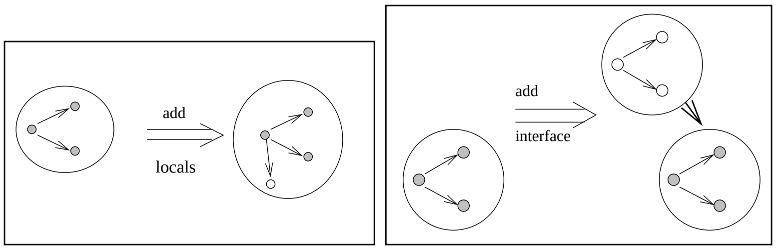

Figure 1.8: Sample distributed rules

Distributed rule "add" can be applied to the left graph of D arbitrarily, because it does not delete anything. It cannot be applied to the right graph of D, because in this case a node and an edge would be created which are not mapped by a local graph morphism to the left-hand graph (connection condition).
The inverse rule of "add" where the left and the right-hand sides are exchanged describes the deletion of a node and an edge. It cannot be applied to the left-hand graph of D in the way, that node "4" is deleted, because its image in the right-hand graph would loose its origin (connection condition). It is applicable, if node "5" is deleted. Moreover, the inverse rule cannot be applied to the right-hand graph of D injectively. A non-injective match would violate the connection condition, because a local node and edge are deleted without their local mappings (connection condition).
Distributed rule "add interface" is allowed to be applied to D, because it fulfills the network condition. The inverse of this rule can be applied to D, because the local matches can be chosen bijective and the network condition is fulfilled, then.

Distributed Graph Transformation Distributed graph transformation is performed by a number of simple graph transformations on both abstraction levels of a distributed graph. Besides the network transformation a number of local graph transformations may be executed in parallel. A local transformation is performed in each network node which is preserved by the network transformation. Relations between transformed local graphs are induced by the transformation steps. Furthermore, network nodes and edges may be deleted and newly created as indicated by the applying rule. The context network remains unchanged.
If the application of a distributed graph rule satisfies all application conditions the result is again a valid distributed graph. This coherence is extremely useful to design consistent data structure transformations in a distributed environment.
Distributed graph transformation supports a model where local components specify their own process. Insertion of a new local component models the start of a new process and deleting a local component means process termination. Each local rule modeling an object interaction can define its own thread of control. Thus, threads of control are modeled here independently of objects and allow concurrency on the level of events. If distributed rule applications are independent of each others they may happen simultaneously or in either order, i.e. truly concurrently.

Designing Local Actions A local action can be modeled by a distributed rule consisting of a local rule which describes the local action and an identic network rule which just describes the preservation of the component where the local action takes place. Changes on export interfaces may also be performed locally as long as graph parts used remotely are not deleted. In this case a synchronization has to be performed to inform the using system that exported parts will be deleted. Adding objects and links to an own export interface is always possible.
Such a distributed graph rule can be applied to a distributed state graph if the local action is compatible with the context where it takes place. The rule application is prohibited if the *connection condition* is not satisfied: *Exported objects must not be deleted, if they are used remotely.* The network condition is always fulfilled, because the network rule is identic in this case. Local actions in different components can be executed simultaneously or in either order, i.e. in a true concurrent way, because they do not affect other components. An example of a distributed rule describing a local action, again in UML-like notation, is shown in Figure 1.9. This rule models the opening of a new account with some id-number, if there is not already an account with this id-number.

Moreover, the new account is put into export interface 'Exp2'.

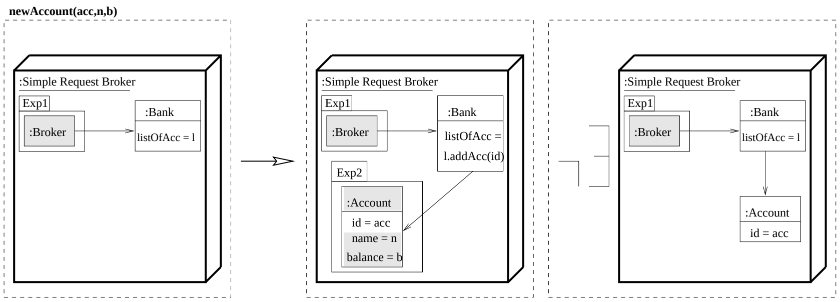

Figure 1.9: Example: NewAccount(acc,n,b): broker's view

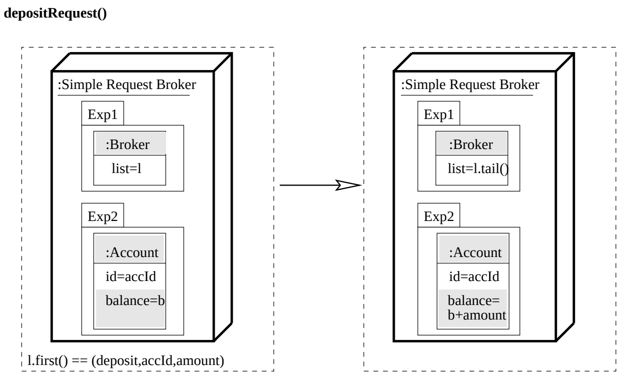

Figure 1.10: Example: Handling a deposit request: broker's view

In Figure 1.10, another local action is depicted handling a deposit request to a broker of a bank. If this rule is applied without any synchronization, the exported account must not be imported by a remote system. Otherwise, the connection condition would be violated.

Complex local interactions such as iterated or case-driven computations can be modeled by a set of local rules all taking place in the same local component.

Dynamic Networks Modeling open distributed systems, reconfiguration of the network may become necessary. With graph transformation we get the possibility to model dynamic networks, i.e. network changes are described

by graph rules. To avoid inconsistencies, the deletion and creation of local components has to be done very cautiously. This means that deletions must not destroy the remote relations, i.e. a component may only be deleted if all its objects are not used remotely. Additionally, deletion of local components must not destroy the network structure, i.e. connected components have to be deleted together with their connections. If components shall be deleted, their final states have to be modeled to specify that moment when deletion is permitted.
Creating a new local component, its initial state has to be consistent with the components it connects. In the easiest case a component's initial state is empty and/or the new component is still disconnected from the network. Compare for example Figure 1.11, where a new printer is created without any network connection. See e.g. Figure 1.12 where a printer is connected to a server. Moreover, connections between components may be deleted and created, too. All these network changing action may run concurrently to each other or to local and synchronized actions.
Altogether a network reconfiguration is also described by a distributed graph rule where its network rule describes the changes of the network topology. Moreover, the initial (final) state of the newly created (deleted) components are described. For preserved components those exports have to be specified which will be involved in remote object relations. See e.g. Figure 1.12 where a new job queue has to be created on the server's side to communicate with the newly connected printer.

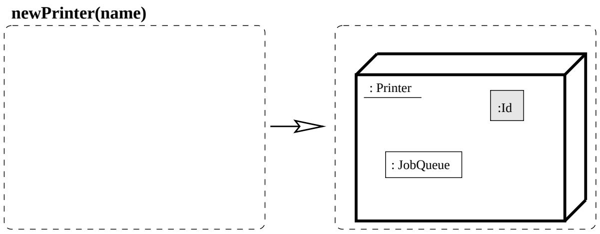

Figure 1.11: Example: NewPrinterComponent: printer's view

Synchronizing Distributed Actions Synchronous actions are modeled by several local rules in different components connected with each other via common interface rules. If two local systems want to synchronize, their views have to overlap in at least one import and one export interface which are connected with each others. The synchronization rules defined in the local views of con-

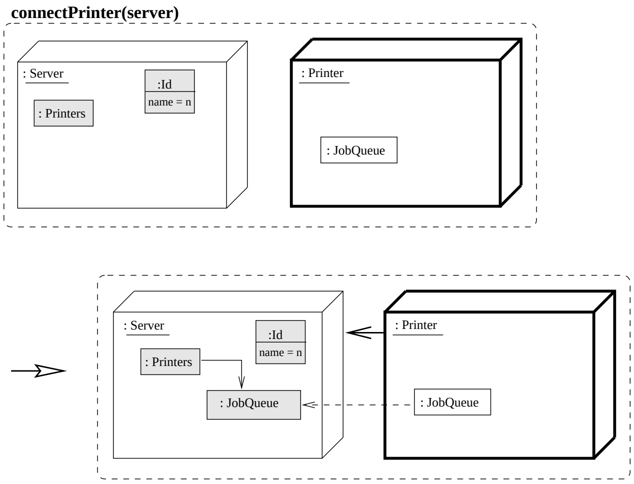

Figure 1.12: Example: Connection of new printer: printer's view

nected systems have to coincide in the subrules of import and export interfaces involved.

Figures 1.13 and 1.14 show two sample rules modeling the opening of a new account in the broker's view and in the view of the security copy as synchronous action. Describing the operation in this way, each account newly opened is immediately replicated for security reasons. The broker's view rule contains the opening of the account at the bank and its export and its import for replication to perform a security copy. In the view of the security copy a new account is put into the export of the bank and replicated for security reasons.

This action can also be performed asynchronously by first opening the new account at the bank using the local rule in Figure 1.9. The new account is immediately exported. Thereafter a rule 'replicateAccount' in the security copy's view may be preformed which replicates the new account. The result is the same as applying the rules in Figures 1.13 and 1.14 simultaneously. A similar comparison between synchronous and asynchronous communication is given at the end of Section 1.4 which different variants of check-in operations for documents of a revision archive are presented.

Considering Figure 1.15 a service is available which shows a list of all jobs of a certain printer. Exporting the job queue this information is permanently offered such that the client does not have to send a request to the server. Clients just copy the information as modeled in Figure 1.15. If attributes of

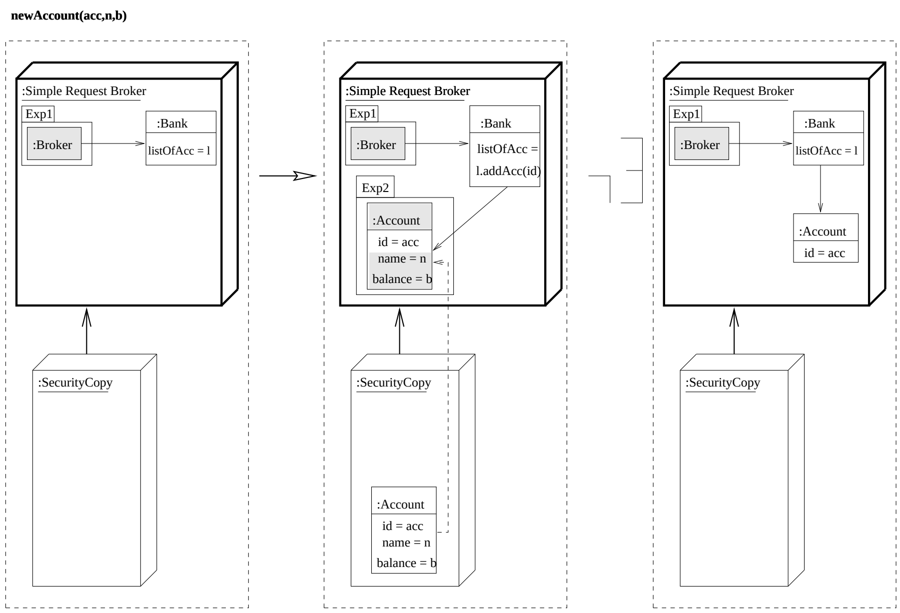

Figure 1.13: NewAccount(acc,n,b) : broker's view

the job queue change a synchronization with the client has to run to update its data. But nevertheless the server is able to remove the *JobQueue* node from the export if none of the clients replicate the job queue, i.e. holds a reference on it. Thus the server's interface may change concerning attributes as well as nodes and edges. Moreover, it is possible that a client triggers a rule application within the export interface of a server to retrieve information. If such a retrieval cannot be performed this corresponds to non-applicability of such a rule.

Local View Rules and Their Application For synchronous as well as asynchronous communication several local view rules have to be combined into one distributed rule. Local view rule applications in several local systems are synchronized if the rules are applied to common interfaces. The synchronization takes place at these common interfaces. Since local view rules which delete e.g. exported objects are not applicable when these objects are imported, such a local view rule has to be synchronized with corresponding rules handling connected import interfaces and performing the same action on their copies of

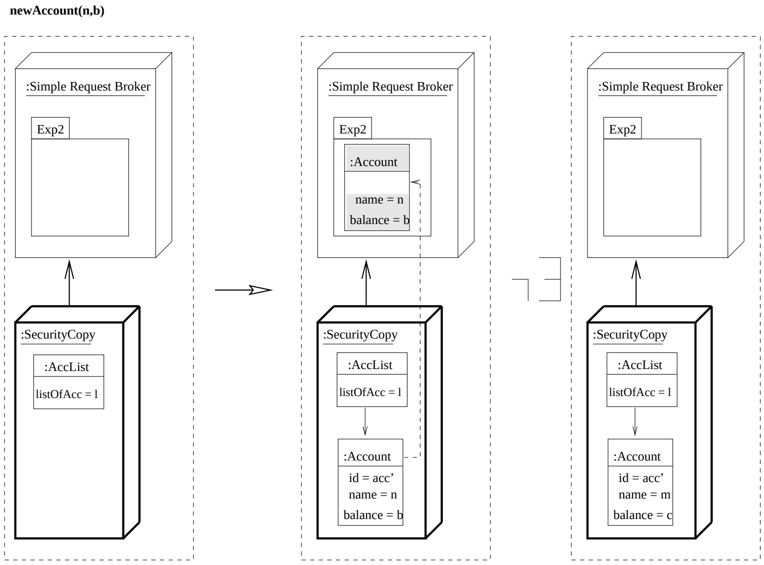

Figure 1.14: NewAccount(acc,n,b) : view of security copy

the export interface. We discuss the constructions of the corresponding synchronized rules at the check-in operations for documents of a revision archive presented in Section 1.4.

Considering Figures 1.33 and 1.37 an example for a synchronous communication is given by two local view rules describing the check-in in the archive's view and in the view of a work space. To describe some action in some view just means to develop a special distributed rule. This rule contains the manipulations of the local system as well as of all interfaces which take part in this action and are visible in this view. To model a synchronization all local view rules which model a part of the synchronous action have to be coupled. The idea for coupling are the common interfaces of local views. Since two local views may overlap in common import and export interfaces, two local view rules can be combined to a synchronized rule at the subrules of their common interfaces. We assume for this operation that the subrules of the interfaces occurring in both views are equal to each others. Considering Figures 1.33 and 1.37, this condition is satisfied for export of the work space and for the import of the archive. (The import interface consists of all nodes which are connected to an exported node by a dashed arrow). The union of both local view rules is easily to obtain. Compare Figure 1.38 for a sample synchronized

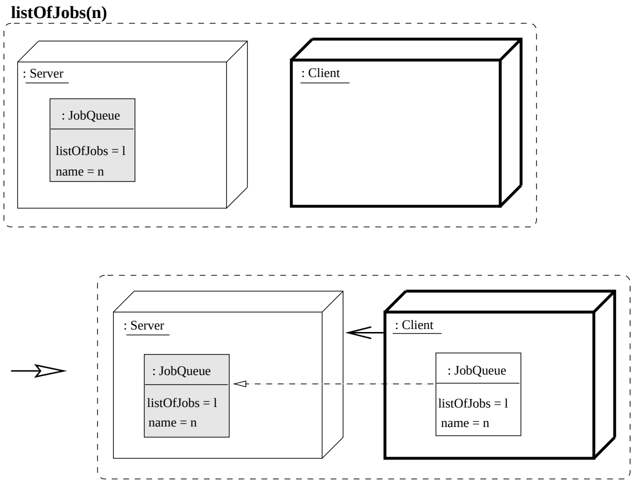

Figure 1.15: Requesting a service: Client's view

rule yielded from the rules in Figures 1.33 and 1.37.

Distributed rules modeling asynchronous communication are obtained differently by constructing a concurrent rule. In Figures 1.33 and 1.35 an asynchronous communication is modeled by two local view rules. Considering the right-hand side of the first local view rule and the left-hand side of the second local view rule, the state graphs of the common interfaces have to be the same. Applying the rule in Figure 1.33 first and thereafter the rule in Figure 1.35, the common export interface of the work space contains a new document after application of the first rule and this document is checked back to the archive by the second rule. A chain of local view rules modeling an asynchronous communication can be used to construct an equivalent concurrent rule which contains all singular actions which the local view rules given contain. The concurrent rule of two rules is constructed by disjointly merging the independent actions of both rules into one and to combine the dependent actions in a way that the resulting effect is computed. Compare Figure 1.38 for a sample concurrent rule yielded from the rules in Figures 1.33 and 1.35. Note that this rule is same as obtained by amalgamation.

LocalView Grammars and Distributed System Grammars An initial local view graph and a set of local view rules for one and the same local system are combined in a *local view grammar*. This grammar describes the structure

and behaviour of one local system. In Section 1.4, two local view grammars are presented: one for work spaces and one for revision archives.
A *distributed system grammar* is the union of a number of local view grammars. Each local view grammar describes one type of local system. In a running distributed system an arbitrary number of local systems of each type may be included and can be started or terminated during runtime.
To start a new local system, the initial local view graph is interpreted as a local view rule, called start rule, in the following sense: On the left-hand side only the remote interfaces occuring in the initial local view graph are present. Onthe right-hand side the local interfaces and the local system are added, i.e. the right-hand side is the initial local view graph. For terminating a local system this start rule has to be applied the other way around.

Comparison with Related Specification Techniques Modeling distributed systems graphs are often used to describe the network structure of the system. Graph transformations which means the rule-based manipulation of graphical structures, are conveniently used to model dynamic changes of the network architecture. Considering the approaches in [DM87], by Δ-grammars in [KLG91], in [Sch92] and by actor graph grammars in [JR91,Kor94], local states are typically coded in some specification or programming text or not considered. None of these approaches use graph transformation on the local level. This is done in [EL93] and [Sch93]. While the distributed graph transformation in [EL93] allows only restricted network reconfiguration, actually switching between global and distributed states, graph transformation can be used on both levels in categorical graph grammars [Sch93]. This kind of graph grammars allows the labeling of nodes by graphs again, but labeling of edges by graph morphism is not considered. Therefore, synchronization of local graph transformation has to be done by additional attributes and cannot be directly modelled by interface graphs.
Comparing graph transformation with the specification techniques mentioned in Section 1.2, we first have to say that the support of modeling by graphical notations is quite restricted. Using distributed graph transformation, the network modeling including reconfiguration as well as the manipulation of local data structures can be specified graphically. Moreover, synchronization and communication can be modeled by graph transformation. In this article, we chose a UML-like notation of distributed graphs to get a compact notation of all relevant features. This notation is also interesting to show how UML might be extended to capture distribution issues more adequately.
Besides graphical notation the other main feature of graph transformation is its rule-based paradigm. For example, the rule-based approach is well suited to

describe network reconfiguration in distributed systems, because it inherently models the asynchrony and non-determinism of reconfiguration control. On the local level, the rule-based approach may also be advantageously used to model concurrency of events. But often, a pure set of rules without any additional control structures leads to mixing up data and control structures in graphs. Graphical notations for rule application control, whether by automatas or Petri nets, are still missing.

1.4 Distributed Version Control: A Reference Application

1.4.1 Problems of Configuration Control

Not only that software projects are much bigger our days than they were before but stricter quality requirements make it even more difficult to plan and to coordinate a project. The result of a development process often has not much in common with the original concept. Additionally, two third of the complete amount of time and money are spent for maintenance and further developments after the initial project was completed. [?,?] Furthermore the knowledge to complete a big project is hardly available in a single company. Hence, external project partners must contribute. More often it is also economically sensible to pass work to external subcontractors. One possible scenario is to have just a core of developers located in the main company. Additional staff is situated around the world wherever the knowledge needed is available for a reasonable price. This is known under the catchword *virtual company*: a company that just exists for a single project. With an ever growing demand on Distributed Software Development one of the results are problems with communication, coordination and quality management, as all project sites must have access to a consistent up to date set of project documents. When one project site changes a document, it must become known to all other project sites, too. This is especially a problem when there is no central archive available for all project partners. In the next section we model a configuration control according to [?].

1.4.2 Modeling Configuration Control with Distributed Graphs

Trying to model Distributed Software Development with Distributed Graph Transformations, it is easy to see which parts of the system are modeled by the different levels of a distributed graph. The network graph describes the different project partners. Each project partner, sometimes also called *site*, has its own *revision archive* containing the documents needed in this site. Each site has just one archive but an arbitrary changing number of workspaces which are

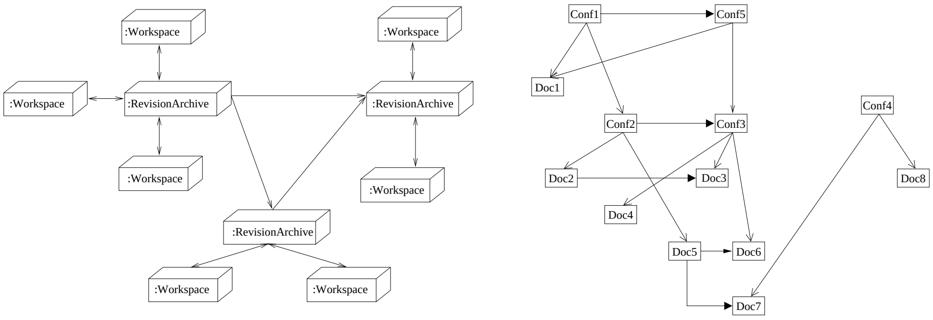

Figure 1.16: A network graph containing three sites consisting of a revision archive and workspaces (left) and an example of a local graph for a revision archive (right)

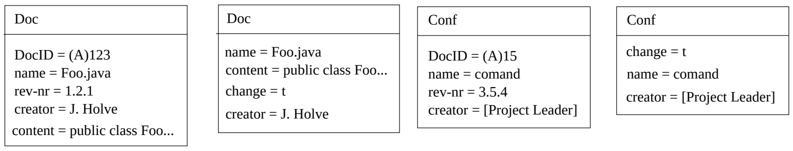

Figure 1.17: Nodes and their attributes modeling a document revision in the revision archive resp. in the workspace and a configuration in the revision archive resp. the workspace (from left to right)

connected to this archive. These workspaces are used to create new revisions of existing documents or to create initial revisions of new documents which are then stored in the revision archive. Consequently there are two network node types (Def. ??) *:RevisionArchive* and *:Workspace*. They each have export interfaces and an arbitrary number of import interfaces. In our case each workspace and each revision archive just has one export archive. For revision archives it would also make sense two have two export interfaces one specially tailored for other archives and one for the workspaces. As not all attributes are needed in workspaces and archives this separation is useful. This interfaces and edges connecting an interface to another resp. interfaces with workspaces and revision archives model the different relations between sites and workspaces. When two revision archives are connected, a *replication direction* is established. The content of the first archive is replicated to the second, so that every archive has a complete set of project documents and their revisions. A revision in the archive is never changed. Instead a revision is *checked out* into a workspace, changed and then *checked into* the archive as a new revision. In Figure 1.16 (left) an example of a network graph is given. Please note that UML notation is used so that no interfaces are explicitly drawn. In the following Local View Rules (Def. ??) are given for *:RevisionArchive* and *:Workspace*. Examples of asynchronous and synchronous communications based on these Local View Grammars are shown.

The local graphs contain the revision structure within a workspace, a revision archive or their interfaces. In the revision archive two sorting principles are used. First revisions are grouped into configurations. Each configuration contains revisions of different documents belonging together. Configurations can even contain other (sub-)configurations. This can be modeled by a tree with configurations at the root and at the inner nodes and having some other revisions e. g. text files, executables or code files on the leaves. Different revisions of a document are connected via another sort of edges where the direction of the edge indicates a predecessor-successor relationship between the two connected revisions. Therefore we have lines connecting different revisions of a document. Additionally different parallel versions of a document may exist (e. g. containing a variant of that document). A configuration never contains more than one revision of the same document; since a workspace only contains one main (root) configuration this also holds for workspaces. Figure 1.16 (right) shows an example with five configurations and their revisions. Revisions, meaning configurations or any other possible revision types are modeled with nodes. Revisions are connected with edges denoting a relation between them. We use the abbreviation *Conf* to denote configurations, *Doc* denotes all types of revisions not being configurations (also called *document revisions*). *Rev* is taken to summarize all kind of revisions including configurations. In Figure 1.17 four nodes of the local graphs are shown with all their attributes: a document revision in the revision archive, a document revision in the workspace, a configuration in the revision archive and a configuration in the workspace. Each document in the revision archive has a unique *DocID* (document identification number) consisting of the name of the site, where it was created, and a number local to this site the revision was created at. Of course each revision has a *name*. This name may be the same for a number of revisions, the revisions can be differentiated via the DocID. Each revision also has a revision number *rev.-nr.*, which is unique with regard to the document of which it is revision. A revision is also attributed with its *creator* and its *content*. Configurations do not have this *content*. In the workspace only those attributes are necessary that can be changed by the user i.e. the *content*, *creator* and the *name* attribute. Additionally a *change* attribute is added in the workspace. It is either true or false depending on whether this node or one of its attributes was changed after it has been checked out into the workspace (see sections 1.4.3 and 1.4.9). In the export interfaces all these attributes mentioned are needed to. The

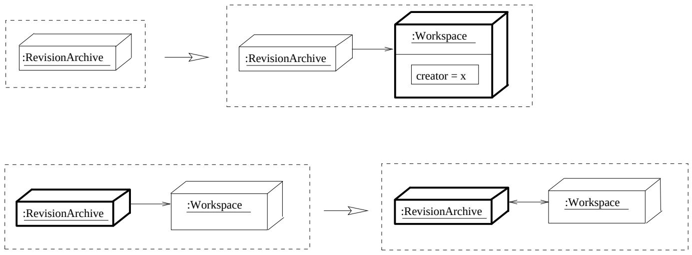

Figure 1.18: Introducing a new workspace to a revision archive

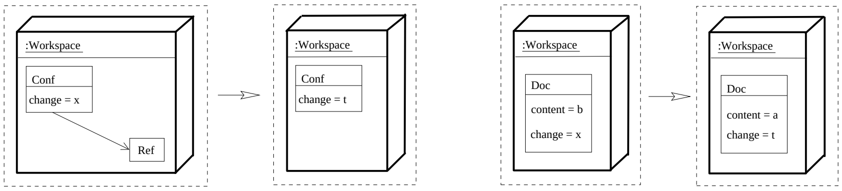

Figure 1.19: In the workspace: Deleting a revision (left) and changing the content of a revision (right)

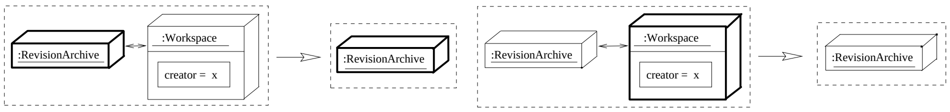

Figure 1.20: Deleting an empty workspace in the local view of the revision archive (left) and the local view of the workspace (right)

situation is different for the import interfaces. A revision interface imports all attributes from another revision interface but only the *creator*, *content* and *name* attribute from a workspace. A workspace on the other hand imports only the *content* and *name* attribute from revision archive. Additionally to the nodes representing revisions, nodes for global information are necessary. For the revision archives this is the name of the site and the next number to be given to a newly created document as part of its DocID. Both are are attributes of a single node. For the workspace the creator of the revisions in the workspace must be stored. Nodes modeling locks are also necessary, since in certain situations it may be necessary to forbid or to force certain rules.

1.4.3 Working with Workspaces

When a programmer is starting to work on a project at a site, a workspace belonging to this programmer must be added to the corresponding revision archive. This can be done with the help of the start rule (Def. ??) for the Local View Grammar of workspaces. In Figure 1.18 such a rule is shown. With this rule, the workspace, its import and its export interface are introduced together with the *creator* node. The import interface of the workspace is connected to the already existing export interface of the revision archive. But additionally a new import interface for the revision archive connected to the export interface of the workspace must be introduced. This is done with the help of the second rule in Figure 1.18. Both rules can be coordinated along the interfaces to form a complete insertion of a workspace. This can be done with either parallel synchronized rules or concurrently synchronized rules (Def. ??, ??).

A document revision can be changed in various ways in the workspace. The most obvious change is a modification of its content as shown in Figure 1.19 (right). As the content is just an attribute of the node representing a document revision, the rewrite rule does nothing but to change this attribute. Additionally the boolean attribute *change* is set to true to indicate that a change took place. This attribute is set to true no matter if it was true or false before as indicated through $change = x$ in the left hand side. The change of other attributes, as e.g. the name of the revision (not the revision number) can be handled the same way. It is also possible to move a revision from one configuration into another. Both configurations taking place in this process are marked as changed. Deleting a revision as shown in Figure 1.19 on the right is similar, the configuration containing the revision to be deleted is marked changed. Please note that several attributes are omitted here to keep the rules small. Inserting a new revision is similar. In both cases the corresponding configuration must be marked changed. When a programmer has finished his/her work and has it checked back into the revision archive, the workspace can be deleted. This is only possible if the workspace is empty i.e. when all changes have been checked into the archive[a], see Figure 1.20. This rule is shown in two views, in the view of the workspace and the view of the revision archive. Both rules can be amalgamated along the interfaces (Def. ??) leading to a synchronous communication.

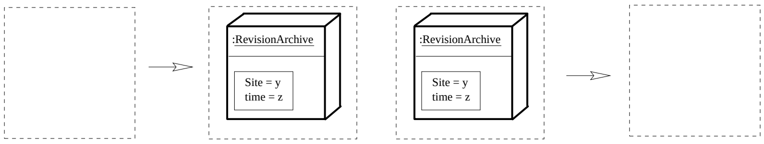

Figure 1.21: Introducing a site via introducing its revision archive (left) and deleting it again (right)

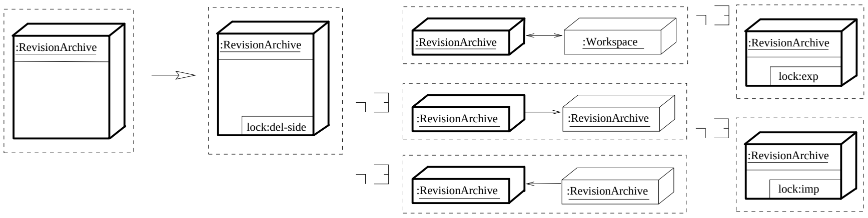

Figure 1.22: Lock a revision archive for deletion

1.4.4 Introducing and Deleting Revision Archives

The structure of a project is not always fixed from the beginning. It may well be that project partners leave or new project partners are included. For our approach this means that revision archives together with their workspaces have to be added or deleted. Adding a new revision archive is easily done with the help of the start rule (Def. ??) for the Local View Grammar of revision archives (see Figure 1.21, left). Please note that with this start rule the interfaces for the corresponding archive are introduced, too. Additional replication edges can be added with the rule shown in Figure 1.23 (see the next section). A site can only be deleted when no workspaces are connected to it (see Fig. 1.20) and all network edges denoting replications have been deleted. In this case several steps are necessary. The archive must be locked first to indicate that no other operations are possible anymore (Figure 1.22). This lock is necessary to forbid e.g. that something is checked out of the archive or that something is replicated during deletion. Rules for check in/out resp. replication must always cross check whether the lock for deleting an archive exists or not. Equally, when setting the lock for deleting an archive, it must be cross checked whether there is an export/import lock indicating that actually a check out or a replication takes place. In this case the lock for deleting the

[a]The rules for checking out configurations are not described until section 1.4.9, since they use the rules explained in the following sections

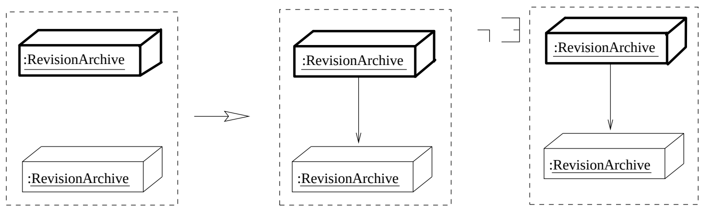

Figure 1.23: Introducing a new replication direction between two sites

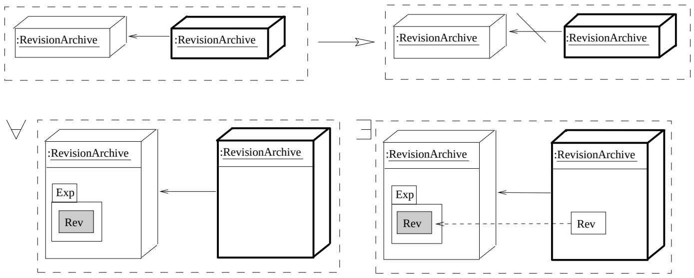

Figure 1.24: Locking a site to prepare deletion of a replication edge

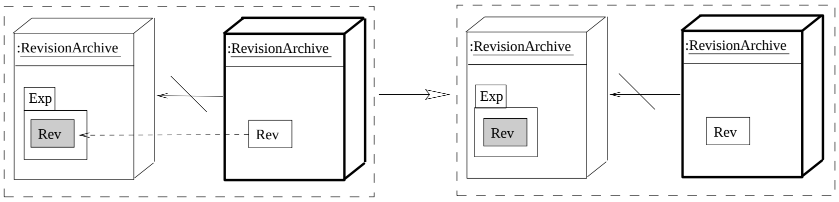

Figure 1.25: Disconnected the replicated revision from its original by deleting it from the import interface together with all morphisms connected

archive can not be set. After the lock is set the archive is emptied with rules similar to the deletion of revisions in the workspace and finally the revision archive is deleted (Figure 1.21, right).

1.4.5 Introducing and Deleting Replication Directions

Two sites working together in a project are connected via a replication edge pointing from the site receiving replicated data to the site this data comes from. As this structure is not fixed during a project it is necessary to add or to

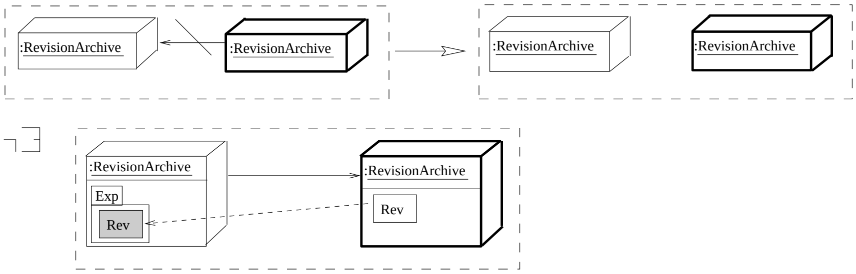

Figure 1.26: Delete the a replication edge

delete replication edges. In Figure 1.23 the introduction of a new replication direction is shown. For the network graph, this means that new network edges are introduced together with its import interface. It is connected to the already existing export interface of the source site. This can only be done if there was no edge before. Revisions are then replicated via this new edge as shown in section 1.4.6.

It is more complex to delete a network edge. Deleting a network edge is cutting of a possible replication direction. This means that everything that can be replicated via this edge must have been replicated first. This can be ensured with an application condition. When deleting a replication edge, the underlying local morphisms (drawn as dashed arrows[b]) pointing from the replicated revision to its original must be deleted. This can be done by deleting the corresponding revisions from the import interface of the importing site. Of course they are not supposed to be deleted from the site itself. What is taking place is a copying by disconnecting the replicate from the original. The export interface of the target site is not changed as this revision may be exported to third sites. The following rules handle the deletion of a replication edge. First the import interface of the source site must be locked. This is shown in Figure 1.24. The lock indicates that this import interface can not be used anymore. As in the UML notation the import interface itself is not shown, the corresponding replication edge is graphically locked. After locking the interface must be emptied. In the UML Notation this means that the dashed morphisms pointing from the imported revision to its original are deleted (see Figure 1.25); the replicate and the original revision are disconnected. A copy is made. After all dashed morphisms have been deleted, the replication edge, its corresponding – now empty – import interface and the lock are deleted (see Figure 1.26).

[b]in the following these morphisms are therefore called *dashed morphisms*

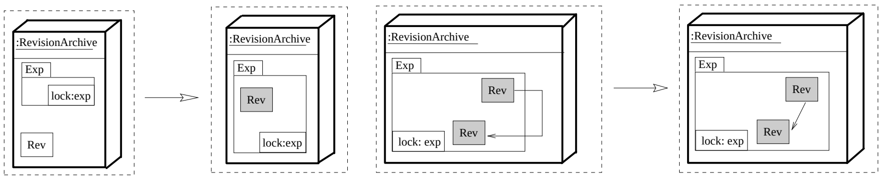

Figure 1.27: Export revisions and connections between revisions

1.4.6 Replication between Sites

Replication consists of two phases. First the revisions to be replicated must be exported. Second they can be imported by any other archive.

1.4.7 Exporting revisions

In this subsection the rules for writing the revision archive's content into the export interface are described. First the export archive must be locked. This must be done to ensure that no site is importing from this export interface while an export takes place. Otherwise incomplete configurations could be imported. The lock can only be set, if there is a new revision in the archive not being exported yet. It also can only be locked when no other lock concerning this interface can be found (no figure given). In a second step revisions and relations between these revisions have to be exported (Figure 1.27). Please note that arrows and revision nodes in these figures are meant to include all different revision types and arrow types introduced above. Since there is no difference in the handling of the different types, only two general rules are shown here. When everything has been replicated, the lock is deleted, so that an import of another site can take place (no figure given).

1.4.8 Importing revisions

When a site wants to import from another site's export interface it must first set a lock to ensure that during the import e.g. no workspace tries to check out an incomplete configuration. The lock is set, if in the corresponding export interface a revision not yet imported can be found. When the lock is set, the current import can start as shown in Figures 1.28 and 1.29. Either a revision was not replicated yet to the target site or it was already replicated via another path than the current replication edge. In Figure 1.28 the first case is shown. In Figure 1.29 the revision to be replicated already exists in the target site. In this case the dashed morphism is inserted (see Figure 1.30). The identification

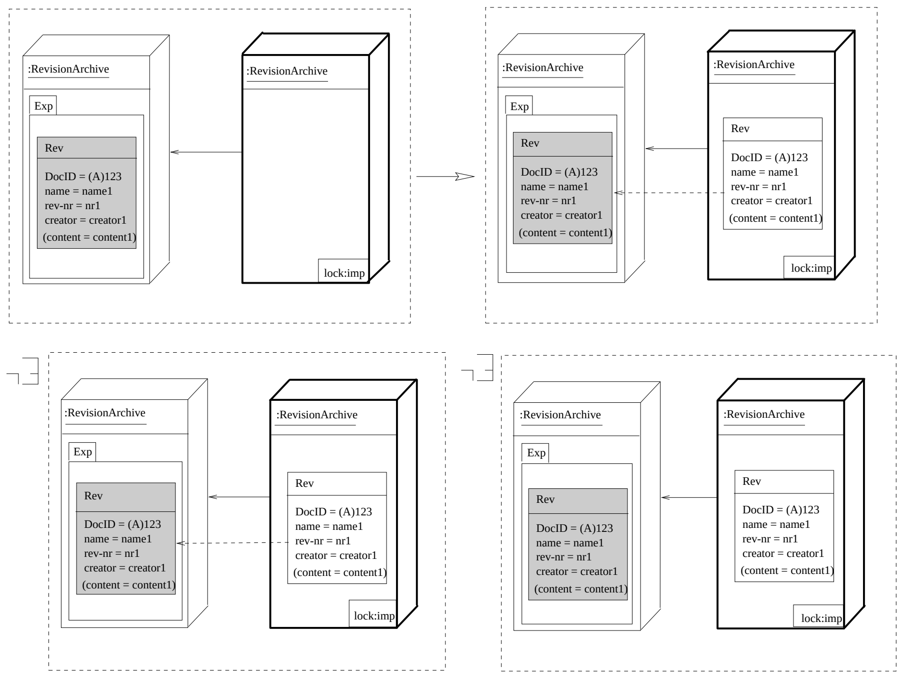

Figure 1.28: A revision that was not replicated before is replicated

of this revision is done with the help of of the unique document identification and its revision number. The rules shown there must be applied to all different sorts of edges for document revisions and configurations. Finally the import lock is deleted if everything was replicated.

1.4.9 Checking out a Configuration

As already mentioned before, workspaces are used to change and to create (document) revisions. When the user of a workspace decides to work on a configuration, this configuration must be *checked out* into his/her workspace. The check out starts from the export interface of the revision archive. No extra rules are necessary to write something into the export archive, they were already given in subsection 1.4.7 for the replication into another revision archive. The difference comes with the import of revisions. For replication everything, all edges and all attributes are imported by another revision archive. In opposite, for the workspace only configurations, not revisions are of further

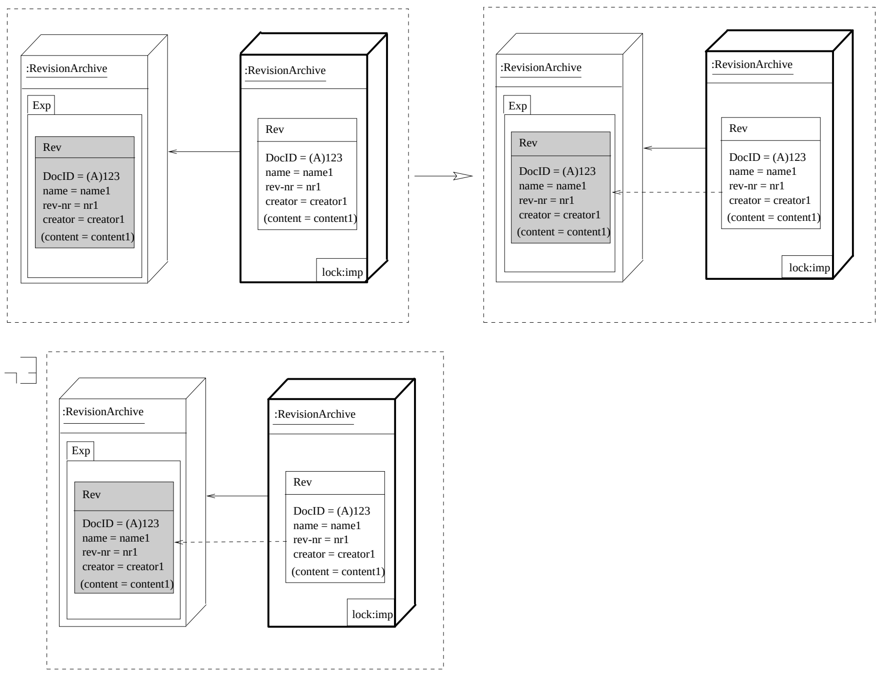

Figure 1.29: A revision that was replicated via other replication edges is connected to the revision in the export interface

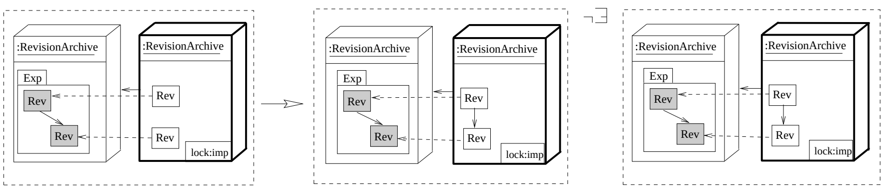

Figure 1.30: Import of edges between imported revisions

interest. Since a workspace shall always contain a consistent set of revisions, it may never contain more than one revision of a single document. Additionally attributes are not replicated but copied as they should not be changed in the archive when they are changed in the workspace. When a configuration is checked out from a revision archive into a workspace, a decision must be made, which configuration to get. This so called *main configuration* must be

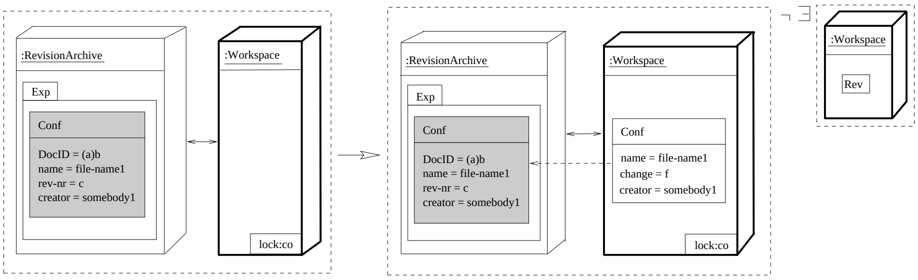

Figure 1.31: Checking out the main configuration

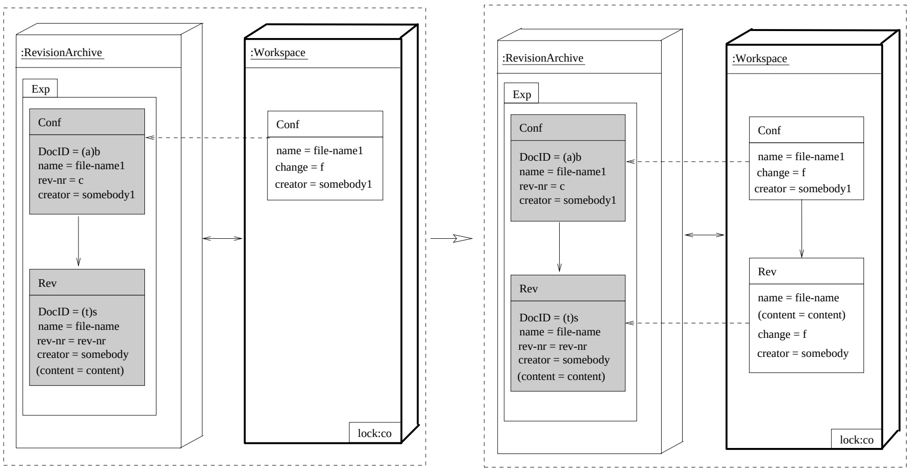

Figure 1.32: Checking out an arbitrary revision

checked out first. This can only be done when the workspace is empty as only one main configuration is allowed in a workspace. To ensure that nothing else takes place within the workspace, a lock is set. Additionally it is necessary that there is no lock in the corresponding export interface of the revision archive. In Figure 1.31 the main configuration is checked out. Not all attributes are necessary, in fact for a configuration only its name is copied. This is the only attribute for which a change by a user makes sense. The *change* attribute is added and set to false in the beginning. In the same way all revisions belonging to this configuration are checked out as shown in Figure 1.32. (An application condition is omitted forbidding a double export). Rules 1.31 and

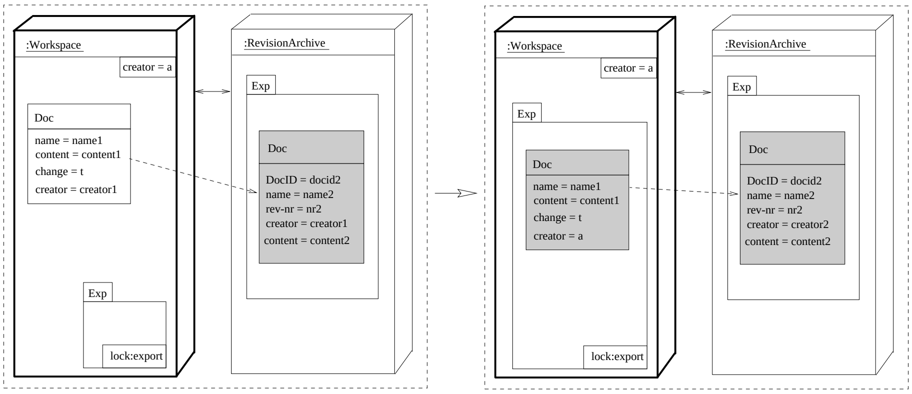

Figure 1.33: Export a changed document revision

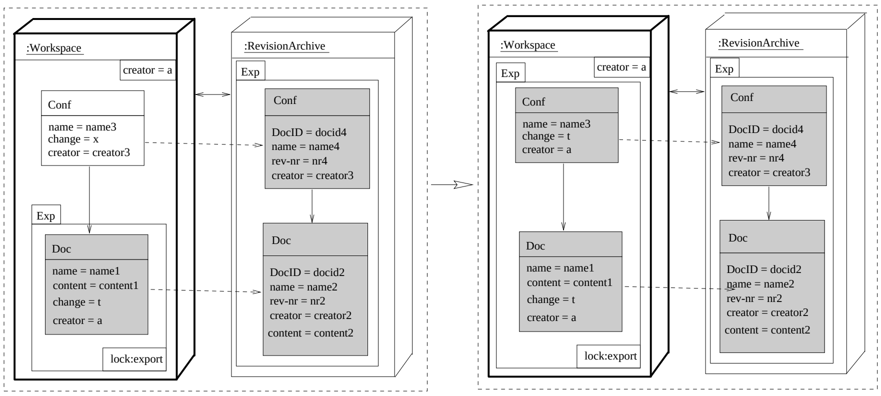

Figure 1.34: Export a configuration containing a changed document revision

1.32 ensure that only one main configuration is checked out. The configuration with all its nodes and edges forms a tree in the workspace. When a complete configuration is checked out, the lock can be removed from the workspace. An application condition ensures, that there is no revision in the revision archive that is contained in a configuration already checked out (no figure given).

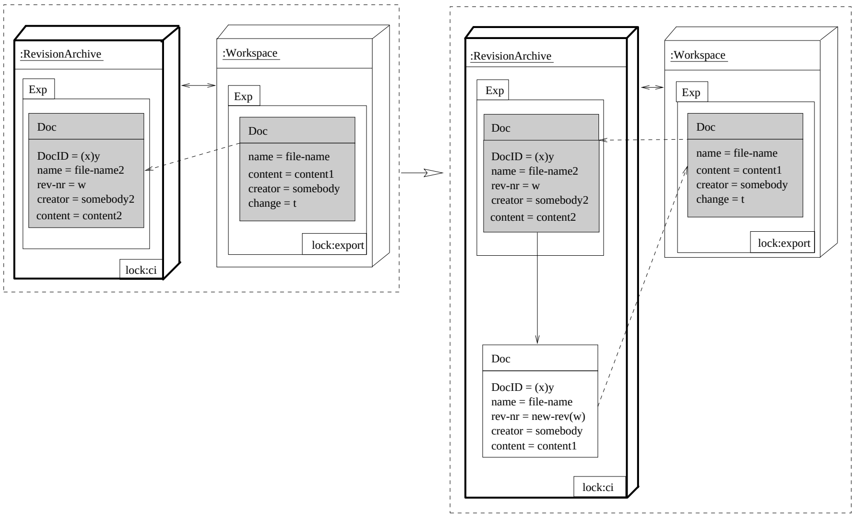

Figure 1.35: Importing a changed document revision into the revision archive

1.4.10 Checking back into the Archive

Checking a complete configuration back into the revision archive needs two steps: the workspace must export the revisions and the revision archive must import them. First rules needed for the export are given. Again several steps must be performed. As usual a lock is set to ensure that no changes happen to the revisions during the export (this rule is omitted here). It is not necessary to write every revision into the export; only those revisions must be exported that are marked changed ($change = t$) or configurations containing changed revisions. In this case also the creator of the revisions can be added: it is the owner of the workspace as stored in a single node in each workspace. Four rules are necessary for exporting all changed revisions. All document revisions that have been changed are exported (Figure 1.33). Then all configurations containing changed document revisions must be written into the export interface (Figure 1.34). Finally, edges pointing from changed revisions to not changed revisions must be taken into account and all edges between already exported revisions. In these rules it is assumed that everything was originally imported from the revision archive. There may also be (initial) revisions of newly created documents. When everything has been exported, the lock can be deleted again.

The next step is to insert the changed changed back into the revision archive.

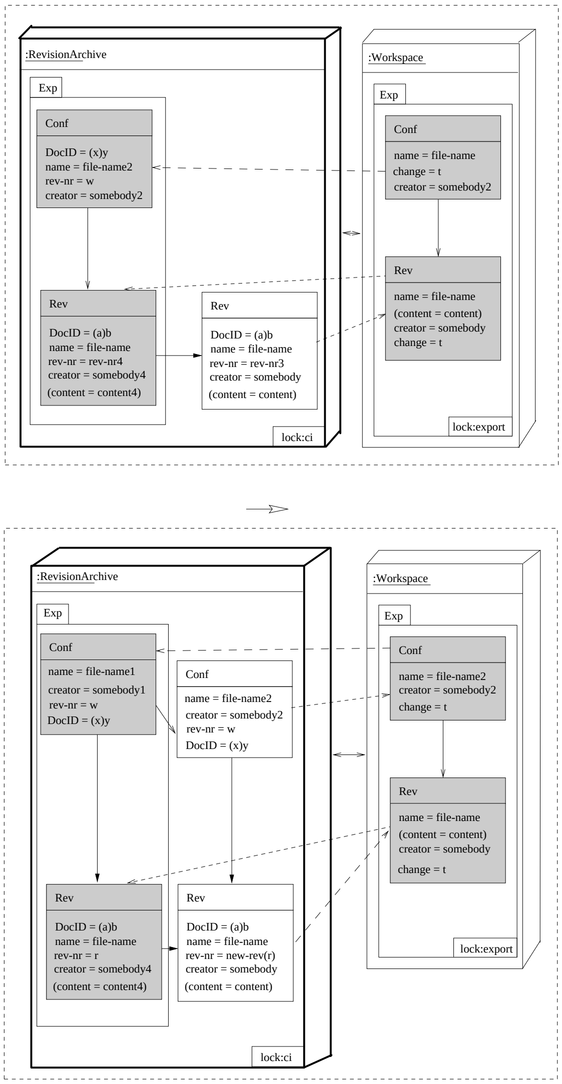

Figure 1.36: Importing a changed configuration into the revision archive

There are two possibilities for the exported revisions of the workspace. Either it is inserted as a direct successor of its original or it is inserted as a new branch when there already is a direct successor of the original revision. The edges

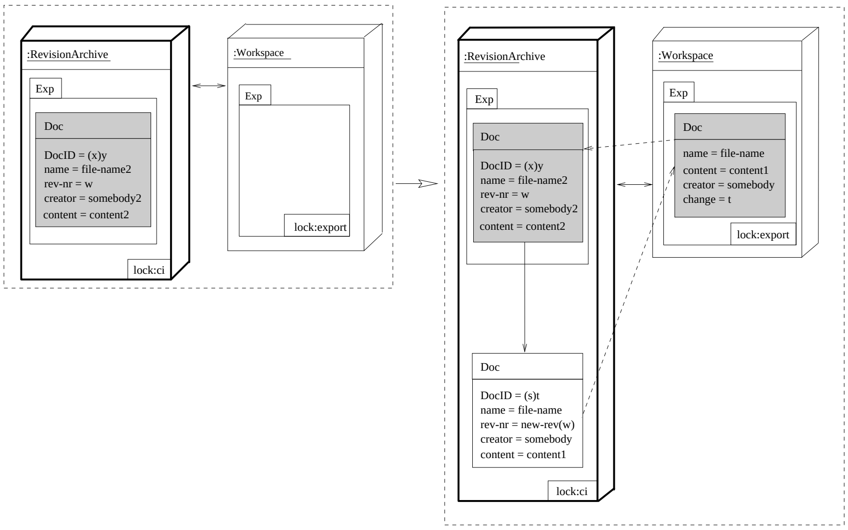

Figure 1.37: Figure 1.35 when using synchronization instead of asynchronous communication

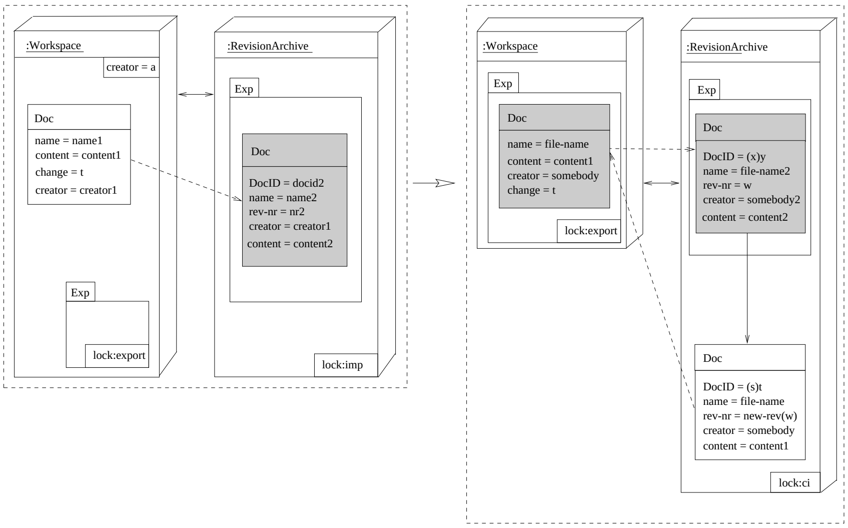

Figure 1.38: The result of a synchronization of Figure 1.37 and Figure 1.33 or an asynchronous communication of Figure 1.35 and Figure 1.33

representing the content of new configurations must be checked in, too. But first a lock must be set to guarantee that no other workspace tries to check in or nothing else is replicated into the revision archive meanwhile. This is necessary to guarantee that there is just one direct successor for each revision and that all other revisions are handled as branches. Then there are rules handling the same situations as mentioned in Figure 1.33 and 1.34, but now realizing the import in the Local View Grammar of the revision archive. Two rules are necessary for each case, one inserting direct successors and the second inserting branches. Here only rules for the insertion of direct successors are given. In Figure 1.35 it is shown how a document revision can be inserted from the export interface of the workspace into the revision archive. It is connected with an edge to its predecessor revision. The attributes *content*, *creator* and *name* are taken over from the workspace, *DocID* from the original revision. *rev-nr* is newly calculated. In Figure 1.36 a configuration containing a changed revision is checked into the revision archive. In our case the configuration already has an equivalent in the revision archive and the revision changed was already checked back into the revision archive. Attributes are handled as in Figure 1.35.

But the rules in Figures 1.35 – 1.36 do not work on their own. As they are now, the new revision could be added everywhere in the revision archive and not only as (direct) successor of the revision which originally has been checked out. This can be realized with the help of a concurrent rule (Def. ??). Synchronizing the right hand side of Figure 1.33 with the left hand side of Figure 1.35 via the interfaces of workspace and revision archive leads to the correct result. In the same way the Figures 1.36, 1.34 must be synchronized. The same must happen to the rules for locking and unlocking and edge handling rules that are not given here. But also an amalgamated rule would lead to the same result (Def. ??) with a parallel synchronization. In this case the left hand sides, the gluing graphs and the right hand sides are glued along the interfaces. Taking Figures 1.33, 1.35 this would not work as the left hand side of Figure 1.35 requires something in the export of the workspace that is not there in Figure 1.33. Taking Figure 1.37 instead of Figure 1.35 solves this problem. Amalgamating Figures 1.33, 1.37 leads to the same result then using concurrent synchronization between Figures 1.33, 1.35. The result is given in Figure 1.38. Similarly for the other cases rules for amalgamation can be constructed.

What is still missing is the insertion of completely new configurations in the revision archive. In this case nothing was ever checked out and no line of revisions must be followed in the revision archive. Rules for this case are omitted here. Finally the lock in the revision archive can be deleted again and the corresponding interfaces of the workspace must be cleaned again. Then a new round of changed at this configurations can take place in the workspace. Alternatively the workspace can be emptied again and a new configuration can be checked out.

1.5 Distributed Graphs and Graph Transformation

A *distributed graph* can be understood as a graph representing the network of a distributed system, whose nodes are *attributed graphs* and whose edges are *attributed graph morphisms.* Attributed graphs can be seen as graphs where nodes and edges are equipped by some additional label. On these labels calculations may take place. Attributed graphs and attributed graph morphisms used in this contribution are introduced in [CL95] for the single-pushout approach [EHK+96]. We adapt this approach for the double-pushout approach [CMR+97].
Next, we briefly review the double-pushout approach of attributed graph transformation according to [CL95]. The main feature of this approach is a partial algebra for the attributes. Afterwards we show how attributed graphs can be distributed via a network graph.

1.5.1 Attributed Graph Transformation

We start this section with a brief introduction in partial algebras to fix the notation used along the chapter. For the sake of space, we are not able to present all concepts in full detail. Therefore, we refer the reader being unfamiliar in this topic to more detailed works as [Bur86,Rei84].
A *signature* $Sig = (S, OP)$ consists of a set of sorts S and a family of sets $OP = (OP_{s_1 \dots s_n, s})_{s_1, \dots, s_n, s \in S}$ of operations. A *signature morphism* $\sigma : Sig \to Sig'$ between two signatures $Sig = (S, OP)$ and $Sig' = (S', OP')$ is a pair of mappings $\sigma = (\sigma_S : S \to S', \sigma_{OP} : OP \to OP')$ such that for each operation $op : s_1 \dots s_n \to s$ in OP and $n \geq 0$ we have $\sigma_{OP}(op) : \sigma_s(s_1) \dots \sigma_s(s_n) \to \sigma_s(s)$ in OP' . A *partial Sig-algebra* A is an S-indexed family of carrier sets $(A_s)_{s \in S}$ together with an OP-indexed family $(op^A)_{op \in OP}$ of partial mappings $op^A : A_{s_1} \times \dots \times A_{s_n} \to A_S$ for each operation op in $OP_{s_1 \dots s_n, s}$. A Sig-algebra is *total* if op^A is a total mapping for each op in OP. We denote by $dom(op^A)$ the *domain* of a partial mapping op^A. A *Sig-homomorphism* $f : A \to B$ between partial Sig-algebras A and B is an S-indexed family of total mappings $f = (f_s : A_s \to B_s)_{s \in S}$ such that for every operation op in $OP_{s_1 \dots s_n, s}$ and for each $(x_1, \dots, x_n)$ in $A_{s_1} \times \dots \times A_{s_n}$ holds that whenever $(x_1, \dots, x_n)$ is in $dom(op^A)$ then $(f_{s_1}(x_1), \dots, f_{s_n})(x_n)$ is in $dom(op^B)$ and

$op^B(f_{s_1}(x_1), ..., f_{s_n}(x_n)) = f_s(op^A(x_1, ..., x_n))$. A *Sig*-homomorphism is injective, if the total mapping f_s for each sort s is injective. Given an S-index set $X = (X_s)_{s\in S}$ of variables, the *term algebra* $T_{Sig}(X)$ w.r.t. Sig and X has as carrier sets for sort s the set of terms of sort s. A term of sort s is each variable x of X_s and each term $op(t_1, ..., t_n)$ for all operations $op : s_1...s_n \to s$ and terms t_i of sort s_i for $i = 1...n$. For each operation $op : s_1...s_n \to s$ and terms t_i in s_i we define $op^{T_{Sig}(X)}(t_1, ..., t_n) := op(t_1, ..., t_n)$.

Attributed graphs are already introduced in a simpler version in [EHK+96]. There, an attributed graph is defined by a graph where graphical objects are labeled by elements of an attribute algebra. Even if the description of an attributed graph given in [EHK+96] is too simplified for our purpose, the idea of an attributed graph consisting of a graphical and an attribute part is adopted into our approach, where an *attributed graph* is defined by a suitable partial algebra over an *attributed graph signature*. This signature reflects the idea given in [EHK+96] by a separation into two sub-signatures and a set of unary operations connecting both sub-signatures. One sub-signature describes the graphical part, the other one the attribute part. The operations connect the elements of the graphical part with elements of the attribute part.

Definition 1.5.1 (attributed graph signature) An *attributed graph signature* $AGS = GS + DS + ATTROP$ consists of a *graph signature* $GS = (S_{GS}, OP_{GS})$, i.e. OP_{GS} contains unary operations only, a signature $DS = (S_{DS}, OP_{DS})$, called *attribute signature*, and an $S_{GS} \times S_{DS}$-indexed family of operations $ATTROP = (ATTROP_{s,s'})_{s\in S_{GS}, s'\in S_{DS}}$.

An attributed graph is a partial AGS-algebra where its GS-reduct algebra is total and the $ATTROP$ operations are total mappings.

Definition 1.5.2 (attributed graph and attributed graph morphism) Given an attributed graph signature AGS as defined in 1.5.1. An *attributed graph* A w.r.t. AGS is given by a total GS-algebra A^{GS}, called *graphical algebra*, a partial DS-algebra A^{DS}, called *attribute algebra*, and a family of total mappings $op^A : A^{GS}_s \to A^{DS}_{s'}$ for all $op : s \to s' \in ATTROP$, called *attribution operations*.
An *attributed graph morphism* $h : A_1 \to A_2$ between two attributed graphs A_1 and A_2 w.r.t. AGS is an AGS-homomorphism. We denote by $h^{GS} : A^{GS}_1 \to A^{GS}_2$ the attributed graph morphism h restricted to the graphical part GS and by $h^{DS} : A^{DS}_1 \to A^{DS}_2$ the restriction to the data part DS.

All attributed graphs and attributed graph morphisms w.r.t. AGS form a cocomplete category **AGr(AGS)** of **A**ttributed **Gr**aphs over AGS [CL95]. The transformation of an attributed graph w.r.t. AGS is defined via an *attributed rule*. An attributed rule consists of a span of injective attributed graph morphisms w.r.t. AGS. The attribute algebras of a rule are given by the *term algebra* for the data signature DS and a set of variables. The attribute homomorphisms of the rule morphisms are identities as we do not intend to rewrite the attribute algebra. For the transformation of arbitrary partial algebras in the DPO approach we refer to [LR98]. This approach can be used if the attribute algebra, being an arbitraty partial algebra, shall be rewritten, too.

Definition 1.5.3 (attributed rule) Given an attributed graph signature AGS as defined in 1.5.1 and an S_{DS}-indexed set of variables $X = (X_s)_{s \in S_{DS}}$. An *attributed AGS-rule* $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ consists of injective attributed graph morphisms l and r w.r.t. AGS and attributed graphs L, I and R w.r.t. AGS, where the the attribute algebras L^{DS}, I^{DS} and R^{DS} are the term algebra $T_{DS}(X)$ w.r.t. DS and X and l^{DS} and r^{DS} are identities. We call L the *left-hand side*, R the *right-hand side* and I the *interface graph* of rule p.

Prerequisite for the application of an AGS-rule to an attributed graph G w.r.t. AGS is an occurrence of the left-hand side L in G. This occurrence is called *match* if the so-called *gluing conditions* are met. The gluing condition is made up of two sub-conditions, namely the *dangling* and the *identification condition*. The dangling condition ensures that all operations in the GS-algebra and all attribution operations of the attributed graph G remain total by the transformation. The identification condition guarantees that an object is not identified with any other object if the object shall be deleted. This condition is necessary for the existence of a pushout-complement, that is needed later on.

Definition 1.5.4 (match, gluing condition) Given an attributed graph signature AGS as defined in 1.5.1, an AGS-rule $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ and an attributed graph morphism $m : L \to G$ w.r.t. AGS.

- Let $op^G : G_{s_1} \to G_{s_2}$ be a mapping of the GS-algebra G^{GS} for operation $op : s_1 \to s_2 \in OP_{GS}$. The morphism m satisfies the *dangling-condition* if for all $x_2 \in (m_{s_2}(L_{s_2} - l_{s_2}(I_{s_2})))$ holds: if $x_2 = op^G(x_1)$ for some $x_1 \in G_{s_1}$ then $x_1 \in (m_{s_1}(L_{s_1} - l_{s_1}(I_{s_1})))$.
- Let s be a sort of S_{GS} of the graphical signature GS. Then the morphism m satisfies the *identification-condition* if for all $x \in (L_s - l_s(I_s))$ and $y \in L_s$ holds: $m_s(x) = m_s(y)$ implies $x = y$.

The morphism m satisfies the *gluing-condition* if m satisfies the dangling and identification condition. In this case, m is called *match* for p and the elements $G - m(L)$ are called *context of the match*.

Please notice, that the dangling and identification condition refer only to the graphical part of the attributed graph morphism m, as these conditions affect only operations of the signature GS.

We present next the construction of a pushout-complement for an attributed rule $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ and a match $m : L \to G$. The rule removes all elements from the graphical part of the attributed graph G that are contained in $m(L - l(I))$. The attribute algebra of G remains unchanged as well as the attribution operations for preserved graphical objects. Our approach of attributed graph transformation does not allow to change directly attributes. To change an attribute of a graphical object requires its deletion and its recreation with the new attribute. Because of that, special graphical sorts are introduced, called attribute holder, carrying the attributes. A more detailed treatment of this problem can be found in [LKW93].

Construction 1.5.5 (context graph) *Given an attributed graph signature AGS, an AGS-rule $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ and a match $m : L \to G$. The* attributed context graph *C is constructed as follows:*

- *We define $C_s = G_s - m_s(L_s - l_s(I_s))$ for all graphical sorts s in S_{GS} and $op^C = op^G|_{C_{s1}}$ for all graphical operations $op : s_1 \to s_2$ in OP_{GS}.*
- *The attribute algebra remains unchanged, i.e. $C^{DS} = G^{DS}$.*
- *We define $op^C = op^G|_{C_s}$ for all attribution operations $op \in ATTROP_{s,s'}$.*

The morphism $g : C \to G$ is defined by the inclusion of C into G and the morphism $c : I \to C$ is defined by the composition $m \circ l$ of m and l.

Proposition 1.5.6 (context graph is a pushout-complement) *Given an attributed graph signature AGS, an AGS-rule $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ and a match $m : L \to G$. Let C be the attributed context graph with morphisms c and g constructed according to 1.5.5. Then, (G, g, m) is a pushout of c and l in* $\mathbf{AGr(AGS)}$.

Definition 1.5.7 (attributed direct derivation) Given an attributed graph signature AGS, an AGS-rule $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ and a match $m : L \to G$. A *direct derivation* via p at m from G to H, denoted by $G \overset{p,m}{\Rightarrow} H$, consists of a double-pushout square in $\mathbf{AGr(AGS)}$ as shown in Figure 1.39.

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

Figure 1.39: direct derivation

$$
\begin{tikzcd}
A_i \arrow[drr, "r"'] & A \arrow[l, "h_i"'] \arrow[dr, "q"] & L \arrow[l, "c"'] \arrow[d, "m"] \\
 & & G
\end{tikzcd}
$$

Figure 1.40: attributed application condition

1.5.2 Attributed Application Conditions

In the following section we introduce *attributed application conditions*. *Negative application conditions* are already introduced in [EHK+96]. They forbid structures in the context of a match. We generalize this idea in order not only to forbid structures, but also to require structures, for example special attributes for graphical objects, or to restrict the conditions only to special matches.

In the same way as attributed graphs are split into a graphical and an attribute part, application conditions for attributed rules are split as well. An *attributed application condition* for a rule consists of a finite set of *attributed graphical constraints*. An attributed graphical constraint specifies on the one hand graphical conditions, on the other hand attribute conditions that have to be fulfilled by a match in order to apply the rule. The main constituent for building attributed graphical constraints are *equational constraints*.

Definition 1.5.8 (equational constraint) Given an attributed graph signature AGS, an AGS-algebra A and a congruence relation Θ on the attribute algebra A^{DS}. An *equational constraint* over A is any attributed graph morphism w.r.t. AGS starting from A, $c : A \to B$, that is injective on the graphical part, i.e. $c^{GS} : A^{GS} \to B^{GS}$ is injective, and the attribute part $c^{DS} : A^{DS} \to A^{DS}_{/\Theta}$ is the natural homomorphism from the attribute algebra A^{DS} to the quotient algebra $A^{DS}_{/\Theta}$.

The natural homomorphism c^{DS} for an equational constraint c is equivalent to a set of equations. Therefore, we can represent the natural homomorphism by the equivalent set of equations. We will use this presentation for the presentation of attributed conditional constraints as shown in Example 1.1. By means of equational constraints we are able to define attributed graphical constraints and attributed application conditions.

Definition 1.5.9 (attributed graphical constraint) Given an attributed graph signature AGS and a set $X = (X_s)_{s \in S_{DS}}$ of variables w.r.t. DS. Let L be an attributed graph w.r.t. AGS where the attribute algebra is the term algebra $T_{DS}(X)$ for DS and X. An *attributed graphical constraint* $cc_L = (c : L \to A, (c_{A_i})_{i \in I})$ over L consists of an equational constraint c over L and an I-indexed set $(c_{A_i} : A \to A_i)_{i \in I}$ of equational constraints over A. The index set I is finite but possibly empty.
An attributed graph morphism $m : L \to G$ w.r.t. AGS *satisfies* an attributed graphical constraint cc_L, denoted by $m \models_L cc_L$, if for all attributed graph morphisms $q : A \to G$ with $m = q \circ c$ there is an $i \in I$ as well as an attributed graph morphism $r : A_i \to G$ such that $q = r \circ h_i$ (cf. Fig. 1.40).

How attributed graphical constraints are represented and interpreted is shown in the following example. We show special cases mostly occuring in a specification, namely *negative graphical constraints* and *attribute constraints*. Former forbid graphical structures in the context of a match, latter require conditions on the assignment of variables by the match.

Example 1.1 (attributed conditional constraints) A *negative constraint* is an attributed graphical constraint $cc_L = (c : L \to A, (c_{A_i})_{i \in I})$ where the index set I is empty. The rule in Figure 1.41 increases the attribute d by one if there is an outgoing edge to a node with attribute 1. This should happen only if the attribute d is the greatest attribute of all nodes having an outgoing edge to the node with attribute 1. The corresponding equational constraint $c_1 : L \to A_1$ is shown in the figure on the top. The GS-homomorphism c_1^{GS} for the graphical part is explicitly shown whereas the DS-homomorphism c_1^{DS} for the attribute part is represented by its equivalent equation. The equation $d1 > d = true$ or short $d1 > d$ is depicted over graph A_1. The equational constraint c_1 requires that there is no other node with outgoing edge to node labeled 1 having an attribute greater than d.
Attribute conditions are further important conditions referring to the assignment of values to the variables of the rule by the match. The corresponding equational constraints have the form (id_L, c_A). In the sample in Fig. 1.41 h is identical on the graph part, the attribute part h^{DS} is equivalent to the equation $d > 5$ shown over graph A_{2_h}. This equational constraint allows the application

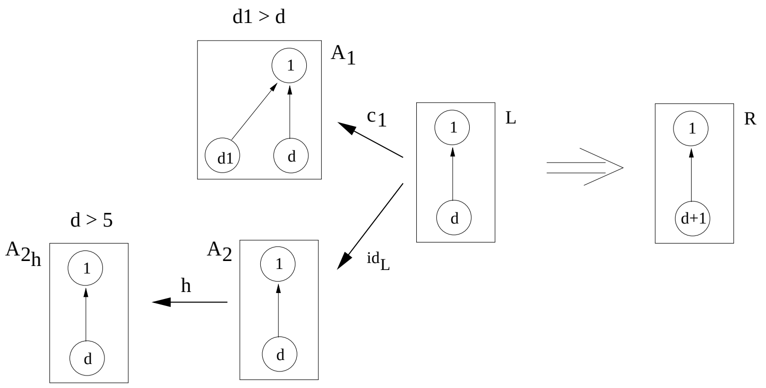

Figure 1.41: A sample application condition

of the production only if the value assigned by match m to variable d is greater than 5.

Definition 1.5.10 (attributed application condition)
An *attributed application condition* A_L over L consists of a finite set of attributed graphical constraints over L. An attributed graph morphism $m : L \to G$ *satisfies* an attributed application condition if $m \models_L cc_L$ for all $cc_L \in A_L$.

A *conditional attributed rule* is an attributed rule equipped with an attributed application condition for the left-hand side. These conditions have to be met in order to applicate the rule.

Definition 1.5.11 (conditional attributed rule and derivation) Let $p = (L \stackrel{l}{\leftarrow} I \stackrel{r}{\rightarrow} R)$ be an attributed rule and A_L an attributed application condition over L. Then $\hat{p} = (p, A_L)$ is a *conditional attributed rule*. The *direct conditional derivation* $G \stackrel{\hat{p},m}{\Longrightarrow} H$ with $\hat{p}$ at match m is given by $G \stackrel{p,m}{\Longrightarrow} H$ if $m \models_L A_L$.

1.5.3 Distributed Attributed Graph Transformation

Distribution of non-attributed graphs is introduced by Taentzer in [Tae96]. In her approach non-attributed graphs are taken to interpret the nodes of a graph and non-attributed graph morphisms are taken for the interpretation of the edges of the graph. *Distributed attributed graph transformation* follows this idea by interpreting nodes of a graph by *attributed graphs* and edges by *attributed graph morphisms*. Non-attributed graphs are well suited for the specification of structural aspects. By means of attributed graphs we are able to incorporate also data aspects into the specification without coding them graphically. Data types are expressed by attributed graph signatures. As data types in different local systems may differ, attributed graph signatures have to differ as well. In order to provide the full variety of attributed graphs over arbitrary attributed graph signatures for expressing local states, we construct category **AGr**. This category consists of all attributed graphs over all attributed graph signatures.

The Category AGr of Attributed Graphs

The category **AGr(AGS)** consists of all attributed graphs and all attributed graph morphisms over one fixed attributed graph signature AGS. We combine the categories $\mathbf{AGr(AGS)}$ for all attributed graph signatures AGS in one category **AGr**. Objects of the combined category are all attributed graphs over all attributed graph signatures AGS. There exists a morphism between attributed graphs over possibly different attributed graph signatures AGS_1 and AGS_2 if there is a signature morphism between the signatures AGS_1 and AGS_2, that preserves the structure of an attributed graph signature, i.e. it maps graphical elements only on graphical ones and data elements only on data elements. We call a structure preserving signature morphism an *attributed graph signature morphism.*

Definition 1.5.12 (category AGr) The category **AGr** has as objects pairs $\langle AGS, A\rangle$ where AGS is an attributed graph signature and A is an attributed graph w.r.t. AGS. Morphisms are pairs $\langle \sigma, h\rangle : \langle AGS_1, A_1\rangle \to \langle AGS_2, A_2\rangle$ where $\sigma : AGS_1 \to AGS_2$ is an attributed graph signature morphism and $h : A_1 \to V_\sigma(A_2)$ is an attributed graph morphism w.r.t. AGS_1 such that $V_\sigma : \mathbf{AGr(AGS_2)} \to \mathbf{AGr(AGS_1)}$ is the forgetful functor over σ (cf. [EM85]). A morphism $\langle \sigma, h\rangle$ is *injective* (*surjective, bijective*) if both σ and h are injective (surjective, bijective).

Proposition 1.5.13 *Category* **AGr** *is cocomplete.*

Proof sketch
Colimits are obtained by constructing first the colimit in the category **AGSig** of attributed graph signatures and attributed graph signature morphisms. Let AGS be the colimit attributed graph signature. Via left adjoints to the forgetful functors w.r.t. the colimit morphisms in **AGSig** all attributed graphs and

attributed graph morphisms involved in the colimit construction are lifted to the category **AGr(AGS)**, where their colimit can be built.

The Category Distr(AGr) of Distributed Attributed Graphs

A *distributed graph* consists of a labeled graph, called *network graph*, and a functor from the small category induced by the network graph to category **AGr**. A L-labeled graph $G = (G_V, G_E, s^G, t^G, l_V^G, l_E^G)$ for a given set L of *labels* consists of a graph (G_V, G_E, s^G, t^G) and labeling mappings $l_V^G : V \to L$ and $l_E^G : G_E \to L$. A L-labeled graph morphism f between L-labeled graphs G and H is a label preserving graph morphism, i.e. $l_V^H(f_V(n)) = l_V^G(n)$ for all nodes n in G and $l_E^H(f_E(e)) = l_E^G(e)$ for all edges e in G.

Definition 1.5.14 (category Distr(AGr)) Given a set of labels L.

- Objects of the category **Distr(AGr)**, called *distributed graphs* , are pairs $\hat{A} = \langle A, \tilde{A} \rangle$ where A is a L-labeled graph and $\tilde{A} : \mathbf{A} \to \mathbf{AGr}$ is a functor from the small category $\mathbf{A}$ induced by A to the category **AGr**. The L-labeled graph A is called *network graph.* For each node $i \in A_V$ the component $\tilde{A}(i) = \langle AGS_i^A, A_i \rangle$ is an attributed graph A_i over AGS_i^A. We call $\tilde{A}(i)$ *local attributed graph* of network node i.
  For each edge $e : i \to j$ in A_E the component $\tilde{A}(e) = \langle \sigma_e^A, h_e^A \rangle$ is an **AGr** morphism.

- Morphisms of the category **Distr(AGr)**, called *distributed morphisms*, are pairs $\hat{f} = \langle f, \tilde{f} \rangle : \langle A, \tilde{A} \rangle \to \langle B, \tilde{B} \rangle$ where $f : A \to B$ is a L-labeled graph morphism and $\tilde{f} : \tilde{A} \to \tilde{B} \circ \mathbf{f}$ is a family of morphisms $\{f_i : A_i \to B_{f(i)} | i \in A_V\}$ in **AGr** such that for each node i in A_V, A_i and $B_{f(i)}$ are attributed graphs over AGS_i^A and f_i is an attributed graph morphism over AGS_i^A. Moreover, for each $e : i \to j$ in A_E we have $f_j \circ h_e^A = h_{f(e)}^B \circ f_i$. We call the f_i for i in A_V *local graph morphisms.*

We call a distributed morphism $\hat{f} = \langle f, \tilde{f} \rangle : \hat{A} \to \hat{B}$ *network-injective* (short n-injective) if f is injective. We call $\hat{f}$ *injective* if f_i for all $i \in A_V$ is injective.

Example 1.2 (distributed graph and distributed morphism) Let $AGS1$ be an attributed graph signature for graphs where nodes are attributed by natural numbers and edges are attributed by strings. Let $AGS2$ be the sub-signature of $AGS1$, where only nodes are attributed by natural numbers, but not the edges. The attributed graph signatures $AGS1$ and $AGS2$ are shown

$AGS1 =$
GS **sorts** $Nodes, Edges$
opns $s, t : Edges \to Nodes$

$DS1$ **sorts** $Nat, String$
opns $+, / : Nat\ Nat \to Nat$
$\lambda : \to String$
$\circ : String\ String \to String$
$length\ : String \to Nat$

$ATTROP1 = a_n : Nodes \to Nat$
$a_e : Edges \to String$

$AGS2 =$
GS **sorts** $Nodes, Edges$
opns $s, t : Edges \to Nodes$

$DS2$ **sorts** Nat
opns $+, / : Nat\ Nat \to Nat$

$ATTROP2 = a_n : Nodes \to Nat$

Figure 1.42: Two attributed graph signatures for local systems. $AGS1$ is intended for graphs, where their nodes are attributed by natural numbers and edges by strings. $AGS2$ attributes only nodes by natural numbers.

in Figure 1.42. Attributed graphs with respect to these attributed graph signatures can be used for expressing local states of a distributed graph. Figure 1.43 shows an example: A distributed graph consists of a network graph where the nodes are interpreted by attributed graphs. In the figure are shown two distributed graphs $\hat{G} = \langle G, \tilde{G} \rangle$ and $\hat{H} = \langle H, \tilde{H} \rangle$ that are connected via a distributed morphism $\hat{f} = \langle f, \tilde{f} \rangle$. The network level shows the network graphs G and H and the network graph morphism f. The local level shows, how the network nodes and edges of G and H are interpreted by attributed graphs. For example, the network node i of G is interpreted by the attributed graph $\tilde{G}(i)$ w.r.t. $AGS1$. Local graphs where edges are attributed by strings are attributed graphs w.r.t. $AGS1$, those one without attributed edges attributed graphs w.r.t. $AGS2$. Local graph morphisms are allowed only between attributed graphs over the same signature. In the example, f_i is an attributed graph morphisms w.r.t. $AGS1$ and f_j is an attributed graph morphism w.r.t. $AGS2$.

We provide now the construction of a pushout in **Distr(AGr)** for n-injective distributed morphisms meeting so-called *locality conditions*. For these morphisms the pushout can be constructed component wise. This is not possible in general. Consider the example of $PO1$ in Fig. 1.44 on the left-hand side.
By pushout construction a new node is generated in the local graph of network node a in $D1$. In order to define the local graph morphism assigned to e_1, an element in the local graph of network node b is generated. Here, the pushout

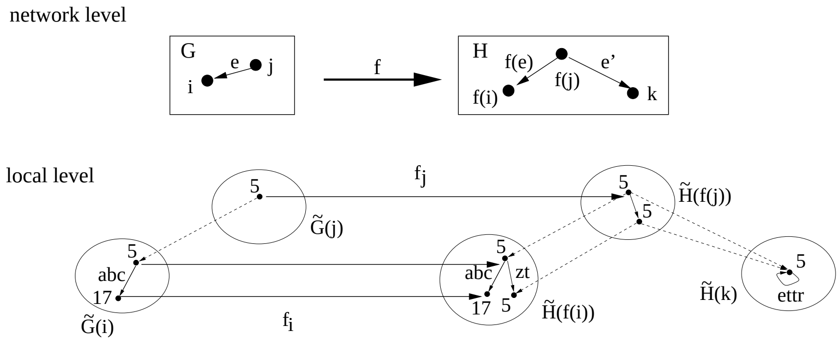

Figure 1.43: Two distributed graphs that are connected via a distributed morphism. In the upper half of the figure the network graphs and the network morphism are shown. The local level shows how network nodes and edges are interpreted by attributed graphs and attributed graph morphisms.

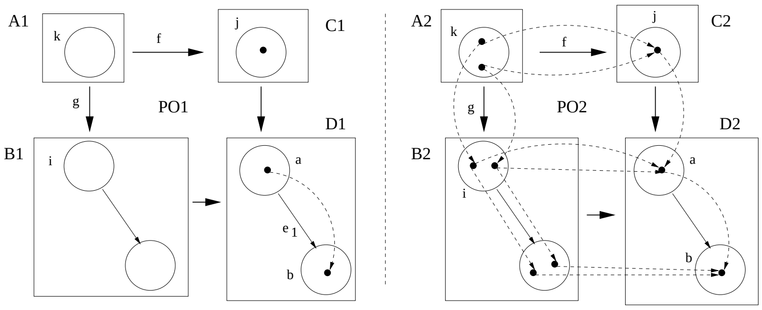

Figure 1.44: Examples for a non component wise pushout construction in **Distr(AGr)**.

can not be constructed locally anymore, since an action in one local graph affects an action in another local graph. The pushout $PO2$ on the right-hand side shows that a gluing of two nodes in a source graph has to be reflected also in the target local graph in order to maintain a graph morphism anyway. We introduce next *locality conditions* that make sure that these non local effects do not occur in the pushout construction. Then, the pushout can be constructed locally for each local system.

Definition 1.5.15 (locality conditions) Two n-injective distributed morphisms $\hat{f} : \hat{A} \to \hat{C}$ and $\hat{g} : \hat{A} \to \hat{B}$ satisfy the *locality conditions*, if

1. for all $e \in C_E - f_E(A_E)$ and for all $y \in A_V$ we have $f_V(y) = s(e)$ implies g_y is bijective.
2. for all $e \in B_E - g_E(A_E)$ and for all $y \in A_V$ we have: $g_V(y) = s(e)$ implies f_y is bijective.

In Fig. 1.44, the distributed morphisms do not satisfy the locality conditions. In the first case, the local graph morphism f_k is not surjective, in the second case it is not injective. If the locality conditions are satisfied by two n-injective distributed morphisms $\hat{f} : \hat{A} \to \hat{C}$ and $\hat{g} : \hat{A} \to \hat{B}$ the pushout can be constructed component wise, i.e. for each i in A_V we construct the pushout of local graph morphisms f_i and g_i in category $\mathbf{AGr}(\mathbf{AGS_i^A})$. The local graph morphisms for the edges of the pushout graph are then uniquely determined by the pushout properties of the local pushouts.

Construction 1.5.16 (component wise pushout construction) *Given two n-injective distributed morphisms $\hat{f} : \hat{A} \to \hat{C}$ and $\hat{g} : \hat{A} \to \hat{B}$ satisfying the locality conditions in 1.5.15. Then, the pushout of $\hat{f}$ and $\hat{g}$ in $\mathbf{Distr}(\mathbf{AGr})$ is constructed as follows:*

1. *Construct the pushout $(D, c : C \to D, d : B \to D)$ of f and g in $\mathbf{Graph}$.*
2. *For all $x \in D_V$ there is at most one $y \in A_V$ with $c_V \circ f_V(y) = x$ as f and g are injective. Then we can uniquely denote by P_x the pushout of f_y and g_y in $\mathbf{AGr}(\mathbf{AGS_y^A})$, and by $c_{f(y)} : C_{f(y)} \to D_x$ and $d_{g(y)} : B_{g(y)} \to D_x$ the pushout morphisms. Then, we define $\tilde{D} : \mathbf{D} \to \mathbf{AGr}$ as follows:*

$$\tilde{D}(x) := \begin{cases} \langle AGS_y^A, P_x \rangle & , \textit{if } \exists y \in A_V : c_V \circ f_V(y) = x \\ \tilde{C}(z) & , \textit{if } \exists z \in C_V - f_V(A_V) : c_V(z) = x \\ \hat{B}(z) & , \textit{if } \exists z \in B_V - g_V(A_V) : d_V(z) = x \\ \langle \sigma_e^A, Ind(P_{s(x)}, P_{t(x)}) \rangle & , \textit{if } \exists e \in A_E : c_E \circ f_E(e) = x \\ \langle \sigma_e^A, c_{t(x)} \circ h_e^C \circ c_{s(x)}^{-1} \rangle & , \textit{if } \exists e \in C_E - f_E(A_E) : c_E(e) = x \\ \langle \sigma_e^A, d_{t(x)} \circ h_e^B \circ d_{s(x)}^{-1} \rangle & , \textit{if } \exists e \in B_E - b_E(A_E) : d_E(e) = x \end{cases}$$

where $Ind(P_{s(x)}, P_{t(x)})$ is the morphism induced by the universal pushout property of pushout $P_{s(x)}$ in category $\mathbf{AGr}$ as $d_{g(t(e))} \circ h_{g(e)}^B \circ g_{s(e)} = c_{f(t(e))} \circ h_{f(e)}^C \circ f_{s(e)}$. Pushout morphisms $\hat{c} : \hat{C} \to \hat{D}$ and $\hat{d} : \hat{B} \to \hat{D}$ are defined as:

$$c_x := \begin{cases} id_{C_x} & , \textit{if } x \in C_V - f_V(A_V) \\ \textit{PO-morphism } c_{f(y)} \textit{ of } PO_{c(x)} & , \textit{if } x = f(y) \end{cases}$$

$$d_x := \begin{cases} id_{B_x} & , \textit{if } x \in B_V - g_V(A_V) \\ \textit{PO-morphism } d_{g(y)} \textit{ of } PO_{d(x)} & , \textit{if } x = g(y) \end{cases}$$

Examples of a component wise pushout construction are depicted in Fig. 1.45. Consider the pushout $PO2$ on the right. Distributed morphisms $\hat{r}$ and $\hat{c}$ are n-injective and satisfy the locality conditions as c_i is bijective. We construct the pushout component wise, i.e. we build the pushout P_i of r_i and c_i in $\mathbf{AGr}(\mathbf{AGS_i^A})$ as well as the pushout P_j of r_j and c_j in $\mathbf{AGr}(\mathbf{AGS_j^A})$. The morphism between the pushout graphs P_i and P_j is the induced pushout morphism in category $\mathbf{AGr}$. The local graphs for the remaining nodes in H are taken from $\hat{C}$ resp. $\hat{R}$.

Proposition 1.5.17 *Let $\hat{f} : \hat{A} \to \hat{C}$ and $\hat{g} : \hat{A} \to \hat{B}$ be two n-injective distributed morphisms satisfying the locality conditions in 1.5.15 and $(\hat{c}, \hat{d}, \hat{D})$ constructed according to 1.5.16. Then, $(\hat{c}, \hat{d}, \hat{D})$ is pushout of $\hat{f}$ and $\hat{g}$ in* $\mathbf{Distr(AGr)}$.

Proof sketch
$\hat{D}$ is a distributed graph and $\hat{c}$ and $\hat{d}$ are distributed morphisms because of the locality conditions and by construction. The commutativity $\hat{c} \circ \hat{f} = \hat{d} \circ \hat{g}$ follows directly from the construction. The universal pushout property holds as $c \circ f = d \circ g$ is pushout in **Graph** and due to the universal pushout properties of the local pushouts P_x for all $x \in D_V$ in $\mathbf{AGr}$. From these properties we can construct component wise the universal distributed morphism $\hat{u}$ for all distributed morphisms $\hat{a}$ and $\hat{b}$ with $\hat{a} \circ \hat{f} = \hat{b} \circ \hat{g}$. The commutativity $\hat{u} \circ \hat{c} = \hat{a}$ and $\hat{u} \circ \hat{d} = \hat{b}$ follow from the commutativity of its components, i.e. from the construction of the network morphism u and all local morphisms u_x for all $x \in D$. The uniqueness of $\hat{u}$ follows also from the uniqueness of the component morphisms. The full proof is given in [FKV99].

Distributed Attributed Graph Transformation

Distributed graphs are transformed via *distributed attributed rules.* A distributed attributed rule consists of a network rule and a set of attributed rules, called *local rules.* The network rule describes the transformation of the network graph. Local systems can be deleted, newly created or simply preserved. For each preserved local system there is a local rule, i.e. an attributed rule according to 1.5.3, that shows how the local system is transformed.

Definition 1.5.18 (distributed attributed rule) A *distributed attributed rule* $\hat{p} = (\hat{L} \xleftarrow{\hat{l}} \hat{I} \xrightarrow{\hat{r}} \hat{R})$ consists of two injective distributed morphisms $\hat{l}$

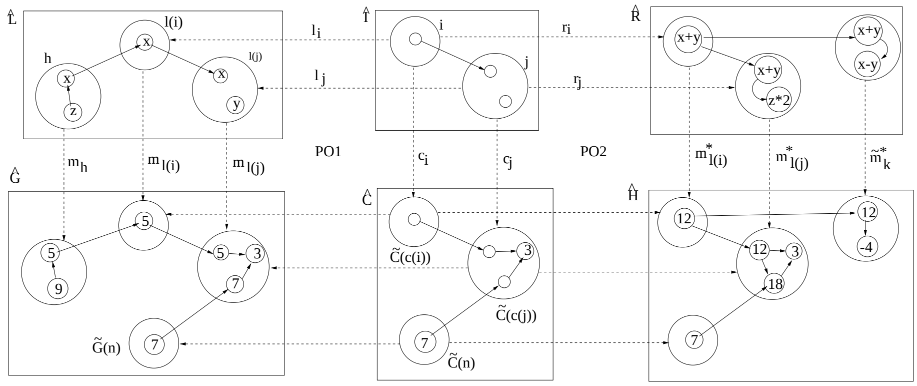

Figure 1.45: A distributed attributed direct derivation. The distributed attributed rule consists of the network rule and two local rules p_i and p_j

and $\hat{r}$ such that for all nodes i in the network interface graph I, the span $p_i = (L_{l(i)} \stackrel{l_i}{\leftarrow} I_i \stackrel{r_i}{\rightarrow} R_{r(i)})$ is an AGS_i^I-rule according to 1.5.3. We call p_i *local rule* for network node i and the rule $p = (L \stackrel{l}{\leftarrow} I \stackrel{r}{\rightarrow} R)$ *network rule.*

Please notice, that the data part of an AGS-rule was the term algebra and the homomorphisms were identities.

Example 1.3 In the upper half of Fig. 1.45 a distributed rule is shown. The nodes in the local graphs have attributes of the term algebra for natural numbers and a set of variables $X = \{x, y, z\}$. Edges, however, have no attributes. The network rule deletes the network node h together with its outgoing network edge and adds a new network node and a new network edge. We have two local rules $p_i = (L_{l(i)} \leftarrow I_i \rightarrow R_{r(i)})$ and $p_j = (L_{l(j)} \leftarrow I_j \rightarrow R_{r(j)})$ for the two nodes i and j in I. The local rule p_i relabels the node in $L_{l(i)}$ from attribute x to attribute $x + y$. Rule p_j relabels the two nodes of $L_{l(j)}$ from x to $x + y$ resp. from y to $z * 2$ and adds an edge between these two nodes. In the newly created local network node two nodes are inserted having attribute $x + y$ and $x - y$. They are connected by an edge.

We need an occurrence $\hat{m}$ of the left-hand side of a distributed rule to a distributed graph in order to apply the rule. A *distributed gluing condition* has to be met by $\hat{m}$ ensuring the existence of a component wise constructed pushout complement. Each of the local rules as well as the network rule of a distributed rule have to fulfill the gluing-condition in Def. 1.5.4. As we are interested in a component wise construction of the pushouts and the pushout complements, two additional conditions have to be considered, namely the *connection* and the *network condition*. An n-injective distributed morphism $\hat{m}: \hat{L} \to \hat{G}$ satisfies the connection condition for a distributed rule $\hat{p}$ if whenever $\hat{p}$ deletes objects in some source local graphs $G_{s(e)}$ resp. in some target graphs $G_{t(e)}$ the local mapping $\tilde{G}(e) = \langle \sigma_e^G, h_e^G \rangle$ must be changed as well. If $\hat{p}$ adds new objects to a local graph $G_{s(e)}$, the local mapping $\tilde{G}(e)$ must be extended, too. $\hat{m}$ satisfies the network condition if a network node is only deleted together with its entire local graph. A network edge e is only deleted if the local graph $G_{s(e)}$ is completely known, so that the local mapping can also be deleted. A new edge e is only inserted at an existing node v becoming the source node of e if G_v is completely known, i.e. the local morphism $\tilde{G}(e)$ is completely specified. The main aim of the connection and network connection is to ensure the consistency of the local graph morphisms. If all these conditions are fulfilled, the context graph can be constructed component wise as shown in Construction 1.5.20.

Definition 1.5.19 (gluing condition and match) Given a rule $\hat{p} = (\hat{L} \overset{\hat{l}}{\leftarrow} \hat{I} \overset{\hat{r}}{\rightarrow} \hat{R})$, a *match* $\hat{m} : \hat{L} \to \hat{G}$ is an n-injective distributed morphism where the following *distributed gluing condition* holds:

1. $m : L \to G$ satisfies the gluing condition for the network rule $p = (L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R)$ (cf. Definition 1.5.4)

2. $m_{l(x)}$ satisfies the gluing condition for the AGS_x^I-rule $p_x = (L_{l(x)} \overset{l_x}{\leftarrow} I_x \overset{r_x}{\rightarrow} R_{r(x)})$ for all $x \in I_V$

3. connection condition: Let $Context(q, n) = H - n(L') + n(l'(I'))$ be the context graph of rule $q = (L' \overset{l'}{\leftarrow} I' \overset{r'}{\rightarrow} R')$ and match $n : L' \to H$:

   (a1) for all $x, y \in I_V$, for all $e : m(l(x)) \to m(l(y))$ [c] $\in Context(p, m)_E$ with $\tilde{G}(e) = \langle \sigma_e^G, h_e^G \rangle$:

   $h_e^G(Context(p_x, m_{(x)})) \subseteq Context(p_y, m_{l(y)})$

   (a2) for all $y \in I_V$, $z \in G_V - m(L_V)$, for all $e : z \to m(l(y)) \in G_E - m(L_E)$ with $\tilde{G}(e) = \langle \sigma_e^G, h_e^G \rangle$: $h_e^G(G_z) \subseteq Context(p_y, m_{l(y)})$

   (b) for all $x \in I_V$, for all $e : m(l(x)) \to z \in G_E - m(L_E) : l_x$ and r_x are bijective.

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

4. network condition: (a) for all $x \in L_V - l(I_V) : m_x$ is bijective
   for all $x \in I_V$, for all $e : l(x) \to y \in L_E - l(I_E) : m_{l(x)}$ is bijective
   (b) for all $x \in I_V$ for all $e : r(x) \to y \in R_E - r(I_E) : m_{l(x)}$ is bijective.

If two n-injective distributed morphisms satisfy the locality conditions the pushout can be constructed component wise.

Definition 1.5.20 (context graph) Given a distributed rule $\hat{p} = (\hat{L} \xleftarrow{\hat{l}} \hat{I} \xrightarrow{\hat{r}} \hat{R})$ and a match $\hat{m} : \hat{L} \to \hat{G}$. Then the *context graph* $\hat{C} = \langle C, \tilde{C} \rangle$ with morphisms $\hat{c} := \hat{m} \circ \hat{l} : \hat{I} \to \hat{C}$ and th e inclusion $\hat{g} : \hat{C} \to \hat{G}$ of $\hat{C}$ into $\hat{G}$ is constructed as follows:

1. The network graph C is the context graph of the network rule p and match m as defined in 1.5.5.
2. We define $\tilde{C} : \mathbf{C} \to \mathbf{AGr}$ as follows:
   - For all $i \in C_V$ with $c(y) = i$ and $y \in I_V$ we define $\tilde{C}(i) = \langle AGS^I_y, C_i \rangle$ where C_i is the context graph of local rule p_y and $m_{l(y)}$ (cf. 1.5.5).
   - For all $i \in G_V - m_V(L_V)$ we define $\tilde{C}(i) := \tilde{G}(i)$.
   - For all $e \in C_E$ with $\tilde{G}(e) = \langle \sigma^G_e, h^G_e \rangle$ we define $h^C_e := g_{t(e)} \circ h^G_e \circ g^{-1}_{s(e)}$.

Example 1.4 (context graph) The construction of the distributed context graph is shown in . The match $\hat{m}$ in Figure 1.45 satisfies the distributed gluing condition for the distributed rule $\hat{p}$ as the match m satisfies the gluing condition for the network rule and the local matches m_x for $x \in \{i, j\}$ satisfy the gluing condition for the local rules p_x. The connection condition is fulfilled as both local rules do not delete objects in local graphs and no local source graph is extended by objects. The network condition is satisfied as the deleted network node corresponds bijectively with that in the rule and $m_{l(i)}$ is bijective. We construct the context graph by constructing the network context graph. This graph is generated by deleting the network node n and the adjacent network edge. Then, the local context graphs $\tilde{C}(c(i))$ and $\tilde{C}(c(j))$ are constructed for l_i and $m_{l(i)}$ resp. l_j and $m_{l(j)}$. Here, the corresponding labels are deleted according to the local rules. The local graph $\tilde{C}(n)$ is taken from $\tilde{G}(n)$ as the network node n is not in the image $m(L)$. The local graph morphisms are then defined in such a way, that commutativity is obtained.

Proposition 1.5.21 (applicability of rules) *Given a distributed rule* $\hat{p} = (\hat{L} \xleftarrow{\hat{l}} \hat{I} \xrightarrow{\hat{r}} \hat{R})$, *a match* $\hat{m} : \hat{L} \to \hat{G}$ *and the context graph* $\hat{C}$ *as well as* *morphisms $\hat{c}$ and $\hat{g}$ as defined in 1.5.20, then $(\hat{G}, \hat{m}, \hat{g})$ is pushout of $\hat{c}$ and $\hat{l}$ in* **Distr(AGr)**. *Furthermore, the pushout-complement $(\hat{C}, \hat{g}, \hat{c})$ is unique up to isomorphism.*

Proof sketch
The context graph $\hat{C}$ is a distributed graph and the morphisms $\hat{g}$, $\hat{c}$ are distributed morphisms as each of the local rules and the network rule satisfy the gluing condition and $\hat{m}$ satisfies the connection condition a1) and a2). The distributed morphisms $\hat{c}$ and $\hat{l}$ satisfy the locality conditions due to the network condition and connection condition b), such that we can construct the pushout of $\hat{c}$ and $\hat{l}$ component wise. It can be shown that this pushout graph is isomorphic to $\hat{G}$. The construction of the isomorphism is done with the help of the pushout properties of the single components of the distributed graph $\hat{G}$, i.e. its network pushout graph and its local pushout graphs.
Uniqueness of the pushout-complement follows from the injectivity of rules and the existence of identities in the attribute part of the rule. A full proof is given in [FKV99].

Definition 1.5.22 (direct derivation) Given a distributed rule $\hat{p} = (\hat{L} \xleftarrow{\hat{l}} \hat{I} \xrightarrow{\hat{r}} \hat{R})$ and a match $\hat{m} : \hat{L} \to \hat{G}$. A *direct distributed attributed graph transformation* $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow} \hat{H}$ is given by a double pushout square in category **Distr(AGr)** as shown in Fig. 1.46.

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

Figure 1.46: distributed direct derivation

$$\begin{tikzcd}
\hat{A}_i \arrow[drr, "\hat{r}"'] & \hat{A} \arrow[l, "\hat{h}_i"'] \arrow[dr, "\hat{q}"] & \hat{L} \arrow[l, "\hat{c}"'] \arrow[d, "\hat{m}"] \\
 & & \hat{G}
\end{tikzcd}$$

Figure 1.47: distributed attributed application condition

An example of a direct derivation is given in Fig. 1.45.

Theorem 1.5.23 (existence and uniqueness) *Let $\hat{p} = (\hat{L} \xleftarrow{\hat{l}} \hat{I} \xrightarrow{\hat{r}} \hat{R})$ be a distributed rule and $\hat{m} : \hat{L} \to \hat{G}$ be a distributed morphism. If $\hat{m}$ satisfies the distributed gluing condition (cf. Def. 1.5.19) then there is a direct derivation $\hat{G} \overset{\hat{p},\hat{m}}{\Rightarrow} \hat{H}$ being unique up to isomorphism.*

Proof
The theorem follows from Proposition 1.5.21 and pushout properties.

1.5.4 Distributed Attributed Application Conditions

In section 1.5.2 attributed application conditions for attributed rules were introduced. We generalize these concepts to distributed attributed rules by defining a distributed attributed application condition by a finite set of *distributed attributed graphical constraints*. A distributed attributed graphical constraint over a distributed graph $\hat{L}$ consists of an attributed graphical over local graph L_i for each node i in L (cf. 1.5.9).

Definition 1.5.24 (distributed attributed application condition)
Given a distributed rule $\hat{p} = (\hat{L} \stackrel{\hat{l}}{\leftarrow} \hat{I} \stackrel{\hat{r}}{\rightarrow} \hat{R})$, a *distributed attributed graphical constraint* $\hat{cc}_{\hat{L}} = (\hat{c} : \hat{L} \rightarrow \hat{A}, c_{\hat{A}})$ over $\hat{L}$ consists of an n-injective distributed morphism $\hat{c}$ and an I-indexed set $c_{\hat{A}} = \{\hat{h}_i : \hat{A} \rightarrow \hat{A}_i\}_{i \in I}$ of n-injective distributed morphisms $\hat{h}_i$ such that for all $x \in L_V$ the attributed graph morphisms c_x are equational constraints over L_x and for all $i \in I$ and $y \in A_V$, the attributed graph morphisms h_{i_y} are equational constraints over A_y.

A distributed morphism $\hat{m} : \hat{L} \rightarrow \hat{G}$ *satisfies* a distributed attributed graphical constraint $\hat{cc}_{\hat{L}}$, denoted by $\hat{m} \models_{\hat{L}} \hat{cc}_{\hat{L}}$, if for all n-injective distributed morphisms $\hat{q} : \hat{A} \rightarrow \hat{G}$ with $\hat{m} = \hat{q} \circ \hat{c}$ there is an $i \in I$ as well as an n-injective distributed morphism $\hat{r} : \hat{A}_i \rightarrow \hat{G}$ such that $\hat{q} = \hat{r} \circ \hat{h}_i$ (cf. Fig. 1.47).

A *distributed attributed application condition* $A_{\hat{L}}$ over $\hat{L}$ consists of a finite set of distributed attributed graphical constraints over $\hat{L}$. A distributed morphism $\hat{m} : \hat{L} \rightarrow \hat{G}$ *satisfies* a distributed attributed application condition if $\hat{m} \models_{\hat{L}} \hat{cc}_{\hat{L}}$ for all $\hat{cc}_{\hat{L}} \in A_{\hat{L}}$.

In the same way as we have defined conditional attributed rules and conditional attributed derivations in 1.5.11, we define now *conditional distributed rules* and *conditional distributed derivations*.

Definition 1.5.25 (conditional distributed attributed rule) A *conditional distributed attributed rule* is a pair $(\hat{p}, A_{\hat{L}})$, where $\hat{p} = (\hat{L} \stackrel{\hat{l}}{\leftarrow} \hat{I} \stackrel{\hat{r}}{\rightarrow} \hat{R})$ is a distributed attributed rule and $A_{\hat{L}}$ is a distributed attributed application condition over $\hat{L}$. The *direct conditional derivation* $\hat{G} \stackrel{(\hat{p}, A_{\hat{L}}), \hat{m}}{\Longrightarrow} \hat{H}$ with $(\hat{p}, A_{\hat{L}})$ at match $\hat{m}$ is given by $\hat{G} \stackrel{\hat{p}, \hat{m}}{\Longrightarrow} \hat{H}$ if $\hat{m} \models_{\hat{L}} \hat{A}_{\hat{L}}$.

1.5.5 Local Views on a Distributed System

In this section we briefly sketch the ideas of a *local view* on a distributed system. Distributed rules as defined in 1.5.18 allow to specify actions on a distributed system affecting several local systems in one rule application. Thus, distributed actions are specified in a global view. In the early stage of system development a global view on the whole distributed system is desirable to get an overview. Once the network structure and its reconfiguration possibilities as well as the interfaces are fixed, the system developers are supposed to take a local view on network nodes and local system parts running on them. Therefore, distributed rules have to be restricted to rules that can be applied locally only. To define locality we propose a network graph consisting of local systems that are connected via export and import interfaces. A local system may have several export and several import interfaces. In export interfaces, local systems present objects accessible from other local systems and import interfaces contain objects used from remote export interfaces. When a local system wants to use exported objects of a remote local system it connects one of its import interfaces with the corresponding export interface. In order to know, when a local system exports objects that are imported by itself, we introduce a *public import interface* between the export and the import interfaces of a local system. Public import interfaces contain objects that are the same in the export and the import interface. Without the public import interface, this information is not visible for remote local systems. An example is given in Figure in Fig. 1.48.

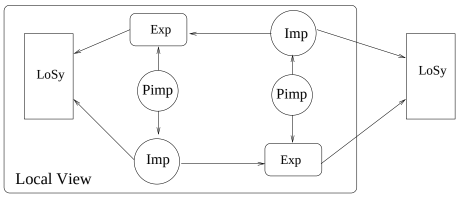

Figure 1.48: A distributed system consisting of two local systems. Each of the local systems has one export and one import interface. Export and import interface are connected via a public import interface.

It shows two local systems and their connection via export and import interfaces. By means of this model of a distributed system we are able to define locality. A local system knows its local state and the states of its export and (public) import interfaces. Furthermore, the local system knows the local states of export interfaces of remote local systems and the local states of remote

(public) import interfaces, that use its export interfaces. While it is natural that a local system knows the exports from where it imports, it is not so clear that this relation is also useful in the other directions. If an export knows all the imports it is related to, this can be advantageously used to inform the imports immediately when the export changes. This is not possible otherwise. The local state of a remote local system is invisible for a local system. All components that can be seen by a local system, i.e. its own local state, the local states of its own interfaces, remote export interfaces and connected remote import interfaces, are in the *local view* of the local system. The local view of the local system in the left-hand side of Figure 1.48 consists of two interfaces belonging to this system and the two remote interfaces. Only the local state of the local system on the right-hand side is outside of the local view of the local system.
A *local view rule* is a distributed rule, where its left-hand side contains at most one local system and interfaces being in its local view.

Communication between local systems may take place synchronously as well as asynchronously via export and import interfaces. Whereas asynchronous communication is simply modeled by sequential application of local view rules, synchronous communication requires more sophisticated mechanisms. Synchronization is expressed by *parallel synchronization* and *concurrent synchronization* of local view rules. Parallel synchronization of rules describes a kind of service request for objects not yet available in the interface. It is formalized by amalgamating the local view rules in category **Distr(AGr)** as presented in [Tae97b] for any cocomplete category. Concurrent synchronization describes just the usage of a service already available in some interface. It comprises the construction of a concurrent rule as shown in [PT95]. However, the amalgamated and the concurrent rule are not any more local view rules.

Local systems can be of different type. In our case study we have two types, namely the type *revision archive* and *workspace*. For each type there exists a *local view grammar* that contains a distributed initial graph, local view rules, where the local system of its left-hand side is of the corresponding type and a *start rule*. The *start rule* for a type generates a local system of this type and initializes it according to the initial graph of the local view grammar. A local system can be started, and the start rule applied, when all remote interfaces exist that are needed for the local system. A *distributed grammar* consists of a local view grammar for each local system type. The distributed grammar of our case study contains a local view grammar for the revision archive and for the workspace.

The *operational semantics* of a distributed grammar (with global view) is constructed by all possible derivations with local view rules and synchronized rules generated by the local view rules. The operational semantics of a local view grammar is defined by the projection from the operational semantics of the distributed grammar onto the local system.

1.6 Conclusions

Modeling distributed systems by distributed graph transformation a powerful and flexible description of *dynamic networks* is obtained by applying graph transformation to network structures. *Communication* and *synchronization* are modeled by graph sharing and common subtransformation on shared graph parts.

Distributed graph transformation is formalized as a structured transformation of structured graphs, characterized by a double-pushout on distributed graph morphisms. Distribution concepts have been integrated in a new approach to attributed graph transformation combined with application conditions for rules.

In this article, we restricted the notion of distributed graph transformation on a double-pushout of n-injective distributed morphisms. The general approach on arbitrary distributed morphisms is presented in [Tae97a]. In this approach also split and join of local systems and parallel execution of several rules in one local system are possible to be described. It is left open for future work to integrated this general approach with attributed graph transformation and application conditions.

All distribution concepts presented (the just mentioned application conditions included) are formalized on the basis of the double-pushout approach to graph transformation as well as diagrams and natural transformations. Since the main concepts are characterized in a purely categorical way they may be generalized to high-level replacement systems (compare article [?] in this handbook) exchanging the category **AGr** of attributed graphs by another cocomplete category.

In this paper, we introduced local views on the distributed transformation of attributed graphs. Local views are formulated by distributed graph grammars containing one local system with import and export interfaces together with remote import and export interfaces to which the local system may have connections. Local systems can communicate asynchronously by simply applying local view rules sequentially, or synchronously by constructing concurrent and amalgamated distributed rules from local view rules. These construction model

the usage and the request of services, resp.
For defining the operational semantics of a distributed system we use concurrent and amalgamated rule constructions describing the synchronization between local views. However, these rules are not local view rules anymore, such that the semantics is specified globally again. Future work has to be done to define the operational semantics of a distributed system grammar by composing the operational semantics of its local view grammars.
The sample application in Section 1.4 has shown that distributed graph transformation has a great potential to naturally design a distributed system. The graphical description of remote object interaction, object migration and replication, communication and synchronization as well as dynamic reconfiguration of distributed systems can be described conveniently in a visual notation. Since the rule-based way of design is well suited to describe highly concurrent events which cannot be temporally ordered, we proposed to use graph transformation for all dynamic distributed structures. Although distributed graph transformation has been applied to a non-trivial sample application, further validation is needed to find out how far these concepts are adequate and which further issues have to be considered.
Especially for the development of distributed systems, consistency is a very important aspect. A notion of consistency conditions, logical formulas on the existence and non-existence of graph structures, and a constructive approach to ensure these conditions are presented in [HW95]. It is due to future work to extend this work to distributed graph transformation.
Currently, we are working on an implementation of our ideas which will be based on the AGG-language and environment [TER98]. Of course, this implementation is designed such that it can run in a really distributed way.

References

[Agh86] G. Agha. *ACTORS: A model of concurrent computation in distributed systems.* MIT Press, Cambridge, Mass., 1986.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, 1994.

[Bur86] P. Burmeister. A model theoretic oriented approach to partial algebras. *Mathematical Research – Mathematische Forschung*, 32:257–274, 1986.

[CL95] Ingo Classen and Michael Löwe. Scheme evolution in object oriented models:A graph transformation approach. In *Proc. Workshop on Formal Methods at the ISCE'95*, 1995.

[CMR+97] 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, *Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations*. World Scientific, 1997.

[DM87] P. Degano and U. Montanari. A model of distributed systems based on graph rewriting. *Journal of the ACM*, 34(2):411–449, 1987.

[EHK+96] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach. In G. Rozenberg, editor, *The Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1996.

[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. The ESPRIT BRWG COMPUGRAPH Computing by Graph Transformations : A survey. In *TCS 109*, pages 3–6. North-Holland, 1993.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1985.

[FKV99] Ingrid Fischer, Manuel Koch, and Victor Volle. Modeling Distributed Configuration Management by Distributed Attributed Graph Transformation. Technical report, TU Berlin, 1999. to appear.

[Har87] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hum89] U. Hummert. *Algebraische High-Level Netze*. PhD thesis, Technische Universität Berlin, 1989.

[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.

[ISO89] ISO. Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour. International Standard ISO 8807, ISO, 1989.

[Jen95] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 2. Springer, 1995.

[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.

[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.

[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.

[LKW93] M. Löwe, M. Korff, and A. Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In *Term Graph Rewriting - Theory and Practice*, pages 185–199. Wiley, 1993.

[LR98] M. Llabrés and F. Rosselló. Pushout complements for arbitrary partial algebras. In G.Engels and G. Rozenberg, editors, *Proc. of 6th International Workshop on Theory and Apllication of Graph Transaformation*, pages 80–87. Universität-Gesamthochschule Paderborn, 98. tr-ri-98-201.

[Mil89] R. Milner, editor. *Communication and Concurrency.* Prentice Hall International, 1989.

[MP92] D.E. Monarchi and G.I. Puhr. A research typology for object-oriented analisys and design. *Communications of the ACM*, 35(9):35–47, Sep. 1992.

[Pet62] C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Schriften des Institutes für Instrumentelle Mathematik, Bonn, 1962.

[PT95] J. Padberg and G. Taentzer. Embedding of derivations in high-level replacement systems. In G. Valiente Feruglio and F. Rosello Llompart, editors, *Proc. Colloquium on Graph Transformation and its Application in Computer Science.* Technical Report B-19, Universitat de les Illes Balears, 1995.

[Rat97] Rational software cooperation. Unified modeling language. available via http://www.rational.com, 1997.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, E. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design.* Prentice Hall International, 1991.

[Rei84] Horst Reichel. *Structural Induction on Partial Algebras*, volume 18. Mathematical Research, Akademie-Verlag Berlin, 1984.

[Rei87] Wolfgang Reisig. Place/transition systems. In *Petri Nets : Central Models and Their Properties.*, volume 254 of *Lecture Notes in Computer Science*, pages 117 – 141. Springer Verlag, 1987.

[Rei91] W. Reisig. Petri Nets and Algebraic Specifications. *Theoretical Computer Science*, 80:1–34, 1991.

[Sch92] G. Schied. *Über Graphgrammatiken, eine Spezifikationsmethode für Programmiersprachen und verteilte Regelsysteme.* Arbeitsberichte des Institus für mathematische Maschinen und Datenverarbeitung (Informatik), PhD thesis, University of Erlangen, 1992.

[Sch93] G. Schied. Konstruktive Spezifikation verteilter Systeme mit Graphgrammatiken. Unpublished, 1993.

[Sti92] Colin Stirling. Modal and Temporal Logics. In *Background: Computational structures*, volume 2 of *Handbook of Logic in Computer Science*, pages 477–563. Clarendon Press, OXFORD, 1992.

[Tae96] Gabriele Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, TU Berlin, 1996. Shaker Verlag.

[Tae97a] G. Taentzer. Distributed graphs and graph transformation. *Applied Categorical Structures*, 1997. Accepted.

[Tae97b] G. Taentzer. Parallel high-level replacement systems. *TCS*, (186), 1997.

[TER98] G. Taentzer, C. Ermel, and M. Rudolf. Agg-approach: Language and tool environment. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Specification and Programming*. World Scientific, 1998. to appear.

Index