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

Gabriele Taentzer[1], Michael Goedicke[2], and Torsten Meyer[2]

[1] Computer Science Department
Technical University of Berlin, D-10587 Berlin
Germany
gabi@cs-tu-berlin.de

[2] Dept. of Mathematics and Computer Science
University of Essen, D - 45117 Essen
Germany
{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 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 pursued, for example, in [4], by Δ-grammars in [11], in [19] and by actor graph grammars in [10, 12].

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 [20]. Distributed graph transformation is extended by an attribution concept for graphs and the concept of application conditions for graph rules in [21]. 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 [14] 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 [15] self-configuring architectures are investigated and in [7, 16] a graphical user interface and a notation for component configurations based on the architecture description language REGIS/DARWIN [17] 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 [8]. The ring database example modeled in this paper was first introduced in the world of dynamic change using DARWIN in [13]. 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 selforganizing software architectures is given in [23]. With the Chemical Abstract Machine [2] an architecture specification and programming model is presented and for covering dynamic issues graph rewrite rules [18] 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. We will show the applicability of our approach by modeling a well-known example in the world of dynamic change in chapter 4 and 5.

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. We follow the model developed in [13, 14]. This model implies a strict separation of the two change aspects mentioned above.

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 [14]. 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 to allow the execution 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.

Below we now introduce the technique used to realize dynamic change management: distributed graph transformation.

3 Distributed graph transformation

In the following, we introduce a recent specification technique, called *distributed graph transformation*, which 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. 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. The concepts as well as the formal definition of distributed graph transformation are

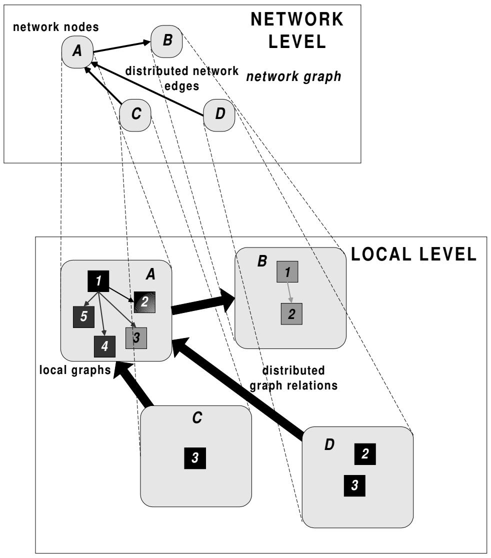

Fig. 1. This example of a distributed graph shows a local view of the system A. In this and in the following distributed graphs body elements of local graphs are depicted in medium gray, exported graph elements are depicted in light gray, imported graph elements are depicted in **black**, and common parameters graph elements (i.e. graph elements which are imported and exported) are depicted in mixed light gray / **black**. Relations between local graph elements are not explicitly depicted, but we assume a mapping between graph elements with identical labels.

based on the *double-pushout approach* to graph transformation [3] where basic concepts from *category theory* are applied. Distributed graph transformation is introduced in [20] and further developed in [21].

The topological structure of a distributed system's network is presented graphically by a *network graph*. Nodes of the network graph represent any kind of processing unit with some memory, edges indicate communication paths between nodes. The combination of the network graph structure and the local object structures is specified by distributed graphs in order to describe distributed object structures. A *distributed graph* consists of one *network graph* where each network node is equipped with a *local graph* describing a locally used data and object structures. Each network edge is equipped with a relation on local graphs defining replicated data and object structures in that way.

In the early stages of system development, a global view on the entire distributed system is desirable to get an overview. Once the network structure is fixed, the system developers are supposed to take a local view on network nodes and local system parts running on them to facilitate concurrent development. The *local view* (cf. [5]) of a local system contains this local system together with remote import and export interfaces to which it may have connections. Communication between local systems takes place via export and import interfaces. In *export interfaces*, local systems present data and objects accessible for other

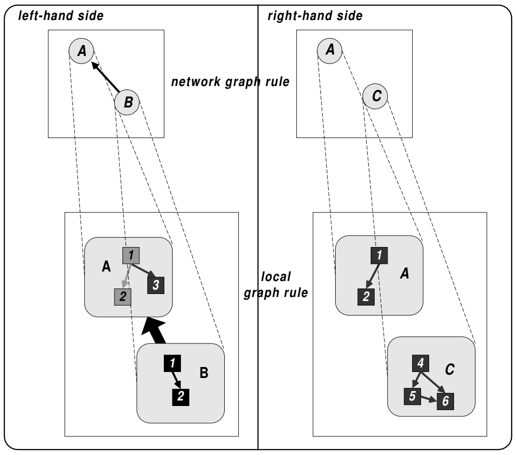

Fig. 2. In this example of a distributed rule the network node B, its local graph as well as the network edge and its local graph relation are deleted. The local graph determines the state in which such a network node is allowed to be deleted. The network node A is preserved. Its local graph has to be manipulated by the local rule given. On the righthand side a new network node C has to be created and its initial state is given by its local graph.

local systems and *import interfaces* contain data and objects from remote export interfaces. Network nodes being the source of a network edge may be interpreted as interfaces. Figure 1 shows an example of a distributed graph where import and export graphs are distinguished by different gray shades of local graph elements.

Each node and edge 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 [21, 6] for more details wrt. attribution of distributed graphs).

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. Furthermore, remote interactions can be described by distributed rules. In this case, all network nodes where parts of the interaction should take place, are incorporated into the network graphs of the rule. 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. See figure 2 for a schematic representation of a distributed rule. In the following, rules are presented by their graphs, the relation between left and righthand side are installed unambiguously obeying compatibility of source, target, type and attribute mappings.

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

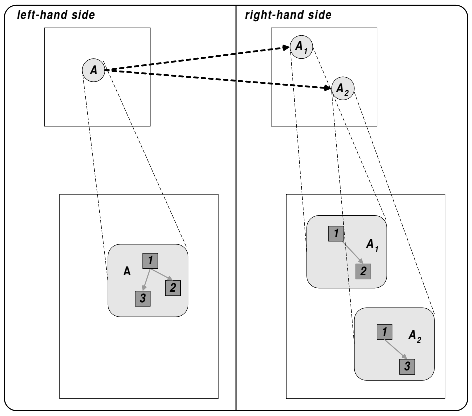

Fig. 3. Example of a slitting and copying distributed rule. The local node 1 in the lefthand side of the rule is copied into both local graphs on the righthand side whereas the local nodes 2 and 3 as well as the local edges are split into the two local graphs on the righthand side.

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.

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.

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 replication relation becomes partial after application. Moreover, the application of distributed rules should not produce implicit sideeffects 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*, a *splitting condition*, and a set of *negative application conditions* (NACs). A detailed description of these application condition can be found in [3, 20, 21]. Compare figures 7, 10, and 11 for examples of NACs stated in the bottom of a rule.

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.

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

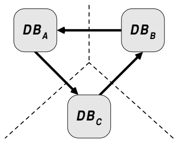

Fig. 4. Ring database network graph. The distributed network edges indicate the direction of the distributed graph relations, i.e. DB_B imports an update which is exported by DB_A, etc.

4 A sample system: distributed ring database

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 [13]. 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 longterm 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.

Let us consider a sample distributed ring database comprising three interconnected databases storing the data of a distributed software development environment. Figure 4 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 is represented by a network node the local graph of which contains graph elements describing the local state of the database (body) as well as graph elements describing its interfaces (export and import). Entire local structures are interconnected via distributed network edges between them. In the following we will assume that node label strings (like those depicted in figure 4) are comprised of a name variables (e.g., DB_A).

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 relation between graphs at the local level. The body graph elements in a network node's local graph 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, if it is not appropriate to model the entire distributed system with graphical means, it is possible to abstract from local states by describing them only as attributes. For example, only data involved with communication processes may have a graph representation. On the other hand, graphs may be labeled by complex attributes like functions or relations [21, 8] but this is beyond the scope of this paper. The initial state of all replicated

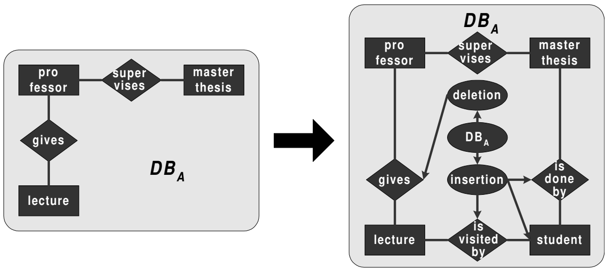

Fig. 5. The left part of the figure depicts the initial database state of the update propagation, the right part shows an update introduced in the originator (local view of DB_A).

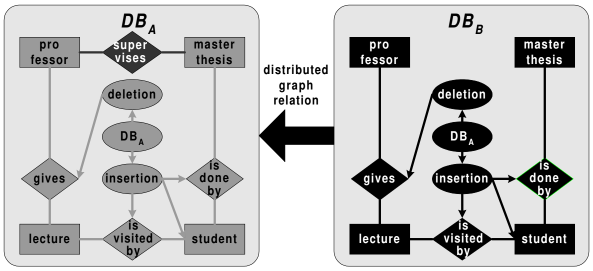

Fig. 6. The update is exported by the originator and imported by the successor database (local view of DB_A). Wrt. the distributed graph relation a mapping between graph elements with identical labels is assumed.

databases comprises the entities *professor*, *lecture*, and *master thesis*, as well as the relations *gives* and *supervises*. The left part of figure 5 shows the local view of the database DB_A in a quiescent state, there exist no export or import graph elements as well as no distributed graph relations to other databases.

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. Now within database DB_A a local update is initiated: three update propagation control nodes are inserted, an additional entity *student*, the relations *is visited by* and *is done by* connecting *student* with the entities *lecture* and *master thesis* are added, and the relation *gives* is deleted (cf. the right part of figure 5). The rules by which the update is introduced are showed in [23] in detail; their application is represented by the middle arrow in figure 5.

The next step is to publish in DB_A the update propagation nodes as well as the ER structure connected with them (including possible interface entities of copied relations, cf. the left part of figure 6). A sample rule for exporting entities is depicted in figure 7, the complete set of rules for exporting entities, relations, and update propagation control nodes are presented in [23].

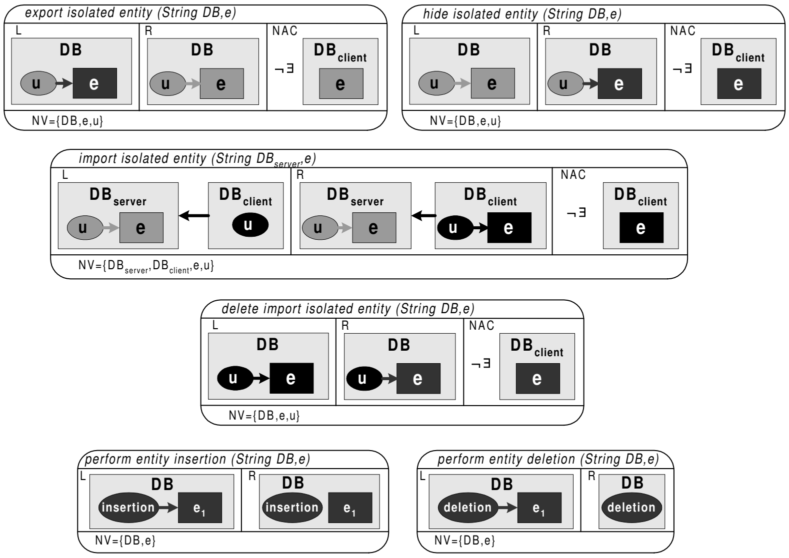

Fig. 7. Rules for exporting and hiding entities, for importing entities and deleting import relations, and for performing entity insertions and deletions (local view of DB).

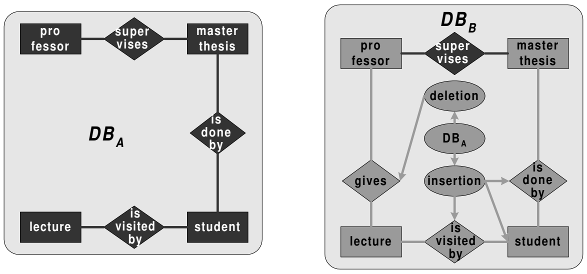

Fig. 8. The update is finally performed in database DB_A (local view of DB_A) and then exported from DB_B (local view of DB_B).

Now a distributed use relation from DB_B's import to DB_A's export has to be established (assuming a mapping between nodes with identical names). As we have presumed synchronous communication channels (cf. above and [13]), the application of the update to DB_B's body graph has to be done at the same time as the update is imported (cf. figure 6). Again, a sample rule for importing entities is shown in figure 7 and further import rules can be found in [23].

Then DB_B's import graph elements and DB_A's export graph elements are deleted (i.e. interfaces are emptied). Finally, the update is performed in DB_A's body graph and the control nodes are deleted (cf. figure 8). The rules by which this is done are depicted in [23], sample rules for deleting export / import entities, as well as sample update perform rules are sketched in figure 7.

The update propagation is continued by exporting the update again from DB_B in the same way as sketched above for DB_A (cf. figure 8). This cycle is

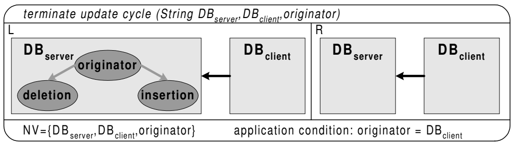

Fig. 9. Rule for terminating a update propagation (local view of DB_{client}).

terminated when a database recognizes itself as the update originator. Then, its predecessor's export structure is directly deleted without importing the update. Figure 9 shows the termination rule which has to be applied by all databases before importing an update.

5 Dynamic change within the ring database: automated database inclusion and removal at runtime

While in the last chapter we have sketched how the ring database example can be described by 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 runtime. Thus distributed graph transformation can be regarded as a natural underlying formalism to realize configurable distributed systems where modifications not envisaged at designtime can be handled at runtime 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 *only* the graph rewrite rules for adding control nodes or importing control nodes [23] are applicable, i.e. the only allowed operation is initiating or importing an update. This means that neither a database imports update propagation control nodes (the database is servicing an update request) nor its body contains control nodes (the database is initiating an update). Quiescence management rules [23] set a *not quiet flag*, if body or import *originator* nodes exist in a database and delete the flag if such nodes do no longer exist. The *not quiet* flag is modeled as an export node so that the change management rules (cf. figures 10 and 11) do not need to inspect a database's body graph to determine its quiescence status.

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 10 sketches a network rule for deleting a database DB. The application condition

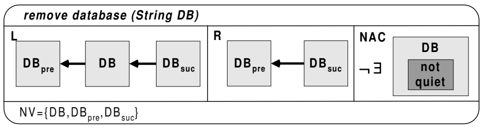

Fig. 10. Change management rule for removing databases (local view of database DB).

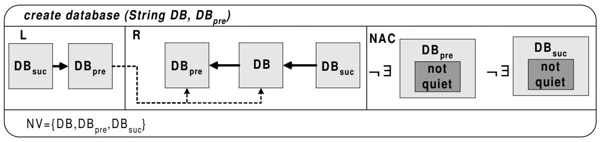

Fig. 11. Change management rule for creating databases (local view of database DB).

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 [13] (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).

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

1. First the entire body database graph of DB_{pre} is exported by the help of export rules [23]. This is done because change management rules must not access the local body graph of a database.
2. The network rule sketched in figure 11 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 [14]). The dotted arrows in figure 11 denote the copying of network nodes as introduced in chapter 2 and figure 3.
3. Finally, DB_{pre}'s export and DB's import graph elements are deleted (i.e. set transferred back to body graph elements) using rules for deleting export / import relations [23].

6 Conclusions 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 [14]:

- *changes are specified in terms of the system structure,*
- *change specifications are declarative,*
- *change specifications are independent of algorithms, protocols, and states of the application,*

– *changes leave the system in a consistent state,*
– *changes minimize the disruption to the application system.*

Moreover, by using graph transformation coherently at both levels - the network level and the local 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 [9]. 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 [22]. 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.

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., Taentzer, G.: Local Views on Distributed Systems and their Communication, TAGT 98, tr-ri-98-201 Dept. of Computer Science, University of Paderborn, Germany.
6. 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.

7. Fossa, H. and Sloman, M.: Implementing Interactive Configuration Management for Distributed Systems, Proc. of the 3rd ICCDS, IEEE Computer Society Press, Annapolis, U.S.A, pp. 44-51, 1996.
8. 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.
9. 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.
10. Janssens, D. and Rozenberg, G.: Structured Transformations and Computation Graphs for Actor Grammars, in Ehrig et al., Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science, p 446, LNCS 532, Springer, 1991.
11. Kaplan, S., Loyall, J., and Goering, S.: Specifying Concurrent Languages and Systems with Δ-Grammars, in Ehrig et al., Proc. 4th Intl. Workshop on Graph Grammars and Their Application to Computer Science, p 475, LNCS 532, Springer, 1991.
12. Korff, M.: Single Pushout Transformations of Equationally Defined Graph Structures with Applications to Actor Systems, in Proc. Graph Grammar Workshop Dagstuhl 93, LNCS 776, pp 234–247, Springer, 1994.
13. Kramer, J. and Magee, J.: Analysing Dynamic Change in Software Architectures: A Case Study, Proc. 4th ICCDS, Annapolis, USA, 1998.
14. Kramer, J. and Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management, IEEE TSE, SE-16, 11, pp. 1293-1306, 1990.
15. Magee, J. and Kramer, J.: Self Organizing Software Architectures, Proc. SIGSOFT Workshops, ACM Press, pp.35-38,1996.
16. Magee, J. and Kramer, J.:Dynamic Structure in Software Architectures, Proc. 4th ACM SIGSOFT Symp. Foundations of Software Engineering, ACM Press, San Francisco, U.S.A., 1996.
17. 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.
18. Métayer, D.L.: Software architecture styles as graph grammars, Proc. 4th ACM SIGSOFT Symp. Foundations of Software Engineering, p. 15-23, ACM Press, 1996.
19. Schied, G.: Über Graphgrammatiken, eine Spezifikationsmethode für Programmiersprachen und verteilte Regelsysteme, Tech. Rep. Institut für math. Maschinen und Datenverarbeitung (Informatik), University of Erlangen, 1992, in german.
20. Taentzer, G.: Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems, PhD thesis, TU Berlin, Shaker Verlag, 1996.
21. Taentzer, G., Fischer, I., Koch, M., and Volle, V.: Distributed Graph Transformation with Application to Visual Design of Distributed Systems, to appear in Graph Grammar Handbook 3 Concurrency and Distribution, 1998.
22. Taentzer, G., Ermel, C., and Rudolf, C.: AGG-Approach: Language and Tool Environment, to appear in Graph Grammar handbook 2: Specification and Programming, 1998.
23. Taentzer, G., Goedicke, M., and Meyer, T.: Dynamic Change Management by Distributed Graph Transformation: Towards Configurable Distributed Systems, TAGT 98, tr-ri-98-201 Dept. of Computer Science, University of Paderborn, Germany.
24. Wermelinger, M.: Towards a Chemical Model for Software Architecture Reconfiguration, Proc. 4th ICCDS, Annapolis, USA, 1998.