A VISUAL MODELING FRAMEWORK FOR DISTRIBUTED OBJECT COMPUTING

Gabriele Taentzer
University of Paderborn, Germany
gabi@upb.de

Abstract Distributed object computing is a computing paradigm that allows objects to be distributed over a heterogeneous network. Infrastructures help to develop distributed object applications by offering necessary services for distributed computing. Having a comprehensive infrastructure to hand, the development of complex distributed object systems is feasible in principle. Flexibly evolving architectures as well as highly dynamic distributed object structures are key requirements for nowadays distributed solutions. They can hardly be well designed on this level of programming, due to their complexity. A visual modeling framework is presented which offers a more abstract and intuitive approach to the relevant aspects of a distributed object system. In this framework, network and object structures as well as their evolution are visualized in a diagrammatic style, e.g. in UML notation. Semantically, this approach relies on graphs and their transformation, i.e. it has a precise background useful for further reasoning.

Keywords: distributed object computing, visual modeling, I/O-automata, graph transformation

1. Introduction

Distributed applications and services are intended to function under various different assumptions and in the face of many possible difficulties. Reasons are among others heterogeneous environments, fast evolving requirements, a high amount of uncertainty, etc. A clear distributed object model which provides an abstract and simplified, but consistent description of the relevant aspects of a distributed object system can help to find common design problems and to avoid wrong design decisions.

Designing a distributed application or service, the following models play important roles: An *architectural model* defines the logical component structure of a distributed system and clarifies how logical components are mapped on an underlying network of processors. The *interaction model* describes the

steps to be taken in each component to communicate with other components. The interaction can be message-based or by shared memory, synchronous or asynchronous. Depending on these key properties of the interaction model the amount of uncertainty differs heavily and can lead to very different design of protocols, service strategies, etc. The key behavior is usually described by distributed algorithms, states are left as simple as possible, focusing mainly on concurrency and failure issues. Additional requirements and effects coming in through concurrent and/or faulty access to complex object structures, are not considered by automata-based methods. On the other hand, object-oriented techniques provide little support for the description of faulty object access. The way how processes and communication channels can fail and when failures can occur, has also to be considered in the *interaction model* in order to discuss their effects. Last but not least, the *security model* clarifies which of the shared resources have to be protected against unauthorized usage.

Several visual modeling techniques are successfully used to describe the architecture and interaction aspects mentioned above. UML (OMG, 2000) and its extension UML-RT are often used for architecture modeling where component diagrams describe the logical structure and deployment diagrams handle its mapping to underlying hardware resources. UML-RT (Lyons, 1998) has been developed for real-time application which may be distributed. A structure diagram shows the static aspects distinguishing capsules, ports, connections, and roles, while some sort of statechart models the behavior of each capsule. Statecharts are also used in other contexts for the behavior description of concurrent processes. Furthermore, Petri nets are used to describe the interaction between processes where state descriptions are simplified as much as possible, focusing mainly on concurrency and failure issues. (Compare (Reisig, 1998) for a discussion of distributed algorithms modeled by Petri nets.)

In this paper, *graph transformation* is proposed as semantic domain for visual modeling of distributed object systems. In the sequel, we start with UML-component diagrams and collaborations and map these modeling techniques to network graphs where nodes are refined by graph transformation components describing process states and behavior. Composing graph transformation components to distributed graph transformation systems (Fischer et al., 1999) we provide a formalization of object interaction useful to define the semantics of a collaboration model. Furthermore, we show that a restricted form of (distributed) graph transformation systems are I/O-automata such that the theory on I/O-automata applies also to graph transformation systems and is usable for formal reasoning on concurrency issues of a restricted form of UML-model. The intermediate graph transformation model adds the possibility to reason also on e.g. consistency of dynamic object structures (Heckel and Wagner, 1995), data dependencies between concurrent activities (Corradini

et al., 1996; Corradini et al., 1997) as well as properties of dynamic network configuration.

The main parts of the distributed computing model are presented on a syntactic level first, providing a UML model. For a syntactic correct model its semantic meaning is provided by a restricted distributed graph transformation system which can be considered as an I/O-automaton.

2. Requirements for Distributed System Modeling

Although distributed applications can be found everywhere, they are seldom backed by a model capturing the main design decisions concerning architecture and interaction. A modeling technique for distributed object computing should be able to cope with a variety of design aspects in a flexible and intuitive way and should support validation of essential properties. In the following, we discuss the main challenges a designer of distributed applications usually meets.

Developing a distributed application, especially on the Internet, confronts the designer with many different sorts of computers and networks. The developer has to deal with heterogeneous hardware and software. Moreover, for larger applications, several developers are engaged following different encoding styles. An adequate model has to offer different abstraction levels, concentrating on the main aspects first, and taking details into account later. Despite all *heterogeneity* the key design should be independent of that.

Distributed applications and services usually include a high amount of *concurrency* which might lead to complex control flow difficult to overlook during programming and even more difficult to test by distributed debugging. Thus, reasoning on concurrent execution is the main reason to develop a distributed computing model. There exist a variety of automaton-based modeling techniques like I/O-automata, statecharts, and Petri nets for this purpose. (Lynch, 1996) is a comprehensive work on distributed algorithms considering key problems for concurrent and distributed processes. Thus, one of the main requirements of a distributed computing model is a clear formal background for reasoning.

In distributed applications, different kinds of *failures* can occur. A distributed computing model should give the possibility to express all kinds of failures which can occur, and moreover, should support the modeling of techniques how to deal with failures such that a high availability of the whole system can be guaranteed.

Moreover, a distributed computing model should offer the possibility to model the increase of resources and clients. Furthermore, testing of scalability, i.e. testing the performance when the system scales up, should be supported.

There are a variety of models for quality of service tests, e.g. based on timed Petri nets (Bestuzheva and Rudnev, 1990).

To achieve openness the key interfaces have to be published. This is usually done on the programming level using standards as CORBA IDL (OMG, 2001) or, more recently, SOAP (W3C, 2001) which is based on XML. Distributed computing models, if there are any used, are not standardized and thus, not published. Here, UML seems to be a natural candidate, since it is already a standard, and there are efforts to extend it for distributed object modeling.

Last but not least, *security* issues are of considerable importance in distributed systems. A distributed computing model should give the possibility to model and reason on security strategies concerning protection against unauthorized users, manipulation and corruption of resources, and accessibility of resources.

Due to its intrinsic complexity another important design issue for a distributed system is the transparency of certain aspects to the users. The degree of transparency is much dependent from the design of a distributed system. Thus, a distributed computing model should support modeling facilities to achieve transparency, e.g. by flexible definition of export interfaces.

3. Visual Modeling of Distributed Object Systems

In the following, we sketch a visual modeling technique which builds upon a small fragment of UML. Its provides diagrammatic notations to describe distributed architecture and their reconfiguration, the distributed object model and interactions in different views. Its visual syntax follows largely the UML notation, extended new features. The semantic domain will be defined by distributed graph transformation presented in the next section.

3.1. The Architectural Model

Considering the architecture of a distributed application we have to distinguish the logical structure, i.e. the component structure, and the deployment on processors. For the logical distribution, component diagrams can be used which contain components, their interfaces and interrelations between components and interfaces. Consider Fig. 1 for a simple client/server architecture of a file server. UML deployment diagrams can be used to define the mapping of logical components to the physical resources.

In open systems, distributed component structures are not static, but may evolve over time. We introduce reconfiguration diagrams which take component diagrams and annotate components, interfaces or interrelations by $\{destroyed\}$ or $\{new\}$, similarly to dynamic object structures in collaboration diagrams. Fig. 2 shows three reconfiguration diagrams for flexible client/server architecture with replication facilities between several file servers (for e.g. load

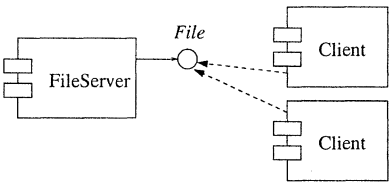

Figure 1. A client/server architecture

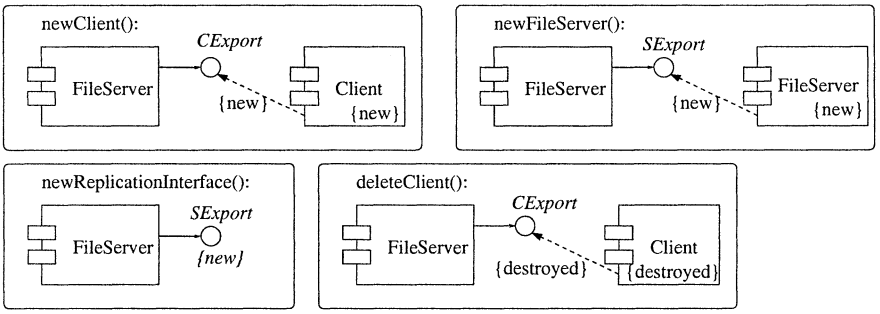

Figure 2. Reconfiguration diagrams

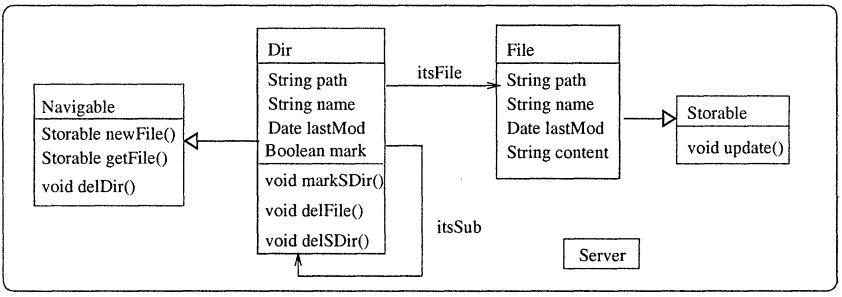

Figure 3. A simple class diagram for file structures

balancing). All these examples contain reconfiguration of the logical structure only, but reconfiguration diagrams can also be used on the deployment level. Note that a file server can have different exports for clients and other servers.

3.2. The Interaction Model

The static aspects of an object-oriented interaction model are usually described by a class diagram. It contains interface classes for export purposes. Consider e.g. the essential part of a class diagram in Fig. 3 for component 'FileServer'.

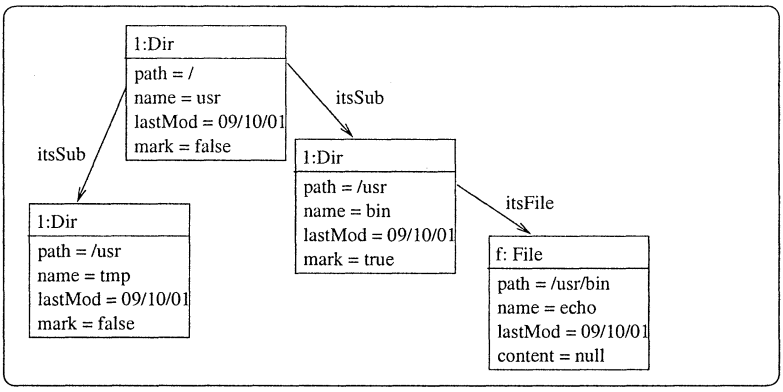

Figure 4. A sample directory structure

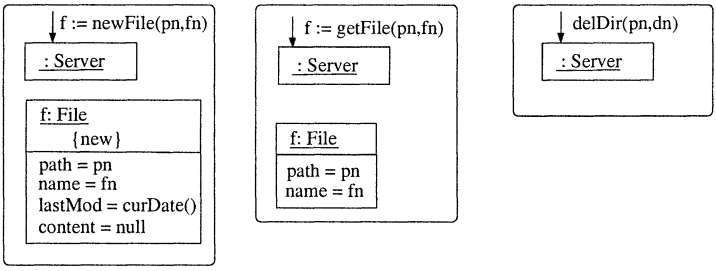

Figure 5. export actions

A concrete object structure has to be an instance of the class model where each object is of a concrete class (not interface). For a sample directory structure being such an instance consider Figure 4.

For the dynamic aspects we restrict ourselves to collaboration diagrams in this paper to describe interactions. A simplified form of collaboration diagrams which models mainly pre and post conditions of actions and their control flow, can be used to describe the import/export and internal views. In Fig. 5 three export actions of a 'FileServer' are described by collaboration diagrams.

Internally, the export can be refined by different collaborations. Two of them are shown in Fig. 6 for the deletion of directories where the left one deletes a directory only if it is empty. Directory 's' does not contain subdirectories or files, otherwise it should not be possible to deleted it due to dangling object links. In the right collaboration, a marker is set, useful to delete a non-empty directory with all its contents step-by-step. The refined parts are put to gray fill style. The marker setting is modeled by changing the value of an internal attribute. This change is indicated by a <<becomes>> relation between two states of one and the same object.

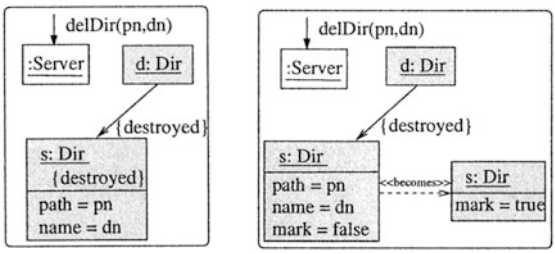

Figure 6. Two different semantics for action "delDir"

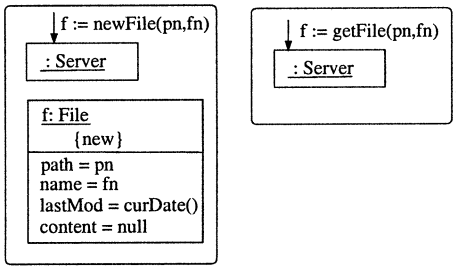

Figure 7. An import action of component 'Client'

A 'Client' component could have two import actions 'newFile' and 'getFile' as depicted in Fig. 7 to communicate with the file server. Note that import and export actions 'newFile' are equal, whereas the clients action 'getFile' is more restricted than the corresponding server action.

Moreover, there should be special collaborations describing there initialization, e.g. initializing a file server at least a 'Server' object has to be created. A detailed specification of various actions in file servers and their clients is given in (Roock, 1998), using a slightly different notation.

The interaction model should also contain failure assumptions, since their presence or absence can lead to solutions which differ heavily. In general, processes or communication channels can fail to perform actions they are supposed to. Processes can crash, i.e. a *stopping failure* can occur. Special stop actions can be added to the interaction model which can occur any time and disable all other actions of the process. E.g. interaction in component 'FileServer' can be permanently disabled by deleting object 'Server', since we assume that this object is initial.

Even worse processes can show Byzantine behavior, i.e. they behave arbitrarily. To model Byzantine behavior of processes, the failure model contains another interaction model with the same external interfaces which can replace the initial one any time.

Communication failures occur when messages are lost or duplicated or their content has changed. Modeling a communication channel as separate component, its concrete behavior is explicitly specified. Modeling a channel as queue, the message loss and duplication can be specified by a random decision for an arbitrary number of message insertions into the queue. This behavior can be modeled by a collaboration diagram using multi objects which indicate any number of messages inserted into the channel.

4. Distributed Graph Transformation as Semantic Domain

In the following, we introduce graph transformation and its distributed extension which is used to define the semantics of the visual modeling concepts discussed above. Furthermore, a restricted form of graph transformation system (GTS) components can be considered as I/O-automata, usually taken to reason about distributed systems. In this way, results for I/O-automata (Lynch, 1996) are available also for GTS components.

4.1. Graph Transformation

A type graph $Types$ together with a set $Rules$ of rules, typed over $Types$, form a graph transformation system $GTS = (Types, Start, Rules)$. For initialization purposes there is a set of start graphs $Start$ in addition which are all typed over graph $Types$. A graph G is typed over graph TG, if there is a graph morphism $t : G \rightarrow TG$. A rule $r : L \rightarrow R$ contains a left-hand side (LHS) L and a right-hand side (RHS) R and a partial graph mapping $L \rightarrow R$ which specifies the correspondence of graph objects by numerical tags. The rule is typed over TG, if both L and R are typed over TG, compatible to rule morphism $r : L \rightarrow R$. The operational semantics of a given GTS is the set of all possible graph transformations using rules of $Rules$, starting at any start graph S if specified, i.e. $opSem(GTS) = \{S \rightarrow_{Rules} G | S \in Start\}$. If $Start$ is empty, S can be any graph typed over TG.

The graphs occurring in a rule as well as the start graphs may be attributed. LHSs are allowed to have variables which have to be instantiated by concrete attribute values. Furthermore, rules may have parameters which are useful to determine e.g. matches and attributes of new graph objects by the user.

Generally, a rule can be applied to a graph at several places. In order to indicate a concrete place where the rule is applied, a rule match has to be defined. A transformation between state graphs G and G' is uniquely described by a rule r and its match m, namely $G \rightarrow_{(r,m)} G'$. Consider Fig. 8 for a schematic notation of a graph transformation step. We follow here the algebraic approach to graph transformation as presented in (Corradini et al.,

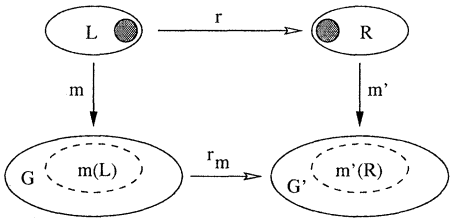

Figure 8. A graph transformation step – schematic

1997). The extension to attributed (and distributed) graph transformation has been worked out in (Fischer et al., 1999).

A rule match is a total mapping, i.e. each graph object of L is embedded into the graph G. If a variable occurs several times in a rule's left-hand side, it must be matched with the same value.

Applying a rule, an occurrence of its LHS is taken out of the state graph and replaced by an appropriate matching pattern for its RHS. Since a match is a total mapping, any graph object o of the LHS has a proper image object $m(o)$ in state graph G. If o has an image $r(o)$ in the RHS, its corresponding graph object $m(o)$ in the state graph is *preserved* during the transformation, otherwise it is *removed.* Graph objects in RHS which are not the image of a graph object in LHS are *newly created* during the transformation. Finally, the graph objects of the state graph which are not covered by the match are not affected by the rule application at all.

Besides manipulating the nodes and arcs of a graph, a graph rule may also perform computations on the graph object's attributes. During rule application, expressions are evaluated with respect to the variable instantiation induced by the actual match.

These ideas are formalized in the notion of a graph transformation step (Corradini et al., 1997) given by a so-called double-pushout, a characterization based on category theory. A graph transformation is a sequence of zero or more single graph transformation steps.

4.2. Semantics of the Architectural Model

Graph transformation is well suited as semantic domain for component, deployment and reconfiguration specification using the diagram techniques presented in section visual. The component and deployment diagrams are mapped to subgraphs of a so-called network graph. Components, interfaces and processing units form nodes where the network edges describe the relations in between. Each reconfiguration diagram can be translated into a network rule. The left-hand rule side contains all that parts of a reconfiguration not annotated by $\{new\}$ and the right-hand side all that not annotated by $\{destroyed\}$. The

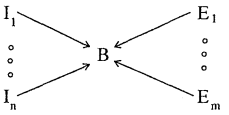

Figure 9. General structure of a GTS component

partial graph morphism between the left and right-hand side relates that part which is neither $\{destroyed\}$ nor $\{new\}$.

In the following, each graph node is refined to a graph transformation system (GTS), more precisely, each interface node will be refined to an export GTS. The export relation between interface and component is described by a GTS morphism. The semantic description of an import relation contains an additional interface node refined by an import GTS and two edges going to the component node on one side and to the export node on the other side.

4.3. Semantics of the Object Model

The abstract syntax of a class diagram can be considered as graph structure. Classes are nodes and class relations (except association classes) are arcs in a so-called type graph. An association class is represented by a node together with arcs to all class nodes where the association relates to. Class nodes may be further attributed, where each attribute is specified by a type, a name and a possible default value. Class diagrams can be formally translated to type graphs and possibly further graph constraints.

4.4. Graph Transformation System Components

A graph transformation component C consists of one body GTS B, a set of import GTS $\bigcup_{i \leq n} I_i$ as well as a set of export GTS $\bigcup_{j \leq m} E_j$, each with an embedding morphism into B. The embedding morphisms relate the type and start graphs as well as rules. Note that not all body rules must have a correspondence in each import/export GTS. See Fig. 9 for the general structure of a GTS component.

4.5. Semantics of the interaction model

Similarly to reconfiguration diagrams, graph rules can be used to formally defined collaborations. The left-hand side of a rule contains all those parts of a collaboration not annotated by $\{new\}$ and not being the target of a <<becomes>>-relation. The right-hand side contains all those not annotated by $\{destroyed\}$ and not being the source of a <<becomes>>-relation. The partial graph morphism between left and right-hand side relates the two copies of that part which is neither $\{destroyed\}$ nor $\{new\}$ as well as the object nodes related by the relation <<becomes>>.

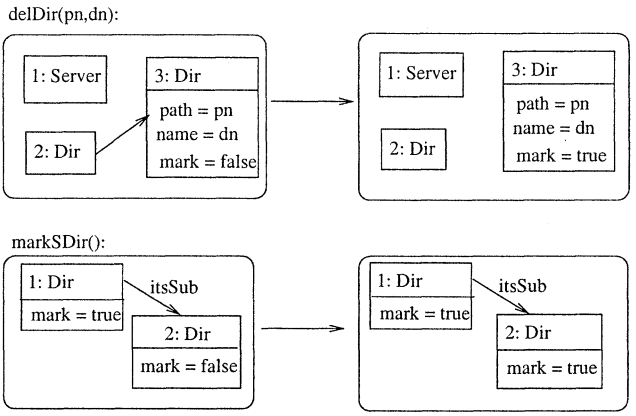

Figure 10. Sample rules for marking directories to be deleted

Considering action "delDir", the collaboration diagram on the right of Fig. 6 shows the first part of a recursive directory deletion, marking the directory to be deleted. In Fig. 10, the intended semantics of the marking action is described as graph transformation rule. Equal numbers establish the partial mapping $L \rightarrow R$. There is no edge between nodes 2 and 3 in R, since this edge is to be destroyed. Attribute values are identical if the corresponding object occurs only once in the collaboration diagram. If there is a <<becomes>> relation between two objects the change of attribute values is precisely determined in the collaboration and can be transferred into the corresponding rule.

Rules $markSDir$, $delFile$ and $delSDir$ in Figures 10 and 11 specify the rest of the whole directory deletion. First $delDir$ marks the directory to be deleted and disconnects it from its parent directory. Then, the other three rules are applied as long as possible to mark all the subdirectories and then, delete all their files and subdirectories. Lastly, directory dn itself is deleted by a rule similar to $delSDir$, but without a marked super directory. (The rule is not depicted.) Thus, a kind of garbage collection is performed.

A sample graph transformation is shown in Fig. 12. Here, rule *delSDir* is applied to the directory indicated by number 1. It has one subdirectory which is empty so that it can be deleted. We only show the attributes $lastMod$ and $mark$ of node $1 : Dir$ in the host graph, because the rule leads to a new attribute values of this node. But of course, all other nodes have a variety of attributes as specified in the class diagram in Fig. 3.

4.6. GTS Components as I/O-automata

A simple model for distributed computing is an I/O-automaton as described in (Lynch, 1996). Each I/O-automaton has three disjoint sets of actions: the input actions $in(S)$, the output actions $out(S)$, and the internal actions $int(S)$.

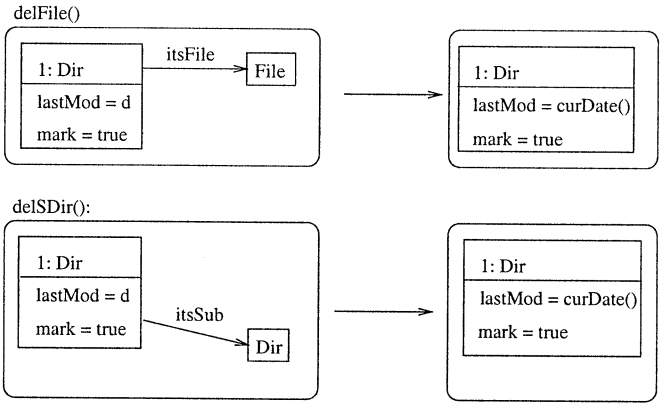

Figure 11. Sample rules for deleting files and directories

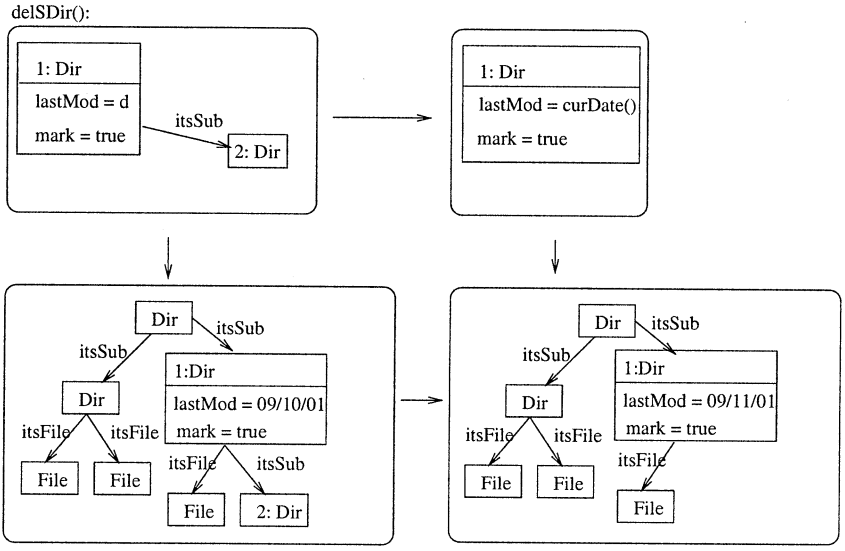

Figure 12. A sample graph transformation

An I/O-automaton consists of a set of actions $act(A)$, a (not necessarily finite) set of states $states(A)$, a non-empty subset $start(A)$ of $states(A)$, a state-transition relation $trans(A) \subseteq states(A) \times acts(A) \times states(A)$ and a task partition $tasks(A)$ being an equivalence relation on internal and output actions. There has to be a triple $(s, a, s') \in trans(A)$ for all $s \in states(A)$ and all $a \in in(A)$.

A restricted form of GTS component, i.e. *input enabled* GTS, can be considered as I/O-automaton. Body rules with related import rules are considered as *input actions*. Body rules with related export rules are considered as *output actions* and body rules without any related import/export rules are considered as *internal actions*. The states are described by graphs, start states by start graphs. A state transition is described by a graph transformation. Action a is enabled in state s if the corresponding rule is applicable to the corresponding state graph. If we restrict input actions to rules where the left-hand sides occur in any state graph produced, transitions with input actions are always enabled in any state (graph) and the GTS is called *input enabled*. E.g. rules with empty left-hand sides fulfill this condition trivially.

4.7. Composition of GTS components

For the composition of GTS components embedding morphisms are added between import and export GTS which have an import/export relation. Such an embedding morphism is a usual GTS morphism. All import GTS not connected to an export GTS form the import of the composed GTS, and similarly for export GTS. Component rules related by embedding morphisms are applied simultaneously. The composition details are described in (Fischer et al., 2000) where GTS components together with their connections to other component interfaces are called *local views*. A restricted form of GTS composition can be viewed as automata composition defined over the product of its components.

Given a countable collection $\{GTS_i\}_{i \in I}$ of input enabled GTS components we define a restricted composition of them by allowing only embedding morphisms between import and export rules where each two rules connected by an embedding morphism are named equally. For example, the 'Client' component is input enabled assuming an initial 'Server' node available in the start graph. Only this node is demanded in the left-hand sides of the import rules being the semantics of the import actions in Figure 7. Furthermore, it should be easy to understand that the set of import rules of component 'Client' can be embedded into the set of export rules of component 'FileServer'. A synchronization between two components composed in such a way, consists of the application of all connected rules simultaneously, e.g. 'getFile' rules are applied within the 'Client' as well as 'FileServer' component.

Due to corresponding results for I/O-automata executions or traces of composed GTSs can be projected execution or traces GTS components and vice versa (compare Theorems 8.1 - 8.3 in (Lynch, 1996)). Furthermore, we have built the starting point for compositional reasoning (as stated in Theorems 8.10 and 8.11 in (Lynch, 1996)).

5. Conclusions

In this contribution, we discussed a formal framework for visual modeling of distributed object systems. Starting with a subset of UML, extended by reconfiguration and import/export view facilities, we presented distributed graph transformation as semantic domain.

Two abstraction levels are offered to concentrate on the overall system architecture first and refine then to the object level where the interaction can be considered in more detail. Due to their rule-based character, graph transformation systems are an adequate means for modeling concurrent actions. Formulating only the preconditions and the effects of an action, the developer is not tempted to think too sequentially, but to cope with the intrinsic concurrency of distributed actions. Conflicts such as deadlocks can be found by analysing the system according to critical pairs (Plump, 1995), a technique known from term rewriting. Actually, graph transformation is used as semantic domain for a number of visual process specification techniques: In (Corradini et al., 1996; Baldan, 2000) Petri nets are shown to be restricted graph grammars and results for Petri nets are lifted to graph grammars. Moreover, a graph grammar-based semantics provide the basis to decide equivalence of high-level message sequence charts, as presented in (Helouet et al., 2000). The modeling of actor systems by graph transformation has been initiated by Jannssens and Rozenberg in (Janssens and Rozenberg, 1989)and further elaborated in various articles. And recently, Statecharts have been formally defined based on graph transformation in (Kuske, 2001).

Due to a new kind of diagram modeling dynamic reconfiguration being mapped to a network rule, scaling can be modeled. Since again rules are used for this purpose, the concurrency issues are well addressed also here and the basis for performance test is laid. Since the model abstracts from concrete time constraints, reasoning on performance has to be restricted to logical time. Proposing a UML-based visual model for distributed object systems, the acceptance as modeling standard for remote interfaces might increase taking into account that there are additional possibilities to publish not only the syntax but also key behavior of interface actions (described by pre/post conditions).

Moreover, we showed how failures can be explicitly modeled and thus, taken into account while reasoning. Furthermore, some sort of transparency

can be achieved by the interface views provided, i.e. access transparency can be achieved, since an input action can trigger some local computation only or start communication with further processes. Last but not least, graph transformation can also be used to model security issues, although this is not explicitly addressed in this paper, but e.g. in (Koch et al., 2000).

Altogether, graph transformation, and moreover, distributed graph transformation seems to be a promising formal framework for visual modeling of distributed object systems, powerful enough to model all key aspects with adequate abstraction. A comprehensive presentation of distributed graph transformation including the full formalization, a UML-like notation on top and a quite elaborated case example can be found in (Fischer et al., 1999). Future work is needed to evaluate the approach with further visual models like UML-RT and to use it for reasoning in concrete settings. Moreover, a distributed graph transformation engine is under development building upon AGG - a tool environment for attributed graph grammars (http://tfs.cs.tu-berlin.de/agg) and communicating by remote method invocations in Java.

References

Baldan, P. (2000). *Modelling concurrent computations: from contextual Petri nets to graph grammars*. PhD thesis, University of Pisa. available as technical report, TD-1/00.

Bestuzheva, I. and Rudnev, V. (1990). Timed Petri Nets: Classification and Comparative Analysis. *Automation and Remote Control*, 51(10):1303–1318.

Corradini, A., Montanari, U., and Rossi, F. (1996). Graph Processes. *Special Issue of Fundamenta Informaticae*, 26(3,4):241–266.

Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., and Löwe, M. (1997). Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In Rozenberg, G., editor, *Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations*, pages 163–246. World Scientific.

Fischer, I., Koch, M., and Taentzer, G. (2000). Local Views on Distributed Systems and their Communications. In Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G., editors, *Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 164–178. Springer Verlag.

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

Heckel, R. and Wagner, A. (1995). Ensuring Consistency of Conditional Graph Grammars – A constructive Approach. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2. http://www.elsevier.nl/locate/entcs/volume2.html.

Helouet, L., Jard, C., and Caillaud, B. (2000). An effective equivalence for sets of scenarios represented by HMSCs. In Ehrig, H. and Taentzer, G., editors, *Proc. Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems (GRATRA'00)*. TU Berlin, FB Informatik, TR 2000-2. Accepted for Mathematical Structures in Computer Science.

Janssens, D. and Rozenberg, G. (1989). Actor grammars. *Mathematical Systems Theory*, 22:75–107.

Koch, M., Mancini, L., and Parisi-Presicce, F. (2000). A Formal Model for Role-Based Access Control using Graph Transformation. In *Proc. of the 6th European Symposium on Research in computer Security (ESORIS 2000)*, volume 1895 of *Lecture Notes in Computer Science*, pages 122 – 139. Springer.

Kuske, S. (2001). A formal semantics of uml state machines based on structured graph transformation. In Gogolla, M. and Kobryn, C., editors, *UML 2001 - The Unified Modeling Language*, volume 2185 of *LNCS*. Springer.

Lynch, N. (1996). *Distributed Algorithms*. Morgan Kaufmann.

Lyons, A. (1998). UML for Real-Time Overview. Technical report, ObjecTime Limited.

OMG (2000). UML Version 1.3.

OMG (2001). CORBA Version 2.4.2.

Plump, D. (1995). On termination of graph rewriting. In *Proc. Graph-Theoretic Concepts in Computer Science*, volume 1017 of *Lecture Notes in Computer Science*, pages 88 – 100. Springer.

Reisig, W. (1998). *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer Verlag.

Roock, A. (1998). Visuelles Design eines verteilten Filemanagers mit Graphtransformation. Master's thesis, TU Berlin.

W3C (2001). SOAP 1.1 Specification.