Visual Design of Distributed Object Systems by Graph Transformation*

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

August 13, 1998

Abstract

A new visual design technique for distributed object systems based on graph transformation is presented. It includes the graphical description of remote object interaction, object migration and replication, communication and synchronization as well as dynamic reconfiguration of distributed systems. The notation is built up on UML providing these concepts as a possible extension of UML. Using graph transformation as underlying formal framework, consistency in distributed systems can be ensured.

1 Introduction

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 based on graphical notations, but altogether distributed systems cannot be designed sufficiently with these techniques.

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 suitable interaction mechanism. 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.

Using graph transformation as underlying formal framework, distributed behavior is designed in a way that consistency of the network as well as of all object and data structures involved is ensured.

2 Design of Distributed Object Systems: Requirements and State-of-the-Art

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 applica-

*Research partially supported by the German Research Council (DFG), the TMR network GETGRATS, and the ESPRIT Basic Research Working Group APPLIGRAPH

tion components have to be modeled to manage the information exchange as well as the synchronization of accesses to resources commonly used. Communication can be synchronous or asynchronous, i.e. one component continues working immediately after sending a message.

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 parts and efficient implementations 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.

There are a number of formal specification techniques such as temporal logics [MP92], state charts [Har87], process algebras [Mil80], Petri nets [Rei87], and actor systems [Agh86] to model the concurrent behavior of distributed systems. The structural aspects of distributed systems are hardly tackled. Dynamic reconfiguration of a distributed system – an aspects which is preferably graphically explained – 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.

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. For the description of allocation of objects and tasks, object replication and migration, remote interactions and dynamic network topologies these techniques promise support, but, however, seem to be insufficient. The most promising language for design of distributed systems is UML. It offers the concept of deployment diagrams which captures the description of a system's network topology and the software components, processes and nodes that live on the network nodes. A missing notational elements to express replication of an object can be invented by a further stereotype which indicates this dependency. But there is no feature to describe the behavior on replicated object structures. Moreover, dynamic network reconfiguration is not addressed by UML.

3 Graph Transformation

Graph transformation means the rule-based manipulation of graphs. Graphs are typically used to describe the static part of a system. In this paper directed graphs are considered where all nodes and edges are *typed* and *attributed*. Typing a graph means that there is a so called *type graph* specifying node and edge types and their possible relations, i.e. which node types may be connected by which edge types (see Figure 1 for an example). A type graph can be read like an entity/relationship schemata. All graphs used in the system must obey this type graph. Attributes are added to nodes resp. edges that may be of any data type describable as e.g. integers, reals, strings, etc.

The dynamic part is formulated with graph rules consisting of a left-hand a right-hand side using an *if-then* style. *If* the left-hand side of the rule exists in a given graph, *then* the occurence of the left-hand side in the graph is deleted and instead the right-hand is inserted. The question is how this insertion takes place as there may be several possibilities how the right-hand side may be inserted into the "hole" and how it may be connected to the rest of the graph. If a connection of the right-hand side and the graph is desired, the rule indicates this by same items in the left-hand and the right-hand side. When deleting the left-hand side these items remain in the graph and serve as gluing points for the right-hand side. Additionally it must be taken care that no dangling edges, that are edges without source or target nodes, are produced through the rule application. This happens if a node is deleted without deleting its connected edges too. If this application condition is satisfied, i.e. all preserved edges are not dangling after deletion, the result of a rule application is again a graph. It is very useful to have rule graph items attributed with variables and computations on these variables. The variables of the left-hand side are instantiated with concrete values when the left-hand side is embedded into the graph

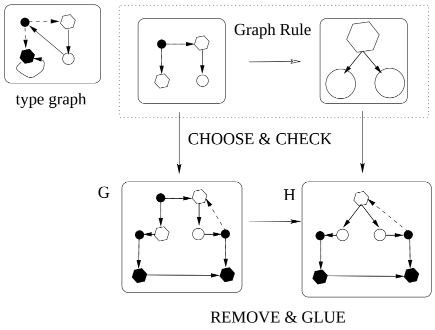

Figure 1: A type graph, a graph rewrite rule and its application to a graph

the rule is applied too. The new values of graph items inserted by the right-hand side are then calculated according to these variable instantiations.

Summarizing a rule application consists of the following steps: *choose* an occurrence of the left-hand side of the rule, *check* for dangling edges, *remove* this occurrence except of the preserved part (if checking was successfull), *glue* the right-hand side at the preserved part and *compute* new attribute values.

On the right in Figure 1 an example of a rule and its application is given. In the upper half of the picture the rule itself is shown. In the lower part its application to graph G with the result graph H are shown. Different node types are indicated by differnt shapes and colors, different edge types by diffrent kinds of lines. How the types can be connected shows the type graph on the left. The left-hand side of the rule is attributed with variables x, y, z, that are instantiated when embedding this side into the graph by values $x = 1, y = 3, z = 2$. With the help of these instantiations new labels on the right-hand side are calculated. Three nodes are preserved, the ones attributes with x, y, z resp. $x + y, x^y, x \cdot y$. They serve as anchors for edges connected to the picture of the left hand side in the graph the rule is applied to.

Graph transformation, i.e. the application of rules, defines a relation on graphs which may be iterated arbitrarily often yielding the transformation process. In this way, a set of graph transformation rules gets an operational semantics.

For the formal definition of graph transformation different approaches occur in the literature [Roz97]. In this article the double-pushout approach is used. The formal background of graph transformations as explained here can be found in [Ehr79, CMR+97].

4 Distributed Graph Transformation

Several approaches were proposed to use graph transformations as a distributed system model. One reason is due to its rule-baseness, that is well suited to describe reconfiguration in distributed systems, because it inherently models the asynchrony and non-determinism of reconfiguration control. All the approaches have the modeling of the network structure by a graph in common. However, local states are typically coded in some specification or programming text or are not considered at all. This idea is followed, for example, in [DM87], by Δ-grammars in [KLG91], in [Sch92] and by actor graph grammars in [JR91, Kor94].

But for application to distributed systems, graph transformation can also be employed twice: the known description of the dynamic reconfiguration of distributed systems on the network level and, in addition, to model evolving data and object structures in the local systems. The advantage of the usage of graph transformation on the local level is the providence of a description of local actions as well as remote interaction, such as object migration, replication, communication and synchronization. In order to be able to describe all these features the combination of the network graph transformation and local transformations is needed.

We propose *distributed graph transformation* (DGT) for fullfilling this aim by describing both the network level and the local level of a distributed system in one *distributed graph*. The concepts as well as the formal definition of distributed graph transformation based on the algebraic approach [CMR+97] are introduced in [Tae96, TFKV98]. A distributed graph can be seen as a hierarchic graph on two levels (cf. Figure 2).

Just as in the other approaches 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*. However, unlike in the other approaches, graph transformation is used on the local level to manipulate local data structures. So, distributed graphs can be nicely used to model a system's network and data structures

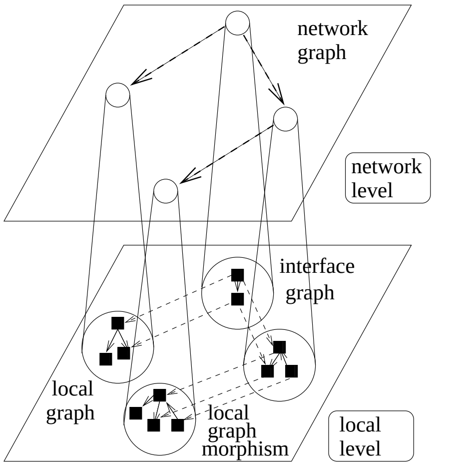

Figure 2: Example of a distributed graph

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.

A distributed rule consists of a network rule and local rules for all network nodes which are preserved by the network rule. Network als well as local rules are graph rules as described in section 3 and may contain a number of attribute computations. `Diese naechsten zwei Saetze versteh ich nicht` 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 the rule match.

The transformation of distributed graphs by *distributed rules* is then performed by a number of simple graph transformations via these network and local rules. All these transformations can be done in parallel. Network nodes as well as edges may be deleted and newly created as indicated by the rule. Newly created network nodes are equipped with a local graph, intended as the initial state of the local system. Deletion of network nodes is only possible if the whole local system is deleted as well.

See Figure 3 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

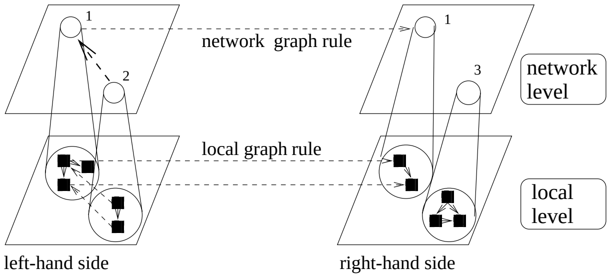

Figure 3: Example of a distributed rule

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 the local graph equipped.

Just as in the description of a rule application in section 3, for the application of a distributed rule in a distributed graph an occurence of the left-hand side has to be chosen as well. In addition to the dangling condition for all local rules and the network rule some more application conditions have to be checked in order to guarantee a distributed result graph after application of the distributed rule (these conditions are explained in more detail in [TFKV98]). The all local rules and the network rule are applied as shown in section 3.

A distributed graph transformation step is formulated by a double-pushout in the category of distributed graphs and graph morphisms [TFKV98]). This coherence is extremely useful to design consistent data structure transformations in a distributed environment. Altogether, 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 a local action 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. truely concurrently.

5 The Design Technique

In the following, we sketch the design technique at a number of key aspects typical for distributed systems. The underlying concept for this design technique is distributed graph transformation.

Since graphs and graph transformation is defined independently of graphical layouts, we adapt our examples presented below as far as possible to the graphical notation of UML [Rat97]. The graphical notation is built up on object and deployment diagrams. This notation is extended below to capture new concepts not available in UML.

In the following, we consider the static aspects of distributed systems first. These are the network and local structures in subsections 5.1 and 5.2 as well as their combination in 5.3. The dynamic aspects are handled in the following subsections. Whereas object interactions in one system component are considered in subsection 5.5, remote object interaction in serveral components and system reconfiguration possibilities are treated in subsections 5.6 and 5.7. Finally, local views on distributed systems are invented in subsection 5.2.

5.1 Network Graph Structure

It is usual to present the topological structure of a distributed system network graphically. In the simplest case, system components are depicted by nodes and component connections by edges between the components. Such *network graphs* may look like those in figure 4. In open systems, network structures are not static, but may evolve over time. It suggests itself to use graph transformation for this purpose. E.g. in [KLG91] client-server-systems and their modifications are defined by Δ-graph grammars. The problem of evolving philosophers or dynamic dining philosophers where philosophers arranged in a ring may be add or removed, is also specified there. Moreover, evolving philosophers are considered by Kramer and Magee in [KM90]. Consider figure 4 for sample network graphs modeling a client-server-system and the five philosopher ring. Network reconfiguration by graph transformation is explained in subsection 5.7 below.

5.2 Local Object Structures

To describe the static and dynamic aspects of object-oriented systems the well-known OOA/D techniques like OMT [RBP$^+$91], OOD [Boo94] and UML [Rat97] may be used. The intention is

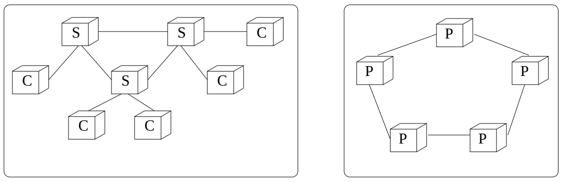

Figure 4: Sample network graphs

to use our design technique in combination with common OOA/D technique.

Class and object diagrams are the standard technique to define the static aspects of object structures. Both kinds of diagrams can be formulated as graphs where classes and objects are modeled by nodes whereas associations and links are defined as edges. Type graphs model some sort of class diagrams, object diagrams can be interpreted as typed graphs.

5.3 Distributed Object Structures

We use distributed graphs to describe distributed object structures. As for local structures, distributed type graphs and distributed state graphs are distinguished. In network nodes local structures are stored. The connection paths between network nodes are described by network edges. The associated local graph morphisms are used to describe replicated object structures.

The graphical notation of distributed object structures in this sense builds up on deployment diagrams. UML version 1.1 provides deployment diagrams to address the physical system topologies. A deployment diagram shows a system's network topology and the software components, processes and nodes that live on the network nodes.

A network consists of nodes representing any kind of processing unit with some memory. Similarly to other structural elements nodes may be typed. 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 arrow between those components.

Figure 5 shows an example of a deployment diagram where components as well as objects

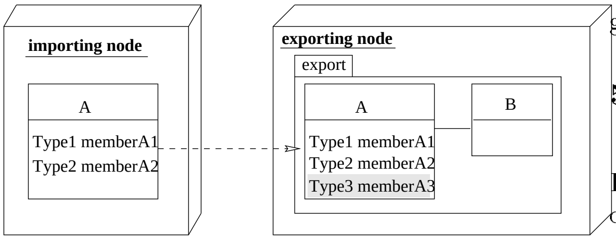

Figure 6: A sample import – export connection between type graphs

are located to nodes and dependencies between different structuring elements are shown.

In the following, we will set up on deployment diagrams to describe the allocation of objects, processes, etc. to network nodes, except of object migration which is modeled differently. This is actually a remote interaction and will be explained in subsection 5.6.

The notation of deployment diagrams is used in our context to describe the static aspects of some distributed system part. We extend this notation to capture, moreover, the information of object replication. If one object is the replication of another one, this is indicated by a dashed arrow between these two pointing to the origin.

Network nodes being the source of a network edge may be interpreted as interfaces. In the following, we distinguish import and export interfaces. Export interfaces offer public parts of local object structures whereas an import interface and the connection to its export interface shows an imported remote object structure. The design of import and export interfaces is handled by type graphs. They are used to define the object types that are required/allowed in principle.

Export type graphs contain all those object types and associations which may be exported. For each export object type it is allowed to export only a subset ofits members. Import type graphs specify all object types and associations to be imported from some export. For each import type graph a suitable export type graph has to be found where the import structures and types coincide with some of the export. Consider figure 6 for a sample import – export connection between type graphs where one of the two exported object types is imported, but memberA3 is private.

Distributed state graphs are used to describe some section of a distributed system state. Connections between import and export state graphs show the current state of replication.

5.4 Design of Dynamics in Distributed Systems

Distributed graph rules are useful to describe dynamic network reconfiguration by a network rule as well as object interaction by a set of local rules applied in each component. The local graph of a network node to be deleted determines the state in which such a network node is allowed to be deleted. On the right-hand side a new network node can be created and its initial state is given by the local graph equipped.

Moreover, distributed graph rules can be used to describe synchronization and communication in distributed systems.

5.5 Local Object Interaction

There are several techniques to describe the dynamic aspects in object-oriented systems, e.g. state charts, sequence and collaboration diagrams. We like to add a new one, i.e. graph transformation. The reason is that the rule-based way of dynamics specification is well suited to be used for specification of remote behaviour. If an object behaviour is described by *before* and *after* states, it is possible to extract public parts, i.e. substates, for exports. Export rules are designed exactly in this way, i.e. being a subrule.

Our proposal is to use graph transformation for all those object structures which may be communicated over the network. Other objects are either independent of these, members of these or include them as members in turn. While all other object behaviours may be described by usual techniques, the object behaviour to be communicated should be defined by graph rules.

Graph rules can describe the creation and deletion of objects as well as links. Moreover, computations on members started by method invocation are modeled by attribute computations. All this is described by showing graphs similar to object diagrams on the left and on the right and relating these both in some way. Consider e.g. the sample graph rule in figure 7. This rule shows the deletion of an object of class B and the creation of a new object of class A. Moreover, member *memberA2* is newly computed for an object of class A which is preserved. This is marked by number 1 in the upper

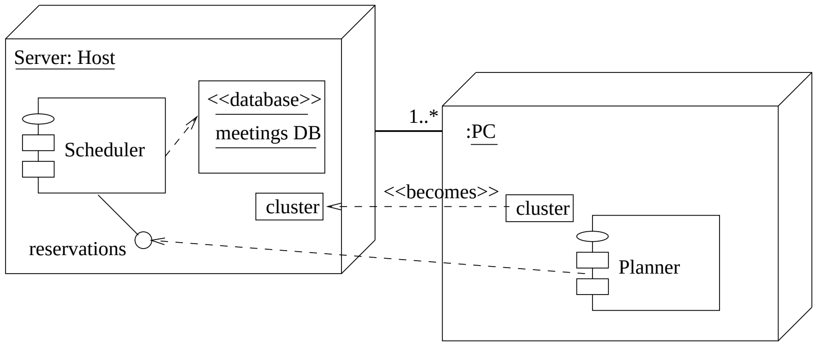

Figure 5: Standard modeling elements in UML

left corners of the objects in the left and in the right-hand side. This preserved object is, moreover, exported in a way, that the third member is private indicated by its grey background. Furthermore, the members of the newly created object are also determined.

Such *local object interaction* can be modeled by a distributed rule consisting of one local rule which describes the local action and an identic 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 and none of their attributes is newly computed. In that case, a synchronization has to be performed to inform the using systems that exported parts will be deleted. Adding objects and links to an export interface, i.e. offering new information, is always possible.

5.6 Remote Object Interaction

Remote object interaction is necessary, if remote services should be used. Services to be offered are specified by the export graph and export rules. Clearly, the contents of the export graph is currently avalable. Services offered by export rules are not available all the time, but only when the corresponding export rule can be applied to its export graph. On the other side, import rules are used to specify certain services requested. These requested services have to coincide with subservices of those offered by the export. This can be achieved twice: Either the import rule is applicable to an export graph, then the service is already offered, or the import rule is a subrule of an export rule and both rules are applicable to their graphs, then the service can be offered.

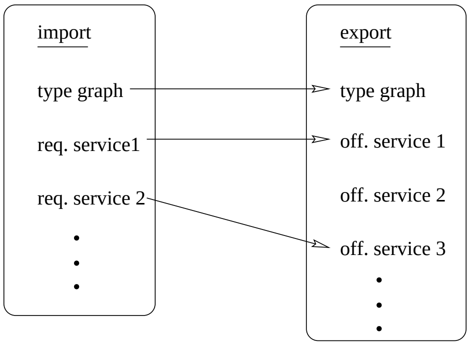

Figure 10: Import – export interconnection

Remote object interactions may be performed synchronously or asynchronously. Asynchronous object communication means sequential application of local rules such that something is 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 other 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 identic one involving several network nodes and edges where the synchronization has to take place. Figure 8 shows how a remote object can be used.

In figure 9 a service request is formulated in the sense that an object of type A in a special context is requested. Figure 10 shows a possible import/export relation between required and offered services.

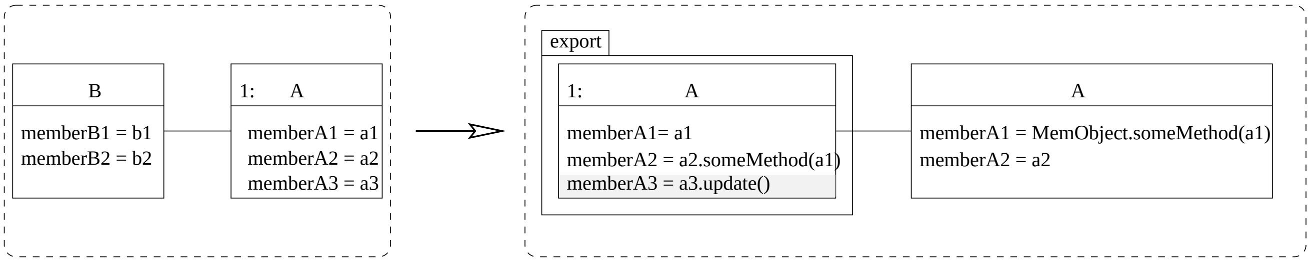

Figure 7: Sample graph rule modeling object behaviour

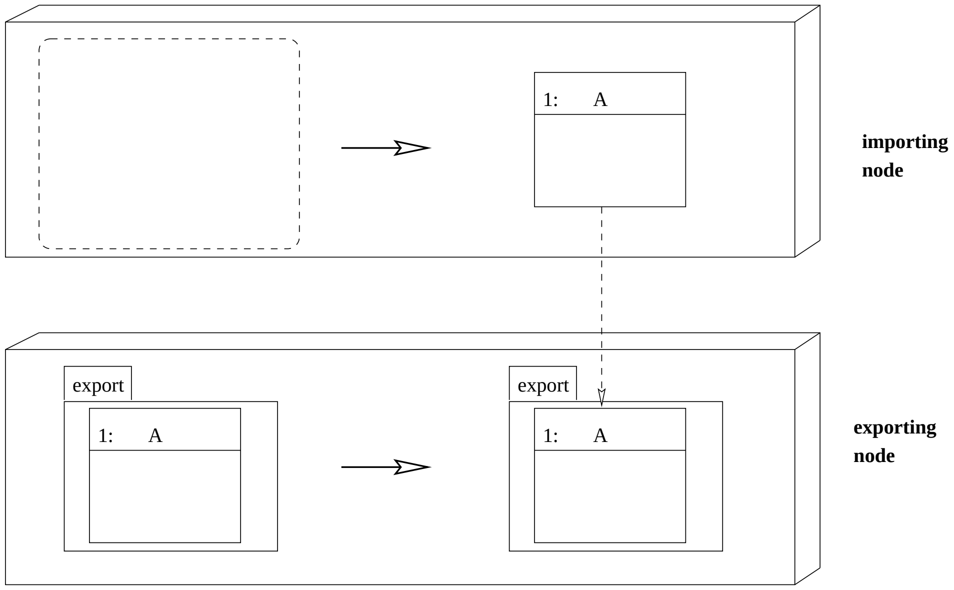

Figure 8: Sample use of a remote object – Local view of importing node

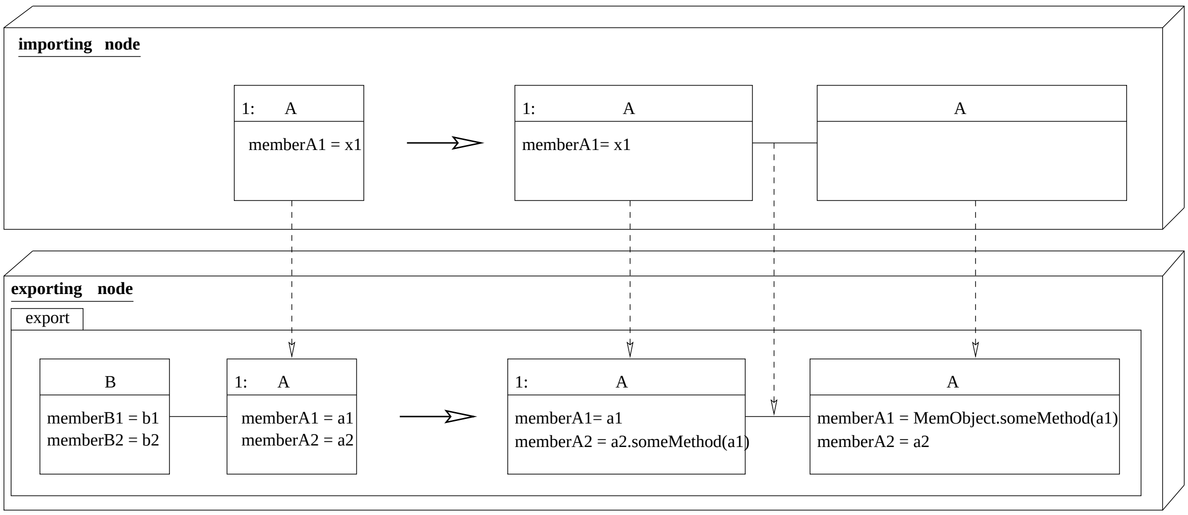

Figure 9: Import – export rule connection

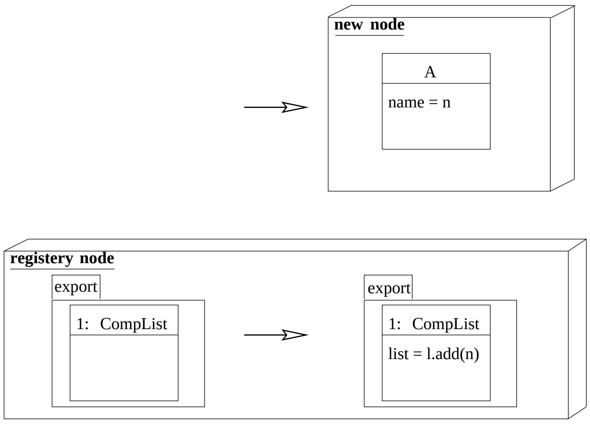

Figure 11: Registering a new component

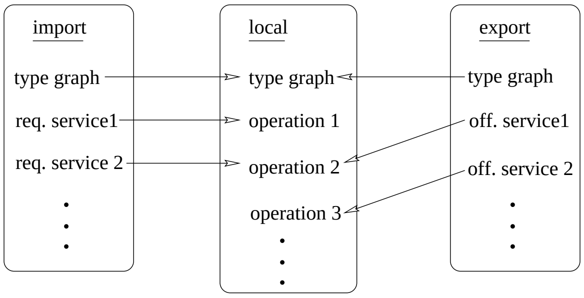

Figure 12: A sample local system - abstract

5.7 System Reconfiguration

A *system reconfiguration* can also be described by a distributed graph rule where its network rule describes the changes in the network topology. Moreover, the initial (final) state of the newly created (deleted) system components are determined. For preserved components those exports have to be specified which will be involved in remote component creation or deletion. Replication into a new network node is always possible. Consider e.g. figure 11 where the registering of a new node is described.

5.8 Local Systems and Views

A local system is described by a local graph transformation system, a set of import and a set of export graph transformation systems. Each graph transformation system contains a type graph to model the static aspects and a set of rules for the definition of the dynamics. Not each local rule must have import and/or export parts. Compare the abstract sample local system in figure 12.

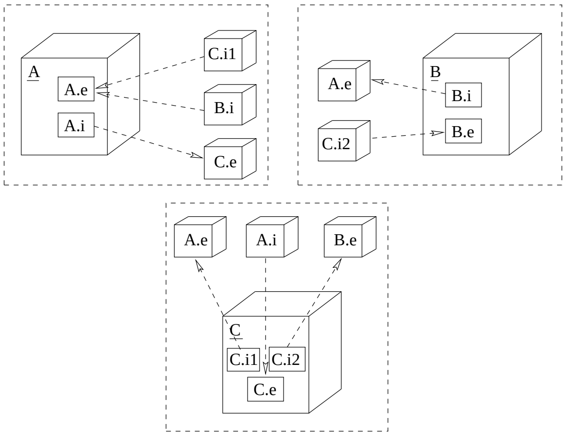

Figure 13: Example of three local views

In the early stages 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 to facilitate concurrent development.

The local view 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 objects accessible from other local systems and *import interfaces* contain objects used from remote export interfaces.

See figure 13 for an abstract example of three local views on local systems which use each others. The local systems and interfaces are modelled by nodes whereas the edges in between are used to describe the relations between these different parts. An export is indicated by its system name followed by a dot and an “e” where the import names end with “i”, “i1” and “i2” indicating different imports.

6 Reference Applications

Two main reference applications for distributed graph transformation have been considered so far to a larger extent: dynamic change management in the area of software design and distributed version control for remote software development. The former application is treated in [GMT98], the latter one in [TFKV98].

6.1 Dynamic Change Management

In the area of design of configurable distributed systems, Kramer and Magee [KM90, MK96] propose a basic model for dynamic change management which separates structural issues from application specific ones. Thus, they distinguish change rules at the architecture configuration level and change actions at the component level. This imples that the intraction between change actions and the local systems is minimal. As a consequence the evolution of a system can be performed in strucutred and systematic way. Kramer and Magee stated the following requirements [KM90] for configuration management of dynamic change:

1. *Changes should be specified in terms of 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.

2. *Change specification should be declarative.* Change specification should be separated from it is achieved. The responsibility how changes are executed - e.g. the specific ordering of actual change operations - has to be left to the configuration management system, not to the user.

3. *Change specification should be independent of algorithms, protocols, and states of application.* The goal is to provide generic configuration management inependent of the application. For example, component quiesence/passivity, i.e. the component is in a consistent state and is not communicating with its environment, include an abstract view on a component's state.

4. *Changes should leave the system in a consistent state.* A system is viewed as moving from one consistent state to the next. But while in progress, transient states may be mutually inconsistent.

5. *Changes should minimize the disruption to the application system.* It should not be necessary to stop the whole of a running application system to modify part of it.

These requirements can be achieved by the design technique presented above:

1. Structural change is formulated by graph transformation on the network level.

2. Since change is described by graph rules, the actual ordering of change actions as well as the degree of parallelism is not fived, but directly depend on the specification.

3. The separation of graph transformation on the network and the local level meets this requirement.

4. Distributed graph transformation has to satisfy a number of application conditions which ensure a consistent graph as result.

5. This is achieved by the rule-based approach, i.e. the left-hand rule sides precisely delineate the involved area.

To verify these statements a sample system consisting of a ring of connected replicated databases where each database can autonomously initiate an update of its containing data is presented in [GMT98].

6.2 Distributed Version Control

The size of a software project and the high quality requirements make it difficult to plan and to coordinate a greater project. As the knowledge for its completion is hardly available in a single company in our days, often external project partners must contribute. It may also be cheaper to give work outside to external experts. In this case communication, coordination and quality management is needed, as all project sites must have access to a consistent actual set of project documents. When a project site changes a document leading to a new revision, it must become known in all other project sites, too. This is especially a problem, when no central online archive can be used by all project partners. Then each project site has its own *revision archive*, that is a local station where all documents and their different revisions are stored. Documents are *replicated* between revision archives to ensure that each project site has an up-to-date document set. When a document of a revision archive is to be changed or a new document has to be inserted this is done with the help of *workspaces*. When the owner of a workspace wants to change a document he/she *checks it out* from the archive into the workspace. Then the actual change can take place or something new can be created. When this work is finished, the documents are *checked back into* the revision archive.

These concepts can be easily realized on the network level of distributed graph transformation. Each workspace and each revision archive is a node of the network graph with im- and export interface. Via these interfaces the replication between different archives and the checking in and out between archive and workspace can be realized.

In the revision archive, documents are sorted according to two principles. First documents are packed into configurations. A configuration contains documents belonging together. These may be other configurations or plain documents containing e.g. text or code. Additionally, different revisions of one document are stored. Each time a document is changed the old version is kept and the new one changed is stored as a new revision of this old version. This is also valid for configurations. In the workspace just one revision of a document can be checked out, the one that is actually changed. Additionally always a complete configuration should be checked out. This way a workspace's local graph contains just a part of the revision archive's graph.

A node in the local graph modeling a revision is additionally labeled with several labels including its creator, its content and a unique document identification number local to the site the document is at.

The key operations are checking in and out of documents. These operations deserve remote object interaction. Performing these operations it must be ensured somehow that a changed document in the workspace is inserted into the revision archive as a direct successor after the revision from that it was originally checked out.

7 Conclusions

In this contribution, we presented a new visual design technique for distributed object systems based on graph transformation. It includes the graphical description of remote object interaction, object migration and replication, communication and synchronization as well as dynamic reconfiguration of distributed systems. 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 object structures which may be communicated over the network. Services are available when the describing rule is applicable to some export graph showing public object structures. Moreover, dynamic reconfiguration can be described in a very flexible way.

Currently, we are working on an implementation of the ideas presented which is based on the visual graph transformation based specification and programming environment AGG [TER98]. Of course, the implementation will be able to run in a distributed manner.

Although this new design technique has been applied to two non-trivial case studies, further validation is needed to find out how far distribution issues are repesented adequately and which further issues have to be considered.

Considering distributed systems a notion of application dependent consistency without having access to the entire state all the time is very important. Since our design technique is based on graph transformation as formal framework, the results of this area can be used. A notion of consistency based on logical formulas on the existence and non-existence of graph structures, and an constructive approach to ensure this consistency are presented in [HW95]. It is due to future work to extend this work to distributed graph transformation.

References

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

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

[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 (ed.), *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.

[Ehr79] H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In V. Claus, H. Ehrig, and G. Rozenberg (eds.), *1st Graph Grammar Workshop,* *Lecture Notes in Computer Science 73*, pages 1–69. Springer Verlag, 1979.

[EKR91] H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.). *4th International Workshop on Graph Grammars and Their Application to Computer Science*. Springer Verlag, 1991. Lecture Notes in Computer Science 532.

[GMT98] M. Goedicke, T. Meyer, and G. Taentzer. Dynamic Change Management by Distributed Graph Transformation : Towards Configurable Distributed Systems. submitted to 6.Int. Workshop on "Theory and Application of Graph Transformation", Paderborn, Germany, Nov. 1998, 1998.

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

[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.

[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. Kaplan, J. Loyall, and S. Goering. Specifying Concurrent Languages and Systems with Δ-Grammars. In Ehrig et al. [EKR91], pages 475–489. Lecture Notes in Computer Science 532.

[KM90] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, SE-16(11):1293–1306, 1990.

[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.

[Mil80] R. Milner (ed.). *A calculus for communicationg systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[MK96] J. Magee and J. Kramer. Self Organizing Software Architectures. In *Proc. SIGSOFT Workshops*, pages 35 – 38. ACM Press, 1996.

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

[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.

[Rei87] W. 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.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

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

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

[TER98] G. Taentzer, C. Ermel, and M. Rudolf. AGG-Approach: Language and Tool Environment. to appear as article in graph grammar handbook 2 "Specification and Programming", 1998.

[TFKV98] G. Taentzer, I. Fischer, M. Koch, and V. Volle. Distributed Graph Transformation with Application to Visual Design of Distributed Systems. to appear as article in Graph Grammar Handbook 3 "Concurrency and Distribution", 1998.