Semantics of Distributed System Specifications based on Graph Transformation*

Gabriele Taentzer, Hartmut Ehrig
Technical University of Berlin, Germany,
{gabi,ehrig}@cs.tu-berlin.de

Abstract

Distributed algorithms and dynamic distributed data structures are basic aspects of distributed systems which usually are modelled separately by different specification techniques. In the running example of the mutual access to file systems, typical problems are shown which are not handled by separate modelling. This paper discusses how these problems can be solved by integrated specification of distributed algorithms and dynamic data structures by graph transformation. From the component point of view it is even more adequate to use distributed graph transformation for the semantics of distributed system specification.

1 Introduction

Specification of distributed systems requires techniques which can cope with their typical problems, like access to shared resources, communication with remote processes, allocation, replication and migration of objects. There are several kinds of distributed systems according to their timing model (synchronous or asynchronous) and the interprocess communication method (shared memory or messages). According to [Lyn96], shared memory systems lead to pretty much the same fundamental results as message-based systems and both kinds of systems can be transformed into each other. We follow the shared memory approach to reason about the semantics of distributed system specifications. In [Lyn96], shared memory systems are modelled by I/O-automata where states and transitions are not refined. In this paper, we consider graph transformation systems which are suitable for the specification of dynamic data structures. Moreover, they can be used as refined model of I/O-automata where states are modelled by graphs and transitions by rule-based graph transformations. This approach has several advantages: graphs are natural means to model any kind of data or object structure, a graph rule can adequately describe the precondition and the effect of a transition without implementing it and graphs as well as

*Research partly supported by the ESPRIT Basic Research Working Group APPLIGRAPH, TMR network GETGRATS, and DFG

graph transformation are formal modelling techniques which also support intuitive visualizations of states and transitions. In this way, graph transformation is a good candidate for integrated specification of distributed algorithms and dynamic data structures and, moreover, for the definition of a visual specification language for distributed systems. It can be used to describe the abstract syntax as well as the semantics of such a visual language. In this paper, we concentrate on the semantics without discussing a concrete language.

As a typical example for a distributed algorithm which is dependent on dynamic data structures we show that mutual access to a file system can be modelled by graph transformation. Since graph transformation is a well-established formalism with a rich theory (following the algebraic approach), it is possible to argue for the validation of desired properties such as well-formedness of executions, progress, termination, etc. in graph transformation-based specifications. Two validation approaches can be found already in [Koc96, Koc99]. However, these validation issues are outside the scope of this paper.

In Section 2 of this paper, we review the basic ideas how distributed algorithms are modeled by I/0-automata ([Lyn96]) and present a simplified version of our running example, the mutual exclusion for one resource. The basic ideas of graph transformation in the sense of the algebraic approach [CMR+97] are reviewed in section 3 and applied to the specification of dynamic data structures. Section 4 discusses how distributed algorithms modeled by I/0 automata in Section 2 can be modelled by graph transformation. Moreover, we show how typical problems in modelling mutual access to file systems – not handled in the I/0-automata model of section 2 – can be solved by integrated specification of distributed algorithms and dynamic data structures using graph transformation. Finally in Section 4, we sketch how to model shared memory systems using the more advanced concept of distributed graph transformation ([Tae96, TFKV99]). Here, graph transformation is employed twice: to describe the network of distributed components and their dynamic reconfiguration, and to model evolving data and object structures within the components. In Section 5, we summarize the main ideas of this paper, discuss the advantages of using graph transformations, and point out directions of future research.

2 Distributed Algorithms Modelled by I/O-Automata

A simple model for an asynchronous shared memory system is an I/O-automaton as described in [Lyn96]. Assuming that the processes in the system are indexed by $1, ..., n$, each process i has an associated set of states, named $states_i$, among which each process has start states $start_i$. Each shared variable x has an associated set of values, $values_x$, among which each variable has start values $initial_x$. A global system state consists of a state in $start_i$ for each process i as well as a value in $initial_x$ for each shared variable x.

Each global transition inferred by some action a satisfies some locality restriction due to the process and variable structure of the system.

1. Action a is associated with process i but with no variable: Transitions can be generated from triples of the form (s, a, s') where $s, s' \in states_i$. All other processes can be in any state and all variables can have any value.

2. Action a is associated with process i and with variable x: Transitions can be generated from triples of the form $((s, v), a, (s', v'))$ where $s, s' \in states_i$ and $v, v' \in values_x$. All other processes can be in any state and all other variables can have any value.

The only type of failures we consider in the following are stopping failures occurring in processes. They are modelled by a special input action $stop_i$, which stops process i permanently but does not affect any other processes.

In the following, we review a typical example from [Lyn96] which is extended later in Section 4.

2.1 Specification of a Resource Allocation Algorithm

The mutual exclusion problem means managing access to a single indivisible resource (e.g. a printer, a database or other data structure) that can only support one user at a time. A user with access to the resource is modelled as being in a *critical region* (C), which is simply a designated subset of states. When a user is not involved somehow with the resource, it is said to be in the *remainder region* (R). In order to gain admittance to its critical region, a user executes a *trying* protocol (T), and after it is done with the resource, it executes an (often trivial) *exit protocol* (E). This procedure can be repeated, so that each user follows a cycle: R $\rightarrow$ T $\rightarrow$ C $\rightarrow$ E $\rightarrow$ R.

Each user corresponds to one process in the shared memory system, the inputs to a process i by user U_i are the following actions: try_i which models a request by user U_i, and $exit_i$ which models an announcement that user U_i is done with the resource. The output actions of process i are $crit_i$ modelling the granting of the resource to U_i and rem_i telling U_i that it can continue with the rest of the work.

A shared memory system solving the mutual exclusion problem for a given collection of processes has to fulfill the following safety and liveness properties:

- *Well-formedness:* In any execution, and for any process i, the cyclic order of process controlled actions try_i, $crit_i$, $exit_i$, rem_i, try_i, ... is obeyed.

- *Mutual Exclusion:* There is no reachable system state in which more than one process is in the critical region C.

- *Progress:* At any point in a *fair execution*:

  1. (Progress for the trying region) If at least one process is in T and no process is in C, then at some later point some process enters C.

2. (Progress for the exit region) If at least one process is in E, then at some later point some process enters R.

A concrete algorithm which implements the mutual exclusion problem is the *Peterson* algorithm [Pet81]. Here, a process i starts to enter the critical region by setting its flag $flag(i)$ to 1, and it proceeds setting $turn$ to i. Then it waits until either the others process' flag $flag(\bar{i})$ is 0, or else that $turn \neq i$. That means, either the other process is not interested to enter the critical region or it tried to enter more recently. The shared variables are $turn \in \{1,2\}$, initially arbitrary, and $flag(i) \in \{0,1\}$, initially 0. The actions process i has to perform are: try_i; $flag(i) = 1$; $turn = i$; $waitfor\ flag(\bar{i}) = 0\ or\ turn \neq i$; $crit_i$, then critical actions can follow, $exit_i$; $flag(i) = 0$; rem_i, and any action in the remainder region. It is rather straight forward to construct an I/O-automaton for this algorithm, e.g. in [Lyn96]. Moreover, safety and liveness properties have been shown for this algorithm there.

Resource allocation generalizes the problem by including many resources instead of one. The mutual exclusion problem addresses the problem of allocating a certain resource as well as ensuring that only one user can access it at one time. Having several resources we have to specify which resources are required to be available all together and which actions exclude each other (exclude specification). Here, the mutual exclusion property is generalized in the following way: There is no reachable system state in which the actions of the set of all processes in their critical regions are excluded by the exclude specification. An extension of the *Peterson* algorithm to arbitrary many resources is given in Section 4 by graph transformation.

3 Specification of Dynamic Data Structures by Graph Transformation

System models based on graph transformation describe state transition systems where states are modelled by graphs and transitions by graph transformations (see [Roz97]).

The nodes and arcs of a graph may be typed and attributed, where each attribute is specified by a type, a name and a value. Each graph node and arc may have several attributes. A state graph is typed, if it is an instance of a certain type graph. In the type graph, each node and arc type occurs once and for arc types their source and target node types are declared.

Each system action is modelled by a graph rule. Applying it to a certain state graph yields a state graph again. Rules contain of a left-hand side (LHS) and a right-hand side (RHS). The graphs occurring in a rule are typed and attributed: LHSs are allowed to have variables which have to be instantiated by concrete attribute values. The left and the right-hand sides of a rule are related by a partial graph mapping $L \to R$ which specifies the correspondence of graph objects by numerical tags. Furthermore, rules may have parameters which are useful to determine e.g. matches and attributes of new graph objects

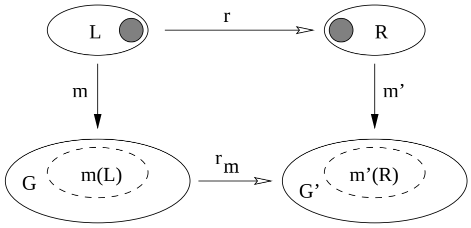

Figure 1: A graph transformation step – schematic

by the user. A rule may contain a set of *negative application conditions (NAC)* which are able to express some graph part that *must not* exist for a rule to be applicable. Basically, this is realized by adding to the rule another dashed graph part which represents the part that must not occur.

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. 1 for a schematic notation of a graph transformation step. We follow here the algebraic approach to graph transformation as presented in [CMR+97]. The combination of attributed graph transformation with negative application conditions has been worked out in [TFKV99].

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. Furthermore, all its NAC's and further attribute conditions have to be satisfied.

Applying a rule, 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 object o of the LHS has a proper image object $m(o)$ in state graph G. Now if o has an image $r(o)$ in the RHS, its corresponding object $m(o)$ in the state graph is *preserved* during the transformation, otherwise it is *removed*. Objects in RHS which are not the image of an object in LHS are *newly created* during the transformation. Finally, the objects of the state graph which are not covered by the match are not affected by the rule application at all.

These ideas are formalized in the notion of a graph transformation step (see [CMR+97]) given by a so-called pushout, a characterization based on category theory. A graph transformation is a sequence of single graph transformation steps.

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

A type graph together with a set of rules form a graph transformation system (GTS). For initialization purposes there might be a start graph in addition. The operational semantics of a given GTS is the set of all possible graph transformations, starting at any state graph compatible to its type graph or at the

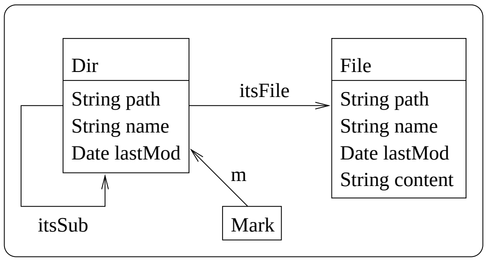

Figure 2: A type graph for file structures

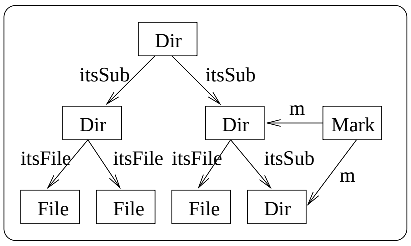

Figure 3: A sample file structure

start graph if specified.

3.1 Specification of Dynamic File Structures

Similarly to entity-relationship or class diagrams we model file structures by graphs which contain separate node types for each kind of object. Files are organized in directories which may have subdirectories additionally. During their life time file structures are heavily dynamic: Typical actions are creating new directories or files, the deletion of those, moving files to another directory, copying, etc. Each of these actions can cause non-trivial modifications of the file structures.

To rely on a file management system the satisfaction of certain consistency properties is required, e.g. each file belongs to exactly one directory, the whole file structure is connected, moreover, a file structure forms a tree, etc. This means that the corresponding file structures ensure certain graph properties which have to be still satisfied after dynamic changes.

In Fig. 2, a simplified type graph for file structures is depicted. Here, an additional type *Mark* occurs which marks directories for deletion. Figure 3 shows an instance to this type graph without attributes. The two rightmost directories are marked for deletion.

In the following, we consider just two modifying graph rules, namely the insertion of a new file and the deletion of a directory, to give a flavour of specifying by graph transformation. Rule $newFile$ in Figure 4 inserts a file with the given name na if such a file does not already exist. Rules $markDir$, $markSubDir$, $delFile$ and $delDir$ in Figures 5 and 6 delete a directory together with its whole

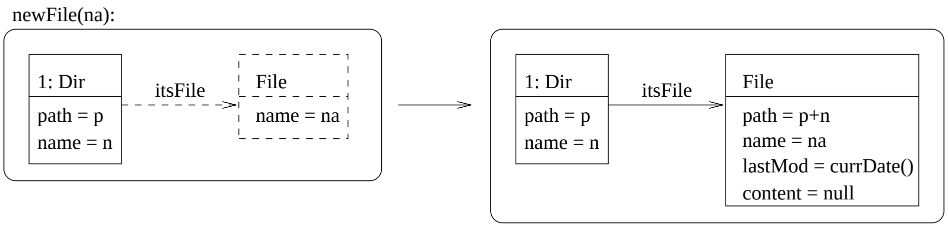

Figure 4: A sample rule for the insertion of a new file

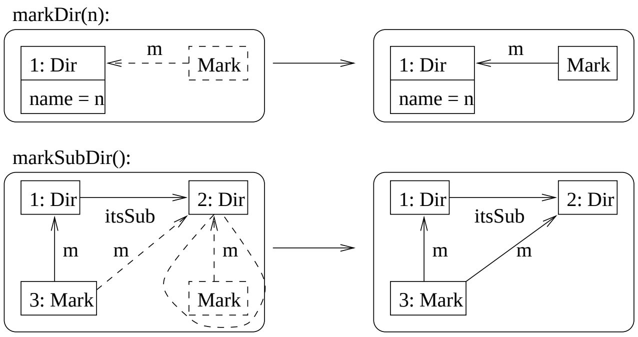

Figure 5: Sample rules for marking directories to be deleted

content. First *markDir* is called for a directory with name n. Due to the NAC this rule can only be applied once. Then, the other three rules are applied as long as possible to first mark all the subdirectories and then, delete all their files and subdirectories. Lastly, directory n itself is deleted by a rule similar to *delDir*, but without a marked super directory.

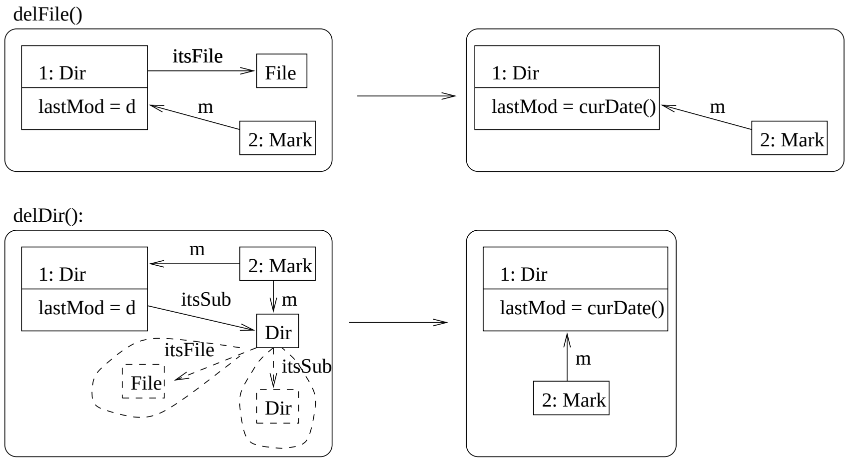

Figure 6: Sample rules for deleting files and directories

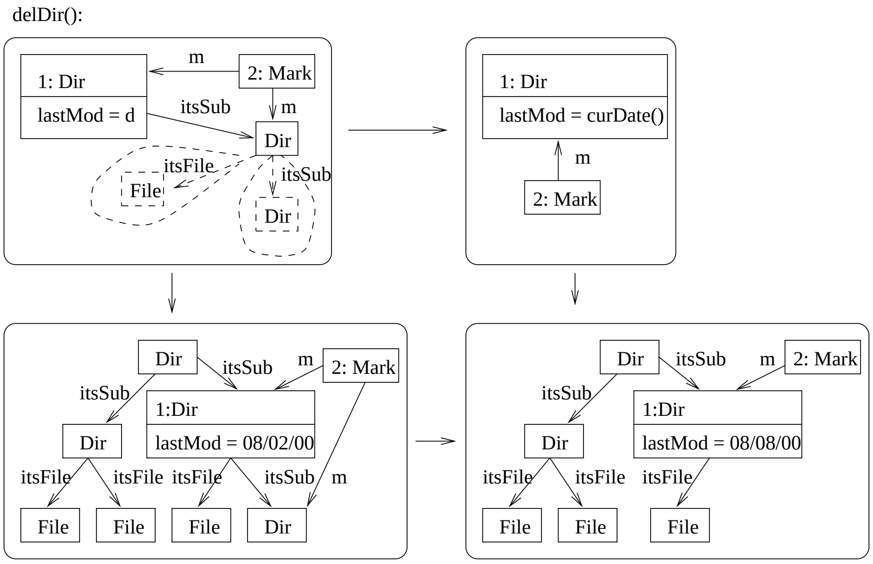

Figure 7: A sample graph transformation

A sample graph transformation is shown in Fig. 7. Here, rule *delDir* is applied to the directory marked by number 1. It has one subdirectory which is empty and already marked so that it can be deleted. Note that the two NACs are satisfied since the subdirectory to be deleted contains no file and no other subdirectory. We only show the attribute $lastMod$ of node $1 : Dir$ in the state graph to which the rule is applied, because the rule leads to a new attribute value of this node.

A detailed specification of various actions in file system based on graph transformation is given in [Roo98].

4 Integrated Specification of Distributed Algorithms and Dynamic Data Structures by Graph Transformation

In the distributed algorithm specified in Section 2, the shared variables model shared resources on an abstract level where all resources are considered to be independent. Any data or object structure as described in Section 3 may be a possible resource to be shared among processes. In order to have the possibility of specifying all requirements and effects which the sharing of complex resources might have, we discuss now an integrated specification of both aspects on the basis of graph transformation. Fortunately, it is not very difficult to lift an I/O-automaton to a GTS. Doing so we have the possibility to combine the graphs and rules of both GTSs to model combined requirements and effects.

Similarly to the shared memory model based on I/O-automata (see Section 2), a global state graph G can be partitioned into subgraphs G_i such that each of them belongs to one process i. Moreover, all shared variables are incorporated as special graph parts which might be connected. All initial process and variable graph parts together form the start graph of the combined GTS.

Each global graph transformation inferred by some rule r and its match m has to satisfy certain locality restrictions due to the process and variable structure of the system. Similarily to Section 2, we distinguish two kinds of actions in shared memory systems:

1. Local action: Rule r models an action of process i, i.e. it can be applied to subgraph G_i which specifies the state of process i. Rule r does not change any variable part. The remaining part of the graph modelling states and values of all other processes and all variables, is not changed.
2. Access of process i to shared variables: The application of rule r models not only a state modification of process i but also some variable changes. Furthermore, variables might be deleted or newly created and dependencies between variables might change. Their graph parts have to correspond to certain initial (final) states to be created (deleted) by process i. The remaining part of the graph modelling states and values of all other processes and all variables, is not changed.

The stopping failure for a graph transformation system can be considered as a rule application which permanently disables the system to apply a rule afterwards. The stopping failure is modelled by a rule changing the current state graph in a sense that no further rule application wrt. the stopping processes, is possible. This rule application stops all processes encouraged in the current graph transformation, but does not affect any other process.

4.1 Specification of Mutual Access to Directories and Files

Considering the specification of dynamic file structures in Section 3, we did not discuss the mutual access of processes to these resources. Since we have arbitrary many resources (directories and files) here and since they can depend hierarchically on each other, we have to extend the mutual exclusion algorithm given in Section 2. The main ideas of an integrated model consisting of such a generalized resource allocation algorithm, the specification of dynamic file structures and the instantiation of abstract resources by directories and files is presented in the following. This integrated model is based on graph transformation.

Processes, resources, flags and turns are represented by nodes, if they are related to each other there are arcs in between. If a process is connected to a resource by a flag, this means that the resource is requested or hold by this process. Each turn node instead is connected to exactly one process node which means no permission for the process which set turn more recently to access the resource. The turn node is interesting only if both processes are trying to get the resource at the same time. In a state graph, a resource node should be

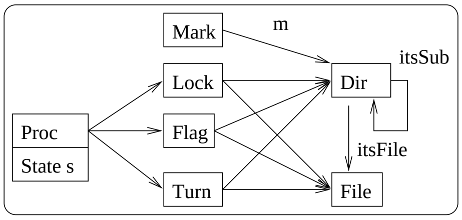

Figure 8: Type graph for mutual access to file structures

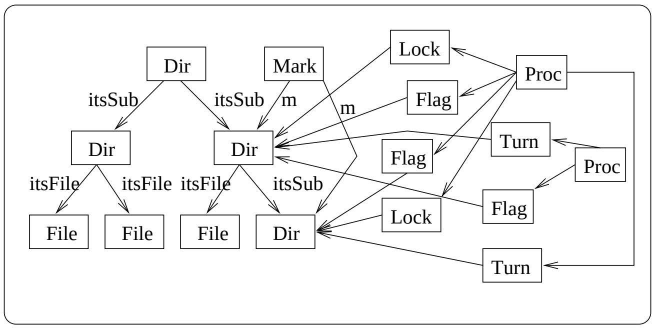

Figure 9: An instance graph showing a state of access to a file structure

always connected to at most one process node via a *turn*-node, otherwise mutual access is not guaranteed. Consider Figure 8 where the type graph in Figure 2 is extended by additional nodes and arcs for mutual exclusion organization. Due to lack of space all arc type labels and the attribute declaration for directories and files (given in Figure 2) are left out. A *Proc* node has its process state s as attribute. Attribute s is of enumeration type *State* with the following values: sf (set-flag), st (set-turn), cft (check-flag-and-turn), lt (leave-try), $crit$, re (reset), le (leave-exit), and rem.

In Figure 9, the state graph of Figure 3 has been extended to show a certain state of access to directories and files. There are two processes where the upper one holds access to two directories indicated by locks. The lower process waits for access to a directory. Note that all directories and files should have a connection to one turn node although only some of them are depicted in Figure 9.

Now, we specify graphically a generalized form of the Peterson algorithm discussed in Section 2 for arbitrary many resources which might depend on each other. Furthermore, the abstract resources and actions in the critical region are instantiated by directories and files and their modifications. Input and output actions are not modelled in the following due to space limitations, but can be modelled by rules, too.

To enter the trying region, rule $try()$ has to be applied. Note that this rule can always be applied. In the trying region a process has to set a flag and the turn. This is modelled by the four rules depicted in Figure 11. Thereafter, the

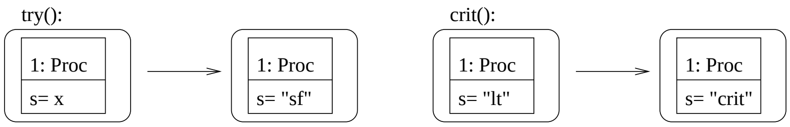

Figure 10: Graph rules to enter and to leave the trying region

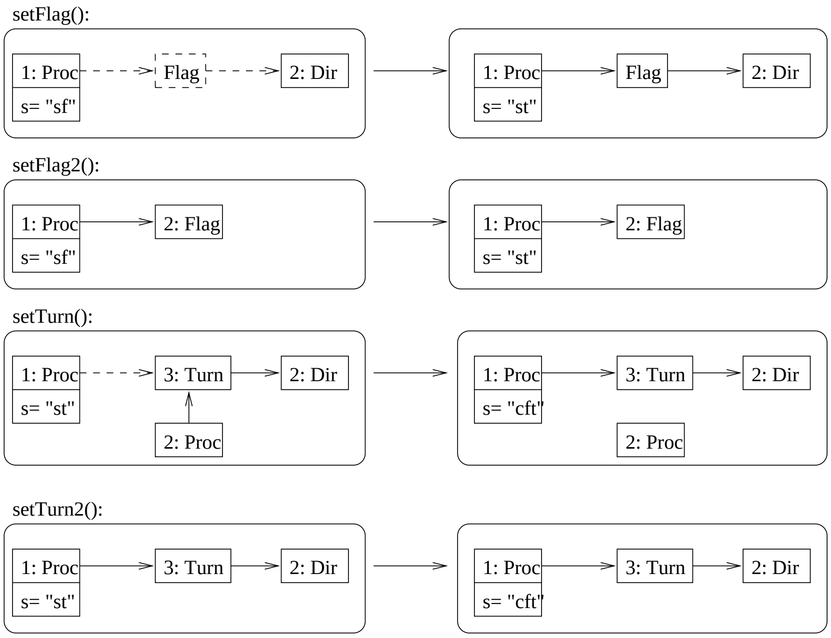

Figure 11: Graph rules setting flags and turns

process has to test the flag of the other process as well as the turn to decide if it can enter the critical region. These two tests are modelled by the rules in Figure 12. As modelled by rule *testTurn()* the *Turn* node is interesting only if both processes are in the trying region, i.e. have set their flags to the same resource. Then, process 1 can access the resource, since process 2 has applied rule *setTurn()* more recently. If one of these tests is successful, i.e. the corresponding rule is applicable, the process can have access to the resource. It *locks* the resource (modelled by a *Lock* node). Afterwards, the process can enter the critical region by rule *crit*() in Figure 10.

We have to make sure that each rule of Section 3 which modifies the file structure is applied only if all participating directories and files are locked. Thus, all these rules have to be extended by *Lock* node pointing to resource nodes as done in e.g. rule *markDir()* in Figure 13. Moreover, dependent resources such as subdirectories and files have to be locked by the same process as shown rule *markSubDir()* in the same figure. If two dependent resources are hold by different processes, rule *markSubDir* cannot be applied.

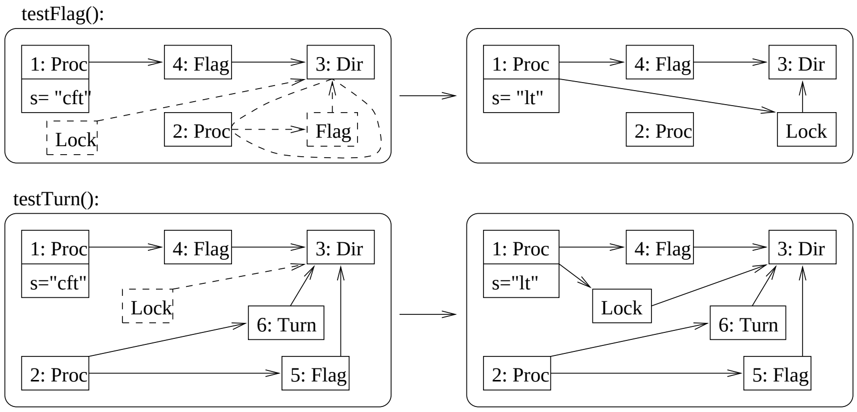

Figure 12: Graph rules testing the access to the critical region

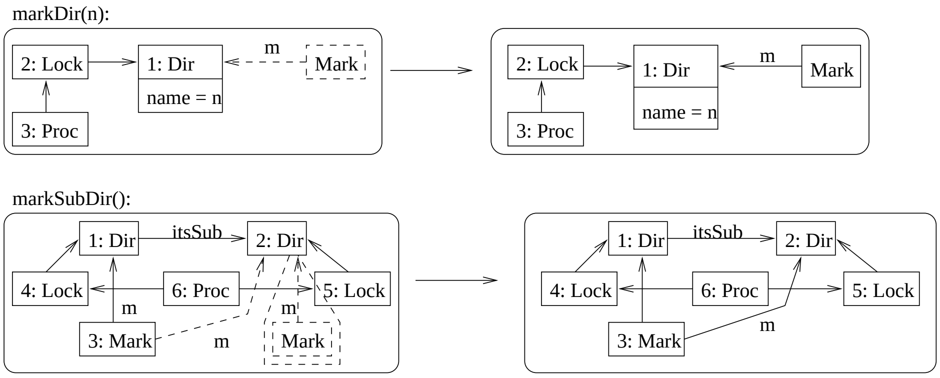

Figure 13: Sample graph rule modelling a critical action on dependent resources (compare Fig. 5)

To exit the critical region rule *exit*() in Figure 14 has to be applied, this rule can always be applied. In the exit region, locks and flags are deleted for a resource, compare rules *reset* and *reset2* in Figure 14. After the application of rule *rem()* the process is in the remainder region again.

In Figures 11, 12, and 14 the mutual access to directories is specified. Similar rules have to be added for mutual access to files. To limit the number of rules, it is also possible to introduce a more abstract node type *resource* instead of *directory* and *file* with an additional attribute specifying the resource as directory or file.

One possible rule to model stopping failures is the deletion of a *Proc* node. Thereafter, none of the rules in this section is applicable to such a stopped process anymore.

For validating the given rules, the safety and liveness properties for the mutual exclusion problem (given in Section 2) have to be checked. This has been partly done by Koch in [Koc96] for similar rules.

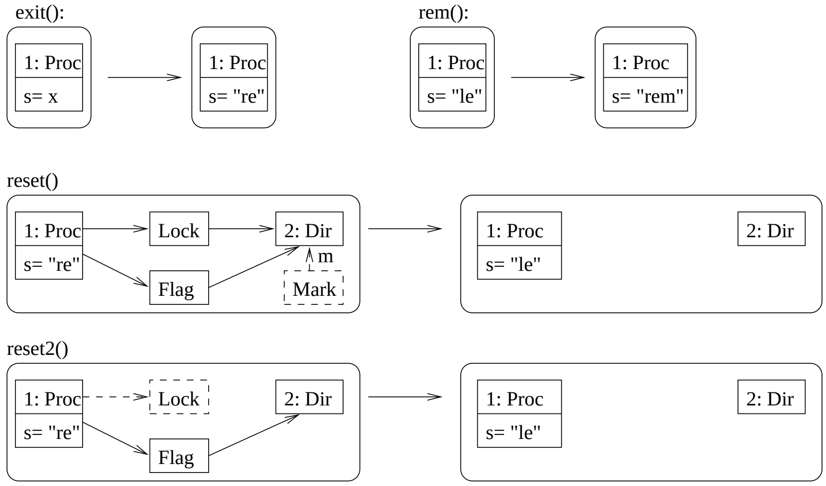

Figure 14: Graph rules modelling the exit of a critical region

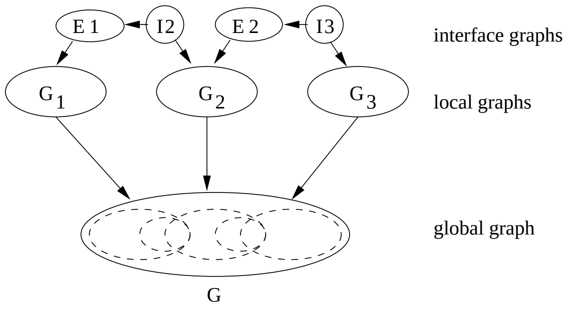

Figure 15: Distributed state graphs

4.2 Component Specification by Distributed Graph Transformation

So far, we specified a distributed system by graph transformation where states for each process and each variable are all contained in one graph. Advanced design specifications care also about where to put the shared variables, on an own server process, to that process which uses it most, etc. Furthermore, variables might be shared between some processes, but not all of the system. This might depend on the type of a process. Furthermore, it might be convenient to allow processes to have copies or replications of shared variables which are automatically updated when their origin has changed.

To model such an advanced state of a distributed system, a global state graph is further structured into local and interface graphs. The local graphs G_i may overlap in interface graphs which are useful to model shared variables. There are two kinds of interface graphs, namely import graphs and export graphs. To share a certain subgraph it has to occur in an export graph and in an import

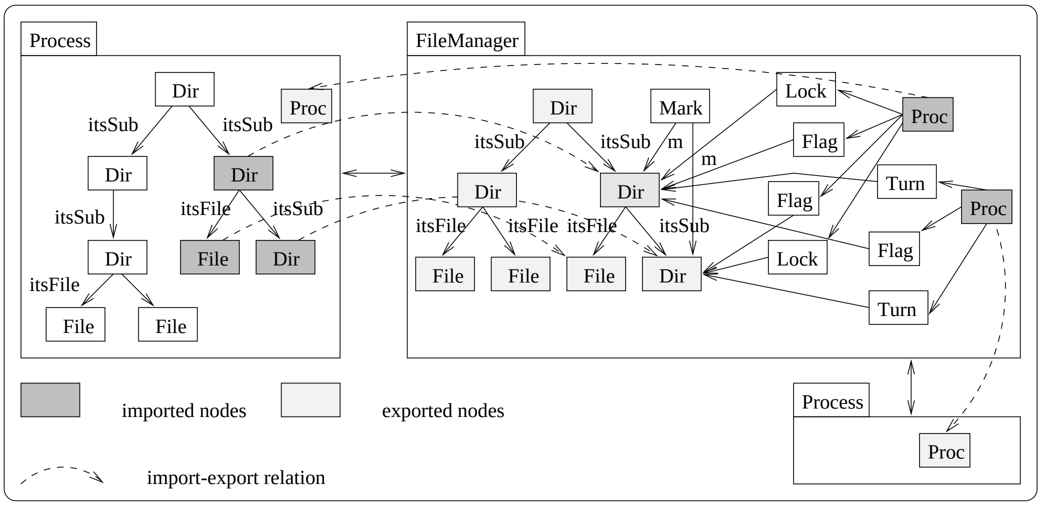

Figure 16: A sample distributed state graph showing distributed file structures

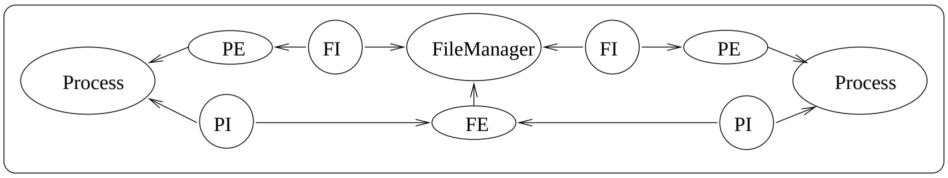

Figure 17: A sample network structure graph

graph which is connected to that export. Exporting processes are meant to care about a shared structure, e.g. they manage mutual access, while importing processes use the shared parts. A graph part is shared between all those processes whose state graphs overlap in it, i.e. all other processes do not share it. The shared part might be any state graph (as used for process states). Consider Fig. 15 for a principle network structure of distributed state graphs (here with 3 processes where graph parts are shared only partially). In graph G, it is depicted how the process and interface graphs overlap. Note that G consists exactly of the union of all process graphs. The arrows indicate graph embeddings. Formally spoken, this situation can be described by the concept of distributed graph transformation [Tae96, TFKV99], where the global graph G can be obtained by building a colimit of the process and interface graphs togehter with their embeddings being graph mappings.

In Fig. 16, a sample distributed graph is depicted. It shows an example for a distributed file structure where one central part is hold by a file manager. This part is intended to be shared between two processes, i.e. it is exported by the file manager. One process using a subdirectory of this shared part is depicted on the left, the other one using no shared resource at the bottom. The left process imports a subdirectory with its content. Note that the imported part does not have to coincide with the part locked by the same process. An imported resource has to be locked only for global changes by the importing process. Reading a directory or file is possible without locks. On the other hand, the file manager is importing the process id nodes of both processes, i.e. if a process wants to access the file systems it has to register at the file manager. The network structure graph is shown explicitly in Figure 17. Note that the file manager has one export FE containing the shared file system, but two imports FI, for each process one. The other way around, each process imports the shared directories and files it wants to use in PI and exports its process node in PE. The correspondence between the interface graphs in Figure 17 and subgraphs in Figure 16 is indicated by the legend at the bottom of Figure 16.

5 Conclusion

In this paper, we have shown by example how distributed algorithms usually modelled by I/0-automata [Lyn96] can also be modelled by graph transformation. Since moreover, graph transformation is suitable to model dynamic distributed data structures, we propose graph transformation as adequate model for the semantics of distributed system specification. The advantage of graph transformation compared with I/0-automata, Petri nets [Rei98] and other classical techniques is the possibility to provide an integrated specifications of distributed algorithms and dynamic data structures. As shown for the example of mutual access to files and directories, such integrated specifications can take care of the dependencies between a distributed algorithm with abstract resources and concrete resources modelled by dynamic data structures. We are confident that the use of graph transformation shown for a typical example in this paper can be extended to model the semantics of distributed system specifications based on graph transformation. Moreover, it is promising to use the concept of distributed graph transformation for this purpose. It remains open to extend not only the basic concepts but also analysis and verification techniques for distributed algorithms based on I/0-automata and Petri nets to the area of graph transformations. First steps in this direction can be found already in [Koc96, Koc99].

References

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In [Roz97], 1997.

[Koc96] M. Koch. Modellierung und Nachweis der Konsistenz von verteilten Transaktionsmodellen für Datenbanksysteme mit algebraischen Graphgrammatiken. Technical Report 96-36, TU Berlin, 1996.

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

[Lyn96] N. Lynch. Distributed Alogirthms. Morgan Kaufmann Publishers, 1996.

[Pet81] G.L. Peterson. Myths about the mutual exclusion problem. Information Processing Letters, 12(3), pages 115-116, June1981.

[Rei98] Wolfgang Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets.* Springer Verlag, 1998.

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

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Vol. 1: Foundations.* World Scientific, 1997.

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

[TFKV99] G. Taentzer, I. Fischer, M Koch, and V. Volle. Visual design of distributed systems by graph transformation. 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.* World Scientific, 1999.