Dynamic Change Management by Distributed Graph Transformation: Towards Configurable Distributed Systems

extended abstract

Gabriele Taentzer
Computer Science Department
Technical University of Berlin, D-10587 Berlin
gabi@cs.tu-berlin.de

Michael Goedicke and Torsten Meyer
Dept. of Mathematics and Computer Science
University of Essen, D - 45117 Essen
{goedicke,tmeyer}@informatik.uni-essen.de

Abstract

In this contribution we consider the application of distributed graph transformation to the problem of specifying dynamic change in distributed systems. Change in distributed systems is related to at least two levels. One is the management of change in a local node of the distributed system and how such a local change is then propagated to those nodes in the systems which need to know about the change. The other aspect is changing the structure of the distributed system itself. This implies e.g. to add and/or remove a local node or an entire subsystem to/from the distributed system. In some important application areas such operations must be done during runtime without disturbing the unmodified rest of the distributed computing system.
We first give an overview of our model of change and how exactly the two aspects of change interact. We describe distributed graph transformation as a technique to realize our change model. An example - a ring database - then shows how our approach can be applied to a small but nontrivial distributed system. This example shows nicely how the two aspects of change can be described uniformly using graph transformation rules and how the interaction of the two change aspects can be defined in an adequate way. Since this is ongoing work we conclude with an assessment of our approach and a brief discussion of further work.

1. Introduction and related work

Graphical representations are an obvious means to describe various aspects of software systems. In the case of distributed systems graphs are often used to describe models of the system's network structure. Graph transformations which here means the rule-based manipulation of graphical structures, can be conveniently used to model dynamic changes of the network architecture. Typically the local states of such systems are coded in some specification or programming text or are not considered at all. This idea is followed, for example, in [4], by Δ–grammars in [10], in [18] and by actor graph grammars in [9, 11].

However, in the case of distributed systems, graph transformation can be employed twice: (1) to describe the dynamic reconfiguration of distributed systems and (2) to model evolving data and object structures in the local systems. The usage of graph transformation at the local level can include the description of local actions as well as remote interaction, such as object migration, replication, communication and synchronization. All these activities are described in a rule-based manner by defining parts of the graph structure before and after a transformation. Thus the application of one or more graph rules, i.e. performing distributed graph transformations, results in a model of distributed computations.

Our approach of distributed system design by distributed graph transformation allows to export parts of the internal structures as well as dynamic behaviour which means to export also parts of the semantics. This information is extremely useful to find, for example, adequate services on remote systems, since syntactic information like names or signatures are often not expressible enough to find the right service. Another advantage is that interfaces may change dynamically which takes care of the fact that not all services can be offered all the time. This property can be described in a natural way using our approach since interfaces are again defined by graphs and graph rules. A service is available if a certain interface rule is applicable to its interface graph.

The concepts as well as the formal definition of distributed graph transformation based on the algebraic approach [3] are introduced in [19]. Distributed graph transformation is extended by an attribution concept for graphs and the concept of application conditions for graph rules in [20]. Using graph transformation as underlying formal framework, consistency of the resulting distributed system structures is ensured.

An obvious application area for distributed graph transformation is distributed systems design where the two levels mentioned above exist in a natural way. The design of such distributed systems involves to layout the architecture of the system which means to find the overall structure and the elements of the structure. Since the system runs in a distributed way care must be taken to achieve a certain notion of consistent state, although not all components of the distributed system's state cannot be accessed all the time. In the case of dynamic (re)-configuration of such systems additional measures must be taken that the exchange, introduction and removal of system components is done in a smooth way without disturbing the rest of the system too much. In general such an architectural change involves to set the involved components into a kind of silent state such that the architectural change actions can be performed and the new parts may be activated in turn.

The research area which is primarily involved in system design of this kind is the area of configurable distributed systems. Prominent work related to this area is done at the Imperial College, London. In [13] J. Kramer and J. Magee propose a basic model for dynamic change management which separates structural issues from application specific ones, thus distinguishing change rules at the architecture configuration level and change actions at the component level. In [14] self-configuring architectures are investigated and in [6, 15] a graphical user interface and a notation for

component configurations based on the architecture description language REGIS/DARWIN [16] are presented. An approach to realize configurable distributed systems based on negotiation of component semantics with the help of graph transformation techniques is presented in [7]. The ring database example modeled in this paper was first introduced in the world of dynamic change using DARWIN in [12]. Important work wrt. dynamic reconfiguration is also done at the University of Maryland, U.S.A. In [1] J. Purtilo et al. present a framework for dynamic reconfiguration of application software, including a notation to express reconfiguration plans and a prototype system.

Another work to describe self-organizing software architectures is given in [22]. With the Chemical Abstract Machine [2] an architecture specification and programming model is presented and for covering dynamic issues graph rewrite rules [17] are introduced. However, this approach includes no distribution or modularization aspects.

In the following we will give in chapter 2 a brief account of dynamic change management in distributed systems. In chapter 3 we will present distributed graph transformation as the technique to realize these goals. Finally, in chapter 4 we will show the applicability of our approach by modeling a well-known example in the world of dynamic change.

2. Dynamic change management and system reconfiguration

Change in distributed systems has at least two aspects. On the one side there is the problem to maintain a certain - application dependent - notion of consistency without having access to the entire state of the distributed system all the time. On the other side there is the general need to evolve a distributed system by adding, removing or substituting software components during the runtime of the system. While the first aspect is a difficult problem in itself in a distributed system without structural change we would like to concentrate here on the second aspect and how to deal with the interaction between structural changes and state changes of the first kind as well. The first aspect is sketched as future work in the conclusion.

We follow the model developed in [13, 12]. This model implies a strict separation of the two change aspects mentioned above.

2.1. Change model

Separation of concerns applied to the design of distributed systems implies the separation of designing the various components from designing the structure of the system. This is naturally transferred to the management of change in such systems. The change of the distributed system structure has to be done at the level of the structure and the node interfaces only. This implies that the interaction between the change actions and the nodes of system is minimal. As a consequence the evolution of the system can be performed in a structured and systematic way.

The idea outlined above requires a certain view on the system structure and its nodes. The nodes offer an interface which provides their publicly available services which are linked to each other according to the system structure via communication channels.

The actions at the structural level - which we will refer to as the network level later on - are addition and / or deletion of a node or subsystem and linking and / or unlinking of nodes. This is the original model of change actions according to [13]. It minimizes the interaction between the change actions at the network level and the related actions necessary within the involved nodes. I.e. involved nodes have to be in a quiescent state in order allow the performance of the change action at network level. A node is said to be in a quiescent state if the application's state is consistent and there is no communication in progress between nodes affected by a change nor between them and their environment. This means that a distributed node

1. is not currently engaged in a communication connection that it initiated,
2. will not initiate new communication connections,
3. it is not currently engaged in servicing a communication connection request,
4. no communication connections have been or will be initiated by other nodes which require service from this node.

If only the first requirement is satisfied a node is said to be in a passive state. Further, the nodes must remain quiescent while the entire change is executed.

Thus the only interaction between the structural change actions and the nodes are services which activate and / or passivate a node.

This view, however, has a certain drawback. It does not deal with the situation that a node is split into more than one node. Also in the case of adding a new node there is the problem of "informing" the new node about the state it has to know in order to cooperate smoothly with the existing nodes of the system. Thus we would like to extend the model by means to describe the interaction between nodes more in detail. Still the interaction has to be done through interfaces which hide internal details from the (mis)use by other nodes. Our approach provides the possibility for multiple interfaces which allow to deal with different aspects using specialized interfaces.

2.2. Requirements and objectives

In this section we present the required characteristics of a configuration management system for managing dynamic change according to J. Kramer and J. Magee [13]. As described above, these objectives are focussed on separating structural configuration from local application specific functions. Thus the configuration management facility is application independent.

The first point is that *changes should be specified in terms of the system structure*. Software evolution at the level of single local components is too detailed and impractical due to the tight coupling of program elements. Thus change should be supported at the architecture level.

The next point is that *change specifications should be declarative*. The goal is to separate the change required from how it is achieved. The responsibility how the change is executed - e.g., the specific ordering of actual change operations - has to be left to the configuration management system, not the user.

Further, *change specifications should be independent of algorithms, protocols, and states of the application*. As mentioned above, the goal is to provide generic configuration management independent of the application. Changes have to be specified in terms of system architecture and application specific aspects like algorithms, protocols, and states have to be abstracted to a more general requirement. In [13] it is shown that component quiescence / passivity represent an abstract view of a component's state.

Changes should leave the systems in a consistent state. A consistent application state is one from which the system

can continue processing normally rather than progressing towards an error state. While the systems moves from one consistent state to the next, application transactions access the internal state of components. In order to avoid the loss of application transitions and achieve a consistent state after the change, the affected system part has to be in a consistent application state before the change.

Finally, *changes should minimize the disruption to the application system*. The final goal is to realize configurable distributed systems where evolutionary change can be accommodated dynamically, i.e. those parts of the system which are not affected by the change are not disturbed in their execution.

Below we now introduce the used distributed graph transformation approach which is then applied to the ring-database problem. This shows how the requirements stated above are satisfied.

3. Distributed graph transformation

Distributed graph transformation is structured graph transformation at two abstraction levels: the network and the local level. The network level contains the description of a system's network topology by a *network graph* and its dynamic reconfiguration during runtime by *network rules*. The application of such a rule to a network graph follows the double-pushout approach to graph transformation as presented in [3]. 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.

At the local level, graph transformation is used to manipulate local data structures. To describe distributed data, replication and migration of data to remote systems, synchronization and communication between distributed systems, as well as dynamic system configuration the combination of network graph transformation and local transformations is needed. Distributed graph transformation has exactly this purpose. Distributed graphs can be nicely used to model a system's network and data structures which are distributed over different local systems. All kinds of local and distributed system interactions as well as dynamic network reconfiguration can be described by distributed graph transformation.

Before we will explain graph transformation in more detail (cf. section 3.2) and discuss how it can be integrated into the change model (cf. section 3.4) we now introduce below distributed graphs on which our graph transformation approach is based.

3.1. Distributed graphs

A distributed graph consists of a network graph where each network node is equipped with a local graph describing a local system state. Each network edge is equipped with a total graph morphism relating local graphs to each others. These graph morphisms are used to describe replicated data structures. Each node in a distributed graph may typed and attributed by further data types. Usually, each local system contains different sets of node, edge and attribute types. The interfaces may contain any subsets of types for communication (compare [20, 5] for more details to attribution of distributed graphs).

Network nodes being the source of a network edge may be interpreted as interfaces. Figure 1 shows an example of a distributed graph. In the following, we distinguish import and export interfaces. Export interface graphs offer public parts of local graphs whereas import graphs and their connection morphisms to export graphs show the imported data structures. Compare figure 1 and figure 5 for sample network graphs with import and export nodes. Figure 6 ff. show sample distributed graphs where import and export graphs are not explicitly presented but recognizable at a different color of local graph elements. In all these examples, local graph morphisms are unambiguously definable obeying compatibility with attributes. Thus, they are not explicitly depicted.

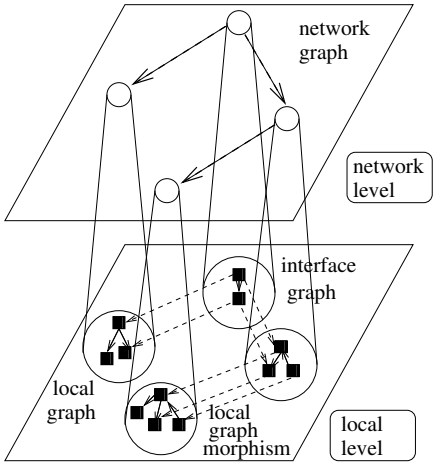

Figure 1 Example of a distributed graph

3.2. Distributed graph rules

Distributed graph rules are useful to describe dynamic network reconfiguration by a network rule as well as dynamic local data structures in each component by a set of local rules. 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 local graphs. Local rules have to be compatible with local graph morphisms. See figure 2 for a schematic representation of a distributed rule. In this example, one local network node, its local graph as well as the network edge and its local morphism are deleted. The local graph determines the state in which such a network node is allowed to be deleted. One network node is preserved. Its local graph has to be manipulated by the local rule given. On the right-hand side a new network node has to be created and its initial state is given by its local graph.

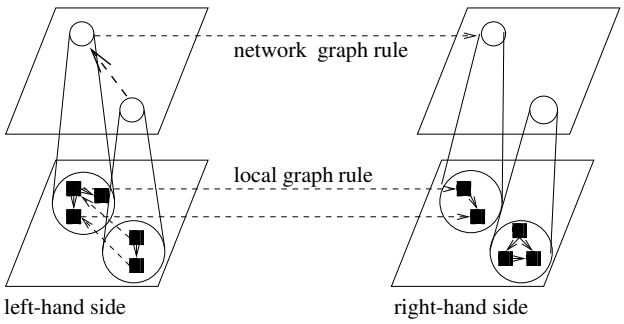

Figure 2 Example of a distributed rule

Moreover, it is possible to split and glue network nodes. The underlying local graphs have to be split and glued, too. The information how to split has to and can be given by distributed rules, too. The local graph of a network node to be split has to be mapped to graph splits. Local nodes and edges

- which are not mapped at all are deleted as usual,

- which are mapped to one local graph, are assigned to that graph,
- which are mapped several times to local graphs, are copied as often as needed.

Consider figure 3 for an example of a splitting and copying rule. The upper local node in the left-hand side of the rule is copied into both local graphs on the right-hand side whereas the local edges and lower nodes are split into the two local graphs on the right-hand side.

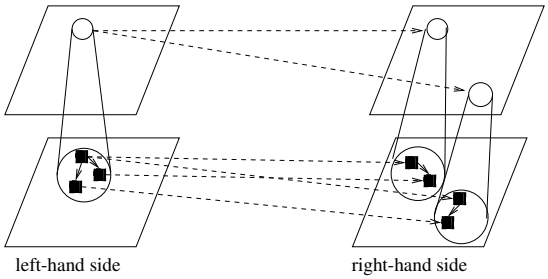

Figure 3 Example of a splitting and copying distributed rule

In the following section, rules are presented by their graphs, the rule morphisms are left out whenever they may be installed unambiguously obeying compatibility with source, target, type and attribute mappings.

Rules may contain variables for attribute values which are globally defined within a rule. Some of these variables are considered as rule parameters which are set from outside while all the others are instantiated by a rule match. In chapter 4, variables for node attributes are given by sets NV and parameters are listed in a rule's head after its name.

Distributed graph rules can be used to describe all kinds of dynamics in a distributed systems. These are manipulations of local data structures, synchronization and communication as well as dynamic reconfiguration of distributed systems.

A *local action* can be modeled by a distributed rule consisting of one local rule which describes the local action and an identical network rule containing 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 export interface is always possible.

A *network reconfiguration* can be also described by a distributed graph rule where its network rule describes the changes of the network topology. Moreover, the initial (final) state of newly created (deleted) components are determined. For preserved components those exports have to be specified which will be involved in remote component creation or deletion. Splitting or copying a network node the corresponding state where such an action is allowed has to be determined by the rule. Replication into a new network node is always possible.

Communicating actions may be performed synchronously or asynchronously. Asynchronous communication means sequential application of local rules such that some graph elements are put into an export interface by one rule which may be used by an importing component formulated in another rule. Synchronous actions are modeled by several local rules in different components connected with each others via common interface rules. If two local systems want to synchronize, at least one import of one component has to be connected to one export interface of another. The corresponding network rule is an identical one involving the network nodes and edges where the synchronization has to take place.

3.3. Distributed graph transformation

To apply a distributed rule, several application conditions have to be checked to ensure that the resulting graph 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 should not produce implicit side-effects on the graph structure. The set of application conditions consists of the well-known gluing condition of the double-pushout approach [3] for network and local rules, a connection condition, a network condition and a splitting condition.

The connection condition states that local nodes and edges are not allowed to be deleted, if they are exported. Moreover, exported nodes and edges are not allowed to be removed from the interface, if they are imported remotely. Analogously, export graphs are only allowed to be extended by new nodes and edges if they exist locally and import graphs must not contain nodes and edges not imported from a remote export graph.

To ensure the network condition the finishing (initial) local state graph of a network node to be deleted (created) has to be completely determined within the distributed rule. Moreover, to install (remove) a network edge, the source graph of its local morphism has to be completely determined, too, such that the whole morphism can be handled.

The splitting conditions states that the edge contexts of local graphs to be split have to be fully determined by the distributed rule.

Moreover, a set of *application conditions* may be appended to each rule 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. Compare figure 9, figure 11, figure 12, figure 19 and figure 20 for examples of negative application conditions (NACs) stated in the bottom of a rule.

Distributed graph transformation is performed by a number of simple graph transformations at 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. It may contain also a number of attribute computations using variable instantiations determined by the corresponding match. 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, since a distributed graph transformation step is formulated by a double-pushout in the category of distributed graphs and graph morphisms [20]. This coherence is extremely useful to design consistent data structure transformations in a distributed environment.

Distributed graph transformation supports modeling local components which specify their own processes. 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 a local action can define its own thread of control. Thus, threads of control are modeled here independently of objects and allow concurrency at the level of events. If distributed rule applica-

tions are independent of each others they may happen simultaneously or in either order, i.e. truly concurrently.

3.4. Integration into the change model

While application-internal states and behaviour can be described by distributed graph transformation at the local level, distributed graph transformation at the network level is well suited for dynamic change management activities (cf figure 4).

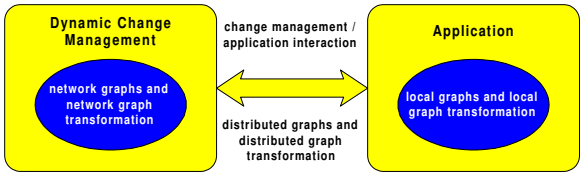

Figure 4 Usage of graph transformation abstraction levels within dynamic change management

Distributed graph transformation satisfies the requirements to dynamic change management postulated in [13]:

- *Changes are specified in terms of the system structure*: this is achieved by formulating the structural change rules as network level graph transformation rules.
- *Change specifications are declarative*: this is achieved by the graph transformation approach which uses rules to define a transformation.
- *Change specifications are independent of algorithms, protocols, and states of the application*: this is achieved by separating network and node level of the distributed system. The approach to distributed graph transformation ensures the necessary conditions for consistent interaction.
- *Changes leave the system in a consistent state*: this is achieved by formulating the transformation rules in the way the notion of system consistency requires.
- *Changes minimize the disruption to the application system*: this is achieved by the rule-based approach; i.e. the left-hand sides of the rules precisely delineates the involved area.

In the next section we will model a sample system well-known in the world of dynamic change management by distributed graph transformation.

4. A sample system: distributed ring database

In section 4.1 we will introduce the distributed ring database as a sample application. Then we will sketch how this sample system can be specified using distributed graph transformation in section 4.2. Finally we will discuss how dynamic accommodation of evolutionary change can be realized with distributed graph transformation in section 4.3.

4.1. Application context

We show the applicability of distributed graph transformation as a natural formalism for realizing dynamic change management by modeling the ring database example introduced by J. Kramer and J. Magee [12]. Let us consider a ring of connected replicated databases where each database can autonomously initiate an update of its containing data. This update is then propagated around the ring.

Copy inconsistencies while different update requests are circulating in the ring simultaneously are tolerated. However, to ensure long-term consistent updates in the presence of node autonomy and concurrency we impose as a requirement to system integrity identity of all databases' states when no updates are circulating and no database is locally initiating an update. Thus all database copies are consistent when the entire ring is quiescent.

In the next section we will sketch how a simplified distributed ring database can be specified with distributed graph transformation.

4.2. Specification of a sample ring database

Let us consider a sample distributed ring database comprising three interconnected databases storing the data of a distributed software development environment. Figure 5 sketches the network level of a distributed graph representing the ring: it comprises three local structures for the distributed databases DB_A, DB_B, and DB_C. Each local structure contains a network node representing the local state of the database (body) and two network nodes for its interfaces (export and import). The interface nodes are connected with the body node via local network edges and the entire local structures are interconnected via distributed network edges between their interface nodes. In the following we will assume that node label strings (like those depicted in figure 5) are comprised of a name variable (e.g., DB_A) and a node type (*.e* for export graphs, *.i* for import graphs and none for body graphs).

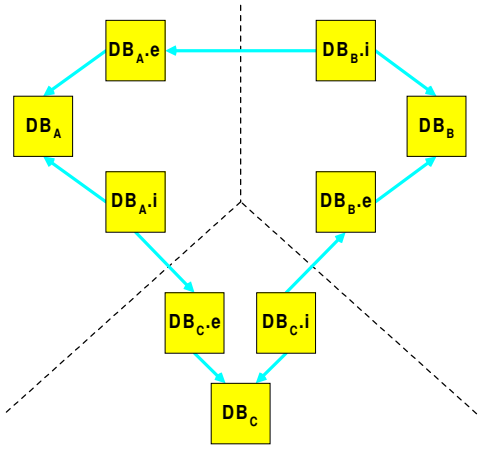

Figure 5 Ring database network graph

For each node at the network level there exists a corresponding graph at the local level and for each network edge there exists a corresponding graph morphism at the local level. The body nodes' local graphs represent the distributed copies of the database and changes to a database's state resulting from updates are propagated over its interfaces.

Now let us consider a simplified sample update propagation assuming Entity-Relationship (ER) diagrams as the data structure. Within this example ER diagrams serve as a data representation which intuitively demonstrates the possibilities of our approach. Of course not all data in a distributed system have to be modeled as graphs, for example only data involved in communication processes may have a graph representation. Also, graphs may be labelled with complex attributes like functions or relations [20, 7] but this is beyond the scope of this paper. The initial state of all replicated databases comprises the entities *professor*, *lec-*

ture, and *master thesis*, as well as the relations *gives* and *supervises* (cf. figure 6). In figure 6 and in the following distributed graphs export and import graphs are not explicitly represented. Only local body graphs are sketched; body graph elements depicted in a lighter color denote that this subgraph is imported or exported assuming a corresponding local graph morphism mapping between nodes with identical names. Figure 6 shows the databases DB_{client} and DB_{server} in a quiescent state, export and import graphs as well as the local graph morphisms and the distributed graph morphism are empty. For demonstrating the effects of the update propagation clearly the following distributed graphs representing database states within a sample update propagation (figure 6, figure 7, figure 10, figure 13, figure 14) depict global views on the distributed system. For depicting a local view wrt. a database only the interface graphs of involved predecessor and successor databases are shown, i.e. their internal body graphs are veiled.

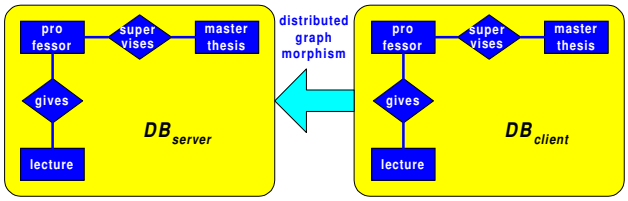

Figure 6 Update propagation (initial state)

Now within database DB_{server} a local update is initiated: three update control propagation nodes are inserted, an additional entity *student*, the relations *is visited by* and *is done by* connecting *student* with the entities *professor* and *master thesis* are added, and the relation *gives* is deleted (cf. figure 7). This change is achieved by applying the following local management rules depicted in figure 8 and figure 9:

- *add control nodes ("DB_{server}", "DB_{server}"),*
- *add isolated entity ("DB_{server}", "student"),*
- *add relation ("DB_{server}", "master thesis", "student", "is done by"),*
- *add relation ("DB_{server}", "lecture", "student", "is visited by"),*
- *delete relation ("DB_{server}", "gives").*

For controlling an update propagation we introduce the nodes *originator*, *insertion*, and *deletion* representing the originator of an update and information about ER structures to be deleted and created. This representation allows us to handle many update requests simultaneously via one communication connection.

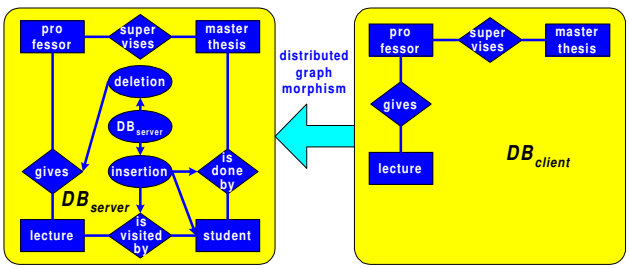

Figure 7 Update propagation (update introduced in originator)

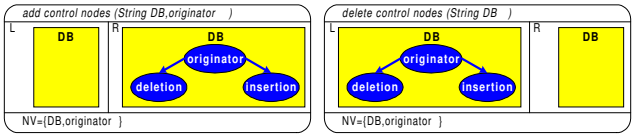

Figure 8 Local control management rules

The next step is to publish the update in DB_{server}'s export graph by copying the update propagation nodes as well as the ER structure connected with them (including possible interface entities of copied relations). In figure 10 exported graph elements of DB_{server} are depicted in a lighter color assuming a local graph morphisms from DB_{server}'s export graph into its body graph mapping between nodes with identical names.

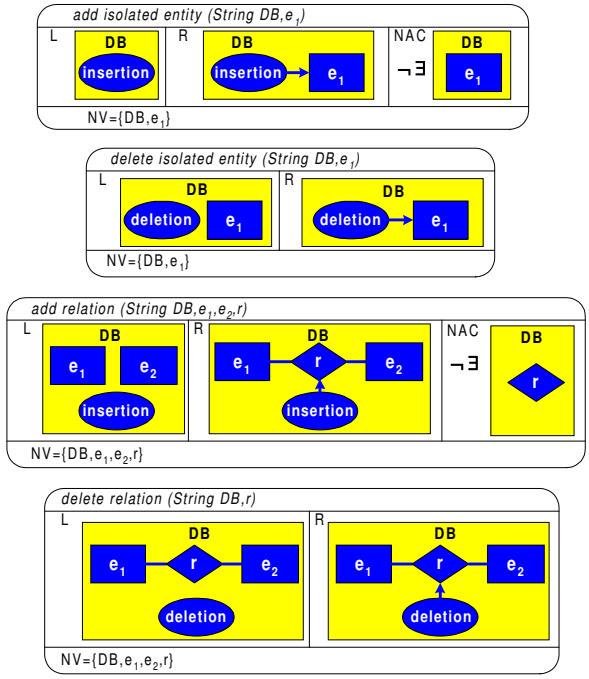

Figure 9 Local ER management rules

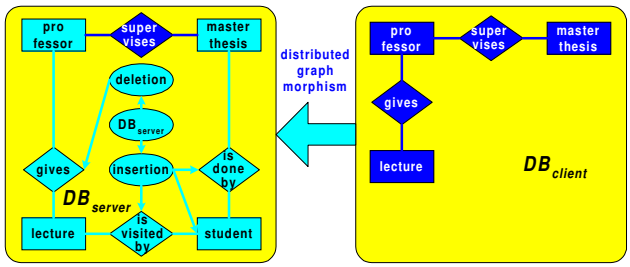

Figure 10 Update propagation (update exported)

The update export is done by the following control and ER copy rules (cf. figure 11 and figure 12):

- *copy control nodes ("DB_{server}", "$DB_{server}e$", "DB_{server}"),*
- *copy isolated entity ("DB_{server}", "$DB_{server}e$", "student") (assuming variable u is an instance of an enumeration type comprising the strings 'insertion', 'deletion'),*
- *copy interface entity ("DB_{server}", "$DB_{server}e$", "master thesis"),*
- *copy relation ("DB_{server}", "$DB_{server}e$", "is done by"),*
- *copy interface entity ("DB_{server}", "$DB_{server}e$", "lecture"),*
- *copy relation ("DB_{server}", "$DB_{server}e$", "is visited by"),*
- *copy interface entity ("DB_{server}", "$DB_{server}e$","professor"),*
- *copy relation ("DB_{server}", "$DB_{server}e$", "gives").*

Now a distributed use relation from DB_{client}'s import to DB_{server}'s export has to be established (cf. figure 13). As we have presumed synchronous communication channels (cf. section 4.3 and [12]), the application of the update to DB_{client}'s body graph has to be done at the same time as the update is imported. Thus DB_{server}'s export graph is copied to DB_{client}'s import and body graph using control copy

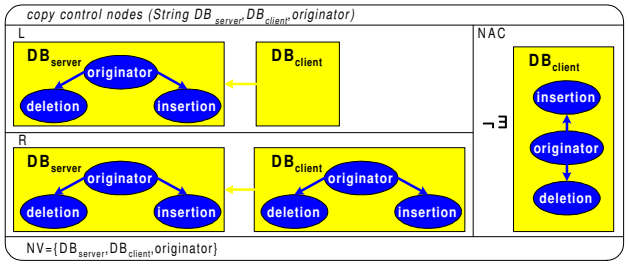

Figure 11 Control copy rule

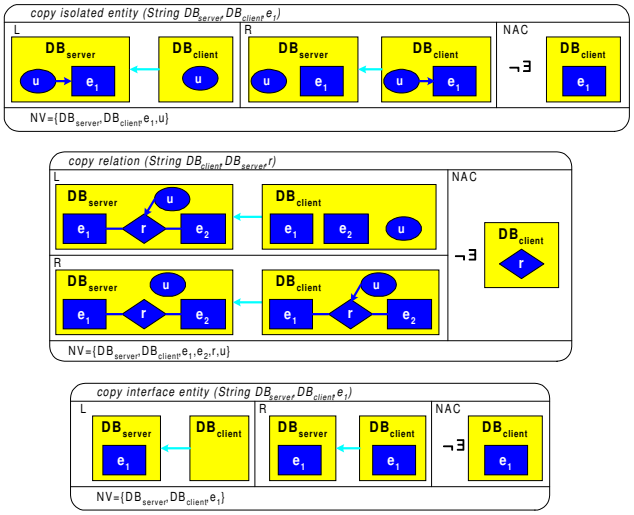

Figure 12 ER copy rules

rules (cf. figure 11) and ER copy rules (cf. figure 12) with + denoting simultaneity:

- *copy control nodes ("DB_{server}e", "DB_{client}.i", "DB_{server}") +*
  copy control nodes ("DB_{client}.i", "DB_{client}", "DB_{server}"),
- *copy isolated entity ("DB_{server}e", "DB_{client}.i", "student") +*
  copy isolated entity ("DB_{client}.i", "DB_{client}", "student"),
- *copy interface entity ("DB_{server}e", "DB_{client}.i", "master thesis") +*
  copy interface entity ("DB_{client}.i", "DB_{client}", "master thesis"),
- *copy relation ("DB_{server}e", "DB_{client}.i", "is done by") +*
  copy relation ("DB_{client}.i", "DB_{client}", "is done by"),
- *copy interface entity ("DB_{server}e", "DB_{client}.i", "lecture") +*
  copy interface entity ("DB_{client}.i", "DB_{client}", "lecture"),
- *copy relation ("DB_{server}e", "DB_{client}.i", "is visited by") +*
  copy relation ("DB_{client}.i", "DB_{client}", "is visited by"),
- *copy interface entity ("DB_{server}e", "DB_{client}.i", "professor")+*
  copy interface entity ("DB_{client}.i", "DB_{client}", "professor"),
- *copy relation ("DB_{server}e", "DB_{client}.i", "gives") +*
  copy relation ("DB_{client}.i", "DB_{client}", "gives").

Note that in figure 13 the lighter gray graph elements depicted in DB_{server} denote exported graph elements while the darker gray graph elements in DB_{client} denote import graph elements.

Further, a distributed graph morphism from DB_{client}'s import graph to DB_{server}'s export graph has to be established (assuming a mapping between nodes with identical names).

Then DB_{client}'s import graph and DB_{server}'s export graph are emptied. Finally, the update is performed in DB_{server}'s body graph and the control nodes are deleted (cf. figure 14). This is done with the help of the update perform

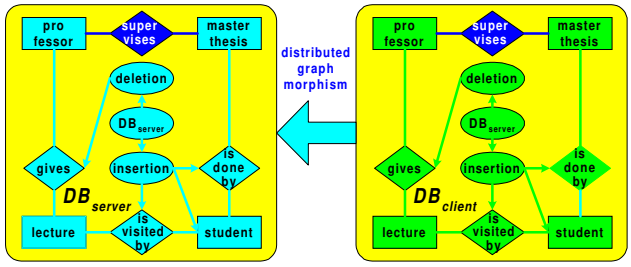

Figure 13 Update propagation (update imported and introduced to client data base)

rules depicted in figure 15 and the clean up rules sketched in figure 16:

- *perform relation insertion ("DB_{client}.i", "is done by"),*
  perform relation insertion ("DB_{client}.i", "is visited by"),
  perform entity insertion ("DB_{client}.i", "student"),
  perform relation deletion ("DB_{client}.i", "gives")
  clean up relation ("DB_{client}.i", "is done by"),
  clean up relation ("DB_{client}.i", "is visited by"),
  clean up entity ("DB_{client}.i", "student"),
  delete control nodes ("DB_{client}.i"),
- *perform relation insertion ("DB_{server}e", "is done by"),*
  perform relation insertion ("DB_{server}e", "is visited by"),
  perform entity insertion ("DB_{server}e", "student"),
  perform relation deletion ("DB_{server}e", "gives")
  clean up relation ("DB_{server}e", "is done by"),
  clean up relation ("DB_{server}e", "is visited by"),
  clean up entity ("DB_{server}e", "student"),
  delete control nodes ("DB_{server}e"),
- *perform relation insertion ("DB_{server}", "is done by"),*
  perform relation insertion ("DB_{server}", "is visited by"),
  perform entity insertion ("DB_{server}", "student"),
  perform relation deletion ("DB_{server}", "gives"),
  delete control nodes ("DB_{server}").

Again, the update is copied into DB_{client}'s export graph. In figure 14 the lighter color graph elements of DB_{client} now denote an exported structure.

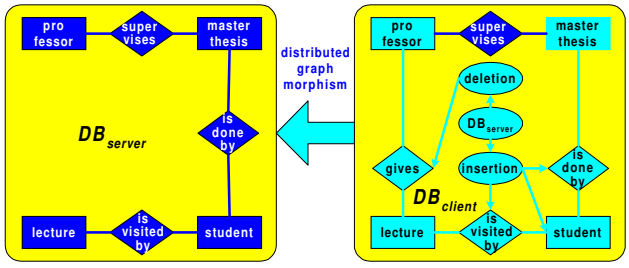

Figure 14 Update propagation (update performed in DB_{server} and exported again from DB_{client})

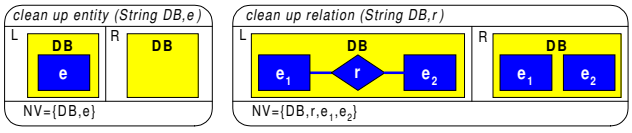

Figure 16 Clean up rules

This update propagation cycle is terminated when a database recognizes itself as the update originator. Then, its predecessor's export structure is directly deleted without importing the update. Figure 17 shows the termination rule which has to be applied by all databases before importing an update.

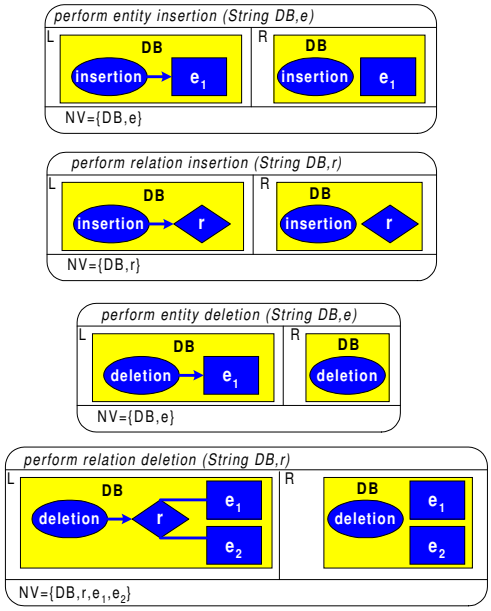

Figure 15 Update perform rules

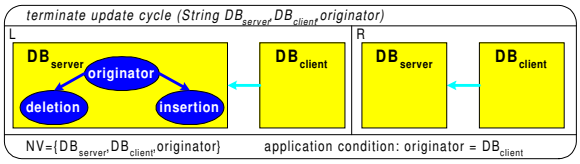

Figure 17 Termination of update propagation

4.3. Dynamic change within the ring database: automated database inclusion and removal at run-time

While in the last section we have sketched how the ring database example can be described with distributed graph transformation, we now discuss how dynamic accommodation of evolutionary changes to the ring's structure can be modeled, i.e. automated inclusion and removal of databases at run-time. Thus distributed graph transformation can be regarded as a natural underlying formalism to realize configurable distributed systems where modifications not envisable at design-time can be handled at run-time while those system parts not affected by the change are not disturbed in their operation.

Following the change model presented in chapter 2 changes to a node - i.e. the deletion or creation of a replicated database - are only permitted when the node is quiescent. As we assume that channels accept/transfer update messages from/to the nodes sychronously and do not loose any messages node quiescence is given when the node is passive: the database is not updating its data. In this passive state change management can delete a node, or if two nodes are in a passive state, a new node can be inserted between them.

We define node passivity as the state in which a database is not updating its data - it is not locally initiating an update nor it is servicing an update request originating from another database of the ring. Thus a node is in a passive state if no graph rewrite rules changing the data representation graph are currently performed - may them originate from the local database or be the update imported from the predecessor database. This means that neither a databases import graph must contain update propagation control nodes (the database is servicing an update request) nor its body and export graphs must contain control nodes (the database is initiating an update). Figure 18 shows quiescence management rules for setting and removing a *not quiet* flag in a database's body and export graph. The *not quiet* flag is included into a database's export graph so that the change management rules (cf. figure 19, figure 20) do not need to inspect a local structure's body graph to determine its quiescence status.

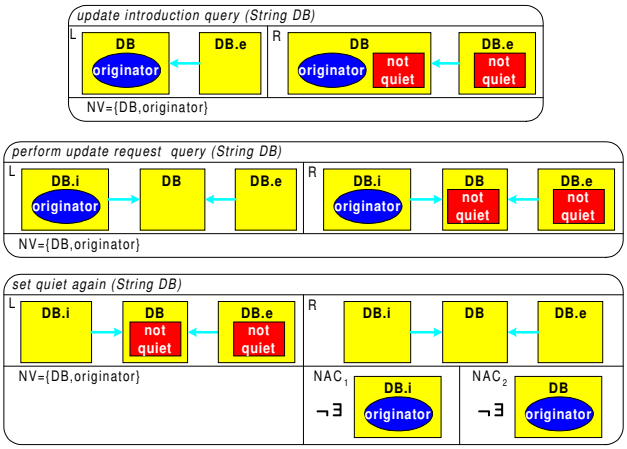

Figure 18 Quiescence management rules

Until now we have sketched the application's graph rewrite rules. Now we investigate how graph rewrite rules of the change management look like. Figure 19 sketches a network rule for deleting a database *DB*. The application condition presupposes that a *not quiet* flag does not exist for *DB*, i.e. the database is not within a data update transaction. As we have assumed synchronous transfer of update messages, there is no need for connected components to also be passive [12] (please note that if an update initiated by a database just deleted is still circulating within the ring, the originator label has to be set to the successor database).

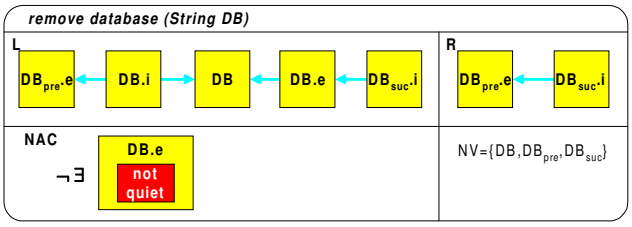

Figure 19 Change management rule for removing databases

The creation of a new database DB between two existing databases DB_{pre} and DB_{suc} is done within four steps:

1. First the entire database graph of DB_{pre} is copied to DB_{pre}*'s* export graph with the help of copy rules similar to those depicted in figure 12 (only without including control nodes). Also, the appropriate local graph morphism from DB_{pre}*'s* export graph into DB_{pre}*'s* body graph is established.
2. The network rule sketched in figure 20 is applied to the ring's network graph presupposing DB_{pre} and DB_{suc} are in a quiescent state (i.e., DB_{pre} must not be exporting and DB_{suc} must not be importing an update [13]). The dotted arrows in figure 20 denote the copying of network nodes as introduced in section 3.2 and figure 3.
3. Finally, DB_{pre}*'s* export graph and *DB's* import graph are emptied using the update perform rules (cf. figure 15) and the clean up rules (cf. figure 16). Also the corresponding graph morphism is emptied.

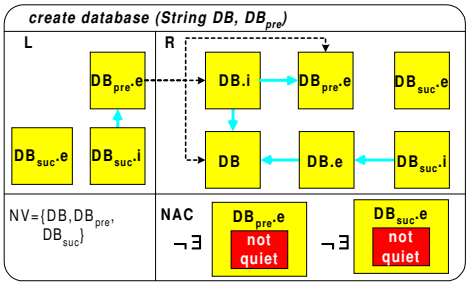

Figure 20 Change management rule for creating databases

5. Conclusion and further work

In this contribution we have introduced our approach to distributed graph transformation applied to the problem of dynamic change management in distributed systems. Distributed graph transformation nicely satisfies all the requirements postulated in [13]. Moreover, by using graph transformation coherently at both levels - the network level and the node level - more detailed and specialized change actions can be described. This is done while the encapsulation provided by the nodes is preserved using only the node's interfaces. The chosen example the ring-database storing ER-diagrams shows the natural way to express the change actions at the two levels.

This is ongoing work and we will improve our notions through implementation and more examples. One aspect - the interaction of change actions at the network level and the node level - leaves still room for improvement. Instead of providing related interfaces for this purpose the necessary mapping of local node state to the network level could also be achieved by specialized morphisms.

As stated in chapter 2, consistency is a very important aspect in dynamic change management. 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 [8]. 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 current AGG - machine [21]. Of course, the implementation will also make use of a distributed computing system. In a first approach, we plan to support the definition of import and export interfaces for graphs and rules realized as substructures of those. The main attention lays on the synchronization of local transformations related by subtransformations. The communication will be realized by remote method invocations (RMI) supported by Java.

6. References

[1] Agnew, B., Hofmeister, C.R., and Purtilo, J.M., "Planning for Change: A Reconfiguration Language for Distributed Systems", *Proc. 2nd Int. Conference on Distributed Computing Systems*, IEEE Computer Society Press, Pittsburgh, U.S.A., 1994.

[2] Berry, G. and Boudol, G., "The chemical abstract machine", *Theoretical Computer Science*, (96) p. 217-248, 1992.

[3] Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M., "Algebraic Approaches to Graph Transformation", in Rozenberg, G. (ed.), *Handbook of Graph Grammrs and Computing by Graph Transformation*, Vol. 1 Foundations, pp 163 - 245, World Scientific, Singapore, 1997.

[4] Degano,P. and Montanari, U., "A model of distributed systems based on graph rewriting", *Journal of the ACM*, 34(2), 1987.

[5] Fischer, I., Koch, M., and Volle, V., "Modeling Distributed Configuration Management by Distributed Attributed Graph Transformation", *to appear as technical report at university of Erlangen*, Germany, 1998.

[6] Fossa, H. and Sloman, M., "Implementing Interactive Configuration Management for Distributed Systems", *Proc. of the 3rd International Conference on Configurable Distributed Systems*, IEEE Computer Society Press, Annapolis, U.S.A, pp. 44-51, 1996.

[7] Goedicke, M. and Meyer, T., "Dynamic Semantics Negotiation in Distributed and Evlving CORBA Systems: Towards Semantics-Directed System Configuration", *Proc. 4th ICCDS*, Annapolis, USA, 1998.

[8] Heckel, R. and Wagner, A., "Ensuring Consistency of Conditional Graph Grammars -- A constructive Approach", *Proc. of SEGRAGR'95 'Graph Rewriting and Computation'*, Electronic Notes of TCS, Vol. 2, http://www.elsevier.nl/locate/entcs/volume2.html, 1995.

[9] Janssens, D. and Rozenberg, G., "Structured Transformations and Computation Graphs for Actor Grammars", in Ehrig et al., *Proc. 4th International Workshop on Graph Grammars and Their Application to Computer Science*, pages 446--460, Lecture Notes in Computer Science 532, Springer, 1991.

[10] Kaplan, S., Loyall, J., and Goering, S., "Specifying Concurrent Languages and Systems with Δ-Grammars", in Ehrig et al., *Proc. 4th International Workshop on Graph Grammars and Their Application to Computer Science*, pages 475--489, Lecture Notes in Computer Science 532, Springer, 1991.

[11] Korff, M., "Single Pushout Transformations of Equationally Defined Graph Structures with Applications to Actor Systems", in *Proc. Graph Grammar Workshop Dagstuhl 93*, Lecture Notes in Computer Science 776, pages 234--247, Springer, 1994.

[12] Kramer, J. and Magee, J., "Analysing Dynamic Change in Software Architectures: A Case Study", *Proc. 4th ICCDS*, Annapolis, USA, 1998.

[13] Kramer, J. and Magee, J., "The Evolving Philosophers Problem: Dynamic Change Management", *IEEE Transactions on Software Engineering*, SE-16, 11, pp. 1293-1306, 1990.

[14] Magee, J. and Kramer, J., "Self Organizing Software Architectures", *Proc. SIGSOFT Workshops*, ACM Press, pp.35-38,1996.

[15] Magee, J. and Kramer, J., "Dynamic Structure in Software Architectures", *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, San Francisco, U.S.A., 1996.

[16] Magee, J., Dulay, N., and Kramer, J., "Regis: A Constructive Development Environment for Distributed Programs", *IEE/IOP/BCS Distributed Systems Engineering Journal 1(5)*,304-312,1994.

[17] Métayer, D.L., "Software architecture styles as graph grammars", *Proc. 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, p. 15-23, ACM Press, 1996.

[18] Schied, G., "Über Graphgrammatiken, eine Spezifikationsmethode für Programmiersprachen und verteilte Regelsysteme", *Arbeitsberichte des Institus für math. Maschinen und Datenverarbeitung (Informatik)*, University of Erlangen, 1992, *in german*.

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

[20] Taentzer, G., Fischer, I., , Koch, M., and Volle, V., "Distributed Graph Transformation with Application to Visual Design of Distributed Systems", *to appear as article in Graph Grammar Handbook 3 Concurrency and Distribution*, 1998.

[21] Taentzer, G., Ermel, C., and Rudolf, C., "AGG-Approach: Language and Tool Environment", *to appear as article in graph grammar handbook 2: Specification and Programming*, 1998.

[22] Wermelinger, M., "Towards a Chemical Model for Software Architecture Reconfiguration", *Proc. 4th ICCDS*, Annapolis, USA, 1998.