Modeling Distributed Systems by Modular Graph Transformation based on Refinement via Rule Expressions *

M. Große-Rhode[1], F. Parisi-Presicce[2], M. Simeoni[2], G. Taentzer[1]

[1] TU Berlin, Germany, {mgr,gabi}@cs.tu-berlin.de
[2] Università di Roma, Italy, {parisi, simeoni}@dsi.uniroma1.it

Abstract. Due to the special requirements of distributed systems, it is important that modeling techniques for this kind of systems offer a stringent module concept. Each module has to support the encapsulation of data structure as well as functionality also at runtime. Modular graph transformation, presented in this contribution, supports these features. Modules are built up of specifications where attributed graphs describe the static data structures, whereas the dynamic behavior is modeled by the controlled application of graph rules. Rule expressions are used to formulate the control flow.
Within one module, we can state a (weak) preservation of export and import behavior wrt. the local behavior in the module's body in the sense that an interface derivation is subsumed by a local derivation if it can be performed. Modules may use each other meaning that each import interface has to be connected with an export interface in a way that the import behavior is subsumed by the export behavior.

1 Introduction

Distributed systems are a special challenge for software development concepts, languages and tools. Objects and tasks are distributed on different processing units and can only be accessed by using certain communication networks. In this context, important issues are allocation of object and tasks, remote interaction as well as object replication and migration.

Good progress has been made in the development of programming languages for distributed applications as well as middle ware providing distribution services to distributed applications. But only a limited amount of work has been reported on extending modeling techniques by new features for the description of distribution issues. For this purpose, in particular UML [UML99] and its recent extension for real-time systems [SR98] promise support which, however, seems to be insufficient, at least concerning the development of dynamic distributed object structures. Considering especially this issue, graphs and graph transformation offer good means for modeling. The data and object structures

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

can be modeled by graphs whereas the dynamic behavior is described by graph transformation.

To manage the complexity of distributed systems the reuse of specifications and software is important. Due to this point, modeling techniques should offer encapsulation and refinement concepts to support the partitioning of a system into concurrent, communicating subsystems. Several approaches to modular graph transformation have been considered (see [HEET98] for an overview on several module concepts including those for GRACE and PROGRES), mainly within an undistributed setting. Within this paper, we consider modular graph transformation with a distributed semantics. Comparing our approach with a previous one, called DIEGO systems [TS95], a module concept for simple graph transformation also coming along with a distributed semantics, our emphasis now lies on modules for attributed graph transformation with application control. The graphical notation of the modules follows the UML-notation as far as possible, i.e. wherever corresponding modeling elements are used.

Each module contains a type graph to model the object structures which are possible in principle. Parts of a type graph (which may overlap) can be declared as export interfaces. This guarantees some information hiding and simultaneously allows direct export of certain object structures. Moreover, parts of a type graph that are required from the environment may be marked as import interfaces. The graphical notation of type graphs follows that of UML class diagrams.

The dynamic behavior in a graph transformation module is described by methods. A basic operation is modeled by just one graph rule. Complex actions are specified by rule expressions which contain the control flow of certain rule applications. Rule expressions are graphically represented by restricted activity diagrams and, thus, resemble story diagrams [JZ98]. By exporting a method not only its name and parameters are published, but also a rule which describes the effect of the method on the exportable types. A subset of a module's rules may be declared as import rules. The import rules state what is needed by the body methods to be performed and can be used to search for the right exports within a network.

Graph transformation modules communicate by connecting an import of one module to an export of another. These connections concern the type graphs as well as the methods select those parts of the export interesting for the connected import. Since modules may be distributed over several sites, remote interaction takes place whenever such an import-export connection is used.

Semantically modular graph transformation is considered twice: first we investigate the behavior relations between different module components. The relations between body, export and import interfaces may also use composed rule applications, corresponding to composed derivations. The important semantic property of such a relation is the preservation of behavior in a stronger or a weaker sense: Given any derivation of e.g. an export interface, there is a corresponding derivation in the related component (body), such that the visible part of this derivation coincides with the export derivation (strong preservation), or the corresponding derivation (in the body) fails and may subsume not only the interface derivation but also further actions (weak preservation). The weak preservation guarantees at least, that nothing wrong happens; it might be, however, that nothing at all happens.

Furthermore, we take a more general point of view: since our goal is to use modular graph transformation to specify distributed systems, we introduce moreover a semantics based on distributed typed graph grammars.

Throughout this paper, all concepts are introduced along a running example which is a (simple) distributed file manager. We consider a client-server system in which the client may replicate files from the server's site. The export of the server contains replicable files, but usually the client just needs a small amount of exported files and replicates only those.

2 Controlling Rule Application by Activity Diagrams

A graph transformation specification consists of a type graph, a start graph and a finite set of methods. Type graphs describe the static structures, i.e. object types and the types of their attributes as well as relations. A method has a head consisting of a name and a parameter list, and a body. The body may be a rule or an activity diagram which may trigger rule applications. Activity diagrams are a well-known sublanguage of the modeling language UML [UML99].

As an example, we consider a file manager with its typical methods such as creating a new file, updating, copying and renaming it and possibly destroying it.

In Figure 1 the type graph as well as the method heads are shown. The start graph is empty. The bodies of the update methods are depicted in Figure 3. The other method bodies are not presented, but should be imaginable. The methods mentioned are just a subset of methods which should be supported by a file manager. Later on additional methods are introduced dealing with the replication of files in a distributed environment.

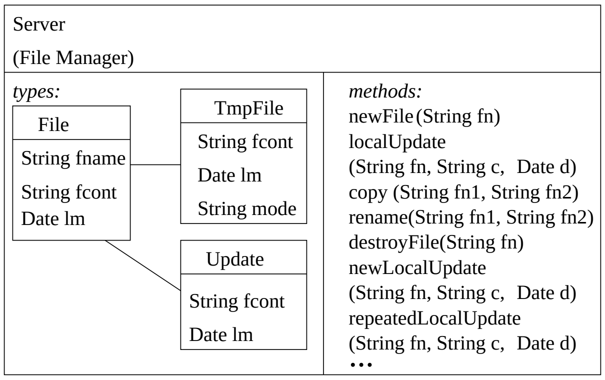

Fig. 1. A sample graph transformation specification (without method bodies)

A basic operation on object structures can be described by one rule in an *if-then* manner. The left-hand side states the premise which contains tests on the existence and non-existence of object structures whereas the right-hand side states the actions which have to be performed. If attributes are just tested and not changed on the right-hand side, they only occur on the left-hand side. On the other hand, attributes which are changed but not tested for rule application only occur on the right-hand side. There is no order among these actions except the fact that object nodes have to be inserted before they can be set into a relation, and deletion has to be done vice versa. Thus, independent basic actions may be performed in parallel.

Sequential execution of actions is expressed by several rules to be applied one after the other. To have the possibility to specify the order of rule application we use activity diagrams. We like to express at least the sequential and parallel application as well as conditional executions. Thus, in the following we use restricted activity diagrams containing the operators in Figure 2.The initial state is indicated by a small solid filled circle. A final state occurs as a circle surrounding a small solid filled circle. The basic activity of rule application is depicted by an active state containing the rule's name with the actual parameter list. The execution of already composed methods is triggered in the same way. Examples for composed methods are given in Figures 3 and 4.

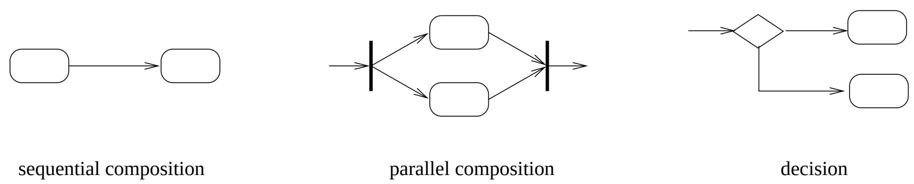

Fig. 2. Operators in restricted activity diagrams

Figure 3 describes the update storing of a file locally on the server. If the file is first updated, rule "newUpdate" is used. This condition is expressed by a negative application condition for this rule stating that there is no "Update"-node with a relational edge to the file to be updated. A negative application condition is depicted by dashed nodes and edges. In the case of a first update, a new "Update"-node is created and initialized. If the file is repeatedly updated, the "Update"-node has to exist and its attribute values are overwritten by new values.

In Figure 4, the publication of an update performed on the client or server site is modeled. Parameter m can have two values: "C2S" (from client to server) or "S2C" (from server to client). In the case of "C2S", the server's file content is updated using the parameter values provided by the client. If a server's update should be published, two cases may occur: there has not been any update, then rule "noUpdate" is applicable. Otherwise rule "serverToClientUpdate" is

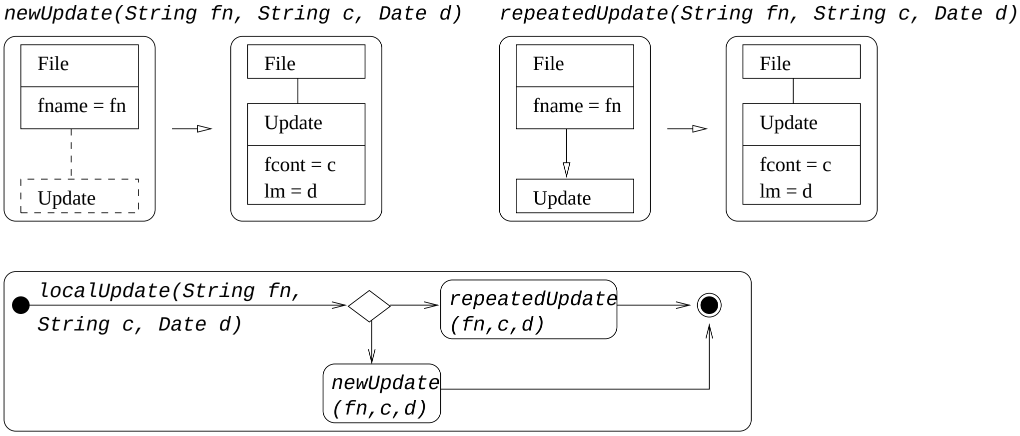

Fig. 3. A sample method performing a local update of a file on the server

triggered which exports the locally stored update. Finally, the temporary stored data for the updated file is deleted.

If a restricted activity diagram does not contain decisions we call it simple, because it can be provided with a formal semantics based on typed graph transformation specifications as given in [GPS98]. In this case, a method's body is semantically equal to a set of composed rules. See section 4 for further details where also requirements and first ideas for the extension to non-simple rules are given.

3 Systems of Graph Transformation Modules

A graph transformation module consists of a body and two sets of interfaces, namely import and export interfaces. All these module parts are typed graph transformation specifications. The type graphs of the interfaces can be embedded into the body's type graph. All export methods – which are just rules – have implementations (refinements) into a body method. Semantically, an export rule has to be subsumed by the composed rule, restricted to the export types, describing the semantics of the body method. Each import method is automatically a body method and can be used in other body methods.

One module uses another one if its import interface is related to the other's export interface. This means that the import's type and start graphs can be embedded into the corresponding graphs of the export. Moreover, each import method is a submethod of the corresponding export one.

In the following, we consider a simple module structure consisting of one server, the file manager, and one client, an application that uses the file manager. It is imaginable to extend this scenario by further clients using the file manager. Each client has an import interface to the server's export interface.

The type graphs of client and server as well as the interfaces together with corresponding relations are shown in Figure 5. Here, a typical case is shown.

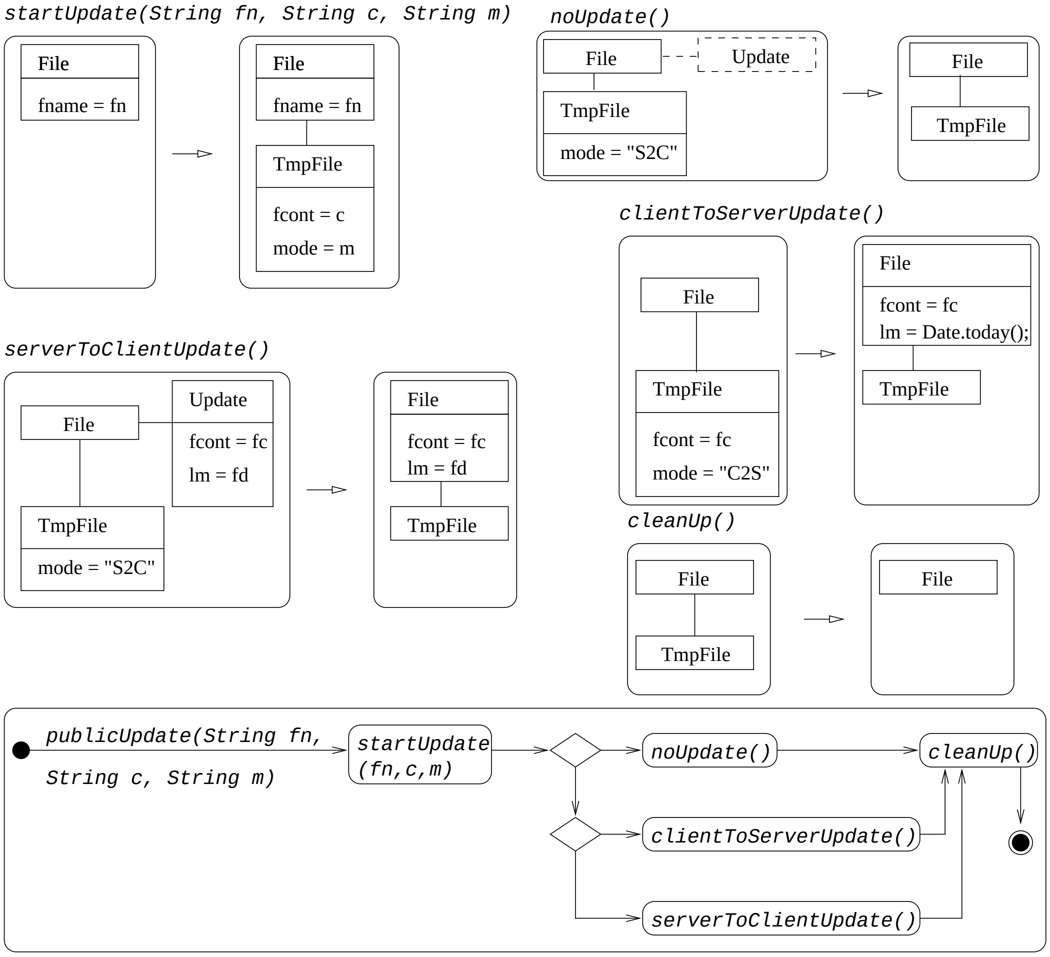

Fig. 4. A sample method performing an update of a replicated file

Client and server communicate via objects of type "File", but not all attributes are exchanged. While the file name and its contents are communicated, the date when it was modified at last and the read and write permissions are set locally. Moreover, client and server have local types such as "Dir" for directories, etc. All interface types and attributes have a grey background.

Figure 6 sketches local and distributed operations as well as the relations in between. The key operations in a distributed file manager are those which deal with export, replication and update of distributed files. While the export of a file is locally performed on the file manager, the replication of a file is local to the client. Furthermore, there are two different update operations distinguished: a local update on the server and a synchronized update on both systems. Clearly, this is possible for replicated files only. These four operations are representa-

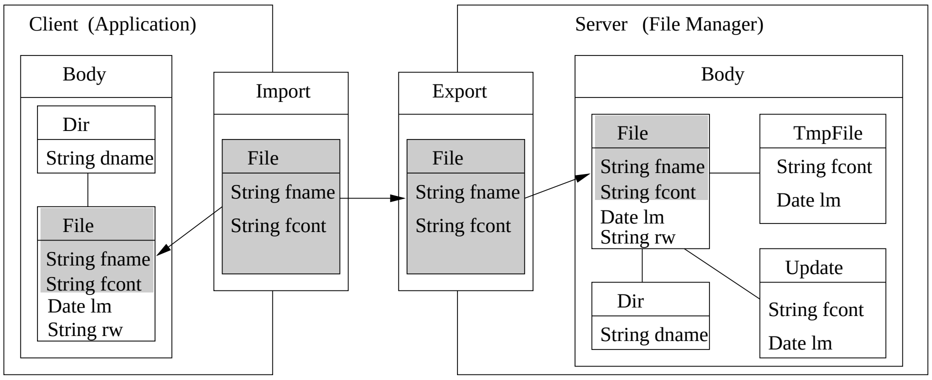

Fig. 5. Relations between type graphs

tives for each kind of distributed action that may occur in a distributed setting: (1) exporting, (2) importing, (3) local and (4) synchronization. They are shown in detail in Figures 3, 4, 7, 8, and 9. (In contrast to all the other operations, "localUpdate2 and "publicUpdate" are not just modeled as rules, but rule compositions. Although there are several simple rules to implement these operations, only the composed ones are mentioned in Fig. 6 to keep the diagram simple.)

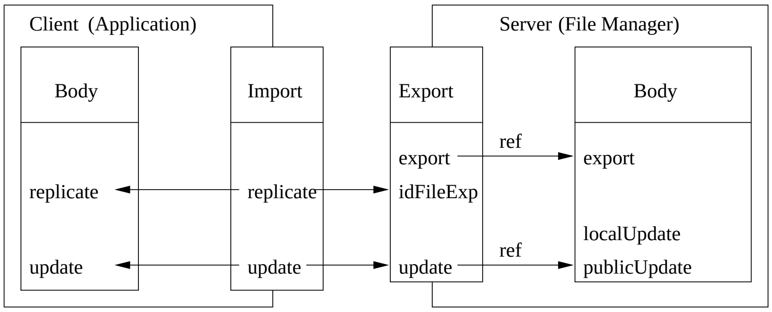

Fig. 6. Relations between module operations

In Figures 7, 8, and 9 exported as well as imported nodes have a grey background. Doing so, corresponding rules in a body and an interface can be depicted in an integrated way.

Figure 7 shows the server's operation "export". This operation can be performed locally by the server. But removing files from the export is possible only when they are not used remotely.

Figure 8 shows the rules for replication within the client's view in an integrated way. The server's export rule contains all gray graph parts. The left-hand side of the import rule is empty, the right-hand side consists of the grey part. The body rule comprises the whole rule shown in Figure 8. Since the export is

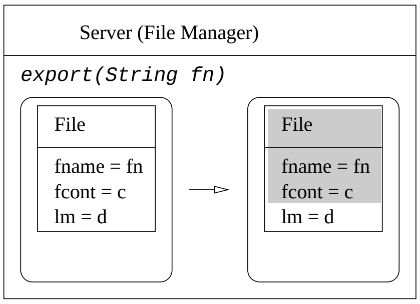

Fig. 7. Exporting a file for replication and abstract rule of "localUpdate" refinement

not changed but only used, this operation can be performed without changing the server's body.

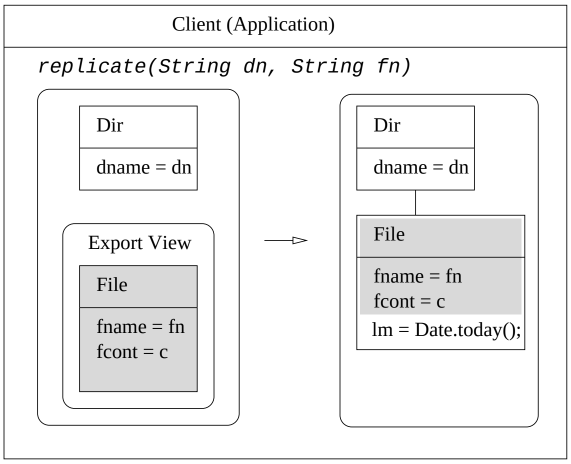

Fig. 8. Replicating a file (showing the client view in an integrated way)

In Figure 9 again the client view, here rule "update", is depicted. The server's export rule looks like the client's import rule. Update is a synchronized operation between client and server. Besides update also retrieval of information where information is requested that is not already available in the export can be modeled by synchronized rule application.

4 Semantics of modular graph transformations

In this section we describe the semantics of modular graph transformation in two steps: first we give the semantics in terms of the behavior of its components and w.r.t. the relations established between them. Subsequentially, we take a more

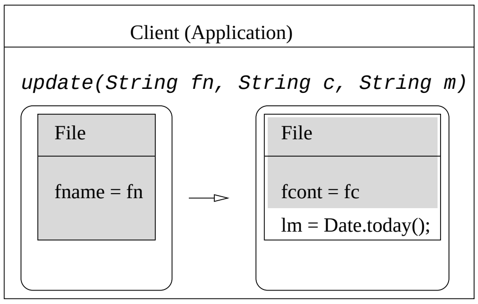

Fig. 9. *update* in the import interface and in the body of the client

general point of view: since our goal is to use modular graph transformation to specify a distributed system, we give its semantics in terms of a distributed typed graph grammar.

4.1 Behavior relations

Among the relations between graph transformation specifications described in this paper, the export–body refinement relation is the most general one: both import–body and import–export relations can be considered as special cases of refinements where each rule of the source specification is associated to just one rule of the target one.

Hence, by analyzing the behavior of two graph transformation specifications related by a refinement, both the behavior of a single module and the behavior of interconnected modules (i.e. of a system of graph transformation modules) can be described.

A refinement of a graph transformation specification is given essentially by associating with each of its rules a composition of rules of a refining system. We distinguish between *strict*, *weak* and *subrule* refinements, depending on the required relation between the original rules and the associated composed ones. The three alternatives are discussed in the following paragraphs. We point out, however, that while for strict refinements a formal theory has already been developed (see [GPS98]), for weak and subrule refinements the theory is still under development.

Strict refinement Strict refinements have been presented in [GPS98]. They require the composed rule to coincide with the translation of the original one to the type graph of the refining system. This means that the composed rule must not use private types (i.e. types not visible from the abstract specification) in the initial and final graph: they have to be hidden in the intermediate steps of the composition.

In [GPS98] the rules are simple DPO rules without attributes and application conditions, and the basic composition operations on graph transformation rules are sequential composition by concatenation and parallel composition by amalgamation, corresponding to temporal and spatial refinements, respectively. In that case a *strict preservation of behavior* property can be proved, stating that each derivation of the more abstract specification can be translated along the refinement into a composed derivation of the more refined one, whose visible part (i.e. its restriction to the smaller type graph of the abstract specification) coincides with the given abstract one.

According to [Koc99] and [Gro99], this theory can easily be extended to the more general case of DPO rules with attributes and application conditions.

With this extension, it would be nice to use application conditions for modeling case distinctions, like *if–then–else* and *case* constructs. The idea would be to map each abstract rule p into a set of concrete rules $\{p_1, \dots, p_n\}$, representing different implementations of p in different contexts, specified by the application conditions of each p_i. This however is not possible for strict refinements because of the condition they require between the abstract rule p and each of its implementation p_i: the translation of p over the refining type graph should coincide with p_i and that means that there could be just one implementation (i.e. no alternatives are possible).

Weak refinement A weak refinement demands a weaker condition than a strict one: it requires the restriction of the composed rule to the more abstract type graph to coincide with the original rule. In this way the composed rule can use private types of the refining specification, and only its visible part over the more abstract type graph has to coincide with the original rule.

On one side this relaxed condition allows to model *if–then–else* and *case* constructs as explained above: an abstract rule is mapped into a set of concrete rules (representing different implementations of it) and the requirement stated above ensures that the visible part of each refining rule over the abstract types coincides with the abstract rule. The same idea can also be used to model refinement by iteration, by mapping an abstract rule to the set of iterations of all lengths.

On the other side, the preservation of behavior property as stated for strict refinements does not hold any longer, and only a *weak preservation of behavior* can be proved. Consider a direct derivation on the more abstract system: the applied rule corresponds to a composed one in the refining specification but, since it can use private types, there need not be a match for it. However, if there is a match and its visible part over the abstract types coincides with the abstract match, then also the visible part of the refining derivation coincides with the abstract one. Thus, the refining system may fail to realize the required behavior, but it does not yield wrong results.

For example, the export view of rule *update* in Figure 9 (server side) and the composed rule *publicUpdate* in Figure 4 are related by a weak refinement. Moreover, the import and body views of rule *update* in Figure 9 (client side) are related by a simple one-to-one weak refinement, too.

Subrule refinement A subrule refinement allows an even more relaxed situation than a weak refinement: it just requires the abstract rule of a refinement to be a subrule of the composed rule restricted to the more abstract type graph.

The subrule relation between two rules is modeled by a rule morphism (i.e. by three graph morphisms relating the three graphs of a rule) and it can be required that the resulting squares are either pullbacks or simply commuting diagrams. In both cases (pullbacks or commuting diagrams) a *subsumption of behavior* property can be proved: if an abstract rule and a composed rule are related by a subrule relation and there are two matches for them such that the visible part of the more concrete match over the abstract types coincides with the abstract one, then also the abstract derivation is subsumed by the visible part of the refining derivation. Even in this case, a match for the composed rule need not exist, i.e the refining system may fail to realize the required behavior, but it does not yield wrong results.

Rule morphisms requiring pullbacks yield obviously a stronger subsumption of behavior than rule morphisms requiring simply commuting diagrams. On the other hand, from the viewpoint of applications, requiring pullbacks means that only synchronous actions can be modeled, while requiring simply commuting diagrams means allowing asynchronous actions, too. In our running example subrule refinements are used to model import–export relations (these are only one-to-one subrule refinements). Both, synchronous (see the import and export views of rule *export* in Figure 7) and asynchronous actions (see the import and export views of rule *replicate* in Figure 8) are present. Moreover, subrule refinement is used within the bodies to model complex local actions. E.g. to define a refinement relation for the operation "localUpdate" defined in Fig. 3 there has to be has to be a circular refinement relation on the server's body. The abstract rule is shown in Fig. 10.

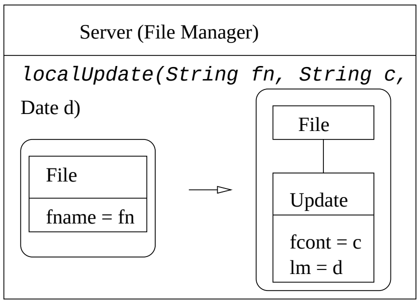

Fig. 10. Abstract rule of "loclaUpdate" refinement in Fig. 3

4.2 Distributed semantics

Our proper aim is to use modular graph transformations for specifying distributed systems. Using distributed typed graph grammars, modular graph transformation can be provided by a distributed semantics in the following sense: Each module has its own local state and can run independently of other modules. To perform synchronous communication certain transformation steps have to be performed in parallel with corresponding steps of other modules.

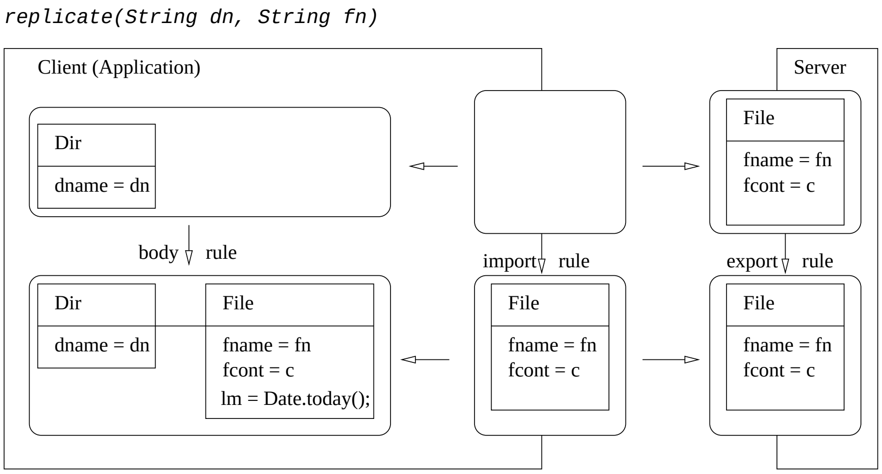

Fig. 11. Detailed version of rule replicate in Figure 8

The schema of a modular graph transformation is a graph having graph transformation specifications as nodes and the relations between them as edges (the schema of our running example is shown in Figure 5 concerning the type graphs and in Figure 6 concerning the module behavior). From the point of view of the distributed system specified by the modular graph transformation, this graph describes the system structure (local components and network connections): it is called *network graph* and we use it for determining the distributed type graph, distributed initial graph and the set of distributed rules characterizing a distributed typed graph grammar.

More precisely, the type graphs (resp. initial graphs) and type graph morphism of all the nodes and edges in the network graph define the distributed type graph (resp. initial graph), while the (composed) rules and relation between them define the distributed rules. Note that a distributed rule shows an import, an export, a local action or a synchronized action between two modules, i.e. each distributed rule is composed of those local rules which are related with each others (consider Figure 6). For example, the distributed rule associated to rule "replicate" is shown in Fig. 11.

Given a distributed rule, a distributed match for it is given by all the matches of the corresponding local rules, and a distributed step consists in applying all the local matches. In [TFKV99], it has been shown that if the distributed match satisfies the so called *distributed gluing conditions*, then the components of the distributed rule can be applied locally and yield a consistent distributed state as result of all these applications.

5 Related Work and Tool Support

Modular graph transformation as presented in this contribution is based on distributed graph transformation. For modular graph transformation, we consider a fixed communication mechanism based on import and export interfaces, visibility constraints for rules and types as well as refinement relations between export and body parts. Two larger case studies for distributed graph transformation have been developed until now: a distributed file manager [Roo98], a small section of it is used as running example here, and a distributed configuration management system [TFKV99]. Both specifications apply distributed graph transformation with corresponding restrictions mentioned above, thus modular graph transformation is used.

Comparing modular graph transformation to other types of module concepts for graph transformation, this is the only one which comes along with a distributed semantics and some kind of control flow, even if it is quite restricted at the moment. (Compare the introduction.)

Simple graph transformation offers just one concept for describing the system behavior: rules. To model a distributed action on a certain level of abstraction, graph rules are attractive and sufficient to describe the behavior. Using rules the actions are clearly modeled and global control flow is modeled locally within the rules. But to model complex local actions, it soon becomes convenient to describe the control flow explicitly. If we allow e.g. operations on directories, the actions of a distributed file manager may soon become quite complex. In [Roo98] rather complex control flows for rule application are specified using Java[1] expressions. Especially concerning control flow we have to mention approaches to modular specifications on the basis of e.g. state charts (O-charts [HG96]) or Petri nets [GGW98] which may have advantages, but they hardly support convenient graphical modeling of data structures.

Furtheron, a recent extension of UML for real time applications [SR98] offers modeling concepts for a modular system design with explicit import and export interfaces, called ports. The modularity units are called capsules which communicate only via ports. Similar to classes, capsules may have several instances which is not possible in our approach until now. While the system structure may be modeled well in this UML extension, there is little support in modeling the system dynamics in a modular way.

To test the practicability of this approach we are currently working on an implementation of the presented ideas based on the graph transformation machine AGG [TER99][2]. AGG performs algebraic graph transformation with negative application conditions for rules. The graphs may be attributed by Java objects

[1] For more information concerning Java see: `http://www.javasoft.com`
[2] See also: `http://tfs.cs.tu-berlin.de/agg`.

and the rules may contain Java expressions as attribute values which are evaluated at application time. The AGG environment comprises graph, rule, attribute and graph grammar editors as well as a graph transformation interpreter. Currently AGG is extended in such a way that import and export interfaces for graphs, rules and graph grammars are realized as substructures of those. The main attention lays on the synchronization of local transformations related by subtransformations in a setting of several AGG-systems possibly running on different hosts. Each AGG-system may hold one or more graph grammars. The communication between import and export interfaces will be realized by remote method invocations (RMI) supported by Java.

6 Conclusion

Modeling distributed systems by modular graph transformation offers visual concepts to describe the network structure, distributed data structures as well as the dynamic behavior. The stringent module concept supports the reuse of specification parts. A module system is defined in a way that subsumption of interface behavior is guaranteed in each module. The behavior of import and export interfaces may be weaker related meaning that import interfaces may start or stop using export items, although the export is not changed.

Modular graph transformation in the double-pushout approach as presented here, supports the modeling of safe distributed systems where a server always knows which parts of its exports are imported by which client. As long as an information is needed, it is exported. This suits very well to model e.g. distributed software management tools [TFKV99]. Modeling e.g. WWW applications may be better supported by using the single-pushout approach to modular graph transformation where exported items may be deleted or updated without informing the using clients [Koc99].

This modeling approach is general enough to consider not only client-server systems, but also other kinds of distributed systems, e.g. broadcasting systems. In this contribution, we have not considered network reconfiguration. A next step is to allow several instances of modules and their creation during runtime leading to a highly dynamic networks. This aspect can also be nicely described by distributed graph transformation [TFKV99] which already serves as a semantical basis for modular graph transformation.

References

[GGW98] H. Giese, J. Graf, and G. Wirtz. Modeling Distributed Software Systems with Object Coordination Nets. In *Proc. Int. Symposium on software Engineering for Parallel and distributed Systems (PDSE'98),Kyoto, Japan*, pages 107–116, jul. 1998.

[GPS98] M. Große–Rhode, F. Parisi Presicce, and M. Simeoni. Refinements of Graph Transformation Systems via Rule Expressions. In G. Engels and G. Rozenberg (eds.), *Proc. 6th International Workshop on Theory and Applications* *of Graph Transformation (TAGT'98)*, number tr–ri–98–201 in Reihe Informatik, pages 96–103. Universität–Gesamthochschule Paderborn, Fachbereich Mathematik–Informatik, 1998.

[Gro99] M. Große–Rhode. Specification of State Based Systems by Algebra Rewrite Systems and Refinements. Technical Report 99–04, TU Berlin, FB Informatik, March 1999.

[HEET98] R. Heckel, H. Ehrig, G. Engels, and G. Taentzer. Classification and comparison of modularity concepts for graph transformation systems. In *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98)*, 1998.

[HG96] D. Harel and E. Gery. Executable object modeling with Statecharts. In *Proceedings of ICSE-18*. IEEE, 1996.

[JZ98] J.-H. Jahnke and A. Zündorf. Specification and Implementation of a Distributed Planning and Information System for Courses based on Story Driven Modelling. In *Proc. of the 9th Int. Workshop on software Specification and Design, Ise-Shima, Japan*, pages 77–86. IEEE Computer Society, 1998.

[Koc99] M. Koch. *Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems.* PhD thesis, Technische Universität Berlin, FB 13, 1999. to defend.

[Roo98] A. Roock. Visuelles Design eines verteilten Filemanagers mit Graphtransformation. Master's thesis, TU Berlin, 1998.

[SR98] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Rational Software Corporation, available at `http://www.rational.com`, 1998.

[TER99] G. Taentzer, C. Ermel, and M. Rudolf. AGG-Approach: Language and Tool Environment. In G. Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Specification and Programming*. World Scientific, 1999. to appear.

[TFKV99] G. Taentzer, I. Fischer, M. Koch, and V. Volle. Visual Design of Distributed Systems by Graph Transformation. In G. Rozenberg, U. Montanari, H. Ehrig, and H.-J. Kreowski (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency and Distribution*. World Scientific, 1999. to appear.

[TS95] G. Taentzer and A. Schürr. DIEGO, Another Step Towards a Module Concept for Graph Transformation Systems. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2, 1995. http://www.elsevier.nl/locate/entcs/volume2.html.

[UML99] Rational Software Cooperation. Unified Modeling Language. available at `http://www.rational.com`, 1999.