Electronic Communications of the EASST
Volume 26 (2010)

Manipulation of Graphs, Algebras and Pictures

Essays Dedicated to Hans-Jörg Kreowski on the Occasion of His 60th Birthday

Assemblies as Graph Processes

Dirk Janssens

18 pages

Guest Editors: Frank Drewes, Annegret Habel, Berthold Hoffmann, Detlef Plump
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Assemblies as Graph Processes

Dirk Janssens

Department of Mathematics and Computer Science
University of Antwerp, Belgium

Abstract: This paper explores the potential of graph rewriting and graph processes as a tool for understanding natural computing, and in particular self-assembly. The basic point of view is that aggregation steps in self-assembly can be adequately described by graph rewriting steps in a graph transformation system: the building blocks of an assembly correspond to occurrences of rewriting rules, and hence assemblies correspond to graph processes. However, meaningful algorithms do not consist only of aggregation steps, but also of global steps in which assemblies are modified or partially destroyed. Thus a number of further operations acting on processes are proposed and it is shown that both kinds of operations (assembly and partial destruction) can be combined to yield meaningful algorithms.

Keywords: natural computing, self-assembly, processes, graph rewriting

1 Introduction

The study of self-assembly has been an interesting and promising part of the fascinating area of natural computing for several years [WLWS98, KR08, Cas06]. The phenomenon is an important aspect of biological systems [ETP+04] and has potential applications in nanotechnology, chemistry and material sciences [WMS91]. The basic idea is that components such as molecules or proteins aggregate to form assemblies that have interesting emerging properties which are not present in the original components. It is obviously important to control this aggregation process, i.e. we want to be able to design the building blocks in such a way that certain a priori known structures emerge as a result of spontaneous aggregation. These structures may in their turn interact in a meaningful way with other components. Components will be called *assemblies* whenever we want to stress that they are built by self-assembly.

The basic step in an assembly process is sketched in Figure 1: two components (left) aggregate to form an assembly (right). It is assumed that this happens because there is a particular relationship between their surface structures: these contain active parts (bold segments) that spontaneously stick together; one may think of atoms or molecules that form bonds between them, like in the case of Watson-Crick complementarity. However it is worth noting that "sticking together" is only a metaphor: often the components are subject to both attracting and repulsive forces, and only if certain complementary structures are present on their surfaces the attracting forces are sufficiently strong to overcome the repulsive ones. Thus if only a part of the active structures is present, then there may be no sticking at all, not even over that part.

The use of graph rewriting [EKMR97, EEPT97] as a tool for studying natural computing and self-assembly has been explored before [HLP08, KGL04]. However there are a lot of possible directions to follow because of the variety of processes that need to be described as well as the

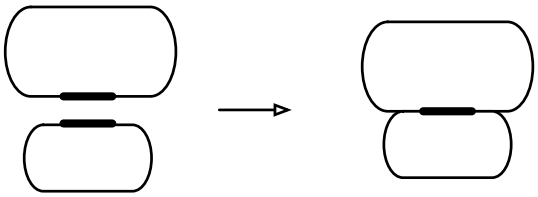

Figure 1: Aggregation

variety of graph rewriting mechanisms. The aim of this paper is to explore, in a preliminary and perhaps somewhat naive way, how the work on graph rewriting with embedding and the corresponding theory of graph processes from, e.g., [VJ02] can be used in this context. It turns out that components, surface structure and assemblies correspond to rules, graphs and graph processes, respectively.

The fact that both the surface structure and the assemblies are described within the same formal framework enables one to switch between the graph/surface structure and the process/assembly view: the former allows one to describe the aggregation steps (combination of various assemblies based on their surface structure) whereas the latter allows one to describe operations that act in a uniform way on the assemblies as a whole (e.g. with the purpose of removing certain parts of them). The first kind of step occurs when a large amount of simple building blocks (molecules) is put into a solution and allowed to form assemblies; the latter kind happens in response to external manipulations such as heating a solution, or selectively extracting assemblies that have a certain property. It is to be expected that the implementation of meaningful algorithms will often require a combination of both kinds of steps: although the self-assembly of cleverly designed building blocks is a powerful tool, its use leads inevitably to ever larger assemblies, and one may expect that at some point a model based on self-assembly alone becomes unrealistic.

Thus in addition to self-assembly we propose three operations acting directly on processes; two of these are very simple (*combine* and *extract*) but the third one (*retain*) allows one to partially destroy assemblies, keeping only a part of each of them. We present two language recognition algorithms in which the various kinds of steps are combined: the first one marks components that encode a certain language, the second one marks those components in a solution X that also occur in another solution Y. Here the "marking" of components consists in attaching a special element to them, making it possible to extract them. In the case of the second algorithm, one may think of Y as a contamination that has to be removed from X; the contamination is not given by specifying its structure explicitly, but by presenting a sample solution.

In Section 2 the basic assumptions underlying this work are given, and the relationship is discussed between components, surface structure and assemblies on the one hand and rules, graphs and graph processes on the other hand. Section 3 provides the basic definitions concerning graph rewriting and processes. In Section 4 the operations dealing with partial destruction and removal of assemblies are defined and illustrated by examples, and the paper ends by a brief discussion section.

2 Graph rewriting and self-assembly

2.1 Basic assumptions

A graph transformation system consists essentially of a set of *rules* that describe local changes applied to graphs. Traditionally, a rule has a *left-hand side* and a *right-hand side*, which are both graphs. A rule is applied to a graph g by matching its left-hand side with a subgraph of g. That subgraph is then removed and replaced by the right-hand side. In the approach used in this paper the rules are equipped with additional information, called "embedding mechanism", which is used to determine the edges of the resulting graph. More details can be found in Subsections 2.2 and 3.1.

The way graph rewriting is related to aggregation is the following. Consider an assembly step, like the one depicted in the upper part of Figure 2: an existing assembly (top) gets larger by aggregating with a building block (bottom). The surface structure of the former is represented by a graph g_1 with nodes a,b,c,d,e. The aggregation leads to a larger assembly with a modified surface structure, represented by a graph g_2 with nodes c,d,e,f,g,h. Thus the effect of adding the new building block on the surface structures is that g_1 is changed into g_2, in a way that can be captured by graph rewriting: the building block is viewed as a graph rewriting rule and the aggregation step corresponds to its application, removing the nodes a,b and replacing them by f,g,h. Evidently one also has to deal with the edges, which represent relationships between surface elements, we come back to this in Subsection 2.2. The approach entails the following three assumptions concerning aggregation.

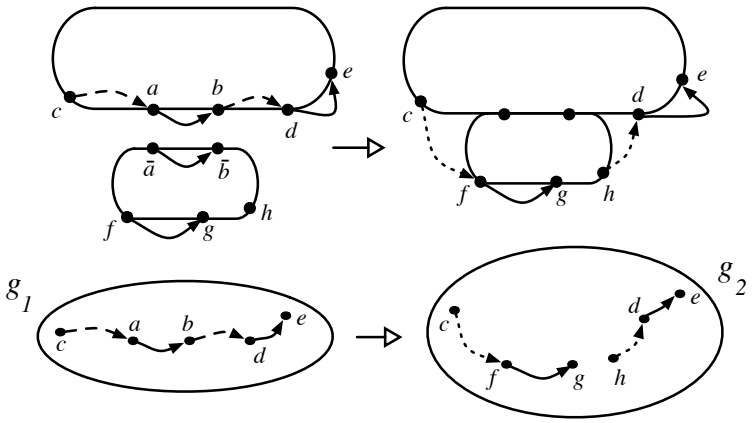

Figure 2: Aggregation and graph transformation

The first working hypothesis is that the surface structure of components, which governs the way in which they aggregate, can be adequately described by a labeled graph. The nodes represent atoms, molecules, ... that are present at specific locations on the surface, the node labels distinguish between various kinds of such surface elements, and the edges describe relationships between these elements that are important for determining whether a group of nodes is active in the sense that it causes aggregation. One may think of spatial relationships or vicinity, but there

may be others. In general, both nodes and edges may be abstractions of physical entities. In this paper we consider graphs that have labels on their nodes but not on their edges. However the approach can be generalized to edge-labeled graphs or even hypergraphs if desired. In Figure 2 the symbols $\overline{a}$ and $\overline{b}$ are used to indicate the fact that binding or aggregation between physical components is caused by elements that are "complementary" in some sense (e.g. having opposite polarities, Watson-Crick complements, ...). In our approach this information is implicitly represented by the fact that building blocks are described by graph rewriting rules which have a designated left-hand side. This is why we will not explicitly need complementary labels such as a and $\overline{a}$ in the next section: it is simply assumed that the label a stands for $\overline{a}$ if it occurs in the left-hand side of a rule.

The second hypothesis is that the relevant relationships between the elements of the new surface (i.e., the new edges) can be determined from (1) the surface of the existing component and (2) the new component. Thus the components, such as the ones depicted in the left part of Figure 1 will not be treated equally: one of them (the upper one in the figures) may be thought of as a large assembly that grows by aggregating with the other one, which is small and simple. The more symmetric case where two arbitrarily large components aggregate is not considered in this paper: there is a "large" assembly that "grows" by adding a new building block. As a result of this, the building blocks of an assembly are partially ordered, making them similar to graph processes. The surface of the assembly after the aggregation step consists of most of the "old" surface combined with a small, new part that belongs to the building block. It is assumed that in determining the new surface structure, one does not need the internal structure of the large assembly. The upper half of Figure 2 depicts an aggregation step where the surface structures are graphs. Technically the letters $a, b, \ldots$ are node labels, but throughout this section we need not to distinguish between nodes and their label. The lower half of Figure 2 depicts the transformation of the surface structure, which is now a graph transformation.

A last assumption is that the effect of an aggregation step is *local*: a surface element that is irrelevant for a location does not suddenly become relevant when an aggregation takes place involving that location: e.g. in Figure 2, e is not relevant to the locations a and b involved in the aggregation – and thus e is not connected to either of them. In terms of graph rewriting, the assumption means that the newly introduced nodes can only be connected to those nodes that were neighbors of the nodes removed by the rewriting. Thus the neighbors of the new nodes f, g, h are either also new or chosen among the "old" neighbors c, d of a and b.

2.2 The embedding mechanism

The lower half of Figure 2 depicts the change in surface structure that corresponds to the aggregation step in the upper half of the figure. This change will be described by the application of a graph rewriting rule to the graph g_1 on the left: the rule removes nodes a, b and creates f, g, h. The edges of the new surface are either edges of the old surface, such as (d, e), or edges of the surface of the newly added building block, such as (f, g), or edges that connect the new nodes with the old ones, such as (c, f) or (h, d). The mechanism for establishing the latter kind of edges is known as an *embedding mechanism*. The embedding mechanism used in this paper is very simple: each of the new nodes may take over the incoming and/or outgoing edges of one or

more of the nodes that have been removed. In Figure 2, f takes over the incoming edges from a and h takes over the outgoing edges of b. The rule applied is depicted in Figure 3: it consists of two graphs (the left-hand side and the right-hand side) and two binary relations *in* and *out* that express the way edges are established. In Figure 3 both relations contain only one pair; in general $in, out \subseteq Lnd \times Rnd$ where *Lnd* and *Rnd* are the sets of nodes of the left-hand side and the right-hand side, respectively.

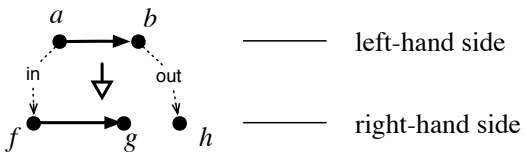

Figure 3: A rule

3 Graph rewriting and processes

The purpose of this section is to provide the basic notions and notation concerning graph rewriting and graph processes. In the first subsection we consider graphs, rules and the way they are applied to rewrite graphs. The graphs are finite, directed, simple, node labeled graphs; the introduction of edge labels should not lead to major difficulties and is left out for simplicity. As pointed out before, the graph rewriting considered uses a very simple embedding mechanism, needed to specify the edges of the graph that results from a rewriting step. In the second subsection graph processes are defined and it is briefly discussed how the process approach is related to the more traditional view of graph rewriting where the emphasis is on sequences of derivation steps.

3.1 Graphs, rules and rewriting

For the remainder of the paper, let *Lab* be a finite set; its elements are called *node labels*.

Definition 1 (graph) A *graph* is a 3-tuple $g = (Nd, Ed, lab)$ where Nd is a finite set, $E \subseteq Nd \times Nd$ and $lab : Nd \to Lab$ is a function. Nd is the *set of nodes* of g, Ed is the *set of edges* of g and lab is the *node labeling function* of g.

For a graph g, its components are denoted by $Nd(g)$, $Ed(g)$ and lab_g. A graph g is a *subgraph* of a graph g' if $Nd(g) \subseteq Nd(g')$, $Ed(g) \subseteq Ed(g')$ and lab_g is the restriction of $lab_{g'}$ to $Nd(g)$, hence the notion of subgraph coincides with pointwise inclusion. For a graph g and a subset X of $Nd(g)$, the *induced subgraph of g on X* is the graph $(X, Ed(g) \cap (X \times X), lab_X)$ where lab_X is the restriction of lab_g to X.

Definition 2 (rule) A *rule* is a 4-tuple $r = (lhs, rhs, in, out)$ where *lhs* and *rhs* are graphs such that $Nd(lhs) \cap Nd(rhs) = \emptyset$ and $in, out \subseteq Nd(lhs) \times Nd(rhs)$. The graphs *lhs* and *rhs* are called

the *left-hand side* and the *right-hand side* of r.

For a rule r, its components are denoted by $lhs(r)$, $rhs(r)$, $in(r)$ and $out(r)$. The notion of a rule will be extended in Subsection 3.2 to allow edges connecting the left-hand side to the right-hand side. Obviously, rules can be used to rewrite graphs: in order to apply a rule r to a graph g, one carries out the following steps.

1. Match the left-hand side of r with a subgraph lhs' of g. This means that one chooses an occurrence (an isomorphic copy) lhs' of $lhs(r)$ that is a subgraph of g. If no such occurrence can be found, then the rule r cannot be applied to g.

2. Remove lhs' from g. This includes the removal of edges between nodes of lhs' and the remaining part of g.

3. Replace lhs' by an isomorphic copy rhs' of $rhs(r)$. The set of nodes of rhs' should be disjoint from the set of nodes of g.

4. Establish edges between nodes of rhs' and the remaining part of g according to the embedding mechanism: for each $(x,y) \in Ed(g)$ and each $x',y' \in (Nd(g) \setminus Nd(lhs')) \cup Nd(rhs')$, add the edge (x',y') to the set of edges of the result graph if both of the following conditions hold: (1) either $x = x'$ or (x,x') corresponds to an element of $out(r)$, and (2) either $y = y'$ or (y,y') corresponds to an element of $in(r)$.

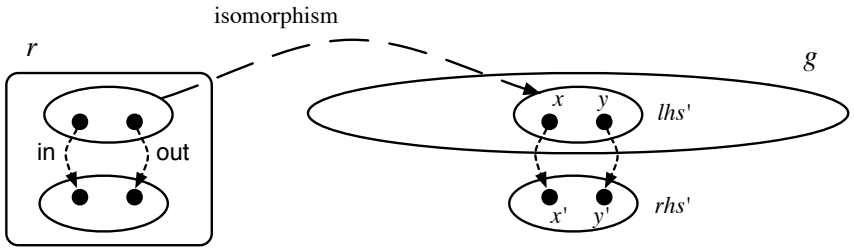

Figure 4: Application of a rule

The situation is illustrated in Figure 4, and it is clear that the graph on the right lower corner of Figure 2 is obtained in this way from the graph in the left lower corner, applying the rule of Figure 3. The embedding mechanism transforms the dashed edges in Figure 2 into the dotted ones. Note also that an edge (x,y) of lhs' such that (x,x') corresponds to an element of $out(r)$ and (y,y') corresponds to an element of $in(r)$ gives rise to an edge (x',y'). In the example algorithms of Section 4, however, this situation does not occur.

In the rather informal description of a rewriting step given above, matching the left-hand side of the rule r with a subgraph lhs' of the graph g amounts to choosing an isomorphism from $lhs(r)$ to lhs'. Such isomorphism is often called *matching morphism*; in many approaches to graph rewriting one allows more general matching morphisms than is the case in this paper. In the formal definition of a derivation step however, matching will be handled in a slightly different

way: it is assumed that an isomorphic copy of the rule r is available such that its left-hand side is *equal*, and not just isomorphic, to lhs'. Whenever a system is specified by a set of rules P, we assume that all isomorphic copies of the rules of P are available for the construction of derivation steps. In this way the only matching morphisms needed are identical mappings, and they can be left implicit. Throughout the paper the term *rule occurrence* is used to emphasize the fact that a certain rule is an isomorphic copy of a rule in a system P. The notion of a derivation step is defined as follows. For a set X, let Id_X denote the identity relation on X.

Definition 3 (derivation step) A *derivation step* is a 3-tuple (g,r,g') where g,g' are graphs, r is a rule and the following holds.

1. $lhs(r)$ is a subgraph of g and $Nd(rhs(r)) \cap Nd(g) = \emptyset$.
2. $Nd(g') = (Nd(g) \setminus Nd(lhs(r))) \cup Nd(rhs(r))$.
3. Let $\overline{in} = in(r) \cup Id_{Nd(g')}$ and $\overline{out} = out(r) \cup Id_{Nd(g')}$. Then

$$Ed(g') = \{(x',y') \in Nd(g') \times Nd(g') \mid \text{there exists } (x,y) \in Ed(g) \cup Ed(rhs(r)) \text{ such that } (x,x') \in \overline{out} \text{ and } (y,y') \in \overline{in}\}.$$

4. For each $x \in Nd(g')$,

$$lab_{g'}(x) = \begin{cases} lab_g(x) & \text{if } x \in Nd(g) \\ lab_{rhs(r)}(x) & \text{if } x \in Nd(rhs(r)). \end{cases}$$

3.2 Processes

In [CMR96, VJ02] graph processes are proposed as a way to describe "runs" of a graph rewriting system. Informally, a graph process is a structure obtained by gluing together the rule occurrences of a run, where the gluing is consistent with the way the rules are applied. For each rule occurrence, its left-hand side is glued over the set of nodes that is removed by that occurrence. Thus a graph process is essentially a directed acyclic graph where the nodes are those that occur in the run and where the edges represent the direct causal dependency relation: whenever a rule occurrence removes a node a and introduces a new node b, then b is directly causally dependent on a. This DAG is further decorated with extra information: the initial graph of the run is given as well as the rule occurrences. However there is no information on the order in which the rules are applied other than the causality relation. Figure 5 depicts a process: the initial graph is the linear structure at the top and there are three occurrences of the rule depicted at the left. There is no information about the relative order of rule occurrences 1 and 2, and so the process describes in fact three possible runs: the three rule occurrences may happen either in the order 1,2,3, or 2,1,3, or 1 and 2 may happen simultaneously, followed by 3. The dotted edges are established according to the embedding mechanism.

When reasoning about processes it is often useful to consider a rule as a graph, rather than a pair of graphs equipped with embedding information. In this way a rule can be viewed as a very simple process, describing a run consisting of just one rule application. However this also enables one to extend the notion of a rule by allowing edges connecting its left-hand side and its

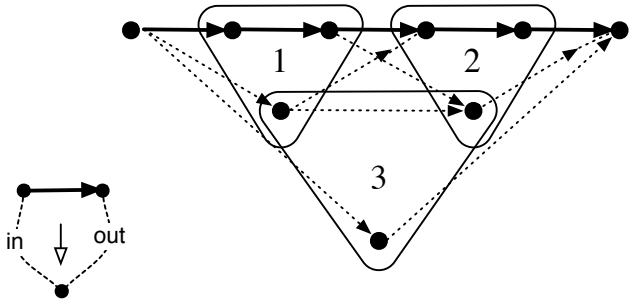

Figure 5: A process

right-hand side. While such edges have no obvious interpretation in the traditional view of graph rewriting, they fit well into the process view developed here and, more importantly, they turn out to be quite useful in the operations and algorithms of Section 4.

Definition 4 (rule,extended) A *rule* is a 5-tuple $r = (gr, Lnd, Rnd, in, out)$ where gr is a graph, (Lnd, Rnd) is a partition of $Nd(gr)$ and $in, out \subseteq Lnd \times Rnd$. The induced subgraphs of gr on Lnd and Rnd are called the *left-hand side* an the *right-hand side* of r, respectively. The set $Nd(gr)$ is the *set of nodes* of r. The *set of edges created by* r is the set $Ed(gr) \setminus (Lnd(r) \times Lnd(r))$.

For a rule r, its components are denoted by $gr(r)$, $Lnd(r)$, $Rnd(r)$, $in(r)$ and $out(r)$ respectively. Its set of nodes, its set of edges created, its left-hand side and its right-hand side are denoted by $Nd(r)$, $Ed(r)$, $lhs(r)$ and $rhs(r)$. Slightly abusing notation, lab_r denotes $lab_{gr(r)}$. Note that an extended rule r such that $Ed(gr(r)) \subseteq (Lnd(r) \times Lnd(r)) \cup (Rnd(r) \times Rnd(r))$ can be described as a rule according to Definition 2, replacing it by $(lhs(r), rhs(r), in(r), out(r))$.

Formally, the notion of a process is defined as follows.

Definition 5 (process) A *process* is a pair $p = (Init, Occ)$ where $Init$ is a graph and Occ is a set of rule occurrences such that the following holds.

1. Let $<^1 = \bigcup_{oc \in Occ}(Lnd(oc) \times Rnd(oc))$ and let $Nd = Nd(Init) \cup \bigcup_{oc \in Occ} Nd(oc)$. Then $(Nd, <^1)$ is a directed acyclic graph. The relation $<^1$ is called the *direct causality relation* of p. Its transitive and reflexive closure is called the *causality relation* of p.

2. $Nd(Init)$ is the set of minimal nodes of $(Nd, <^1)$.

3. There exists a function $lab : Nd \to Lab$ such that, for each $g \in \{Init\} \cup Occ$, lab_g is a restriction of lab. The function lab is called the *labeling function* of p.

4. For each $oc, oc' \in Occ$ such that $oc \neq oc'$, the sets $Lnd(oc)$ and $Lnd(oc')$ are disjoint, and the sets $Rnd(oc)$ and $Rnd(oc')$ are disjoint.

For a process p, its components are denoted by $Init(p)$ and $Occ(p)$, respectively. Its set of nodes, direct causality relation, causality relation and labeling function are denoted by $Nd(p)$,

$<_p^1$, $\leq_p$ and lab_p. The relations $in(p)$ and $out(p)$ are defined by

$$in(p) = Id_{Nd(p)} \cup \bigcup_{oc \in Occ} in(oc) \quad \text{and} \quad out(p) = Id_{Nd(p)} \cup \bigcup_{oc \in Occ} out(oc).$$

The set of *edges created in p*, denoted by $Ed(p)$ is defined by

$$Ed(Init(p)) \cup \bigcup_{oc \in Occ} Ed(oc).$$

Note that a graph g can be identified with a process$(g, \emptyset)$, and that a process imposes a partial order on its rule occurrences: oc directly precedes oc' if $Rnd(oc) \cap Lnd(oc') \neq \emptyset$.

Given a set of rules P, one has to associate with P a set of processes that is "valid" in the sense that they describe a possible run or rewriting process of P. Informally, there are two obvious conditions to be satisfied by such a process:

1. the process should be built from an initial graph and occurrences of rules from P, and
2. for each rule occurrence oc of the process, its left-hand side should be a subgraph of the graph obtained by applying the occurrences that precede oc, starting from the initial graph.

In order to formalize the second condition one needs to associate a set of graphs with a process: the graphs that occur as intermediate configurations in the course of the rewriting process. it is well known in the theory of processes that these configurations correspond to *slices*, i.e. maximal sets of causally unrelated nodes. Thus what remains to be done is to equip each slice with a suitable set of edges (the node labels are determined by the node labeling functions of the initial graph and the rule occurrences). This leads to the following definitions.

Definition 6 (slice) Let p be a process. A *slice* of p is a maximal subset S of $Nd(p)$ such that, for each $x, y \in S$, $x \leq_p y$ implies that $x = y$.

Definition 7 (configuration) Let p be a process and let S be a subset of $Nd(p)$. The *configuration of p on S*, denoted $Conf(p,S)$, is the graph (S, E, lab_S) where

$$E = \{(x', y') \in S \times S \mid \text{ there exists } (x,y) \in Ed(p) \text{ such that } (x,x') \in out(p) \text{ and } (y,y') \in in(p)\}$$

and lab_S is the restriction of $lab(p)$ to S.

Intuitively, this means that the graph $Conf(p,S)$ contains all the edges that can be derived by the embedding mechanism from the edges of the initial graph and the edges created in the rule occurrences. The graph $Conf(p,S)$ is defined for arbitrary subsets S of $Nd(g)$, not only for slices. Since the nodes of $Init(p)$ are minimal w.r.t. the causality relation, one easily verifies that $Conf(Nd(Init(p)), p) = Init(p)$. The second condition mentioned above (a rule can only be applied if there is a match for its left-hand side) is captured by the notion of validity, defined as follows.

Definition 8 (validity) Let p be a process.

1. p is *valid* if, for each $oc \in Occ(p)$ and each slice S of p such that $Lnd(oc) \subseteq S$, the graph $lhs(oc)$ is a subgraph of $Conf(p,S)$.

2. Let P be a set of rules. Then p is *valid for* P if p is valid and each $oc \in Occ(p)$ is a rule occurrence of P.

If p is a valid process and S is a slice, then $Conf(p,S)$ is an intermediate configuration of the rewriting process represented by p; i.e. it is the graph obtained by applying the rule occurrences that precede S, in the way specified by p, to the graph $Init(p)$. In the case where one uses only restricted rules, i.e. rules according to Definition 2, then this view is consistent with the way derivation steps are defined in Definition 3. This follows from the fact that $Conf(Nd(Init(p)),p) = Init(p)$ and the following lemma.

Lemma 1 *Let p be a valid process, let S,S' be slices of p, let $oc \in Occ(p)$ be such that $Ed(gr(oc)) \subseteq (Lnd(oc) \times Lnd(oc)) \cup (Rnd(oc) \times Rnd(oc))$, $Lnd(oc) \subseteq S$ and $S' = (S \setminus Lnd(oc)) \cup Rnd(oc)$. Let r_{oc} be the rule (according to Definition 2) $(lhs(oc), rhs(oc), in(oc), out(oc))$. Then $(Conf(p,S), r_{oc}, Conf(p,S'))$ is a derivation step.*

Thus the process view extends the traditional way of looking at graph rewriting: if p is a valid process such that all rule occurrences can be viewed as rules according to Definition 2, S is a slice of p and oc is a rule occurrence such that the nodes of $lhs(oc)$ belong to S, then oc transforms the graph $Conf(p,S)$ in the way described by Definition 3. It is also worth noting that this implies that the graph rewriting considered in this paper is *confluent*: in general the causality relation does not impose a total order on the rule occurrences of a process, and applying them in any order consistent with the causality leads to the same result $Conf(p,Max)$, where Max is the set of maximal nodes of the causality relation.

Using the notions introduced above one has three ways to view the components that act as building blocks in aggregation steps such as the one considered in Figure 2: a component with a surface structure, a graph rewriting rule, and a process. Since such components are not composed of smaller ones they are called "atomic". Similarly, the processes that represent a single rule are called atomic processes. Figure 6 depicts the three views; the arrows/lines labeled *in* and *out* represent the embedding mechanism.

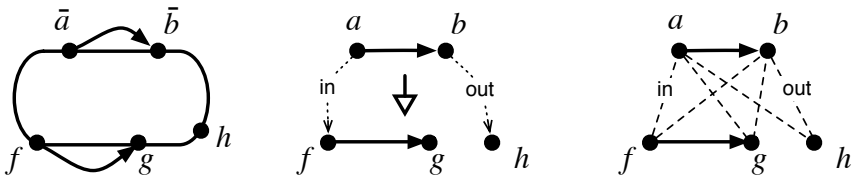

Figure 6: Atomic component, rule and atomic process

The relationship between processes and assemblies is illustrated in Figure 7 (in the process, on the right, only the direct causality relation is represented). The confluency property is illustrated by Figure 7: one may e.g. consider the situation of the assembly after building blocks 1 and 2

are added. The corresponding slice consists of the square nodes. The confluency property then implies that the surface structure at this point of the aggregation process does not depend on the order in which blocks 1 and 2 were added; a posteriori inspection of an assembly (which blocks are present and how are they glued together) suffices to determine its surface structure. Since the order in which the aggregation takes place would probably be very hard to control, this property is of crucial importance.

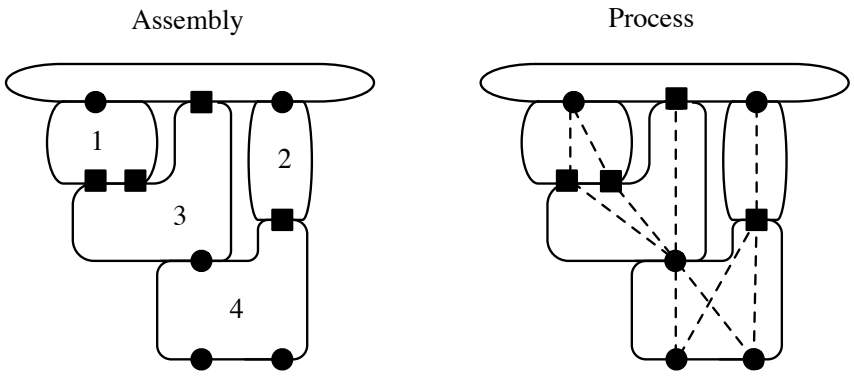

Figure 7: Assembly and process

4 Operations on assemblies

We wish to study algorithms that manipulate solutions containing potentially large amounts of assemblies: the input is a set of solutions (test tubes) and the output is another set of solutions, one or more of which should have a certain, desired property. In order to realize this we need a set of operations to be used as instructions or steps in the algorithms. Self-assembly provides a first powerful operation of this kind: one may add a number of cleverly designed atomic components to the solution, let them aggregate, and obtain in this way a modified solution. The design of aggregation steps is thus viewed as the design of a suitable set of graph transformation rules. It is implicitly assumed that for each of those rules a component can be constructed which has the right surface structure, and that this component interacts in the right way with the other components. It has to be noted that the latter assumption is not obvious, but there is evidence that unintended interactions can be made improbable by a clever design of the components; the problem is somewhat similar to the DNA code word problem, which is an interesting research topic on its own.

It is to be expected that self-assembly alone is insufficient to describe most meaningful algorithms or processes taking place in living organisms, because its repeated use leads to ever larger assemblies. In many forms of natural computing operations occur that partially break down or disassemble larger components, e.g. cutting strands of DNA in a controlled way. Thus there is a need for modeling operations that have a similar effect on assemblies. In this way one also introduces the possibility to reactivate elements that have become inactive: in the building of

assemblies, we assumed so far that a surface element that is glued to a new building block ceases to be on the surface of the resulting assembly and therefore becomes unavailable for further interactions. However, partially breaking down an assembly may bring such elements back to the surface and hence reactivate them. Since we describe assemblies by (valid) processes, we need one or more operations that acts on processes. These have to be implemented by manipulations such as heating, applying an electromagnetic field, irradiation, etc., where all components in a solution are modified in a uniform way. Such "global steps" are not necessarily local changes based on the surface structure of components, and thus the rewriting of graphs describing their surface structure is not a natural way to formalize them. However the more complete description of an assembly by a graph process provides information that is sufficient to express the global steps: the atomic components it is built from and their surface elements.

4.1 Operations on processes

Apart from self-assembly, we propose three operations. Two of them, *combine* and *extract*, are very simple, but the third one, *retain*, requires some more explanation. It enables one to partially break down assemblies.

1. *combine*. This operation allows to combine a number of graphs as one: the resulting graph simply consists of disjoint copies of the original graph. The combined graph can then serve as the initial graph of a self-assembly process. Thus *combine* does not correspond to a physical manipulation of test tubes, it is merely a technical convenience.

2. $extract(m)$, where m is a node label. The symbol m represents a marker, i.e. a part of a component that can easily be detected by its physical properties. The operation removes all components in which the marker occurs from a test tube.

3. $retain(R)$, where R is a subset of the set of node labels. The operation removes, from each component in a test tube, the part that does not correspond to R. The remaining part is again considered as a component; its surface structure is derived from the internal structure of the original component. Such effect can be achieved by designing these components in such a way that the elements in R are they are more stable or more resistant to heat or radiation than the other elements. Since the surface of the remaining part of an assembly may contain elements that have been added at various stages of the assembly process, it may contain elements that are causally related in the process description of the assembly. In this first approach it is therefore assumed that the causality relation of the resulting assembly is trivial: thus all its elements are maximal, and belong to the surface. Formally, the effect of $retain(R)$ on a set of processes is that each process p is replaced by the graph $Conf(p, Ret)$ where Ret is the set $\{x \in Nd(p) \mid lab_p(x) \in R\}$.

Evidently, the operation $retain(R)$ is rather limited in that it completely destroys the internal structure of an assembly. It is easy to conceive more sophisticated variants of the operation, e.g. a version where the substructure of the process p induced by the nodes with labels in R is preserved. The structures obtained in that way would not be valid according to Definition 8, since they are not obtained by composing elements of a designated set of basic components, but nevertheless one may decide to introduce them if desired.

Example 1 *As a first example of the use of retain, consider the situation of Figure 8. One wishes to let the string-like component grow by adding a building block. The assembly step would lead to a disconnected graph, but the desired result (bottom) appears if that step is followed by the operation retain*$(\{a,b,c\})$*. Note that the rule corresponding to the building block contains an edge connecting its left-hand side to its right-hand side.*

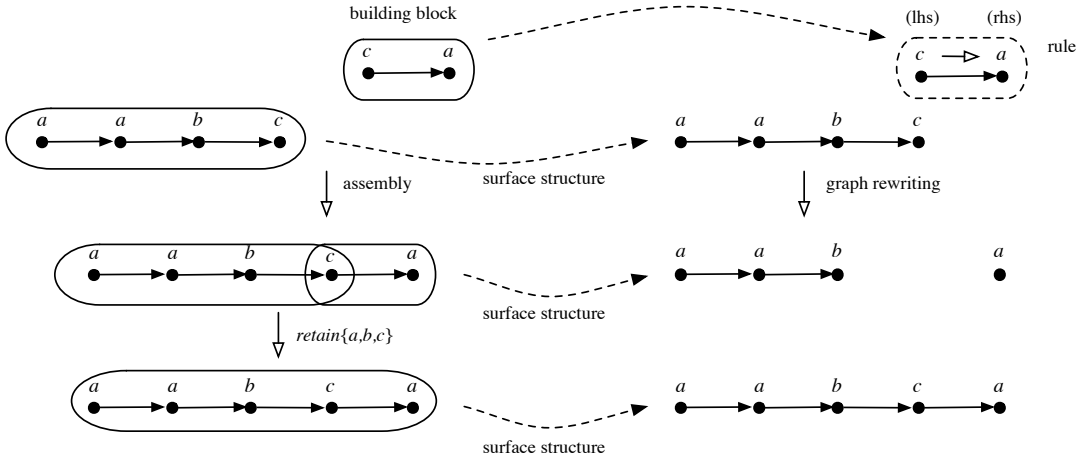

Figure 8: Growing a string

4.2 Example algorithms

The aim of this section is to sketch how aggregation steps (building assemblies) and global steps (acting in a uniform way on all components) may be combined into a meaningful algorithm. An algorithm describes a sequence of steps in which test tubes containing a solution are manipulated in order to obtain a solution with certain desired properties.

We present two language recognition algorithms: in both cases the input is a test tube containing string-like components such as the one depicted in Figure 9, which is used as the encoding of the word $x_1x_2\ldots x_n$. In both cases the output is a test tube from which the components encoding words of a certain language L are removed. This is achieved by using self-assembly to attach a "marker" to them, which allows their extraction. In both cases the *retain* operation can be used to return to a situation where the process of self-assembly and extraction can be iterated. In the first algorithm the language L is given by a context-free grammar, and the assembly process essentially builds a parse tree. In the second algorithm the language L is given in the form of a test tube that contains its encoding by components, so the structure of the words in L is not known explicitly. In this case one may think of L as a contamination that is to be removed from the input solution.

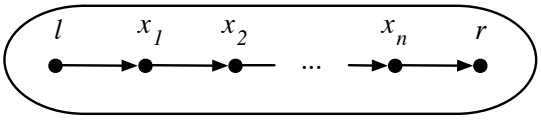

Figure 9: Component encoding a string

4.2.1 Recognizing the language of a context-free grammar

The method demonstrated works for any context-free grammar, but for the example consider the following one.

$$S \to AB \quad A \to aab \quad B \to ab$$

The set of node labels is $\{l,r,a,b,S,A,B,m\}$. The algorithm consists of repeating the following steps for as long as step 3 yields a significant amount of extracted components:

1. Mark the components encoding words in L, by adding the encodings of the rules of Figure 10 to the test tube and letting them aggregate. Note that the rightmost rule contains an edge from the only node of its right-hand side to a node of its left-hand side.

2. Execute $retain(\{l,r,a,b,m\})$ to partially destroy the assemblies formed, so that components which have not been correctly parsed return to their original form.

3. Execute $extract(m)$ to remove the correctly parsed components from the test tube.

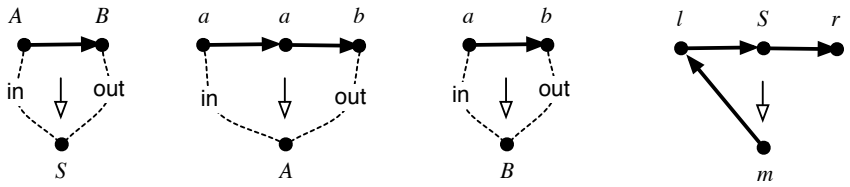

Figure 10: Rules for first algorithm

Figure 11 depicts a process that represents the correct parsing of a component encoding a word from L. The dotted edges are the edges added by the embedding mechanism, these are the edges of the graphs $Conf(p,S)$. An application of the rightmost rule of Figure 10 leads to the process sketched in Figure 12, where the shaded triangle represents the parse tree. A subsequent execution of $retain(\{l,r,a,b,m\})$ leads to the graph of Figure 13. The edge from m to l in the rightmost rule of Figure 10 is not really necessary; it was only added to avoid an isolated node in Figure 13. In general Step 1 yields also processes corresponding to failed attempts at parsing, e.g. reducing ab in the word aab to B. One can only expect to remove all components encoding a word of L if the procedure can be iterated, and thus one needs to break down the unmarked assemblies so that step 1 can be repeated; this happens in step 2.

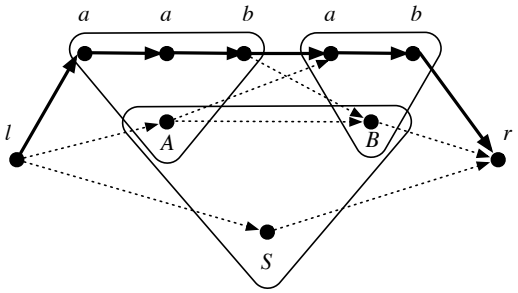

Figure 11: Parsing

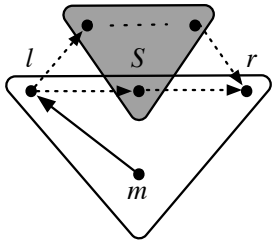

Figure 12: Attaching the marker

4.2.2 Recognizing the components of a test tube

In the second algorithm we start from two test tubes T and T_L: the first one contains a solution that has to be modified, whereas the second one only serves as an additional input: it contains the encodings of the language L. The aim is to remove from the first test tube the components that encode words of L.

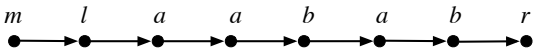

Figure 13: Marked component

Let the set of node labels be $\{l, r, a, b, q, c, m\}$. The algorithm consists of the following steps:

1. Mark the components encoding words in L, by first adding the rule depicted at the left of Figure 14 to test tube T_L and letting it aggregate, and then executing $retain(\{l, r, a, b, c\})$ to obtain components such as the one on the right of Figure 14: hence a node with label c is attached.

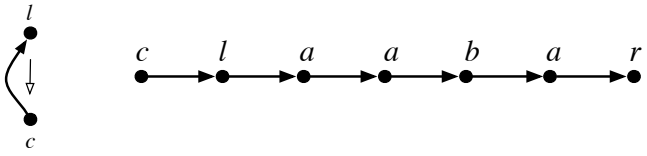

Figure 14: Step 1

2. Repeat the following steps for as long as step (c) effectively yields a significant change:
   (a) Add part of the contents of T_L to T, and allow pairs of components to be considered as one; formally, apply a *combine* operation.
   (b) Add the rules depicted in Figure 15 and let them assemble. The aggregation process traverses both words from left to right, as illustrated in Figure 16. The rightmost rule, which attaches the marker m, can be applied only to an assembly that has been built on a pair of components that both encode the same word.

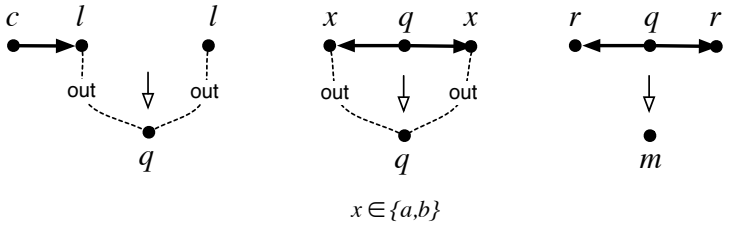

Figure 15: Step 1

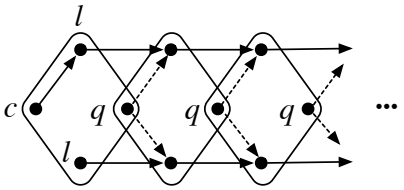

Figure 16: Step 1

(c) Execute $extract(m)$ to remove the marked components from the test tube.

(d) Execute $retain(\{l,r,a,b,c,m\})$. This has the effect of partially destroying the assemblies formed, so that components representing unsuccessful comparisons (starting from pairs of components that encode different words) return to their original form: a string starting with c ad a string starting with l. As a result, step 2 can be repeated.

5 Discussion

The aim of the paper is to explore the use of graph rewriting based on embedding for the understanding of self-assembly and natural computing. The basic idea is that graphs capture the active surface structure that controls the way components in a solution aggregate, and that the way in which such aggregation changes the surface structure can be captured by graph rewriting. However one may expect that most meaningful algorithms in this context do not only require the building of ever larger assemblies, but also operations that break down or modify such assemblies. Thus what seems to be needed is an interplay between aggregation operations, which are described by graph transformation rules, and which act on the graphs that describe the active surface of components, and global operations in which all assemblies of a given kind in a solution are modified. Since assemblies correspond to processes of the graph rewriting systems that describe their formation, the theory of graph rewriting and graph processes may provide a way to obtain a formal framework in which both kinds of operations can be combined in an elegant way.

Obviously, the material presented here is of a very speculative nature, since the implicit assumptions concerning the possible realization of the approach in the physical world may turn out to be naive or unrealistic. To mention just a few: when reducing the problem of controlling self-assemby to the problem of writing a suitable graph transformation system, it is assumed that each rule written down can be realized by a component that behaves exactly in the right way: not only does it aggregate with another component when the structure corresponding to its left-hand side matches part of the structure of the other component, but this is also the *only* way it interacts with other components. Another thorny issue is the assumption about the information to be encoded into the edges, information that is handled by the embedding mechanism: what are exactly the relationships between locations on a component that are relevant? How to encode spatial information into those edges? Also for the the global operations many questions remain: on the one hand they may seem rather ad-hoc, but on the other hand they are quite simple and one can easily define more sophisticated variants. It seems also rather straightforward to generalize the approach to edge labeled graphs or hypergraphs, providing a more expressive language to describe the surface structures. Finally, in this paper only assembly steps are considered where a component combines with a basic component. However it may be more realistic to also include the possibility that large assemblies, perhaps each already composed of a number of basic components, combine. It is probably not too hard to adapt the approach to this: on the one hand, one may introduce operations that enable one to treat an assembly as a rule; hiding or encapsulating part of its internal structure, and on the other hand one may define suitable composition operations on processes, gluing them together over parts representing the surface of the components they represent. Similar operations have been investigated for processes in the context of

concurrency theory, and it seems likely that a number of ideas studied there can be useful.

It seems fair to conclude that, in spite of a number of reservations, the correspondence between graph rewriting and graph processes on the one hand and aggregation and assemblies on the other hand is simple and natural enough to deserve further attention.

Bibliography

[Cas06] L. D. Castro. *Fundamentals of natural computing: basic concepts, algorithms, and applications*. Chapman and Hall/CRC Press, 2006.

[CMR96] A. Corradini, U. Montanari, F. Rossi. Graph processes. *Fundam. Inf.* 26(3-4):241–265, 1996.

[EEPT97] H. Ehrig, K. Ehrig, U. Prange, G. Taenzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer Verlag, 1997.

[EKMR97] H. Ehrig, H.-J. Kreowski, U. Montanari, G. Rozenberg. *Handbook of Graph grammars and Computing by Graph Transformation*. World Scientific, 1997.

[ETP+04] A. Ehrenfeucht, T.Harju, I. Petre, D. Prescott, G. Rozenberg. *Computation in Living Cells - Gene Assembly in Ciliates*. Natural Computing Series. Springer Verlag, 2004.

[HLP08] T. Harju, C. Li, I. Petre. Graph theoretic approach to parallel gene assembly. *Discrete Applied Mathematics* 156(18):3416–3429, 2008.

[KGL04] E. Klavins, R. Ghrist, D. Lipsky. Graph Grammars for Self Assembling Robotic Systems. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation*. Pp. 5293–5300. 2004.

[KR08] L. Kari, G. Rozenberg. The many facets of natural computing. *Commun. ACM* 51(10):72–83, 2008.

[VJ02] N. Verlinden, D. Janssens. Algebraic properties of processes for Local Action Systems. *Mathematical. Structures in Comp. Sci.* 12(4):423–448, 2002.

[WLWS98] E. Winfree, F. Liu, L. Wenzler, N. Seeman. Design and self-assembly of two-dimensional dna crystals. *Nature* 394:539–544, 1998.

[WMS91] G. Whitesides, J. Mathias, C. Seto. Molecular self-assembly and nanochemistry - a chemical strategy for the synthesis of nanostructures. *Science* 254:1312–1319, 1991.