Graph Rewriting, Constraint Solving and Tiles for Coordinating Distributed Systems

Ugo Montanari and Francesca Rossi
Università di Pisa, Dipartimento di Informatica
Corso Italia 40, 56125 Pisa, Italy
E-mail: {ugo,rossi}@di.unipi.it

Abstract. In this paper we describe an approach to model the dynamics of distributed systems. For distributed systems we mean systems consisting of concurrent processes communicating via shared ports and posing certain synchronization requirements, via the ports, to the adjacent processes. The basic idea is to use *graphs* to represent states of such systems, and *graph rewriting* to represent their evolution. The kind of graph rewriting we use is based on simple context-free productions which are however combined by means of a synchronization mechanism. This allows for a good level of expressivity in the system without sacrifying full distribution. To formally model this kind of graph rewriting, however, we do not adopt the classical graph rewriting style but a more general framework, called *the tile model*, which allows for a clear separation between sequential rewriting and synchronization. Then, since the problem of satisfying the synchronization requirements may be a complex combinatorial problem, we suggest to exploit existing techniques for *constraint solving*. This is based on the observation that the synchronization problem can be modeled as a (finite domain) constraint problem. In this respect, we propose to use both local consistency techniques, to remove the possible redundancies in a system state, and a distributed backtracking search algorithm, as used in distributed constraint solving. Our method has the following advantages: first, it provides a formal description of the way a distributed system evolves; second, it seems promising from the performance point of view, since the techniques we propose to combine productions together have been proven to be very convenient in several cases; finally, the kind of system evolution we describe here is just a particular instance of what can be described by using the tile model in its most general form, thus suggesting the possibility of extending our approach to modeling more complex distributed systems.

1. Introduction

Among the many formalisms that can be chosen to represent distributed systems and their evolutions, we believe that graphs and graph rewriting systems [14, 15, 39, 40, 9] are one of the most convenient, both in terms of expressivity and of technical background. In fact, graphs describe in a natural way net topologies and data sharing, and moreover they possess a wide literature and technical results which make the whole field of graph rewriting very formal and its notions precisely definable. Therefore, consider a distributed system consisting of concurrent processes communicating via pieces of shared data (or channels, or ports). Then such a system can be represented as a graph where arcs (or in general hyperarcs) can represent processes (or subsystem abstractions) and nodes can represent the shared data. To avoid confusion on the notion of a process, in this paper by a process we mean the state of a system component. Thus, at any given point during its evolution, a system consists of a collection of components with a certain current state (that is, processes).

Then, the evolution of a system can be modeled by graph rewriting: at each rewriting step a subpart of the current graph is chosen to evolve, and its evolution consists of being replaced by another graph. In the literature there are many graph rewriting formalisms, which range from the simplest context-free edge rewriting [24], to the more general context-sensitive subgraph rewriting [14, 15], to synchronized rewriting via the concept of amalgamation [2]. In order to choose one of them to model the evolution of a distributed system, we must make some assumptions on the system we want to model. For this paper, we assume that processes, in order to evolve, may require to synchronize with adjacent processes. That is, if two processes share a piece of data, then each of them may pose some conditions on this shared piece of information. If they agree on these conditions, then both of them (but not one of them alone) can evolve. Thus the evolution of a single process may happen only if the process is isolated, or if it does not require any synchronization to the adjacent processes.

A possible simple way to model process evolution in such a kind of systems is to use a two-phase approach: first *context-free* process productions are specified (a set for each process), as well as synchronization requests for each of these possible moves. Then, *context-sensitive* subsystem rewriting rules are obtained by combining some context-free productions. These context-sensitive rules allow for the evolution of a subpart of the system consisting of several processes that agree on the synchronization on the shared data. Applying the rule means making all such processes (and not a proper subset of them) evolve, each with one of its context-free productions. Thus the resulting rewriting mechanism is not completely general, since it is obtained by combining context-free productions and not general ones. This may pose some restrictions on the way a system evolves, but it provides a natural formalism to model the behavior of distributed systems with decentralized control. In fact, once the processes have agreed on how to behave on the shared data, they may move without any need of communicating with each other: each process needs only know its context-free productions.

This kind of synchronization and process evolution has been already considered in [11], and modeled via a special graph rewriting formalism, called *grammars for distributed systems* (GDS), where graphs represented not only the current state of the system but also its history, so to be able to derive concurrency and dependency information among the various computation steps. Also, a variation of GDSs which uses a more general rewriting mechanism allowing for system reconfiguration has been proposed in [6]. In this paper we use instead a simpler graph rewriting approach, since we are just interested in the present of the system, and not in its history.

Moreover, we model this kind of graph rewriting within the *tile model* [23], a general framework which allows for a clear and formal separation between sequential evolution and synchronization. This is in line with most coordination languages and models [5], which separate the two issues of *computation* and *coordination*. Therefore, distributed systems are described by sequential agents which take care of the evolution of a subpart of the system (that is, the *computation*), plus coordination agents which state how the subparts relate to each other (that is, the *coordination*). What is new in the tile model w.r.t. standard coordination models is that a distributed system is decomposed in sequential agents and coordinators not only statically but also dynamically. Thus the separation concerns not only the structure of the system but also its behavior.

More formally, the tile model [23, 22] describes the behavior of a wide class of systems via certain rewriting rules. These rules define a *transition system* where both the states and the transitions may be complex and be equipped with an algebraic structure. The whole rule-based system is seen as a *logical theory*, and each transition step is described by a *sequent* entailed by this theory. According to this theory, systems can be composed in three ways. In fact, besides sequential composition (to model the effect of a sequence of transitions) and parallel composition (to allow different components to act simultaneously), the tile model allows also for sub-component synchronization. Each sequent of the logic theory describing a system has the form $s \xrightarrow[b]{a} t$ and it means that s can rewrite into t, producing the effect b, only if the sub-components of s can be rewritten with the effect a. While s and t describe parts of the system which are subject to rewriting, a and b are used to describe the way the behavior of such parts is coordinated. This is done both by requiring certain synchronizations among sub-parts of the system and by defining the effects of a rewriting (to be used by subsequent rewritings). Thus, for example, term rewriting systems [26], which do not allow for any synchronization nor restriction on the class of possible rewritings, are modeled by a simple version of tile transition systems where sequents have the form $s \to t$.

Rewriting models from which the tile model drew inspiration are the SOS approach [38] and the rewriting logic approach [32]. But the work which is most related to the tile model is that on context systems [27]. However, the tile model allows for more general formats, where e.g. each process variable can appear several times in process contexts (see [23], Definition 22).

The name of the tile model comes from the fact that each sequent can be described by a square tile, whose horizontal borders correspond to the parts to be rewritten (s and t) and the vertical borders to the trigger (a) and the effect (b) of the rewriting. In this graphical representation, parallel composition corresponds to putting one tile on top of another, while sequential composition corresponds to vertical justapposition of tiles and synchronization to horizontal justapposition.

The tile model can also be characterized by using *double-categories* [25, 23], which can be seen as the superposition of a vertical and an horizontal category. Starting from a tile transition system, it is in fact possible to obtain a double-category whose cells directly correspond to the sequents entailed by that system. It is interesting to remark that this construction can be characterized as universal.

Tiles have already been used to model several formalisms. For example, the combined used of tiles and term graphs[1] [1, 7, 8] for modeling asynchronous π-calculus and CCS with locations has been described in [16, 17, 18]. Also, extensions of Petri nets equipped with flexible synchronization primitives and inspired by the tile model are presented in [3, 4]. Finally, it is studied in [34] how to translate the tile model into rewriting logic [33], in order to take advantage of important features of the latter, like execution strategies and reflective logics, and to employ its existing implementations.

While modeling our notion of graph rewriting via the tile model, a graph is just a multiset of edges, and the application of a context-free production is a tile which rewrites one edge (described in the upper border of the tile) into a graph (in the lower border), if the synchronization requirements (in the left border) are satisfied. More formally, it is a sequent $s \xrightarrow[b]{a} t$ where s is the rewritten hyperarc, t the generated hypergraph, a the required synchronization, and b is the vacuous effect, since in graph rewriting no side effect is obtained in rewriting a graph via a production. In this paper we formally describe graph rewriting sequences (also considering the synchronization requirements) as tiles, and we give a version of the tile logic which contains, as sequents, all and only those tiles which represent graph rewriting sequences.

The fact that rewriting is possible only if all synchronization requirements related to nodes shared by rewritten edges are satisfied is here

[1] Term graphs are different from terms in that they allow for controlled subterm sharing.

modeled in terms of an operator ∇, called *duplicator*. This operator can be rewritten only via a tile which requires that the same synchronization action be imposed on the shared node, by having such actions (as many as the edges sharing the node) in its right border. By synchronizing such a tile with the tiles corresponding to the productions, we get an allowed rewriting step.

The step from context-free productions to context-sensitive rules can be very costly in practice. Thus, to make the above described model for distributed system rewriting realistic, one needs to give an effective, and possibly efficient, way to tackle this problem, which is called the *rule-matching problem* [6]. That is, one needs to say how a set of processes is chosen, their synchronization requirements checked, and the corresponding context-free productions combined to get the resulting context-sensitive rule. Of course, all these tasks are to be performed in a distributed way, since we assumed to be in a distributed environment.

In this respect, *finite domain constraint problems* [30, 35], and the propagation and solution techniques associated to them [30, 35, 29, 19, 31, 12], may be helpful. A finite domain constraint problem can be described as a set of variables taking values over corresponding finite domains, and a set of constraints. Each constraint involves a subset of the variables and specifies all the possible ways such variables can satisfy the constraint. A solution for a constraint problem is an assignment to all the variables which satisfies all the constraints.

The idea here, already considered in [6], is to represent the rule-matching problem in a distributed system as a constraint problem, where variables are associated to processes and constraints to ports (that is, sets of adjacent processes). The domain of a variable is then the set of all context-free productions for the corresponding process, and each constraint is satisfied by the tuples of context-free productions (one for each adjacent process) whose synchronization requirements agree on the considered port. In this kind of constraint problem, a solution is thus a choice of a context-free production for each process, such that all synchronization requirements are satisfied. Note that a solution does not necessarily make every process evolve. In fact, to model the case of a rewrite rule which synchronizes only a subset of the processes, we can assume that an *idle* production has been chosen for each inactive process.

Given this representation of the rule-matching problem as a finite domain constraint problem, it is natural to consider the techniques used to solve constraint problems of this kind. Usually, finite domain constraint problems are solved by a *backtracking search* over the tree of the possible alternatives for each variable: variables are assigned to elements of their domain, in some order, until a failure to satisfy (even a subset of) the constraints occurs; at that point, the assignment of the latest assigned variable is retracted and another assignment tried. Otherwise, a complete assignment is reached, which is a solution of the constraint problem, and the backtracking search assures that all solutions are tried.

To make the basic backtracking algorithm more efficient (although it always remains exponential in the worst case, since the problem is NP-complete), many *constraint propagation* (or *local consistency*) algorithms have been proposed in the literature [35, 29, 19, 31, 12]. The basic idea behind such algorithms is the following: if, by looking at only some of the constraints of the constraint problem, one can see that there is no way to assign some variable to a value, then that assignment will never participate in any complete solution. Thus it can be eliminated. By iterating this removal until no more removals are possible in any variable domain, we get an algorithm which is polynomial, although exponential in the size (that is, number of variables) of the subsets of constraints considered at each step. This idea can be instantiated in several different ways depending on which and how many constraints one considers at each step. For example, by considering only one constraint at a time (and thus only the variables involved in such constraint), we have the so-called *arc-consistency* algorithm [29]. More generally, by considering subsets of constraints involving k variables, we get the so-called k*-consistency* algorithm [19]. It is clear how such algorithms may help the standard backtracking algorithm outlined above: if applied before starting the backtracking algorithm, then each variable has less alternatives; thus the branching factor of the search tree is smaller, causing failures to be discovered earlier. Therefore the process of finding a solution is faster.

In our formulation of the rule-matching problem as a finite domain constraint problem, eliminating values from a variable domain means eliminating, for a process, the choice of some of its context-free productions (because their synchronization requirements do not match with those of the productions of adjacent processes). This could lead to a great speed-up in the overall search for a solution, especially when the graph representing the current system state has special topologies [20].

In classical constraint solving over finite domains, constraints are usually stored in one memory. Thus local consistency algorithms are usually centralized. However, it is easy to see that such algorithms have an intrinsic distributed nature, since one can consider agents corresponding to clusters of constraints, which represent the selected subproblems. In particular, the arc-consistency algorithm can be implemented in a completely distributed environment, since each agent corresponds to just one constraint [37].

Once a local consistency algorithm has been applied over the current state of the system, so to possibly eliminate some redundant choices for the processes, one has however still the problem of solving the rule-matching problem, which, as said above, amounts to choosing a subset of the processes, and a context-free rule for each of them, so that all synchronization requirements of the chosen productions are satisfied. This can be done by a backtracking search algorithm (as described above), which however has to be distributed as well. Examples of distributed version of such algorithm for finite domain constraint problems can be found in [42, 43].

In the following of this paper, we provide the details on the formalisms and the techniques we informally described in the introduction, and on the way they can be used for our purposes. In particular, Section 2 gives an algebraic description of graphs, Section 3 gives a summary of the tile model, and Section 4 contains the basic notions of graph rewriting, and their modification to model distributed systems with synchronization. Then, Section 5 shows how to model graph rewriting within the tile model. After this, Section 6 defines constraint problems, local consistency algorithms, and distributed backtracking search, and Section 7 describes how to model and efficiently solve the rule-matching problem using constraint-related techniques. Finally, Section 8 gives an example of our methodology, and Section 9 concludes the paper summarizing the main points of our approach and pointing out its advantages.

A shorter version of this paper, where however the modeling of distributed systems did not involve the tile model, has appeared in [36].

2. Graphs

Graphs are well-known structures which are usually described as a set of nodes and edges, but the issue of an algebra of graphs has received little attention up to now. In this section we first describe graphs in the usual way, then we introduce a generalization of graphs, called *open graphs*, and give an algebra for them, and finally we show that graphs can be characterized as arrows of the *gs-monoidal theory* generated over the signature of all possible edges.

Our notion of graphs is very general. In fact, we will consider *labelled multi hyper graphs*, which means that each edge is labelled and may connect more than two nodes (thus “hyper”), and that there may be several edges with the same label and which connect the same nodes (thus “multi”).

Definition 1. (labeled (multi-hyper) graphs)

Given a ranked set $S = \{S_k\}_{k\in\omega}$ of labels, where $|s| = k$ is the rank of $s \in S_k$, a *labeled multi hyper graph*, or simply a *graph*, is a pair $G^n = (N, E)$ where

- $N = \{x_1, \ldots, x_n\}$ is an ordered set of *nodes*. Number n is called the *rank* of G;
- E is a multiset of *labeled edges*, where each edge is a pair $\langle s, (x_1, \ldots, x_m)\rangle$, with $s \in S_m$ and $x_i \in N$ for each $i \in \{1, \ldots, m\}$.

Two graphs $G_1^n = (N_1, E_1)$ and $G_2^n = (N_2, E_2)$ are *isomorphic* if there is a bijection $f : N_1 \to N_2$ such that the number of occurrences of $\langle s, (y_1, \ldots, y_m)\rangle$ in E_1 is the same as the number of occurrences of $\langle s, (f(y_1), \ldots, f(y_m))\rangle$ in E_2. □

In the following, an edge $\langle s, (x_1, \ldots, x_m)\rangle$ will usually be represented by using the notation $s(x_1, \ldots, x_m)$, and an entire graph will be denoted by a multiset of such edges. A last missing thing in this representation of a graph is the set of its nodes. Assuming that a graph with n nodes has always the nodes $x_1, \ldots, x_n$, we just need to write the number of such nodes, that is, n. Therefore in the following a graph will be denoted by a multiset of the form

$$\{t_1, \ldots, t_k\}^n$$

where n is the number of its nodes, k the number of its edges, and each t_i, for $i = 1, \ldots, k$, denotes an edge occurrence.

We now introduce *open graphs*, which are a generalization of graphs. More precisely, open graphs are just graphs plus a tuple of output nodes (possibly with repetitions). The intuitive motivation behind this notion of open graphs is that they represent systems which are not *closed* but may have to interact with other systems (via their output nodes). Then, graphs, which are open graphs without output nodes, represent closed systems, which do not have any way to interact with the rest of the world.

Definition 2. (open graphs)

Given a ranked set S of labels, an *open graph* is a typed triple $P : \underline{n} \to \underline{m} = (N, E, O)$ where $G^n = (N, E)$ is a graph and $O = (y_1, \ldots, y_m)$ is a tuple of *output* nodes $y_i \in N$, $i = 1, \ldots, m$, possibly with repetitions. Thus a graph is an open graph with no output nodes. Underlined natural numbers $\underline{n}$ and $\underline{m}$ are called the *source* and the *target* of the open graph respectively.

Two open graphs $P_1 : \underline{n} \to \underline{m} = (N_1, E_1, O_1)$ and $P_2 : \underline{n} \to \underline{m} = (N_2, E_2, O_2)$ are *isomorphic* if there is a bijection $f : N_1 \to N_2$ with[2] $O_2 = f(O_1)$ which makes (N_1, E_1) and (N_2, E_2) isomorphic. □

For example, Figure 1 shows an open graph with nodes $\{x_1, x_2, x_3, x_4\}$ and four labelled edges. Note that two edges have the same label (s_1) and connect the same nodes (x_1 and x_2). The tuple of output nodes is $\langle x_2, x_4, x_2 \rangle$. This is represented in the figure by numbers labelling the output nodes and giving the position(s) in which each output node occurs in the tuple O.

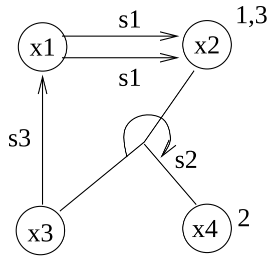

Figure 1. An open graph.

To give an example of the notation for graphs introduced after Definition 1, consider the open graph in Figure 1 and forget the tuple of output nodes. In this way we get a graph which is described via the notation

$$\{s_1(x_1, x_2), s_1(x_1, x_2), s_2(x_3, x_2, x_4), s_3(x_3, x_1)\}^4.$$

Open graphs can be seen as a special case of *gs-graphs* [18], which are axiomatically defined by *gs-monoidal theories*. The latter is a logical theory similar to, but weaker then, the ordinary algebraic theory of terms and substitutions. In the Appendix we give the finitary axiomatization of (one sorted) gs-monoidal theories as presented in [21, 23, 7, 8]. The models of gs-monoidal theories are gs-monoidal categories, a special case of symmetric monoidal categories. Here we present an algebra of open graphs which coincides with the algebra in the Appendix in the case the signature Σ contains only edge labels.

The *basic* open graphs are the following:

Generators: for every $s \in S_k$ the graph $s : \underline{k} \to \underline{0}$ with $N = \{x_1, \ldots, x_k\}$ and one occurrence of the edge $s(x_1, \ldots, x_k)$.

Empty: the graph $1 : \underline{0} \to \underline{0}$ with no nodes and no edges.

[2] Function f is obviously extended to tuples of nodes.

Identity: the open graph $id : \underline{1} \to \underline{1}$ with $N = \{x_1\}$, no edges and $O = (x_1)$.

Permutation: the open graph $\rho : \underline{2} \to \underline{2}$ with $N = \{x_1, x_2\}$, no edges and $O = (x_2, x_1)$.

Duplicator: the open graph $\nabla : \underline{1} \to \underline{2}$ with $N = \{x_1\}$, no edges and $O = (x_1, x_1)$.

Discharger: the graph $! : \underline{1} \to \underline{0}$ with $N = \{x_1\}$ and no edges.

Figure 2 gives a pictorial representation of these basic open graphs (except the empty one).

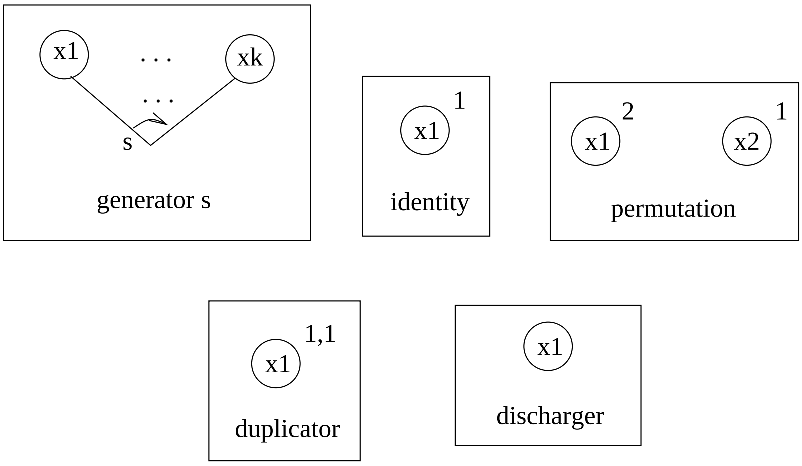

Figure 2. Basic open graphs.

We now introduce two operations on open graphs, which can be used to build larger open graphs from smaller ones: sequential and parallel composition. The *sequential composition* of two open graphs is obtained by gluing the list of output nodes of the first graph with the list of nodes of the second, and it is defined only if the lengths of such two lists are equal. The *parallel composition* instead is always defined, and it is a sort of disjoint union where the lists of nodes and of output nodes are concatenated.

Definition 3. (parallel and sequential composition of open graphs) Given two open graphs (with disjoint sets of nodes)

$$P : \underline{n} \to \underline{m} = (\{x_1, \ldots, x_n\}, E, (y_1, \ldots y_m))$$

and

$$P' : \underline{n'} \to \underline{m'} = (\{x'_1, \ldots, x'_{n'}\}, E', (y'_1, \ldots y'_{m'}))$$

their *parallel composition* is

$$P \otimes P' \ : \ \underline{n+n'} \to \underline{m+m'} = \\ (\{x_1, \ldots, x_n, x'_1, \ldots, x'_{n'}\}, E \cup E', (y_1, \ldots y_m, y'_1, \ldots y'_{m'}))$$

where $\cup$ is multiset union. We write P^n for the open graph obtained composing P in parallel n times, i.e. $P^0 = 1$ and $P^{n+1} = P \otimes P^n$.

The sequential composition of P and P' is defined only if $m = n'$ and can be written as

$$P \, ; P' : \ \underline{n} \to \underline{m'} = (\{x_1, \ldots, x_n\}, E \cup E'[\vec{y}/\vec{x'}], (y'_1, \ldots, y'_{m'})[\vec{y}/\vec{x'}])$$

where substitution

$$(\vec{y}/\vec{x'}) = (y_1/x'_1, \ldots, y_m/x'_{n'}) \quad \text{with} \quad m = n'$$

is applied in the obvious way to multisets of edges and tuples of nodes:

- $\langle s, (y_1, \ldots, y_m)[y/x] \rangle$ has multiplicity n in $E[y/x]$ if and only if $\langle s, (y_1, \ldots, y_m) \rangle$ has multiplicity n in E;
- $(y_1, \ldots, y_m)[y/x] = (y_1[y/x], \ldots, y_m[y/x])$;
- $x'[y/x] \ = \ $ if $x' = x$ then y else x'. □

Figure 3 shows an example of sequential composition of two open graphs.

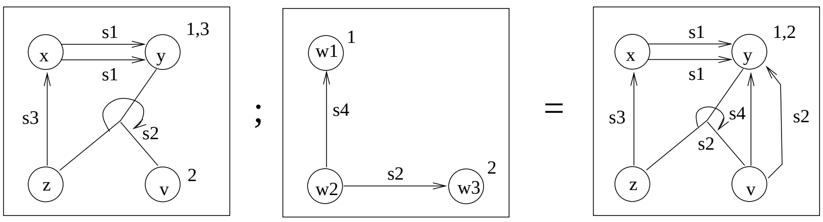

Figure 3. Sequential composition of open graphs.

Given this algebra of open graphs, it is now possible to show that there is a normal form for open graphs, which contains just basic open graphs, suitably combined via parallel and sequential composition.

Proposition 4. (decomposition of open graphs) Every open graph (and thus in particular every graph) can be obtained by evaluating some expression containing only basic open graphs as constants, and sequential and parallel composition as operators. □

To prove the above proposition, it is convenient to introduce a *normal form* of open graphs. First it is easy to see that every open graph $(\{x_1, \ldots, x_n\}, E, (x_{i_1}, \ldots x_{i_m}))$ with no edges (i.e. $E = \emptyset$) can be built from identities, permutations, duplicators and dischargers. In fact, this open graph is fully determined (up to isomorphism) by the substitution $\sigma_n = (i_1/1, \ldots, i_m/m)$ and will be just denoted by σ_n in the following. For every σ_n, we have[3]

$$\sigma_n = d_1 \otimes \ldots \otimes d_n \; ; \; r$$

where r is a bijective substitution, i.e. a permutation, and $d_i = m_k$ if node x_i occurs k times in the list $(x_{i_1}, \ldots x_{i_m})$. Also,

$$m_0 = ! \quad m_1 = id \quad m_{i+1} = m_i \; ; \; \nabla \otimes id^{i-1}$$

where $i = 1, 2, \ldots$. Basically, m_i makes i copies of a node. Note that when $i = 0$ this means that a node is deleted and when $i = 1$ that the node is maintained as it is.
Finally, any permutation can be obtained by repeated sequential and parallel composition of the basic permutation ρ and of the identity id.

Constructively, what the above formula says is that the normal form of an open graph with no nodes can be obtained by taking the nodes of the graph, deleting those that do not appear as output nodes (via the m_0's), maintaining those that appear as output nodes only once (via the m_1's), and duplicating those that appear as output nodes more than once (via the m_i's with $i > 1$). At the end of this process, which is performed via $d_1 \otimes \ldots \otimes d_n$, we have a tuple of as many node occurrences as the length of the tuple of output. Therefore, to get the output tuple, we just need to suitably permute such occurrences so that they are in the right order for the output tuple. This is done via permutation r.

Any open graph P can now be obtained as

$$P = \sigma_n ; id_m \otimes \bigotimes_{s \in S} s^{n_s} \tag{1}$$

where n_s is the sum of the number of occurrences of all the edges of P labelled by s, and $\sigma_n : \underline{n} \rightarrow \underline{m + \sum_{s \in S} n_s |s|}$ is a suitable substitution. In the above formula, the open graph P is obtained by first composing in parallel, for every s, as many generators s as are the edges of P labeled by s. Also, m identities are composed in parallel to account for the output nodes. At this point, every node is either an output

[3] Note that in this formula and in all the following ones we assume that the parallel composition operator $\otimes$ has precedence over the sequential composition. This will allow us to avoid the need for a lot of parenthesis.

node, or it is connected to exactly one edge. Substitution σ provides the connection relation of P by linking nodes, output nodes and edges. Notice that if a total ordering of the edges of P is provided (e.g. the lexicographic ordering), then decomposition (1) is unique.

Again, we can look at the above formula from a constructive point of view. To represent an open graph, we first need to start from its nodes, duplicate them (or delete them, or maintain them), so to get as many node occurrences as the length of the tuple of output nodes plus the nodes involved separately in all edges. This is done via substitution σ_n. Then, each such node occurrence is either an output node, or it participates in one edge. Therefore the sequential composition of this open graph with no edges with $id_m \otimes \bigotimes_{s \in S} s^{n_s}$ allows to include the edges in the representation and thus to get the entire open graph.

As anticipated at the beginning of this section, we now give a characterization of graphs and open graphs in terms of gs-monoidal theories. However, we first need the notion of *signature*.

Definition 5. (one-sorted (hyper-)signatures)

Given a set S of sorts, a *hyper-signature* (in the following simply a *signature*) Σ is a family $\Sigma = \{\Sigma_{h,k}\}_{h,k \in \omega}$ of sets of *operators* from h to k, where h and k are natural numbers. $\square$

Proposition 6. (characterization of graphs and open graphs)

The open graphs (up to isomorphism) on a set $S = \{S_k\}_{k \in \omega}$ of labels, are the arrows of the gs-monoidal theory $\mathbf{GS}(\Sigma)$ (see the Appendix for the formal definition of a gs-monoidal theory) generated by the signature $\Sigma = \{\Sigma_{h,k}\}_{h,k \in \omega}$, with $\Sigma_{h,k} = \emptyset$ if $k \neq 0$ and $\Sigma_{h,0} = S_h, h \in \omega$, where $1 = id_{\underline{0}}$, $id = id_{\underline{1}}$, $\rho = \rho_{\underline{1},\underline{1}}$, $\nabla = \nabla_{\underline{1}}$ and $! = !_{\underline{1}}$. An arrow is a graph iff its target is 0. $\square$

For the proof of the first part, besides Proposition 4, it is sufficient to notice that every arrow can be brought to normal form (1) by using the axioms of gs-monoidal theories, and different normal forms cannot be proved equal. The second part derives from the definition of graphs as open graphs with no output nodes.

3. The Tile Model

In this section we describe the basic features of the *tile model* [21, 23]. In the general case, tiles are double cells of monoidal double categories [13]. Thus their theory can be seen as the tensor product of the theory of categories and of the theory of monoidal categories [34], while their algebras are internal categories in the category of monoidal categories.

Additional axioms may introduce extra horizontal and vertical structure useful for certain applications. In this paper we need the horizontal categories to be gs-monoidal in order to represent graphs. In addition, we are not interested in associating proof terms to tile sequents[4] and thus we introduce the *flat* version of our tile logic. Formally, its theory can be obtained by adding a *flattening* axiom which identifies two cells (tiles) whenever they have the same border, i.e. the same horizontal and vertical sources and targets. Thus the tile theory relevant here is the tensor product of the theory of categories and of the theory of gs-monoidal categories, with the additional flattening axiom. Flat gs-monoidal tile logic is introduced directly below, simply by omitting the proof terms in the sequents.

In the following definitions we will refer to the gs-monoidal theory $\mathbf{GS}(\Sigma_h)$, to the monoidal theory $\mathbf{M}(\Sigma_v)$, and to the graph theory $G(\Sigma_v)$. All three of them are defined in the Appendix. The main difference between a gs-monoidal theory and a monoidal theory is that the inference rules generating the arrows of a monoidal theory do not allow for duplication, permutation, and discharging, and these arrows are subject to many less axioms than the arrows of a gs-monoidal theory. A graph theory is even simpler than a monoidal theory, since its arrows cannot be composed, but just summed. When Σ_v, as it is the case in this paper, contains only operators from 1 to 1, then $\mathbf{M}(\Sigma_v)$ consists of all the tuples of strings of operators, and $G(\Sigma_v)$ consists of all the tuples of operators, in Σ_v.

Definition 7. (tile sequent, tile rewrite system)

Let Σ_h and Σ_v be two (one sorted) signatures, called the *horizontal* and the *vertical* signature respectively.

A Σ_h-Σ_v *tile sequent* is a quadruple $l \xrightarrow[b]{a} r$, where $l : \underline{n} \to \underline{m}$ and $r : \underline{p} \to \underline{q}$ are arrows of $\mathbf{GS}(\Sigma_h)$ while $a : \underline{n} \to \underline{p}$ and $b : \underline{m} \to \underline{q}$ are arrows of $\mathbf{M}(\Sigma_v)$. Arrows l, r, a and b are called respectively the *initial configuration*, the *final configuration*, the *trigger* and the *effect* of the tile. Trigger and effect are called *observations*. Underlined natural numbers $\underline{n}$, $\underline{m}$, $\underline{p}$ and $\underline{q}$ are called respectively the *initial input interface*, the *initial output interface*, the *final input interface* and the *final output interface*.

A *short* tile sequent is a tile $l \xrightarrow[b]{a} r$ where observations a and b are arrows of the graph theory $G(\Sigma_v)$ (i.e. no sequential composition is allowed to build them).

[4] In this paper we are not interested in giving a true-concurrency semantics to distributed systems. Instead, proof terms and their axiomatization are essential to this aim.

A *tile rewrite system* (TRS) $\mathcal{R}$ is a triple $\langle \Sigma_h, \Sigma_v, R\rangle$, where R is a set of Σ_h-Σ_v short tile sequents called *rewrite rules.* $\square$

A TRS $\mathcal{R}$ can be considered as a logical theory, and new sequents can be derived from it via certain inference rules.

Definition 8. (flat gs-monoidal tile logic) Let $\mathcal{R} = \langle \Sigma_h, \Sigma_v, R\rangle$ be a TRS. Then we say that $\mathcal{R}$ *entails* the class $\mathbf{R}$ of the *tile sequents* $s \xrightarrow[b]{a} t$ obtained by finitely many applications of the following inference rules:

basic rules:

$$(generators)\ \frac{s \xrightarrow[b]{a} t \in \mathcal{R}}{s \xrightarrow[b]{a} t \in \mathbf{R}}$$

$$(h\text{-}refl)\ \frac{s : \underline{n} \to \underline{m} \in \mathbf{GS}(\Sigma_h)}{id_s = s \xrightarrow[id_{\underline{m}}]{id_{\underline{n}}} s \in \mathbf{R}} \qquad (v\text{-}refl)\ \frac{a : \underline{n} \to \underline{m} \in \mathbf{M}(\Sigma_v)}{id_a = id_{\underline{n}} \xrightarrow[a]{a} id_{\underline{m}} \in \mathbf{R}};$$

composition rules:

$$(p\text{-}comp)\ \frac{\alpha = s \xrightarrow[b]{a} t, \alpha' = s' \xrightarrow[b']{a'} t' \in \mathbf{R}}{\alpha \otimes \alpha' = s \otimes s' \xrightarrow[b \otimes b']{a \otimes a'} t \otimes t' \in \mathbf{R}}$$

$$(h\text{-}comp)\ \frac{\alpha = s \xrightarrow[c]{a} t, \alpha' = s' \xrightarrow[b]{c} t' \in \mathbf{R}}{\alpha * \alpha' = s; s' \xrightarrow[b]{a} t; t' \in \mathbf{R}}$$

$$(v\text{-}comp)\ \frac{\alpha = s \xrightarrow[b]{a} u, \alpha' = u \xrightarrow[b']{a'} t \in \mathbf{R}}{\alpha \cdot \alpha' = s \xrightarrow[b;b']{a;a'} t \in \mathbf{R}};$$

auxiliary rules:

$$(perm)\ \frac{a : \underline{n} \to \underline{m}, b : \underline{n'} \to \underline{m'} \in \mathbf{M}(\Sigma_v)}{\rho_{a,b} = \rho_{\underline{n},\underline{n'}} \xrightarrow[b \otimes a]{a \otimes b} \rho_{\underline{m},\underline{m'}} \in \mathbf{R}}$$

$$(dupl)\ \frac{a : \underline{n} \to \underline{m} \in \mathbf{M}(\Sigma_v)}{\nabla_a = \nabla_{\underline{n}} \xrightarrow[a \otimes a]{a} \nabla_{\underline{m}} \in \mathbf{R}} \qquad (disch)\ \frac{a : \underline{n} \to \underline{m} \in \mathbf{M}(\Sigma_v)}{!_a = !_{\underline{n}} \xrightarrow[id_{\underline{0}}]{a} !_{\underline{m}} \in \mathbf{R}}.$$

$\square$

The different sets of rules are self-explaining. Basic rules provide the generators of the sequents, together with suitable identity arrows, whose intuitive meaning is that an element of $\mathbf{GS}(\Sigma_h)$ or $\mathbf{M}(\Sigma_v)$ can be rewritten to itself (showing no effect/using no trigger, so to say). Composition rules provide all the possible ways in which sequents can be composed, while auxiliary rules are the counterpart of ρ, ∇ and ! for gs-monoidal theories.

A *quasi-gs-monoidal* tile logic differs from a gs-monoidal tile logic in that rule *dupl* applies only to some of the constructors in $\mathbf{M}(\Sigma_v)$. That is, rule *dupl* above is replaced by the following rule:

$$(\textit{quasi-dupl})\ \frac{a : \underline{n} \to \underline{m} \in (\mathbf{M}(\Sigma_v - I))}{\nabla_a = \nabla_{\underline{n}} \xrightarrow[a \otimes a]{a} \nabla_{\underline{m}} \in \mathbf{R}}$$

where I is a subset of Σ_v. Then, the set of sequents obtained by $\mathcal{R}$ by applying the rules above, where rule *(dupl)* is replaced by rule *(quasi-dupl)*, depends on the set I, and is denoted by $Q(\mathcal{R}, I)$. The notion of quasi-gs-monoidal tile logic is needed to allow for the elimination of those *(dupl)* tiles whose targets and effects contain vertical constructors in a subset I of Σ_v. This will be useful later when we will model graph rewriting systems within the tile model (in Section 5), and we will prove that there is a precise correspondence between tiles and graph derivations (in Theorem 15).

Another notion that will be useful in that theorem is that of *horizontal tiles*, which are obtained by using all the inference rules in Definition 8 except rule *(v-comp)*. In the following the class of horizontal tiles generated by a given TRS $\mathcal{R}$ will be denoted by $HT(\mathcal{R}, \mathcal{I})$, where again I is the subset of Σ_v consisting of the constructors among which we do not want any node sharing. Finally, we also define the notion of *derivation tiles*, which are tiles obtained by vertical composition of horizontal tiles. These are the tiles that will correspond to sequences of graph rewriting steps in Theorem 15. The class of all derivation tiles generated by a given TRS $\mathcal{R}$ will be denoted by $DT(\mathcal{R}, \mathcal{I})$.

4. Graph Rewriting for Distributed Systems

We describe here the basic notions of graph rewriting, following the presentation of graphs introduced in Section 2, and then we show the modifications to be done to the classical approach to make graph rewriting suitable for describing the behavior of distributed systems with synchronization.

4.1. Graph Rewriting

Graphs have already been defined at the beginning of Section 2, where we also introduced a notation as multisets of labelled edges, plus the number of their nodes. Let us now come to the description of the process of rewriting a graph into another graph. This is done via the concept of *graph production*, which, when applied to a graph, replaces a part of such graph with another graph, and in doing so it may delete some elements (nodes and edges), generate new ones, and preserve others. In this paper we will just consider *context-free* productions, which tell how to rewrite a graph containing a single edge, connecting as many distinct nodes as its rank, into an arbitrary graph, while preserving the nodes connected by the rewritten edge. Therefore, in a context-free production, no node is ever deleted.

Context-free graph productions will be written as $L \to R$, where L is the (graph containing only the) edge to be rewritten and R is the graph to be generated. For example, $\{p(x_1, x_2)\}^2 \to \{q(x_1, x_3), r(x_3, x_2)\}^3$ is a production which replaces edge p, which connects nodes x_1 and x_2, with the two following edges: q, which connects nodes x_1 and x_3, and r, which connects nodes x_3 and x_2. Note that nodes x_1 and x_2 are preserved, while node x_3 is generated (and shared between q and r). The fact that the rank of the left-hand side and of the right-hand side of the production coincide with the number of different nodes occurring in their terms means that there are no nodes left isolated in this production.

Due to the special shape of context-free productions, the specification of the nodes in the left-hand side, as well as the multiset constructor $\{...\}$, are superfluous, if we know the rank of the rewritten edge. Thus the above production will in fact be written as $p^2 \to \{q(x_1, x_3), r(x_3, x_2)\}^3$, where we assume that the nodes occurring in the rewritten edge are always the first ones in the ordering given by their indices. Thus in this case p involves nodes x_1 and x_2.

Definition 9. (graph productions) Given a ranked set (of edge labels) S, a graph production p over S is a pair $\langle L^n, R^{n+m} \rangle$ where

- $L^n \in S_n$, and
- R^{n+m} is a graph over S with rank $n + m$.

A production $prod = \langle L, R \rangle$ will be written as $prod = (L \to R)$. □

A production $prod = (L \to R)$ can be applied to a graph G yielding H if there is an *occurrence* of L in G. The result of applying p to G is a graph H which is obtained from G by removing the occurrence of L and adding R.

Let us consider again the production $p^2 \to \{q(x_1, x_3), r(x_3, x_2)\}^3$, and the graph $G = \{p(x_1, x_2), s(x_1, x_3)\}^3$. This production can be applied to G because there is an occurrence of p in G. Then a possible result of its application to G is the graph $H = \{q(x_1, x_4), r(x_4, x_2), s(x_1, x_3)\}^4$. Other possible results of applying this same production to G can be obtained by suitably permuting the indices of the nodes of H.

Definition 10. (production application) Consider a graph $G^h = \{t_1, \dots, t_l\}^h$ and a production $prod = (p^n \to H^{n+m})$. This production is applicable to the graph G^h if $t_1 = p(x_{j_1}, \dots, x_{j_n})$. Note that by choosing t_1 and not a generic t_i we do not loose in generality, since G^h is a multiset and thus by permuting its elements we get exactly the same graph. Then we can write

$$\{p(x_{j_1}, \dots, x_{j_n}), t_2, \dots, t_l\}^h \Rightarrow_{prod}$$

$$\{H[x_{j_1}/x_1, \dots, x_{j_n}/x_n]\rho_1, t_2, \dots, t_l\}^{h+m}\rho_2$$

where $\rho_1 = (x_{h+1}/x_{n+1}, \dots, x_{h+m}/x_{n+m})$ and ρ_2 is any permutation of the nodes of the resulting graph, with the constraint that $i, j \in 1, \dots, h$ and $i < j$ implies $\rho_2(x_i) < \rho_2(x_j)$. □

In the above definition, ρ_1 is a substitution which changes the names of the new nodes introduced by the production application (that is, $x_{n+1}, \dots, x_{n+m}$) in a way that the set of nodes of the right hand side of the production becomes $x_1, \dots, x_n, x_{h+1}, \dots, x_{h+m}$. In this way, the new nodes are not identified with any of the old nodes. Instead, ρ_2 is a permutation which may reorder the nodes of the resulting graph, with the constraint that the ordering of the old nodes (that is, $x_1, \dots, x_h$) must be preserved. The reason why we allow for such a reordering is that, when we will model graph rewriting via the tile model, in Section 5, we will get these permutations automatically. Therefore we need ρ_2 to get a precise correspondence between that description of graph rewriting and the one contained in this section.

Therefore, during a rewriting of a graph via a production, we have three sources of nondeterminism:

- the choice of the production to be used in the rewriting;
- the choice of the edge to be rewritten;
- the choice of the ordering of the nodes in the resulting graph, which can be chosen in any way that respects the ordering of the old nodes already in G before the rewriting.

Consider the graph $G = \{p(x_1), q(x_2)\}^2$, and the production $p^1 \to \{r(x_1), s(x_2)\}^2$. This production is obviously applicable to G. A possible result of its application is the graph $H_1 = \{r(x_1), s(x_3), q(x_2)\}^3$. Note that this choice of indices for the nodes in the resulting graph respects the ordering of the old nodes, since x_1 comes before x_2. A second possible choice would be the one which leads to the following graph $H_2 = \{r(x_1), s(x_2), q(x_3)\}^3$, and a third (and last) choice leads to $H_3 = \{r(x_2), s(x_1), q(x_3)\}^3$. More generally, the new nodes generated by a rewriting can be inserted in any possible way among the old nodes, which are already ordered, provided that this old ordering is not affected.

Definition 11. (graph rewriting system, derivations) A graph rewriting system is a triple $GRS = \langle S, G_0, P\rangle$, where S is a set of edge labels, G_0 is a graph over S, and P is a set of graph productions over S. A derivation for GRS is a finite sequence of direct derivation steps of the form $G_0 \Rightarrow_{p_1} G_1 \Rightarrow_{p_2} \ldots \Rightarrow_{p_n} G_n = H$, where $p_1, \ldots, p_n$ are in P. $\square$

Note that this substitution-oriented description of the application of a production to a graph could also be denoted by a simple rewriting in the algebraic style of Section 2. In fact, given a graph

$$G^h = P; p$$

where p is an edge and P is an open graph, and a production

$$p^n \to H^{n+m},$$

the application of such a production over G leads to a new graph

$$\rho_2; P \otimes id^m; H.$$

In fact, apart from the reordering given by permutation ρ_2, seen as an open graph, the graph $P \otimes id^m; H$ is exactly the graph obtained by taking P (that is, the part of G which is not rewritten), adding m new nodes to it (via the parallel composition of P and id^m), and then adding graph H (via sequential composition).

4.2. Graph Rewriting for Distributed Systems

It is easy to see how to model a distributed systems via a graph: system components are represented by edges (labeled with the current state of the component), and ports by nodes. The connection function specifies which ports are connected to which components. Then, for each component there is a set of context-free productions, all of which have the hyperedge representing that component in their left member. Thus, the productions associated to a component specify all its possible moves.

For the purposes of this paper, where we intend to use graph productions to model the behavior of distributed systems with synchronization, we need to define a notion of production slightly different from the one of the previous section, by adding some labels to the nodes in its left member. Assuming to have an alphabet of synchronization requirements, say A, then we need a partial function $f : nodes(L) \rightharpoonup A$, which associates a synchronization requirement to some of the nodes connected by the rewritten edge. In this way we specify that the replacement of the edge in L with the graph R poses some requests to the adjacent edges: that they move as well, and with matching synchronization requirements. Function f is partial since it is possible that, for a specific move, no requirement is posed on some of the ports adjacent to the process. We will specify the presence of synchronization requirements in a production by means of a tuple of as many elements as the rank of L, where each element may be either a synchronization requirement or a special symbol (ε) which specifies the absence of synchronization requirement on the corresponding node. For example, $p^2 \stackrel{\langle \varepsilon,a \rangle}{\longrightarrow} \{q(x_1,x_3), r(x_3,x_2)\}^3$ poses the requirement a on node x_2. This means that, in order to apply this production to a graph G, not only we have to find an occurrence of $p(x_1,x_2)$, but also we need to make sure that all edges which are attached to the node of G corresponding to x_2 move as well, and with moves which specify the synchronization requirement a over it. These productions will be called *synchronized productions*.

Definition 12. (synchronized productions) Given two sets S_1 (of edge labels) and S_2 (of synchronization requirements), a synchronized production p over S_1 and S_2 is a triple $\langle L, K, R \rangle$ where $\langle L, R \rangle$, with $L \in (S_1)_n$, is a production as by Definition 9, and K is an n-tuple of elements of $S_2 \cup \{\varepsilon\}$. A synchronized production $prod = \langle L, K, R \rangle$ will be written as $prod = (L \stackrel{K}{\to} R)$. □

If $p \in (S_1)_n$, production $p^n \stackrel{\langle \varepsilon,\ldots,\varepsilon \rangle}{\to} \{p(x_1,\ldots,x_n)\}^n$, where there are n occurrences of ε on top of the arrow, is called the *synchronized idle production* for an edge (labeled) p, and will be written as s-idle(p).

Note that the presence of synchronization requirements makes the applicability condition for a synchronized production more complex than that for a production. In fact, it is not enough that an occurrence of L can be found in the graph to be rewritten. We also have to make sure that all edges sharing a node with the edge in L are rewritten as well, and by using synchronized productions which pose the same synchronization requirements on that shared node. In other words, a rewriting step in a graph rewriting system using synchronized productions involves the application of several productions (in any order), so that all edges sharing a synchronization node evolve according to one of their productions.

Definition 13. (synchronized rewriting) Consider a graph $G^n = \langle t_1, \ldots, t_m \rangle^n$ and a tuple of synchronized productions $P = \langle prod_i = (L_i \stackrel{K_i}{\to} R_i), i = 1, \ldots, m \rangle$. Then P is applicable to G if:

- for each $i = 1, \ldots, m$, $prod_i$ is applicable to t_i;
- for $h, h' \in \{1, \ldots, m\}$, if $t_h^k = p(x_{j_1}, \ldots, x_{j_i}, \ldots, x_{j_k})$ and $t_{h'}^{k'} = p'(x_{j'_1}, \ldots, x_{j'_{i'}}, \ldots, x_{j'_{k'}})$, with $x_{j_i} = x_{j'_{i'}}$, then it must be $K_h(i) = K_{h'}(i')$, where $K(i)$ is the i-th element of tuple K.

The rewriting of G via P, written as $G \Rightarrow_P H$, applies each production $prod_i = (L_i \to R_i)$, for $i = 1, \ldots, n$, to term t_i of G, in any order. □

Note that the application of an idle production s-idle(p) to t_i means that the edge represented by t_i is actually not rewritten, and therefore it will be present also in the graph resulting after the rewriting. Note also that $p(x_1, x_1)$ can only be rewritten using productions where $K(1) = K(2)$.

Definition 14. (synchronized rewriting system, synchronized derivations) A synchronized graph rewriting system is a quadruple $SGRS = \langle S_1, S_2, G_0, P \rangle$, where S_1 is a set of edge labels, S_2 is a set of synchronization requirements, G_0 is a graph over S_1 and P is a set of synchronized graph productions over S_1 and S_2. A synchronized derivation for $SGRS$ is a finite sequence of synchronized rewritings of the form $G_0 \Rightarrow_{P_1} G_1 \Rightarrow_{P_2} \ldots \Rightarrow_{P_n} G_n = H$ where each P_i, for $i = 1, \ldots, n$, is a tuple of synchronized productions in $P \cup \{s - idle(p)$, for all $p \in S_1\}$. □

Given a graph G^n and a synchronized graph rewriting system $SGRS = \langle S_1, S_2, G_0, P \rangle$, before rewriting G we have to find an n-tuple of productions of P which is applicable to G as by Definition 13. The problem of obtaining all such n-tuples (although formulated in a different way) is called the *rule-matching problem* in [6].

Note that, after the system moves via a certain synchronized rewriting step, we have a different distributed system. Thus a new search for an applicable set of synchronized productions has to be performed in order to understand which are the subsequent possible moves. On the other hand, note also that, in order to specify the behavior of a distributed system, one does not need to worry about the possibly complex problem to find such applicable sets of productions. In fact, the rule-matching problem is left to the system. This makes the specification task much easier. However, the system may take a long time to solve such a complex problem (that, as noted above, has to be solved before being able to actually make a move). This is why in Section 6 we will suggest some techniques to solve this problem in a distributed but efficient way.

5. Tiles for Graph Rewriting

We will now show how to model our notion of graph rewriting within the tile model.

First we need to choose the two signatures Σ_h and Σ_v, and the set of rewrite rules R, of Definition 7. In this way we have a tile rewrite system from which to obtain a gs-monoidal (or quasi-gs-monoidal) tile logic via Definition 8. We recall that, if $l \xrightarrow[b]{a} r$ is in R, then l and r are built over Σ_h while a and b are built over Σ_v, and it means that l is rewritten into r with a trigger a and an effect b. In our case, in which we want to describe graph rewriting, l and r are graphs, and more precisely l is a generator, while a represents the synchronization requirements for the rewriting; b is instead empty, since l and r are *graphs* (and not *open graphs*), and thus they have no output nodes. More precisely, b is the identity $id_{\underline{0}}$ of the neutral element $\underline{0}$ of the graph theory $G(\Sigma_v)$.

Thus Σ_h must contain operators corresponding to the edge labels, that is, labels $s \in S_n$, where n is the number of nodes connected by an edge and s is the edge label. More precisely, as noted in Proposition 6, $\Sigma_h = \{\Sigma_{n,m}\}_{n,m\in\omega}$ such that $\Sigma_{n,m} = \emptyset$ if $m \neq 0$, and $\Sigma_{n,0} = S_n$, with $n \in \omega$. In this way, the gs-monoidal theory $\mathbf{GS}(\Sigma_h)$ contains the arrows representing all and only the open graphs with labelled edges over $S = \{S_n\}_{n\in\omega}$.

Let us now consider signature Σ_v. This signature must contain the constructors which help us represent the effects of a graph rewriting, which have to do with the presence of synchronization requirements and the generation of new nodes. Therefore we have:

- $k : \underline{1} \rightarrow \underline{1}$ where k varies over the alphabet of synchronization requirements,
- $new : \underline{0} \rightarrow \underline{1}$ to model the creation of new nodes during a rewriting.

Now we come to the set R of rewrite rules, which specifies the generators of the gs-monoidal tile logic. In our case, the rewrite rules are exactly the synchronized productions of the graph rewriting system we want to model.

We recall that a synchronized production has the form $L \stackrel{K}{\to} R$. Let us assume that $L = p^n$ is the label of the edge to be rewritten, that $K = \langle k_1, \ldots, k_n \rangle$ is the tuple of synchronization requirements, and that R is a multiset of k edges containing m nodes which do not appear in L. To describe this synchronized production within the tile model, we use a tile $s \xrightarrow[b]{a} t$ where s represents L, t represents R, a represents K, and b is empty. More precisely,

- s is the arrow $L : \underline{n} \to \underline{0}$;
- t is the arrow $R : \underline{n+m} \to \underline{0}$;
- b is the arrow $id_{\underline{0}} : \underline{0} \to \underline{0}$
- a is the arrow $a_1 \otimes \ldots \otimes a_n \otimes b_1 \otimes \ldots \otimes b_m : \underline{n} \to \underline{n+m}$, where
  - $a_i = id_{\underline{1}} : \underline{1} \to \underline{1}$ if $k_i = \varepsilon$, otherwise $a_i = k_i : \underline{1} \to \underline{1}$;
  - $b_i = new : \underline{0} \to \underline{1}$ for all $i = 1, \ldots, m$.

Notice that the tile corresponding to an idle production of the form $p^n \to \{p(x_1, \ldots, x_n)\}^n$ is the tile id_p (see its definition in rule *(h-refl)* of Definition 8).

Notice also that Σ_h and Σ_v do not depend on the given synchronized graph rewriting system, while R does depend on it, since it contains a description of all its synchronized productions. Therefore, given a synchronized graph rewriting system S, we will write $R(S)$ to denote the corresponding set of tiles, and $TRS(S) = \langle \Sigma_h, \Sigma_v, R(S) \rangle$ to denote the corresponding tile rewrite system. Also, since arrow b in a tile $s \xrightarrow[b]{a} t$ representing a synchronized production is always the same (that is, b is the arrow $id_{\underline{0}} : \underline{0} \to \underline{0}$), in the following we will not specify it.

For example, consider again the synchronized production $p^2 \stackrel{\langle \varepsilon, c \rangle}{\longrightarrow} \{q(x_1, x_3), r(x_3, x_2)\}^3$. The corresponding 4-tuple is $s \xrightarrow[b]{a} t$, where

- s is the arrow $\{p(x_1, x_2)\}^2 : \underline{2} \to \underline{0}$;
- t is the arrow $\{q(x_1, x_3), r(x_3, x_2)\}^3 : \underline{3} \to \underline{0}$;
- a is the arrow $id_{\underline{1}} \otimes c \otimes new : \underline{2} \to \underline{3}$.

Another example is the synchronized production $p^2 \xrightarrow{\langle\varepsilon,c\rangle} \{q(x_1,x_3), r(x_3,x_1)\}^3$. Note that node x_2 does not appear in the right hand side of the production. However, this node must be considered in the corresponding tile, which is as follows:

- s is the arrow $\{p(x_1,x_2)\}^2 : \underline{2} \to \underline{0}$;
- t is the arrow $\{q(x_1,x_3), r(x_3,x_1)\}^3 : \underline{3} \to \underline{0}$ (note that the source of this arrow is $\underline{3}$ although there are only two nodes appearing in R);
- a is the arrow $id_{\underline{1}} \otimes c \otimes new : \underline{2} \to \underline{3}$.

In other words, node x_2 is not deleted from the graph, but it is just treated as an isolated node.

We do not need any other rewrite rule in R, besides those representing the synchronized productions of our graph rewriting system, since the basic, composition, and auxiliary rules of the tile logic built on top of the given tile rewrite system (see Definition 8) automatically generate all we need.

What is crucial in this modeling of a synchronized graph rewriting system within the tile model is that the obtained tile logic "satisfies" the synchronization requirements posed by the synchronized productions. In other words, the sequents $s \xrightarrow[b]{a} t$ of the tile logic, where s is a graph, correspond exactly to those derivations which are allowed in the given synchronized graph rewriting system.

However, obviously tiles $s \xrightarrow[b]{a} t$ of the form id_a, $\rho_{a,b}$, and ∇_a cannot correspond to any derivation, since in these cases s does not represent a graph. Nevertheless, this does not mean that these tiles cannot be used to create more complex tiles which do correspond to graph derivations.

Another tile that should be avoided in the correspondence between a tile logic and a graph rewriting system is the tile ∇_{new}, which would allow the identification of two nodes generated by one or more productions. In fact, this kind of identification is not possible in the graph rewriting approach of Section 4 (and also in all other graph rewriting systems we know of), and thus no derivation can use tile ∇_{new}. To show that this restriction is indeed very reasonable in graph rewriting, let us consider an example with a production (without synchronization requirements) $p^1 \to q(x_1,x_2)$ and a graph $G = \{p(x_1), p(x_1)\}^1$, and let us apply this production twice to graph G. One of the possible two results of the first application is graph $G_1 = \{q(x_1,x_2), p(x_1)\}^2$. The other result is $\{q(x_2,x_1), p(x_2)\}$. Then, if we apply the given production to G_1, one of the possible (three) results is $G_2 = \{q(x_1, x_2), q(x_1, x_3)\}$. All other resulting graphs differ for permutations of the nodes, but all of them have different nodes as second nodes of the two edges labelled q. In other words, nodes x_2 and x_3 in G_2 were not present in G and are generated by two different rewriting steps. Therefore it is reasonable to avoid having the possibility of merging them into one. It is exactly to allow for the elimination of ∇_{new} that in Definition 8 we introduced the notion of a quasi-gs-monoidal tile logic, which, we recall, does not consider some constructors of Σ_v in applying rule *dupl*. In particular, in the following theorem we will use this notion to avoid the use of constructor *new* in that rule.

Theorem 15. (derivation tiles and rewriting sequences) Consider a synchronized graph rewriting system S, the corresponding TRS $R = TRS(S)$ and the derivation tiles generated by its quasi-gs-monoidal tile logic $DT(R, \{new\})$. Given a graph G_0, there exists a synchronized derivation $G_0 \Rightarrow_{P_1} G_1 \Rightarrow_{P_2} \ldots \Rightarrow_{P_n} G_n$ if and only if there is a tile $s \xrightarrow[b]{a} t$ in $DT(R, \{new\})$ such that $s = G_0$ and $t = G_n$.

Proof. We show that given a graph G there is a synchronized rewriting step $G \Rightarrow_P G'$ if and only if there exists a horizontal tile $s \xrightarrow[b]{a} t$ in $HT(R\{new\})$ with $s = G$ and $t = G'$. Then the statement of the theorem follows by composing in sequence the rewriting steps and vertically the corresponding horizontal tiles.

First we recall that, as explained in Section 2, the initial graph $G = \langle t_1, \ldots, t_m \rangle^n$ can be represented in a canonical way as $G = \sigma_n; (t_1 \otimes \ldots \otimes t_m)$.

To prove the "only if" part of the theorem, we observe that, since the synchronization requirements are satisfied in $G \Rightarrow_P G'$, a corresponding expression

$$\sigma_2 * \sigma_n * \sigma_1 * (t_1 \otimes \ldots \otimes t_m) \tag{2}$$

can be evaluated on tiles, yelding a tile $G \xrightarrow[b]{a} G'$. To evaluate σ_n and $(t_1 \otimes \ldots \otimes t_m)$ on tiles, it is sufficient to replace homomorphically:

- every edge t_i with the production applied to it in the given synchronized derivation;
- every identity $id = id_{\underline{1}}$, permutation $\rho = \rho_{\underline{1},\underline{1}}$, duplicator $\nabla = \nabla_{\underline{1}}$, and discharger $! = !_{\underline{1}}$ in σ_n, with tiles id_a, $\rho_{a,a'}$, ∇_a and $!$, where a and a' are either $id_{\underline{1}}$ or some synchronization requirement k_i.

The choice among the two alternatives depends on the corresponding node: we choose k_i if the node is a synchronization node with requirement k_i, $id_{\underline{1}}$ otherwise;

- parallel ($\otimes$) and sequential (;) composition are replaced with parallel ($\otimes$) and horizontal ($*$) composition on tiles.

Tiles σ_1 and σ_2 correspond to substitutions ρ_1 and ρ_2 of Definition 10. They are obtained by composing horizontally tiles of the form $\rho_{a,new}$, $\rho_{new,a}$ and $\rho_{new,new}$, where a is as above. Tile σ_1 moves all the new nodes to the end, while tile σ_2 can interleave them in all ways with the old nodes. Notice that all these tiles have $id_{\underline{1}}$ as top side, i.e., they do not change the top side of the resulting tile, i.e. G. Notice also that the order of old nodes (among them) cannot be changed.

To prove the "if" part of the theorem, one assumes to have a tile $s \xrightarrow[b]{a} t$ where $s = G$. The tile must be constructed using the rules of Definition 8, except rule (*v-comp)*. Its derivation can be projected on the top side and yields a derivation for G. By essentially mimicking the proof which brings this derivation to its canonical representation, it is possible to bring also the tile derivation to the corresponding normal form (2) above. But the evaluation of this form corresponds to the construction of the rewriting step according to Definitions 9-13. $\square$

As noted above, this theorem uses a quasi-gs-monoidal tile logic, instead of a gs-monoidal one, to eliminate the possibility of having tiles of the form ∇_{new} (see rule *quasi-dupl*) of Definition 8. In fact, the effect of this rule, which is the merging of nodes generated via the constructor *new*, cannot be obtained in the kind of graph rewriting we have defined. Notice however that this does not mean that new nodes cannot appear more than once in the right-hand side of a production (and thus in the graph resulting after the rewriting). In fact, to express this kind of sharing we do not need a ∇_{new} tile, but we just need to use the $\nabla_{\underline{n}}$ constructor while describing the state t in the tile $s \xrightarrow[b]{a} t$ representing the production. For example, production $p^1 \overset{\langle\varepsilon\rangle}{\to} \{q(x_1, x_2), r(x_2)\}^2$ generates a new node, x_2, which is shared among edges q and r. The corresponding tile would be $s \xrightarrow[b]{a} t$, where

- s is the arrow $p : \underline{1} \to \underline{0}$;
- t is the arrow $\{q(x_1, x_2), r(x_2)\}^2 : \underline{2} \to \underline{0}$, which is obtained as $(id \otimes \nabla); (q \otimes r)$;
- a is the arrow $id_{\underline{1}} \otimes new : \underline{1} \to \underline{2}$.

Let us now see an example of the correspondence between a derivation and a tile. Consider a one-step derivation starting from graph $G = \{p(x_1, x_2), p(x_1, x_2)\}^2$ and using the synchronized production $p^2 \stackrel{\langle k, \varepsilon \rangle}{\rightarrow} \{q(x_1, x_2)\}^2$. The only non-trivial synchronized rewriting step applies this production on both edges obtaining the graph $H = \{q(x_1, x_2), q(x_1, x_2)\}$. Let us now see how to construct the tile corresponding to this derivation. First we consider the tile $\alpha_p = s_p \xrightarrow[b_p]{a_p} t_p$ representing the production, where s_p is the arrow $p : \underline{2} \to \underline{0}$, t_p is the arrow $q : \underline{2} \to \underline{0}$, b_p is the arrow $id_{\underline{0}} : \underline{0} \to \underline{0}$, and a_p is the arrow $a \otimes id_{\underline{1}} : \underline{2} \to \underline{2}$. To represent the fact that this production has to be applied on both edges, we compose in parallel α_p with itself, obtaining the tile $\alpha = \alpha_p \otimes \alpha_p = p \otimes p \xrightarrow[id_{\underline{0}} \otimes id_{\underline{0}}]{k \otimes id_{\underline{1}} \otimes k \otimes id_{\underline{1}}} q \otimes q$. Now we must represent the fact that the two occurrences of p share two nodes. This is done by first reordering the nodes, if needed, and then by using the ∇_a tiles. In our case it is enough to consider the tile $\sigma = \nabla_{k \otimes id_{\underline{1}}} = \nabla_{\underline{2}} \xrightarrow[k \otimes id_{\underline{1}} \otimes k \otimes id_{\underline{1}}]{k \otimes id_{\underline{1}}} \nabla_{\underline{2}}$. This tile can be composed horizontally with α obtaining $\sigma * \alpha = s \xrightarrow[b]{a} t$, where s is $\nabla_{\underline{2}}; (p \otimes p) : \underline{2} \to \underline{0}$, t is $\nabla_{\underline{2}}; (q \otimes q) : \underline{2} \to \underline{0}$, a is $k \otimes id_{\underline{1}} : \underline{2} \to \underline{2}$, and b is $id_{\underline{0}} : \underline{0} \to \underline{0}$. Therefore s represents $\{p(x_1, x_2), p(x_1, x_2)\}^2$ and t represents $\{q(x_1, x_2), q(x_1, x_2)\}^2$. Thus, this is the tile corresponding to the given derivation.

Note that this last example is particularly simple because no new nodes are created by the rewriting step. If instead some new nodes are indeed created, then the construction of the tile representing the rewriting will follow the same lines, but will also require to insert a tile σ_1 in $\sigma * \sigma_1 * \alpha$ (as mentioned in the proof of Theorem 15 above) to move all the new nodes to the end. If we stop here, the resulting tile would represent just one result of the graph rewriting step: the one where all the new nodes are ordered after the old ones. However, nothing would prevent us from composing horizontally this tile with other tiles like $\rho_{k,new}$, $\rho_{id_{\underline{1}},new}$, or $\rho_{new,new}$ (see rule *(perm)* of definition 8), thus obtaining a resulting tile where the upper border is the same as before, but in the lower border the new nodes are ordered in a different way. This is the reason why we used permutation ρ_2 in the definition of graph rewriting, otherwise we would not have had a precise correspondence between graph rewriting and tile rewriting. That is, we would have had some tiles which did not correspond to any graph rewriting sequence. A simple example which shows this issue is the following one. Consider a graph $G^1 = \{p(x_1)\}$ and a production $p^1 \to \{q(x_1, x_2)\}^2$. Then the tile which is obtained by the constructive method we used for the previous example is $s \xrightarrow[b]{a} t$, where

- $s = p : \underline{1} \to \underline{0}$,
- $t = q : \underline{2} \to \underline{0}$,
- $a = id_{\underline{1}} \otimes new : \underline{1} \to \underline{2}$.

This tile represents the rewriting of G via the given production which obtains the new graph $H_1 = \{q(x_1, x_2)\}$. Now, this tile, say α, can be horizontally composed with the tile $\sigma_2 = \rho_{new,id_{\underline{1}}}$, which just reorders the two nodes (the new one and the old one). Therefore, $\sigma_2 * \alpha$ would again represent a possible rewriting of G via the given production, but the result would now be the graph $H_2 = \{q(x_2, x_1)\}$.

6. Constraint Problems and Constraint Propagation

A (finite domain) constraint problem [30, 35] consists of a set of variables ranging over a finite domain, and a set of constraints. A solution to such problem is an instantiation of all the variables such that all the constraints are satisfied. Formally, a (finite domain) constraint satisfaction problem (CSP) is a tuple

$$\langle V, D, C, con, def \rangle$$

where

- V is a finite set of *variables* (i.e., $V = \{v_1, \ldots, v_n\}$);
- D is a finite set of values, called the *domain*;
- C is a finite set of *constraints* (i.e., $C = \{c_1, \ldots, c_m\}$); C is ranked, i.e. $C = \bigcup_k C_k$, such that $c \in C_k$ if c involves k variables;
- con is called the *connection function* and it is such that $con : \bigcup_k (C_k \to V^k)$, where $con(c) = \langle v_1, \ldots, v_k \rangle$ is the tuple of variables involved in $c \in C_k$;
- def is called the *definition function* and it is such that $def : \bigcup_k (C_k \to \wp(D^k))$, where $\wp(D^k)$ is the powerset of D^k.

Function con describes which variables are involved in which constraint, while function def gives the meaning of a constraint in terms of a set of tuples of domain elements, which represent the allowed combinations of values for the involved variables. Then, the solution $Sol(P)$

of a CSP $P = \langle V, D, C, con, def \rangle$ is defined as the set of all instantiations of the variables in V (seen as tuples of values) which are consistent with all the constraints in C.

Local consistency algorithms [30, 35, 29] remove from a CSP some domain elements or also some tuples from the constraint definitions if these objects are found to be inconsistent with some other object in the CSP. This is safe (that is, it does not change the set of solutions of the CSP), because local inconsistency implies global inconsistency, and thus such objects would never appear in any solution of the CSP. However, there may be objects (tuples and/or domain elements) which are inconsistent but are not recognized as such and therefore are not removed. Thus in general only local consistency is achieved (and not global consistency, which would mean that the problem is solved).

The first local consistency algorithms have been called arc-consistency [29] and path-consistency [35]. Later, both of them were generalized to the concept of k-consistency [20]: a CSP is k-consistent if, for each k-1 variables, and values for them which are allowed by all the constraints involving subsets of them, and for each other variable, there is at least a value locally allowed for such k-th variable which is compatible with all the other k-1 variables. In this line, arc-consistency [29] is just 2-consistency and path-consistency is 3-consistency.

A local consistency algorithm is an algorithm which achieves a certain degree of local consistency (2, 3, or k) in a constraint problem. This is usually done by an iterative scheme which stops only when no other domain value or tuple can be deleted. Here we will just consider local consistency algorithms which remove values from variable domains, and not tuples from constraints. This poses some limitations on the power of these algorithms, but it assures that no new constraint is added by them. In fact, it has been proved that in general achieving k-consistency may add constraints involving at most k-1 variables [19]. The fact that no constraint is added makes the algorithm much more natural to be used in a distributed setting, where a change in the structure of the processes interconnection may be undesirable.

The simplest, but most of the times quite effective, local consistency algorithm is the one which achieves arc-consistency [29]: only one constraint at a time is considered, and, for each variable connected to it, it is checked whether there is a value of its domain which is not compatible with any other value of the other variables (according to the constraint). If so, such value is deleted.

Local consistency algorithms are very useful because they are polynomial in the worst case (in the number of variables) [31] and at the same time they are able, in several cases, to remove many redundancies in the domains. This result can make the subsequent search algorithm, used to find solutions of the constraint problem, much faster.

Once the given constraint problem has been processed by a local consistency algorithm and some redundant values have been removed from each variable domain, we can now obtain all possible solutions of the current constraint problem. This can be done via a distributed backtracking search algorithm, where each of the concurrent agents corresponds to one variable (to be instantiated over its domain) and communicates with the other agents via the constraint topology.

We will not go here into the detail of any backtracking-search algorithm, pointing instead to an already developed algorithm presented in [42, 43], and presenting here just its basic features. Agents of the algorithm are processes, to be assigned one of their context-free productions, among the ones that are left after the application of some local consistency algorithm. We recall that each process has always the idle production, which for sure is still one of the possibilities after local consistency. This allows to obtain partial solutions besides complete ones. Each agent knows the possible choices for its process and also the ports such a process shares with others. Assuming a fixed total ordering of the agents, each agent, concurrently to all the others, waits until new synchronization requirements are posted on the ports it shares with agents above it in the ordering. When this happens, it chooses one of its context-free productions which agrees with the requirements, and posts the corresponding new requirements on the ports the process shares with agents below it in the ordering. If it has no choice which satisfies such requirements, then it posts a failure on the ports the process shares with the agent which is the lowest one among those above it. When an agent sees one or more failures from below, it tries to make another choice, still compatible with the information coming from above. Attention has to be put to the fixed ordering of the agents, in order to avoid incorrect behaviors of the algorithm. In fact, if the order does not correspond to a path in the graph (that is, processes which are handled by agents which are one the successor of the other one in the ordering are not adjacent), then it is possible that failures are not communicated properly, and that thus a global failure is returned while instead there is a solution. Another way to solve this problem is to dynamically change the structure of the connections among agents, in particular by adding new connections whenever needed [42].

7. Constraints for the Rule-Matching Problem

The rule-matching problem can be represented as a finite domain constraint problem as follows: variables represent processes and constraints represent ports (that is, sets of adjacent processes). The domain of a variable is the set of all context-free productions for the corresponding process, and the definition of a constraint is the set of tuples of context-free productions (one for each adjacent process) whose synchronization requirements match on that port.

A solution to this kind of constraint problem is, according to the above definition, a choice of a context-free production for each process, such that all synchronization requirements are satisfied. Thus such solution specifies a legal move of the distributed system, where each process proceeds according to the chosen production. This move is legal (that is, it corresponds to one of the rewriting rules that would be generated by merging some of the context-free production, as informally stated above) because it is obvious that no synchronization requirements are violated, otherwise some constraint would not be satisfied. It is possible to show that the number of solutions of the constraint problem is the same as the number of rewrite rules obtainable via the definition above. Thus this approach actually solves the rule-matching problem.

A distributed algorithm which takes the representation of the current state of the system and produces a finite domain constraint problem representing the current rule-matching problem can be easily outlined. In fact, variables are associated to processes, and constraints to ports. Since adjacent processes communicate via ports, also this information is directly and locally available.

It is now clear that local consistency algorithms can be applied to the constraint problem representing the current rule-matching problem in order to remove redundant values in the domain of the variables (that is, redundant choices for context-free productions which would never be used in any context-sensitive rewrite rule). The resulting problem (after the application of the chosen local consistency algorithm) therefore represents a new rule-matching problem (but equivalent to the given one, since it generates exactly the same possible moves), where some of the redundant context-free productions have been deleted.

In particular, the arc-consistency algorithm is the most natural in a distributed environment, since it has to deal with only one constraint at a time. Thus one may use a distributed version of this algorithm where a set of concurrent agents cooperate to remove redundant values from the variable domains, each agent corresponds to one of the constraints, and communication occurs via shared variables [37].

8. An Example

Let us now clarify the ideas described above via a simple but expressive example, adapted from [11]. Consider a distributed system representing a synchronous communication network connecting a number of stations. The system can increase its size dynamically but its topology is always tree-like. Each station may decide whether to request a communication or not. In a certain state of the system, a communication is established between two communication-ready stations.

There are three types of processes:

- process A represents a station which is ready to communicate,
- process B represents a station which does not want to communicate (thus it can only make the system increase its size or transform itself into a communication-ready station), and
- process C represents a communication switch with three ports, which has three possible ways to let the communication pass (by choosing any two of its three ports).

The synchronized productions specifying how the single processes may evolve can be seen in Figure 4. The graphical way we represent these productions consists of specifying hypergraph L (the left hand side) and graph R (the right hand side). Multiple productions with the same left hand side are represented by giving the left hand side only once and separating the right hand sides by a bar. In a graph, nodes are circles and edges are boxes, connected to as many nodes as their arity. The way nodes of L and R map to each other, for this example, should be clear from the picture. When a production has some synchronization requirements, they are written on the corresponding nodes of the right hand side. For this example, there is only one kind of synchronization requirement, which we call a.

¿From Figure 4 we can see that:

- A can only transform itself into B, while synchronizing over the only node it is connected to. Thus this production can be written as $A \stackrel{\langle a \rangle}{\rightarrow} \{B(x_1)\}^1$.
- B can evolve in two different ways: it can either transform itself into A, with the production $B \rightarrow \{A(x_1)\}^1$, or it can change the structure of the communication net by introducing a new switch. Two ports of such new switch are connected to B processes, while the other one is free. After rewriting, all the nodes of the graph which are connected to the node marked 1 in the left hand side are

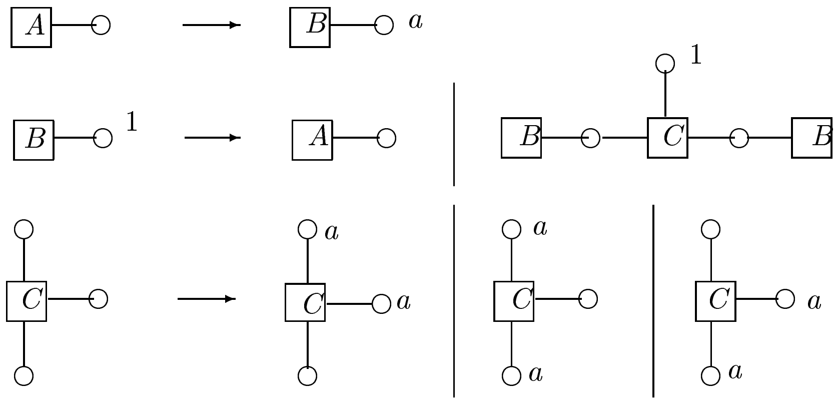

Figure 4. A set of context-free productions.

attached to the node marked 1 in the right hand side. No marking has been used for the other production because the geometrical correspondence suffices. The production representing this second possibility is $B \rightarrow \{B(x_2), C(x_2, x_1, x_3), B(x_3)\}^3$. Note that none of these two moves require any synchronization. Thus B can move in isolation according to any one of the two productions.

- C can only move in a synchronized way: it can choose any pair of its three ports and decide to synchronize over such two ports. The three productions representing this behavior are:

$$C \stackrel{\langle a,a,\varepsilon\rangle}{\rightarrow} \{C(x_1, x_2, x_3)\}^3$$

$$C \stackrel{\langle a,\varepsilon,a\rangle}{\rightarrow} \{C(x_1, x_2, x_3)\}^3$$

$$C \stackrel{\langle \varepsilon,a,a\rangle}{\rightarrow} \{C(x_1, x_2, x_3)\}^3$$

The synchronization required by the only production for A means that the station represented by these A-process is chosen for the communication. In the same way, the choice of any of the productions for C means that communication passes through the two ports over which synchronization has occurred. Since rewriting can only occur via these productions, any state of this system is a tree-like structure of C-processes with As or Bs at the leaves. Due to the synchronizations required by the productions, each context-sensitive rewriting rule for the system (whatever its current state is) involves either a single B-process (and it is specified by one of its context-free productions), or a subpart of the system consisting of two A-processes and a chain of C-processes. This subpart obviously specifies a path in the tree-like structure of the system which goes from one leaf to another one.

Thus the rule-matching problem, for this example, consists the problem of choosing one of the possibly many paths going from one leaf labeled A to another leaf labeled A. Consider for example the states depicted in Figure 5 a) and b). In Figure 5 a), there are only two A-processes, and thus only one possible communication path (outlined by a bidirectional arrow). Instead, in Figure 5 b), there are three A-processes, and thus three possible communication paths.

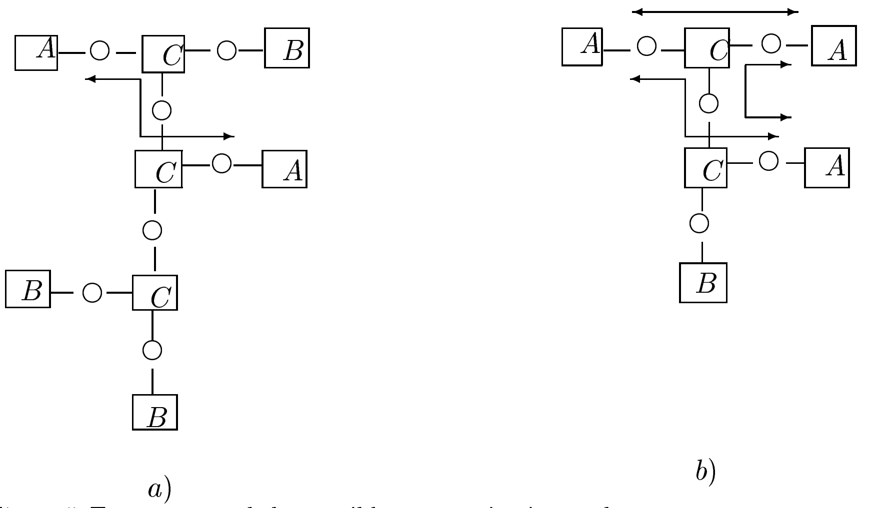

Figure 5. Two states and the possible communication paths.

Consider now the state in Figure 5 b). The constraint problem corresponding to this state has as many variables as the processes (thus 6 variables), and as many constraints as the ports (thus 5 constraints). Each constraint connects only two variables, since every port of this example connects only two adjacent processes. The domain of each variable is the set of context-free productions available for the corresponding process. Thus, the domain corresponding to an A-process consists of just one production, that of a B-process consists of two productions, and that of a C-process consists of three productions (apart from the idle production which we did not explicitly specify in the example but which is always assumed to be a possible production for any process).

Consider now what an arc-consistency algorithm would do on this constraint problem. We recall that an arc-consistency algorithm looks at each variable, say x, and, for each other variable connected to it, say y, checks whether there are values in the domain of x which are not compatible with any other value for y. If so, that value for x is deleted. This continues until no more deletions are possible. In this case, consider the variable corresponding to the lower C-process. Since a port of such process is connected to a B-process, which does not have any production with synchronization, there is no way that C can use any one of its two productions which require synchronization over that port. Thus these two productions can be eliminated. Therefore the only production remaining for that C-process is the one which lets the communication pass through the upper and right ports of the process. No other pruning is done by the algorithm.

Consider now the state in Figure 5 a). In this case, the domain of the variable corresponding to the lower C-process becomes empty. In fact, since such a process is connected to two B-processes, it cannot use any one of its three productions, since all of them require synchronization over one or two of the ports connected to a B-process. Moreover, because of this, also the domain of the variable corresponding to the central C-process is pruned. In fact, the two productions which require synchronization over the lower port of this process cannot be used. Thus the only production left for this C-process is the one which lets the communication pass in the upper and right port. Also, for the same reason as the pruning occurred in Figure 5 b), the domain of the upper C-process gets pruned as well, remaining with just the production which lets the communication pass in the left and lower ports. At this point the arc-consistency algorithm stops because no more pruning is possible.

Notice, however, that in both cases the pruning performed by arc-consistency is enough to directly solve the rule-matching problem without any search. In fact,

- in the state of Figure 5 a), all the C-processes present in the system have either one or zero possibilities. Thus no choice has to be made during the search for a solution: the C-processes with one possibility will move according to that, while the one with the empty domain will stay idle.

- in the state of Figure 5 b), even though a C-process still has more than one possibility, it is still possible to find a solution of the rule-matching problem without any search. In fact, since the constraint problem has a tree structure and it has been made arc-consistent, by choosing a search order which traverses the tree top-down, given the values chosen for the variables already instantiated, we can always extend this partial solution to a total solution. This is a classical result in constraint solving [20]. In our example, since the variables are the A, B, and C processes, we can choose to order them in the following way: first the upper C, then the the lower C, and then all the A and B processes. This is one possible ordering which visits the tree top-down. Given this ordering, the first variable (the upper C) has three possible values. Let us choose one of them: the one that lets the communication pass in the left and lower ports. Then we pass to the lower C process: this variable has just one possibility (the one that lets the communication pass in the upper and right port). Given these choices for the two C processes, all the other processes have no choice: the left A and the lower right A will communicate, while the B and the other A process will remain idle. It is possible to see that any other top-down ordering and any other choice for the value of the upper C will lead to a solution as well, although different from this one.

Therefore, the analysis of these two cases shows that in any state generated by the grammar of this example, arc-consistency plus a linear search is enough to solve the problem. Of course we cannot generalize this statement to any grammar, but only to those where each state is tree-like. This means that the initial graph is tree-like and that the right hand side of every production is tree-like. If one, or both, of these two conditions does not hold, then we may obtain a state which is not tree-like. Therefore, we may have to search and backtrack over the search tree. However, arc-consistency will always improve the search by performing some pruning in such a tree.

9. Conclusions

To summarize, we may list the main points of our proposal as the following ones:

- adopt *graphs* to describe system states and *graph rewriting* to describe system evolution;
- exploit the expressive power and generality of the *tile model* to describe the kind of graph rewriting we use for distributed systems with synchronization;
- use *constraint problems* and the corresponding *local consistency* algorithms to remove redundancies in the description of a system state;
- apply a *distributed backtracking search* algorithm, like those used for distributed constraint solving, to solve the rule-matching problem, and thus choose one possible move for the system. Such a move will in general involve several processes acting in synchrony.

The novelty of our approach is not in its individual ingredients, which have all being developed previously (apart from the need of small variations for making them fit for the desired distributed environment), but instead in the combination of all of them, which seems to us a convenient alternative way to model the dynamic of distributed systems, w.r.t. the methods usually used. In particular, the convenience of our proposal comes from various aspects:

- graphs are a very natural formalism for describing situations where there are agents or processes (that is, edges) connected via ports (that is, nodes);
- graph rewriting allows for a faithful model of synchronized process evolution, since all the parts which belong to the left hand side of the same rewrite rule must be rewritten simultaneously, and thus synchronously;
- context-free graph rewriting is natural for distribution, since context-freeness assures that each process can evolve independently on the others (once the desired synchronization requirements have been checked and satisfied);
- the ability of the tile model to formally separate the description of sequential rewriting steps from the coordinating of such steps (here given by the matching of the synchronization requirements) is crucial in making the whole methodology very flexible and modular, since it is possible to change the rewriting aspect without influencing the coordination part, and vice versa; moreover, its generality (here we defined a simpler version of the tile model defined in [23]) suggests that this same style could be applicable also to other models of computations;
- constraint problems model very naturally the rule-matching problem, where there are processes with several choices (that is, variables with their domains) and synchronization requirements (that is, constraints);
- local consistency algorithms have been proved, by a long theoretical study and practical experimentation, to achieve a great amount of redundancy removal, and thus a great improvement in the solution process, in many cases, which are related for example to the structure of the constraint problem [10], or to the semantics of the constraints [35], or to the size of the domains [10], or also to the tightness of the constraints [41];
- distributed backtracking has been shown to be convenient when the structure of the process connection is not too dense [42]; thus in

this cases the performance of this way of solving the rule-matching problem is feasible in practice.

References

1. H.P. Barendregt, M.C.J.D. van Eekelrn, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, M.R. Sleep. Term Graph Reduction, in *Proc. PARLE*, Springer LNCS 259, 141–158, 1987.
2. Paul Boehm, Harald-Reto Fonio, and Annegret Habel. Amalgamation of graph transformations: A synchronization mechanism, in *Journal of Computer and System Sciences*, 34(2/3):377-408, April/June 1987.
3. R. Bruni and U. Montanari. Zero-Safe Nets, or Transition Synchronization Made Simple, in: Catuscia Palamidessi, Joachim Parrow, Eds, *Proc. EXPRESS'97*, ENTCS, Vol. 7, `http://www.elsevier.nl/locate/entcs/volume7.html`.
4. R. Bruni, and U. Montanari. Zero-Safe Nets: The Individual Token Approach, in: Francesco Parisi-Presicce, Ed., *Proc. 12th WADT Workshop on Algebraic Development Techniques*, Springer LNCS, 1998, to appear.
5. P. Ciancarini, C. Hankin (Eds.), *Coordination Languages and Models*, Proceedings, Springer-Verlag, LNCS 1061, 1996.
6. A. Corradini, P. Degano, and U. Montanari. Specifying highly concurrent data structure manipulation. In G. Bucci and G. Valle, editors, *COMPUTING 85: A Broad Perspective of Current Developments*. Elsevier Science, 1985.
7. A. Corradini, F. Gadducci, An Algebraic Presentation of Term Graphs via Gs-Monoidal Categories, *Applied Categorical Structures*, to appear. Available at http://www.di.unipi.it/ gadducci/papers/aptg.ps, 1997.
8. A. Corradini, F. Gadducci, A 2-Categorical Presentation of Term Graph Rewriting, in *Proc. CTCS'97*, LNCS 1290, 1997.
9. A. Corradini and U. Montanari, editors. *Proc. SEGRAGRA95 (Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation)*. Elsevier, Electronic Notes on Theoretical Computer Science, 1995.
10. Rina Dechter. From local to global consistency. *Artificial Intelligence*, 55:87–107, 1992.
11. P. Degano and U. Montanari. A model for distributed systems based on graph rewritings. *Journal of ACM*, 34(2), 1987.
12. R. Dechter and J. Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. In Kanal and Kumar, editors, *Search in Artificial Intelligence*. Springer-Verlag, 1988.
13. C. Ehresmann, *Catégories Structurées*: I and II, Ann. Éc. Norm. Sup. 80, Paris (1963), 349-426; III, Topo. et Géo. diff. V, Paris (1963).
14. H. Ehrig. Introduction to the algebraic theory of graph grammars. In *Proc. International Workshop on Graph Grammars*. Springer Verlag, LNCS 73, 1978.
15. H. Ehrig. Tutorial introduction to the algebraic theory of graph grammars. In *Proc. 3rd International Workshop on Graph Grammars*. Springer Verlag, LNCS 291, 1987.
16. G. Ferrari and U. Montanari, A Tile-Based Coordination View of π-Calculus, in: Igor Prívara, Peter Ruzicka, Eds., *Mathematical Foundations of Computer Science 1997*, Springer LNCS 1295, 1997, pp. 52-70.
17. G. Ferrari, U. Montanari, Tiles for Concurrent and Located Calculi, in: Catuscia Palamidessi, Joachim Parrow, Eds, *Proc. EXPRESS'97*, ENTCS, Vol. 7, `http://www.elsevier.nl/locate/entcs/volume7.html`.
18. G. Ferrari and U. Montanari, Tile Formats for Located and Mobile Systems, paper submitted for publication.

19. E. Freuder. Synthesizing constraint expressions. *Communication of the ACM*, 21(11), 1978.
20. E. Freuder. Backtrack-free and backtrack-bounded search. In Kanal and Kumar, editors, *Search in Artificial Intelligence*. Springer-Verlag, 1988.
21. F. Gadducci, *On the Algebraic Approach to Concurrent Term Rewriting*, PhD Thesis, Università di Pisa, Pisa. Technical Report TD-96-02, Department of Computer Science, University of Pisa, 1996.
22. F. Gadducci and U. Montanari, Tiles, Rewriting Rules and CCS, in *Proc. 1st international workshop on Rewriting Logic and Applications*, J. Meseguer Ed., ENTCS 4 (1996), pp.1-19.
23. F. Gadducci, U. Montanari, *The Tile Model*, in G. Plotkin, C. Stirling and M. Tofte, Eds., *Proofs, Languages and Interaction: Essays in Honour of Robin Milner*, MIT Press, 1998.
24. A. Habel. *Hyperedge Replacement: Grammars and Languages.* PhD thesis, University of Bremen, 1989.
25. G.M. Kelly, R.H. Street. Review of the Elements of 2-categories, *Lecture Notes in Mathematics* 420, 1974, pp. 75-103.
26. J.W. Klop. Term Rewriting Systems, in *Handbook of Logic in Computer Science*, Vol. I, eds. S. Abramsky, D. Gabbay, and T. Maibaum, Oxford University Press, 1992, pp. 1-116.
27. K.G. Larsen, L. Xinxin. Compositionality Through an Operational Semantics of Contexts, in *Journal of Logic and Computation*, vol.1, n.6, 1991 (conference version in Proc. ICALP'90, Springer-Verlag, LNCS 443, 1990).
28. F. W. Lawvere. Functorial Semantics of Algebraic Theories, *Proc. National Academy of Science* **50**, 1963, pp. 869-872.
29. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1), 1977.
30. A.K. Mackworth. *Encyclopedia of AI*, chapter Constraint Satisfaction, pages 205–211. Springer Verlag, 1988.
31. A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25, 1985.
32. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency, *Theoretical Computer Science*, 96, 1992, pp. 73-155.
33. J. Meseguer. Rewriting Logic as a Semantic Framework for Concurrency: A Progress Report, in: U. Montanari and V. Sassone, Eds., *CONCUR'96: Concurrency Theory*, Springer LNCS 1119, 1996, 331-372.
34. J. Meseguer and U. Montanari. Mapping Tile Logic into Rewriting Logic, in: Francesco Parisi-Presicce, Ed., *Proc. 12th WADT Workshop on Algebraic Development Techniques*, Springer LNCS, 1998, to appear. Paper available from `http://www. csl.sri.com/ ugo/wadt.ps`.
35. U. Montanari. Networks of constraints: Fundamental properties and application to picture processing. *Information Science*, 7, 1974.
36. U. Montanari and F. Rossi. Graph Rewriting and Constraint Solving for Modelling Distributed Systems with Synchronization. *Proc. COORDINATION96*, Springer-Verlag, LNCS 1061, 1996.
37. T. Nguyen and Y. Deville. A distributed arc-consistency algorithm. In *Proc. of the CCP95 international workshop*, 1995.
38. G. Plotkin, *A Structural Approach to Operational Semantics*, Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
39. H.J. Schneider and H. Ehrig, editors. *Graph Trasformations in Computer Science.* Springer Verlag, LNCS 776, 1994.
40. M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors. *Term Graph Rewriting: Theory and Practice.* John Wiley & Sons, 1992.

41. P. van Beek and R. Dechter. Constraint tightness versus global consistency. In *Proc. KR94*. Morgan Kaufmann, 1994.
42. M. Yokoo, T. Ishida, E.H. Durfee, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proc. IEEE International Conference on Distributed Computing Systems*, 1992.
43. M. Yokoo. Asynchronous weak-committment search for solving distributed constraint satisfaction problems. In *Proc. CP95 (Principles and Practice of Constraint Programming)*. Springer-Verlag, LNCS 976, 1995.

Appendix: Gs-Monoidal Theories

Here the interested reader may find the presentations (taken from [21, 23, 7, 8]) of (one sorted) graph, monoidal and gs-monoidal theories. Gs-monoidal theories are similar to ordinary algebraic (Lawvere) theories [28], the differences being the extended signature and the missing naturality axioms for duplicators and dischargers.

Definition 16. (graphs) A *graph* G is a 4-tuple $\langle O_G, A_G, \delta_0, \delta_1 \rangle$: O_G, A_G are sets whose elements are called respectively *objects* and *arrows* (ranged over by $a, b, \ldots$ and $f, g, \ldots$), and $\delta_0, \delta_1 : A_G \to O_G$ are functions, called respectively *source* and *target*. A graph G is *reflexive* if there exists an *identity* function $id_{\_} : O_G \to A_G$ such that $\delta_0(id_a) = \delta_1(id_a) = a$ for all $a \in O_G$; it is *with pairing* if its class O_G of objects forms a monoid; it is *monoidal* if it is reflexive with pairing and also its class of arrows forms a monoid, such that id, δ_0 and δ_1 respect the neutral element and the monoidal operation. □

Definition 17. (graph theories). Given a (hyper)signature Σ, the associated *graph theory* $G(\Sigma)$ is the monoidal graph with objects the elements of the commutative monoid $(\underline{\mathbb{N}}, \otimes, \underline{0})$ of underlined natural numbers (where $\underline{0}$ is the neutral object and the sum is defined as $\underline{n} \otimes \underline{m} = \underline{n+m}$); and arrows those generated by the following inference rules:

$$(generators)\ \frac{f \in \Sigma_{n,m}}{f : \underline{n} \to \underline{m} \in G(\Sigma)} \qquad (sum)\ \frac{s : \underline{n} \to \underline{m}, t : \underline{n'} \to \underline{m'}}{s \otimes t : \underline{n} \otimes \underline{n'} \to \underline{m} \otimes \underline{m'} \in G(\Sigma)}$$

$$(identities)\ \frac{\underline{n} \in \underline{\mathbb{N}}}{id_{\underline{n}} : \underline{n} \to \underline{n} \in G(\Sigma)}$$

satisfying the *monoidality* axiom $id_{\underline{n} \otimes \underline{m}} = id_{\underline{n}} \otimes id_{\underline{m}}$ for all $\underline{n}, \underline{m} \in \underline{\mathbb{N}}$. □

Definition 18. (monoidal theories).
Given a signature Σ, the associated *monoidal theory* $\mathbf{M}(\Sigma)$ is the monoidal graph with objects the elements of the commutative monoid

$(\underline{\mathbb{N}}, \otimes, \underline{0})$ of underlined natural numbers and arrows those generated by the following inference rules:

$$(generators)\ \frac{f \in \Sigma_{n,m}}{f : \underline{n} \to \underline{m} \in \mathbf{M}(\Sigma)} \qquad (sum)\ \frac{s : \underline{n} \to \underline{m}, t : \underline{n'} \to \underline{m'}}{s \otimes t : \underline{n} \otimes \underline{n'} \to \underline{m} \otimes \underline{m'} \in \mathbf{M}(\Sigma)}$$

$$(identities)\ \frac{\underline{n} \in \underline{\mathbb{N}}}{id_{\underline{n}} : \underline{n} \to \underline{n} \in \mathbf{M}(\Sigma)} \qquad (composition)\ \frac{s : \underline{n} \to \underline{m}, t : \underline{m} \to \underline{k}}{s; t : \underline{n} \to \underline{k} \in \mathbf{M}(\Sigma)}$$

Moreover, the composition operator ; is associative, and the monoid of arrows satisfies the *functoriality* axiom

$$(s \otimes t); (s' \otimes t') = (s; s') \otimes (t; t')$$

(whenever both sides are defined) and the *identity* axiom $id_{\underline{n}}; s = s = s; id_{\underline{m}}$ for all $s : \underline{n} \to \underline{m}$. □

Definition 19. (gs-monoidal theories)
Given a signature Σ, the associated *gs-monoidal theory* $\mathbf{GS}(\Sigma)$ is the monoidal graph with objects the elements of the commutative monoid $(\underline{\mathbb{N}}, \otimes, \underline{0})$ of underlined natural numbers and arrows those generated by the following inference rules:

$$(generators)\ \frac{f \in \Sigma_{n,m}}{f : \underline{n} \to \underline{m} \in \mathbf{GS}(\Sigma)} \qquad (sum)\ \frac{s : \underline{n} \to \underline{m}, t : \underline{n'} \to \underline{m'}}{s \otimes t : \underline{n} \otimes \underline{n'} \to \underline{m} \otimes \underline{m'} \in \mathbf{GS}(\Sigma)}$$

$$(identities)\ \frac{\underline{n} \in \underline{\mathbb{N}}}{id_{\underline{n}} : \underline{n} \to \underline{n} \in \mathbf{GS}(\Sigma)} \qquad (composition)\ \frac{s : \underline{n} \to \underline{m}, t : \underline{m} \to \underline{k}}{s; t : \underline{n} \to \underline{k} \in \mathbf{GS}(\Sigma)}$$

$$(duplicators)\ \frac{\underline{n} \in \underline{\mathbb{N}}}{\nabla_{\underline{n}} : \underline{n} \to \underline{n} \otimes \underline{n} \in \mathbf{GS}(\Sigma)} \qquad (dischargers)\ \frac{\underline{n} \in \underline{\mathbb{N}}}{!_{\underline{n}} : \underline{n} \to \underline{0} \in \mathbf{GS}(\Sigma)}$$

$$(permutations)\ \frac{\underline{n}, \underline{m} \in \underline{\mathbb{N}}}{\rho_{\underline{n},\underline{m}} : \underline{n} \otimes \underline{m} \to \underline{m} \otimes \underline{n} \in \mathbf{GS}(\Sigma)}$$

Moreover, the composition operator ; is associative, and the monoid of arrows satisfies the functoriality axiom

$$(s \otimes t); (s' \otimes t') = (s; s') \otimes (t; t')$$

(whenever both sides are defined); the identity axiom $id_{\underline{n}}; s = s = s; id_{\underline{m}}$ for all $s : \underline{n} \to \underline{m}$; the monoidality axioms

$$\rho_{\underline{n} \otimes \underline{m}, \underline{p}} = (id_{\underline{n}} \otimes \rho_{\underline{m},\underline{p}}); (\rho_{\underline{n},\underline{p}} \otimes id_{\underline{m}})$$

$$!_{\underline{n} \otimes \underline{m}} = !_{\underline{n}} \otimes !_{\underline{m}} \qquad \nabla_{\underline{n} \otimes \underline{m}} = (\nabla_{\underline{n}} \otimes \nabla_{\underline{m}}); (id_{\underline{n}} \otimes \rho_{\underline{n},\underline{m}} \otimes id_{\underline{m}})$$

$$!_{\underline{0}} = \nabla_{\underline{0}} = \rho_{\underline{0},\underline{0}} = id_{\underline{0}} \qquad \rho_{\underline{0},\underline{n}} = \rho_{\underline{n},\underline{0}} = id_{\underline{n}}$$

for all $\underline{n}, \underline{m}, \underline{p} \in \underline{\mathbb{N}}$; the *coherence* axioms

$$\nabla_{\underline{n}}; (id_{\underline{n}} \otimes \nabla_{\underline{n}}) = \nabla_{\underline{n}}; (\nabla_{\underline{n}} \otimes id_{\underline{n}}) \qquad \nabla_{\underline{n}}; \rho_{\underline{n},\underline{n}} = \nabla_{\underline{n}}$$

$$\nabla_{\underline{n}}; (id_{\underline{n}} \otimes !_{\underline{n}}) = id_{\underline{n}} \qquad \rho_{\underline{n},\underline{m}}; \rho_{\underline{m},\underline{n}} = id_{\underline{n}} \otimes id_{\underline{m}}$$

for all $\underline{n}, \underline{m} \in \underline{\mathbb{N}}$; and the *naturality* axiom

$$(s \otimes t); \rho_{\underline{m},\underline{q}} = \rho_{\underline{n},\underline{p}}; (t \otimes s)$$

for all $s : \underline{n} \to \underline{m}, t : \underline{p} \to \underline{q}$. $\square$