Electronic Notes in Theoretical Computer Science 55 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume55.html pp. 1 –

Visual Modeling of Distributed Object Systems by Graph Transformation

Gabriele Taentzer [1]

University of Paderborn, gabi@upb.de

Abstract

A visual modeling technique for distributed object systems based on graph transformation is presented. It includes the graphical description of the network and its dynamic reconfiguration as well as the component interfaces and local object systems and their behavior. Typical issues in distributed systems like remote object interaction, object migration and replication, communication and synchronization are expressible in this technique. The notation is close to UML. It extends the UML notation where needed. 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.

1 Introduction

Distributed systems demand a number of requirements on specification techniques which have to be taken into account in addition to the development of non-distributed software. Allocation of objects and tasks to network nodes, 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, such as object allocation, are already handled by common modeling techniques for object-oriented systems, such as UML [20], but others like system reconfiguration cannot be designed sufficiently with the techniques available. Furthermore, there are a number of formal specification techniques, such as temporal logics [17], process algebras [14], Petri nets [15], and actor systems [1], to model the concurrent behavior of distributed systems. The structural aspects of distributed systems are hardly tackled by these approaches. Moreover, dynamic reconfiguration of a distributed system as well as distributed data handling cannot be addressed directly by most of the techniques. Considering all the techniques mentioned, actor systems are that technique meeting most of the features required. But

[1] On the leave of: Technical University of Berlin, e-mail: gabi@cs.tu-berlin.de

also this technique does not support the handling of distributed data sufficiently, since e.g. replication cannot be addressed directly.

In this article, the concepts for visual modeling of distributed object systems presented are based on *distributed graph transformation* as the 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 by rules. The advantage of using rules in this context is to avoid purely sequential or series-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. Throughout this contribution all formal concepts are introduced on an intuitive level in order to be comprehensible. Distributed graph transformation is formally presented in [18,6].

2 Distributed Graph Transformation

Several graph transformation approaches exist which have been used to model distributed systems. One big advantage of these approaches is the rule-based nature of dynamics description. Rules are well suited to describe reconfiguration in distributed systems, because they inherently model the asynchrony and non-determinism of reconfiguration control. All the graph transformation 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, e.g. in [2], by Δ-grammars in [10], in [16] and by actor graph grammars in [9].

But for the application to distributed systems, graph transformation can be employed twice: to describe the dynamic reconfiguration of distributed systems on the network level and, in addition, for modeling evolving data and object structures in local systems. The advantage of using graph transformation on the local level is manifold: a description of replicated objects in remote systems, of local actions as well as of remote interaction which includes object migration, replication, communication and synchronization. In order to be able to describe all these features in a modeling technique for distributed systems the combination of the network graph transformation with local transformations is needed.

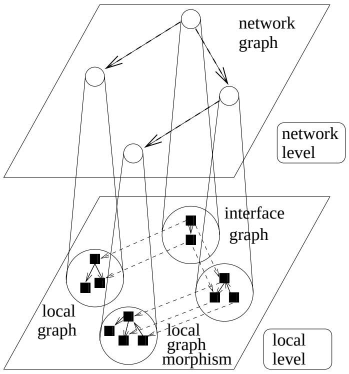

Fig. 1. Example of a distributed graph

Distributed graph transformation (DGT) fulfills this aim by describing both the network and the local level of a distributed system in one *distributed graph*.

A distributed graph can be seen as a hierarchical graph on two levels (cf. Figure 1). 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 most approaches, graph transformation is used also on the local level to manipulate local data structures. Each network node is equipped with a graph representing its local state and each network edge is equipped with a graph morphism, called *local graph morphism*, representing a relation between two local states. Moreover, nodes and edges on both levels may be attributed by any element of some data algebra. Local graph morphisms also map data attributes by containing a homomorphism between some algebra and a superalgebra. Please note that a graph object with a smaller number of attributes can be mapped to one with more attributes if each of the attributes can be mapped. So, 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.

A *distributed rule* consists of a network rule and local rules for all network nodes which are preserved by the network rule. Network as well as local rules are graph rules and may contain a number of attribute computations.

See Figure 2 for a schematic representation of a distributed rule. In this example, one local network node (with number 2), its local graph as well as the network edge and its local morphism are deleted. The local rule graph determines the state in which such a network node is allowed to be deleted, i.e. an isomorphism has to be established between the local rule and the corresponding local host graph. This restricted form of deletion corresponds directly to the way how local systems are created. The whole initial state is determined in the rule, in the deletion case the whole final state has to be described within the rule as well. Going on with the example, we can see that one network node is preserved (number 1). Its local graph can be manipulated.

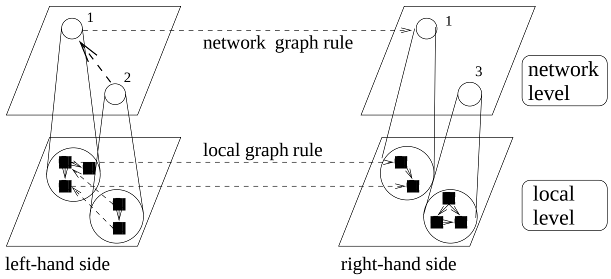

Fig. 2. Example of a distributed rule

This is done by the local rule given. On the right-hand side a new network node (number 3) has to be created. Its initial state is given by the local graph equipped.

The transformation of distributed graphs by *distributed rules* is performed by a number of graph transformations using the network and local rules contained in a distributed rule. All these transformations can be done nondeterministically. Network nodes as well as edges may be deleted or newly created as indicated by the rule. Newly created network nodes are equipped with a local graph describing the initial state of this local system. Deletion of network nodes is only possible if the whole local system is deleted as well. Relations between transformed local graphs are induced by the transformation steps. For the application of a distributed rule in a distributed graph an occurrence of the left-hand side has to be chosen. In addition to the dangling condition for all local rules and the network rule some additional application conditions have to be checked in order to guarantee a distributed graph as result. (These conditions are explained in more detail in [6]).

Formally, a distributed graph transformation step is formulated by a double-pushout in the category of distributed graphs and graph morphisms. On this basis, the existence and uniqueness of a distributed graph transformation step can be shown. This coherence is extremely useful for modeling consistent data structure transformations in a distributed environment which get soon too complex hence, difficult to understand. 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. truly concurrently.

3 The Modeling Technique

In the following, we sketch the modeling technique on a running example showing a number of key aspects of distributed systems. The running example deals with distributed version management for remote software development (see [6] for a detailed model).

The underlying concept for this design technique is distributed graph transformation. Since graphs and graph transformations are defined independently of graphical layouts, we adapt the examples presented below as far as possible to the graphical notation of UML. The notation is built up on object and deployment diagrams and is extended to capture new concepts not available in UML.

At first, we introduce our running example. Afterwards, we consider the static aspects of distributed systems in section 5.1, then the dynamic aspects in section 5.2.

3.1 Running Example: Distributed Version Management

Because of the costs, the size of the projects, high quality requirements etc. software is more and more developed in a distributed way nowadays. Therefore, communication, coordination, and quality management are needed, because all project sites must have access to a consistent, up-to-date 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.

One approach to distributed version management is that each project site has its own *revision archive* which 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 independent 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 in *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, i.e. a new revision is created whereas the previous one remains unchanged.

3.2 Modeling the Statics in Distributed Systems

At first, the static aspects of the network level are considered. Afterwards, local structures are modeled. The distributed object structures are specified in local views. Here, both levels are combined.

3.2.1 Network Graph Structure

The topological structure of a distributed system's network is presented graphically by a typed network graph. Nodes of the network graph represent any kind of processing unit with some memory, edges indicate communication paths between nodes. In our running example, the network of revision archives and workspaces can be easily modeled by a network graph depicted on the left

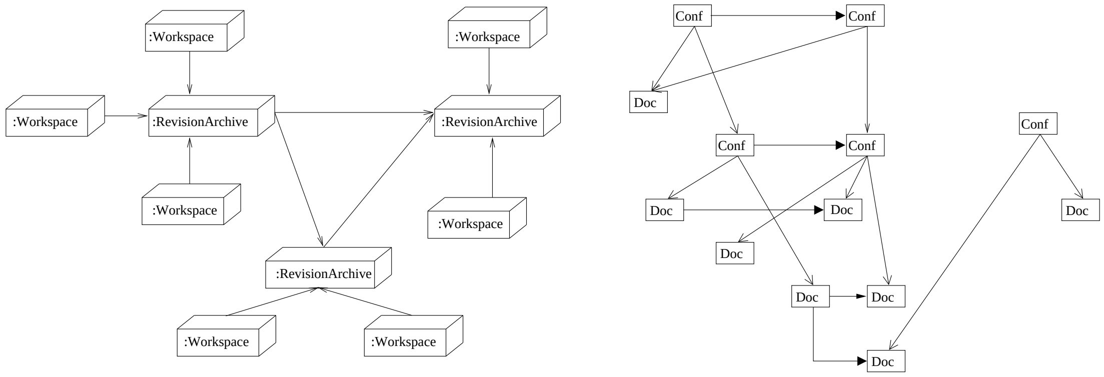

Fig. 3. Sample network and local graph

one in Figure 3. Each workspace and each revision archive is a node of the network graph. Each site has only one archive, but a possibly changing number of workspaces which are connected to their local archive.

In open systems, network structures are not static, but may evolve over time. Graph transformation has proved to be well suited for specifying this evolution. The dynamic part of networks, i.e. network reconfiguration by graph transformation, is explained in section 3.3.3.

3.2.2 Local Object Structures

To describe static and dynamic aspects of object-oriented systems well-known OOA/D techniques like OMT, OOD and UML may be used. The underlying abstract structure of these models are usually graphs. In the following examples, the graphical notation extends the UML-notation where needed. Class and object diagrams are the standard techniques 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 which have to be type compatible with it are described by local graphs.

Consider e.g. a local graph of a revision archive on the right of Figure 3. It contains the revision structure of documents following two sorting principles. At first, revisions are grouped into configurations which may be nested. Configurations are hierarchically structured with documents as leaves. Moreover, different revisions of documents are related where the arrow direction indicates the development direction. Moreover, configuration and documents may carry attributes concerning identity, name, revision number, creator, contents, etc. Extending the boxes in Figure 3, these attributes can be denoted in a way similar to static structure diagrams in UML. Object relations are depicted in a slightly different way from UML-associations. Formally, local object structures are considered as attributed graphs.

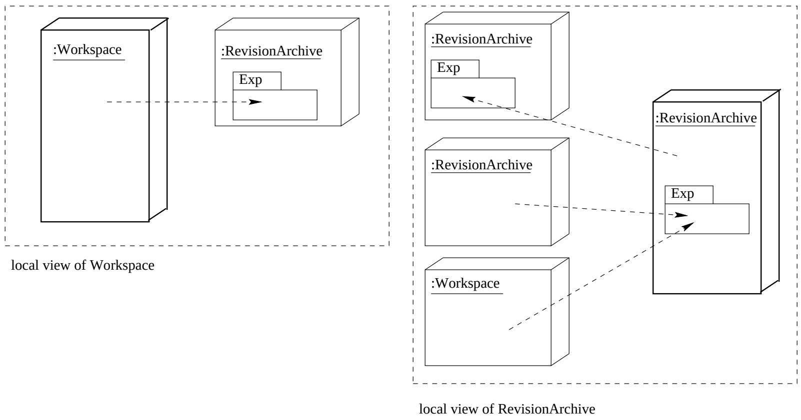

Fig. 4. Local views of workspaces and revision archives

3.2.3 Local Systems and Views

In the early stages of system development, a global view on the entire distributed system is desirable. Once the network structure and its reconfiguration possibilities 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 the local system itself 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 from remote export interfaces.

Figure 4 shows two local views: the local view of a workspace and that of a revision archive. A workspace knows the export interface of the revision archive from where it checks out the documents. A revision archive may contain replicas of configurations or documents in other archives, i.e. it imports them from other archives. Furthermore, it can export configurations and documents for replication by other archives or for checkout by a workspace. In each local view, the local system is depicted in bold face.

3.2.4 Distributed Object Structures

The combination of the network graph structure and the local object structures is specified by distributed graphs in order to describe distributed object structures. The graphical notation for distributed object structures used in the sequel builds up on UML deployment diagrams. A deployment diagram shows a system's network topology and the software components, processes and nodes that live on the network nodes. 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 are located in nodes and dependencies between different

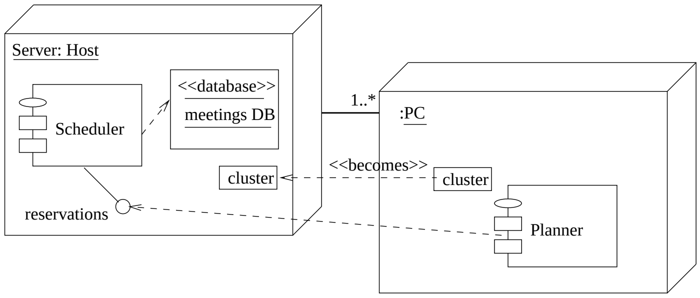

Fig. 5. Standard modeling elements in UML

structuring elements are shown.

Deployment diagrams can be easily regarded as distributed graphs where the big boxes constitute the network graph and the components, objects, etc. within these boxes are the local graphs. Therefore, we will capitalize on deployment diagrams to describe the allocation of objects, processes, etc. to network nodes. On the contrary, object migration which is a dynamic issue will be modeled by graph rules. This is actually a remote interaction and will be explained in subsection 3.3.2.

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

Moreover, we distinguish export- and import interface graphs. Export interface graphs offer public parts of local object structures whereas an import interface graph determines the part to be imported from another system. The connection between import and export interfaces, i.e. the connecting graph morphism, shows an actual imported remote object structure.

The design of import and export interface structures is handled by type graphs. Such a type graph indicates also which imports can be connected to which export interfaces. 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 of its members, i.e. its attributes. An import type graph specifies all object types and associations that may be imported from some export. 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. Compare 6 for a schematic presentation of distributed state graphs.

Consider Figure 7 for a very small example of a distributed state graph in the local view of a workspace. It consists of two networks nodes which are connected. Within the workspace node there is one local configuration containing a document. The revision archive also contains these two objects which are actually replicated by the workspace, i.e. they are checked out (imported) by the workspace. This is possible, because both objects are exported by the

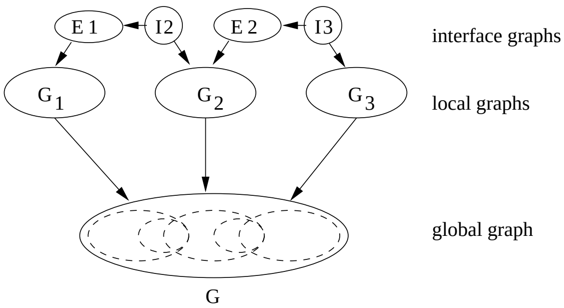

Fig. 6. Distributed state graphs

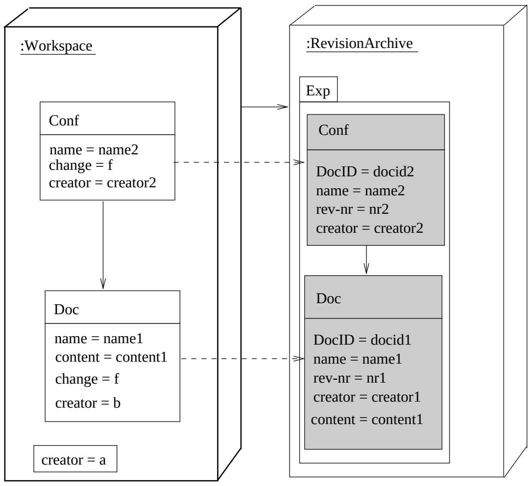

Fig. 7. A sample distributed object structure

archive which is indicated by a box labeled by “Exp”. This import-export relation is indicated by dashed arrows pointing to the originals. Please note that the objects in the archive contain partly different attributes than their correspondents in the workspace. Attributes like the internal document identity DocID are not relevant within a workspace, but for other revision archives. It is also possible not to export single attributes for hiding local informations, although the entire object is exported. On the other side, attributes may be local such as *change* is local to workspaces.

3.3 Modeling the Dynamics in Distributed Systems

The following paragraphs deal with the dynamic aspects of a distributed system. At first, object interactions in one system component are considered, then remote object interaction in several components and system reconfiguration possibilities are treated in the following paragraphs.

Dynamics are described by distributed graph rules. They 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. A local graph rule can also have *export* and *import rules* as sub rules. Modeling reconfiguration, the local graph of a network node to be deleted determines the state

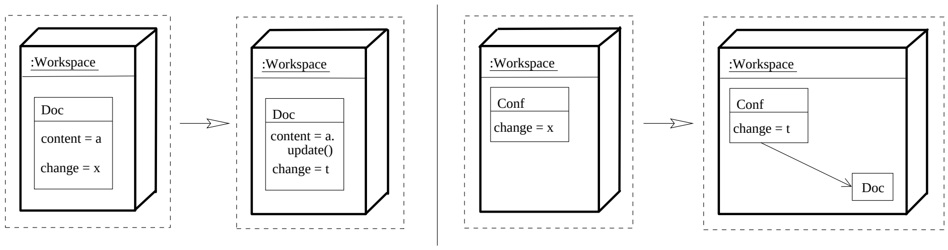

Fig. 8. Updating a document (left) and creating a new document (right)

in which the corresponding local system is allowed to be terminated. A new network node can be created and its initial state is given by the local graph equipped. Mostly, distributed graph rules are used to describe synchronization and communication in distributed systems.

3.3.1 Local Object Interaction

There are several techniques to describe the dynamic aspects in object-oriented systems by e.g. state charts or sequence and collaboration diagrams. We propose additionally graph transformation for this purpose as the rule-based way of dynamics specification is well suited to be used for specification of distributed behavior. If an object behavior is described by a rule, it may be possible to separate this rule into several sub rules. In this way, we separate also the object behavior in several parts. This allows us easily to separate the behavior into the behavior visible for other objects and the internal behavior by extracting two sub rules.

It is worth mentioning again, that we do not intend to use graph transformation as the one and only formalism for specifying software systems. We recommend it for the specification of the communication between local components. For the specification of the dynamic aspects restricted to one local component only any other technique, e.g. state charts or collaboration diagrams, may be used. Although on the formal level, graph rules can be used to define the semantics of e.g. collaboration diagrams [4]. Thus, different behavior description techniques can be semantically mapped on one formalism, i.e. graph transformation.

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 by a graph morphism. Consider e.g. Figure 8, where local actions in a workspace are modeled. The rule on the left models an update of a document. The document is preserved, only two attributes are newly computed. On the right, a rule for document creation is depicted. It has to belong to a configuration. Thus, the configuration has to occur in the rule, a document node and a relating edge are newly created. In this case, all attributes of the new document are unset. Further rules are needed to set the attribute values.

Such *local object interaction* can be modeled by a distributed rule where the network graph of both – the left and right-hand side – contains exactly one node, namely the component where the local action shall take place. This network node is preserved. The local rule for this component describes the local action. Changes on export interfaces can also be performed locally as long as graph parts used remotely are not deleted and none of their attributes is newly computed. Otherwise, a synchronization has to be performed to inform the using systems that parts which are exported will be deleted. Adding objects and links to an export interface, i.e. offering new information, is possible without any problem.

3.3.2 Remote Object Interaction

Remote object interaction is necessary, if remote services should be used. These services are specified by the export graph and export rules. The contents of an export graph shows which services are available at this moment in time. An export service (modelled by a export graph rule) offered in principle may not be available, if the corresponding export rule cannot be applied to the actual export graph. On the other side, import rules are used to specify certain services requested. These requesting services have to coincide with sub services 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 sub rule of an export rule and both rules are applicable to their graphs, then the service can be offered.

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 which are interconnected by 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 rules are identical ones which have to overlap in those interface nodes where the synchronization should take place.

In Figure 9, the checkout of a document is specified. Please note that all attributes relevant for workspaces are replicated from objects in the archive. Moreover, local attribute *change* is set. Before the checkout of a document can take place, it has to be exported by the revision archive first. Thus, a revision archive has to offer the service to export a certain document. This service may be requested by workspaces for checkout and by other revision archives for replication, i.e. checkout and replication are modeled asynchronously.

Replication between revision archives is not shown explicitly here due the space limitations. Roughly spoken, it consists of two steps: first new revisions have to be exported (as for checkout by a workspace), then connected archives can imported them. During the import a revision archive has to make sure

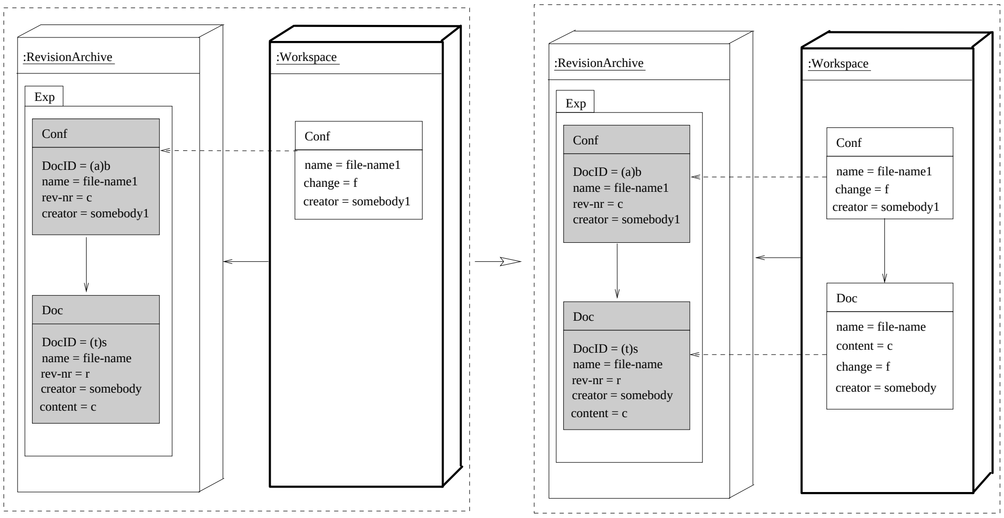

Fig. 9. Checkout of a document

that no workspace tries to check out an incomplete configuration. For this purpose a lock has to be set first on that configuration which is updated. These replication rules are intended to be applied as soon as they are applicable, i.e. the update between distributed archives is immediately done when possible (the corresponding document is not currently checked out), and can be performed automatically.

3.3.3 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 is determined. For preserved components those exports have to be specified which will be involved in remote component creation or deletion.

Figure 10 shows two rules modeling the creation of a workspace and a revision archive. In both rules, also local nodes are created, modeling local information. For the creation of a workspace there has to be already an archive to which it will be related. Deletion of workspaces and revision archives can be modeled by the inverse rules. But note that these rules can only be applied when the workspace or archive, resp., is empty, i.e. does not contain a configuration. This is due to the application conditions for distributed graph transformation (compare section 2). To achieve such a state further rules have to be applied first which delete each document in a workspace or an archive, resp. Furthermore, an archive to be deleted first has to be disconnected from others.

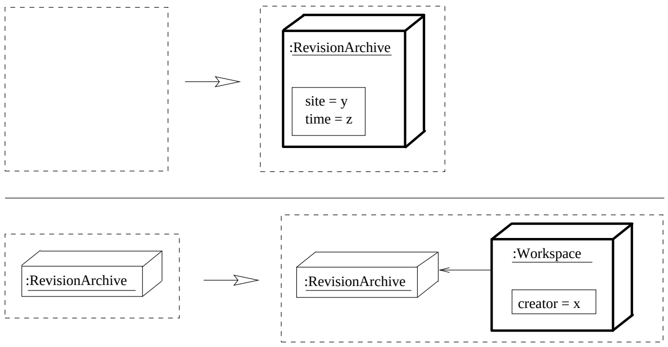

Fig. 10. Creating a new revision archive (top) and a new workspace (bottom)

4 Conclusions

In this contribution, we presented a visual modeling technique for distributed object systems based on distributed 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 modeling is well suited to describe highly concurrent events, 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.

A prototypical implementation of the ideas presented is based on the graph transformation machine AGG [5]. The execution of a remote object interaction is performed in a true distributed manner relying on Java RMI.

The modeling technique has been applied to two case studies, i.e. distributed version control for remote software development (the running example) and dynamic change management in the area of software design [12], [13]. The former application is treated in more detail in [6], the latter one in [19]. Although these cases studies are non-trivial, further validation is needed to find out how far distribution issues are represented 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 modeling 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 a constructive approach to ensure this consistency are presented in [8]. Another approach to consistency checking has been followed in [7,11] by graph-interpreted temporal logics. It is left to future work to extend these approaches to distributed graph transformation as it has been presented in this paper.

References

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

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

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

[4] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioural Diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language*, volume 1939 of *Lecture Notes in Computer Science*, pages 323 – 337. Springer Verlag, 2000.

[5] C. Ermel, M. Rudolf, and G. Taentzer. The AGG-Approach: Language and Tool Environment. In H. Ehrig, G. Engels, J.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools*, pages 551–603. World Scientific, 1999.

[6] I. Fischer, M. Koch, G. Taentzer, and V. Volle. Distributed Graph Transformation with Application to Visual Design of Distributed Systems. In H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency, Parallelism, and Distribution*, pages 269–340. World Scientific, 1999.

[7] F. Gadducci, R. Heckel, and M. Koch. A Fully Abstract Model for Graph-Interpreted Temporal Logic. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 6th International Workshop on Theory and Application of Graph Transformation (TAGT'98)*, Springer LNCS 1764, pages 310 – 322, 2000.

[8] 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.

[9] D. Janssens and G. Rozenberg. Structured transformations and computation graphs for actor grammars. In Ehrig et al. [3], pages 446–460. Lecture Notes in Computer Science 532.

[10] S.M. Kaplan, J.P. Loyall, and S.K. Goering. Specifying concurrent languages and systems with Δ-grammars. In Ehrig et al. [3], pages 475–489. Lecture Notes in Computer Science 532.

[11] M. Koch. *Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems.* PhD thesis, Technische Universität Berlin, FB 13, 1999.

[12] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, SE-16(11):1293–1306, 1990.

[13] J. Magee and J. Kramer. Self organizing software architectures. In *Proc. SIGSOFT Workshops*, pages 35 – 38. ACM Press, 1996.

[14] R. Milner, editor. *A calculus for communicationg systems.* Springer LNCS 92, 1980.

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

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

[17] C. Stirling. Modal and temporal logics. In *Handbook of Logic in Computer Science*, volume 2, Background: Computational structures. Clarendon Press, Oxford, 1992.

[18] G. Taentzer. Distributed Graphs and Graph Transformation. *Applied Categorical Structures*, 4(4):431–462, December 1999.

[19] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic Change Management by Distributed Graph Transformation: Towards Configurable Distributed Systems. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Theory and Application of Graph Transformations*, volume 1764, pages 179 – 193. Springer Verlag, 2000.

[20] *Unified Modeling Language – version 1.3*, 2000. Available at `http://www.omg.org/uml`.