Simulation of self-assembly processes using abstract reduction systems

Jean-Louis Giavitto & Antoine Spicher

LaMI UMR 8042 CNRS – Université d'Évry, Génopole,
523 place des Terrasses de l'Agora, 91000 Évry, France

Abstract

We present in this chapter the use of MGS, a declarative and rule-based language dedicated to the modeling and the simulation of various morphogenetic and developmental processes, like self-assembly processes. The MGS approach relies on the introduction of a topological point of view on various data structures called *topological collections*. This topological approach enables a uniform handling of theses data structure by a new kind of rewriting rules called *transformations*. Using (local) rewriting rules to specify self-assembling processes is particularly adequate because it mimics closely the incremental building mechanism of the real phenomena. The MGS approach is illustrated on the fabrication of a fractal pattern, a Sierpinsky triangle, using two approaches: by *accretive growth* and by *carving*. More generally, the notions of topological collections and transformations available in MGS enable the easy and concise modeling of cellular automata on various lattice geometries as well as more arbitrary constructions of multi-dimensional objects.

1 Introduction

Self-assembly is a process that creates incrementally complex hierarchical spatial structures. Nature presents a lots of examples, ranging from crystallization in physics to morphogenesis in developmental biology. There is no unified general theory of self-assembling, nor a unique definition. However, understanding the principles underlying self-assembly processing will open entire new opportunities for our technological capabilities. In this chapter, self-assembled systems can be thought to be built of basic building elements (molecules, cells, etc.); together these basic elements exhibit a new, often highly, complex behavior.

Email address: [giavitto,aspicher]@lami.univ-evry.fr.
URL: http://mgs.lami.univ-evry.fr.

For a computer scientist, self-assembly processes are particularly inspiring because the dynamic organization of the involved entities emerge from many decentralized and local interactions that occur concurrently at several time and space scales. As a matter of fact, they have inspired several new computational models like *amorphous computing* (1) or *autonomic computing* (9).

The emergence of the global structure of self-assembled systems cannot be deduced from the individual composing elements. Simulation models are often, if not the only available way, to obtain a deeper insight of these complex systems. However, the modeling and the simulation of self-assembly can be very difficult to achieve, because of the representation of the underlying space and of the handling of complex spatial structures build in this space.

1.1 Self-Assembly by Accretive Growth and by Carving

A central thema in the research in self-assembly processes is the organizational principles that can be used to structure a population of basic elements. The structure is incrementally built and often corresponds to a spatial structure. In this paper we will focus on the modeling of two kinds of self-assembly that rely on the addition or the removal of materials. That is, we do not consider rearrangement of matter, nor its modification (e.g. by stretching). Our approach has a *combinatorial* nature and hide the burden of the physical implementation.

Self-Assembly by Accretive Growth. One of the most fundamental kind of self-assembly is certainly processes where basic elements are aggregated into a shape during a *growth process*. An incremental growth process can be described as the iteration of a basic aggregation : in each growth stage, new basic entities (e.g., material) are added to the preceding growth stage (10). The aggregation depends on the available material and also on the the form of the object in the preceding growth stage. We use the term *accretive growth* to qualify a growing process that takes place on the boundaries of the system. This kind of growth is to oppose to "intercalary growth" where the growing process is from the inside of the assembly.

Self-Assembly by Carving. Manca et al. have introduced a somewhat unusual type of computation strategy called *computation by carving* (12). The idea is to generate a (large) set of candidate solutions of a problem, then remove the non-solutions such that what remains is the set of solutions. This idea to remove unwanted elements is also present in building shapes by space carving (11), an algorithm to compute a volume that is consistent with a set

of photos of a 3D shape. Transposed in the domain of self-assembly, this leads to the idea to iteratively remove elements, starting from an initial shape. A better name is perhaps *self-disassembling*.

1.2 Domain Specific Languages for the Simulation of Self-Assembly

As noted above, the simulation of self-assembly can be very difficult to achieve. In this paper, we advocate the use of a domain specific language (DSL) for the modeling and the simulation, in an abstract and uniform setting, of accretive growth and carving.

DSLs are specially tailored programming languages designed for solving problems in a particular domain. To this end, a DSL provides abstractions and notations for the domain at hand. DSLs are usually small, and more declarative than imperative. Moreover, DSLs are more attractive for programming in the dedicated domain than general-purpose languages because of easier programming, systematic reuse, better productivity and flexibility. Our approach relies on two dedicated notions:

- dedicated data-structures, called *topological collections* are used to represent the space underlying a self-assembly process and/or the self-assembled system; and
- rewriting rules on topological collection, called *transformations*, are used to implement the local evolution rules usually used to specify the self-assembly process.

These two notions are studied in an experimental programming language called MGS. MGS is a vehicle used to investigate the notions of topological collections and transformations and to study their adequacy to the simulation of various biological and self-assembly processes (6; 4). Using (local) rewriting rules to specify the self-assembling process is particularly adequate because it mimics closely the incremental building mechanism of the real phenomenon. In addition, the declarative style enabled by rule-based programming corresponds to a mathematical specification of the self-assembly process, which opens the way to mathematical reasoning and proofs.

1.3 Organization of the Paper

The rest of this paper is organized as follows. The next section provides a quick introduction to MGS. Group-based data fields are sketched: they are topological collections used to define various lattices used in the modeling of accretive growth. Section 3 presents three short and well-known examples of

growth by aggregation processes in MGS. Section 4 details the self-assembly of Sierpinsky triangles. Section 5 introduces the notion of abstract cellular complexes and their handling in MGS. Cellular complexes are used to model arbitrary shape for growth and carving. The use of abstract cellular complexes is illustrated to sketch the tridimensional assembly of proteins. The process sketched is rather abstract and does not refer to a physical example. However it gives a good flavor of the possibilities offered by abstract cellular complexes and the very concise style enabled by the MGS transformations. In section 7 we build again Sierpinsky triangles but in 3 dimensions and using a carving process. The conclusion reviews some previous, related and future work.

2 A Short MGS Presentation

In this section, we present the notions needed to understand the MGS coding of the following computation processes. MGS is a declarative programming language aimed at the representation and manipulation of local transformations of entities structured by *abstract topologies* (4). A set of entities organized by an abstract topology is called a *topological collection*. Topological means here that each collection type defines a neighborhood relation specifying both the notion of *locality*, *path* and *sub-collection*. A path is a finite sequence of elements e_i where e_{i+1} is a neighbor of e_i. A sub-collection B of a collection A is a subset of elements of A defined by some path and inheriting its organization from A. The *global transformation* of a topological collection C consists in the parallel application of a set of *local transformations*. A local transformation is specified by a rewriting rule r that specifies the change of a sub-collection. The application of a rewrite rule $\beta \Rightarrow f(\beta, ...)$ to a collection A:

(1) selects a sub-collection B of A whose elements match *pattern* β,
(2) computes a new collection C as a function f of B and its neighbors,
(3) and specifies the insertion of C in place of B into A.

The collection types can range in MGS from totally unstructured with sets and multisets to more structured with sequences, "group-based data fields" and "abstract cellular complexes".

There are two kinds of patterns that can be used in a transformation for selecting a sub-collection: path patterns and patch patterns.

Path Patterns. Path patterns match paths in a collection. A path pattern is a sequence of elements separated by a comma. Path pattern `x, y` defines a path of two elements, where `y` must be a neighbor of `x`. Arbitrary condition can be tested using guards inserted in a path pattern: `(x / x>0), (y / y>x)`

matches two elements `x` and `y` such that the value of `x` is strictly positive and `y` is a neighbor of `x` and the value of `y` must be greater than the value of `x`.

Patch Patterns. Patch patterns allow the matching of arbitrary sub-collection. A patch pattern is specified using a set of clauses. We will present the patch pattern features we need on section 7.

2.1 Group-Based Data Field

Group-based data fields (GBF in short) are used to define topological collections with *uniform* neighborhood. A GBF is an extension of the notion of array, where the values are indexed by the elements of a group, called the *shape* of the GBF (5). The elements of the group are called the *positions* of the GBF. For example:

```
gbf Grid2 = < north, east >
```

defines a GBF collection type called `Grid2`, corresponding to the Von Neuman neighborhood in a classical array (a cell above, below, left or right – not diagonal). The two names `north` and `east` refer to the directions that can be followed to reach the neighbors of an element. These directions are the *generators* of the underlying group structure. The right hand side (r.h.s.) of the GBF definition gives a finite presentation of the group structure.

The list of the generators can be completed by giving equations that constraint the displacements in the shape:

```
gbf Hex2 = < east, north, northeast; east + north = northeast >
```

defines an hexagonal lattice that tiles the plane, see figure 2. Each cell has six neighbors (following the three generators and their inverses). We use an additive notation for the group operation. The equation `east + north = northeast` specifies that a move following `northeast` is the same as a move following the `east` direction followed by a move following the `north` direction. In this chapter we will use only commutative groups, that is, commutation equations between generators (like `east+north = north+east` always hold implicitly.

For convenience, we identify the type of a GBF with the presentation of the underlying group. A GBF g of type G can be formalized as a partial function g from the group specified by G to some set of values: g associates a value to some positions. In other words, the group elements act as indices of a generalized array. An empty GBF is the everywhere undefined function. A special constant `<undef>` can be used in path patterns to match a position in a GBF that has no associated value.

The topology of the collections of type G is easily visualized as the Cayley graph $\mathcal{G}$ of G: each vertex in the Cayley graph is an element of the group G and vertex x and y are linked by an edge labeled by `u` if there is a generator `u` in the presentation of G such that $x + \mathtt{u} = y$.

The relationships between Cayley graphs and group theory are pictured in figure 1. A word (a sum of generators) is a path. Path composition corresponds to the group addition. A closed path (a cycle) is a word equal to e (the identity of the group, denoted also by `0`). An equation $v = w$ can be rewritten $v - w = e$ and then corresponds to a cycle in the graph. There are two kinds of cycles in the graph: the cycles that are present in all Cayley graphs and corresponding to group laws (intuitively: a backtracking path like `east+north-north-east`) and closed paths specific to the own group equations (e.g.: `east - north - east+north`). The graph connectivity (there is always a path going from P to Q) is equivalent to say that there is always a solution x to equation $P+x = Q$.

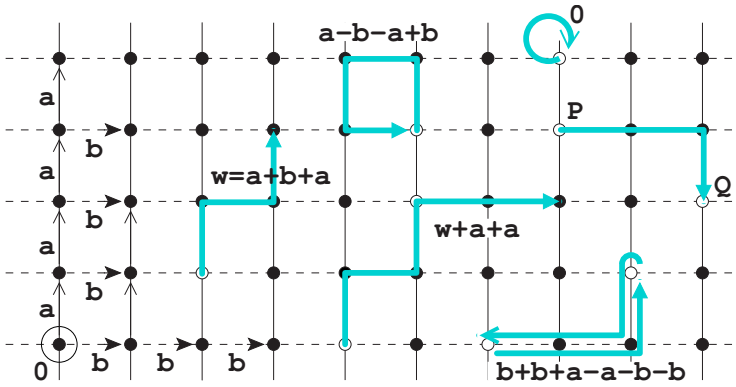


Fig. 1. Graphical representation of the relationships between Cayley graphs and group theory. The GBF pictured is `Grid2` where, for brievety, `a` is used to denote `north` and `b` is used to denote `east`.

3 Aggregation Processes in MGS

Eden's Process. We start with a simple model of growth sometimes called the Eden model (3). The model has been used since the 1960's as a model for such things as tumor growth and growth of cities. In this model, a 2D space is partitioned in empty or occupied cells. We start with only one occupied cell. At each step, occupied cells x with an empty neighbor are selected, and the corresponding empty cell is made occupied (by a copy of the content of x).

The Eden's aggregation process is simply described as the following MGS global transformation:

```
trans Eden = { x, <undef> => x, x }
```

The r.h.s of the rule lists a set of values: in a path transformation, the first value in the r.h.s replaces the first value matched in the left hand side (l.h.s), the second value in the r.h.s replaces the second value matched in the l.h.s, etc.

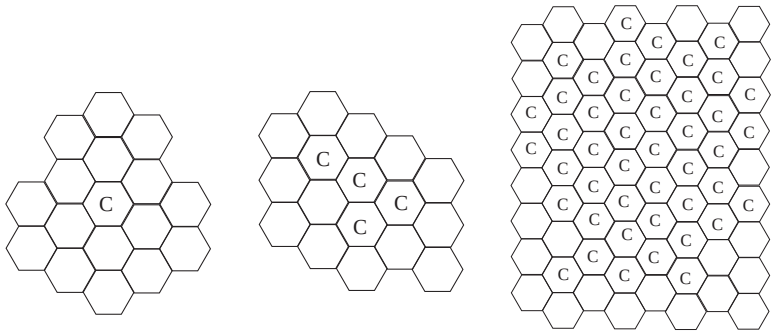


Fig. 2. Eden's model on an hexagonal mesh (initial state, and states after 3 and 7 time steps). This shape corresponds to the Cayley graph of `Hex2` with the following conventions: a vertex is represented as a face and two neighbors in the Cayley graphs share an edge in this representation. An empty cell has an undefined value. Only a part of the infinite domain is figured.

The Growth of a Snowflake. A crystal forms when a liquid is cooled below its freezing point. Crystals start from a seed and then grows by progressively adding more molecules to their surface. As an idealization, the molecules of a snowflake lie on an hexagonal grid and when a piece of ice is added to the snowflake, the heat released by this process inhibits the addition of ice nearby.

This phenomenon leads to the following cellular automata rule (19): a black cell (value `1`) represents a place of the crystal filled with ice and a white cell (value `0`) is an empty place. A white cell becomes black if it has exactly one black neighbor, otherwise it remains white. The corresponding MGS transformation is:

```
trans SnowFlake = { 0 as x / 1 == FoldNeighbor[+,0](x) => 1 }
```

The construction *`path`* `as x` enables the naming of a sub-collection matched by *`path`*: here the identifier `x` refers to some occurence of `0` in the GBF. The construct `FoldNeighbor` is not a function but an operator available only within a rule: it enables to fold a function on the defined neighbors of an element matched in the l.h.s. Here, this operator is used to compute the number of neighbors with value of `x` (the accumulating function is the sum and the initial value is `0`). This transformation acts on a value of type `Hex2` and a possible run is illustrated in figure 3.

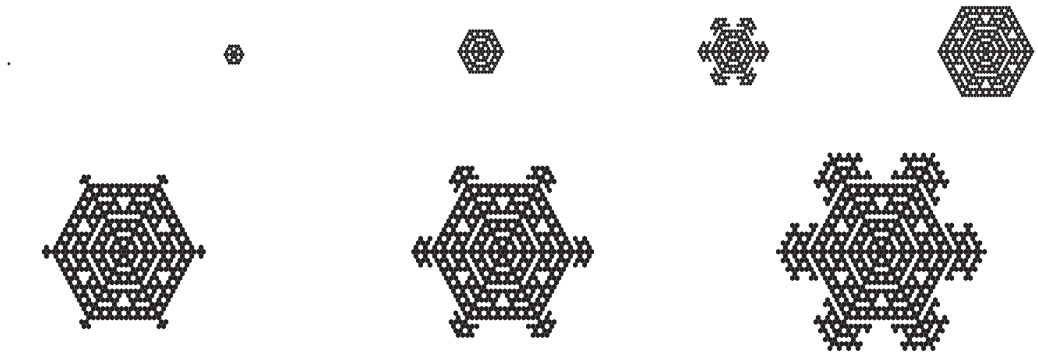


Fig. 3. Formation of a snowflake. The pictured states are the step at time states 1, 4, 8, 12, 16, 18, 20 and 23.

Diffusion Limited Aggregation. In a *diffusion limited aggregation* process, or DLA (18), a set of particles diffuse randomly on a given spatial domain. Initially one particle, the seed, is fixed. When a mobile particle collides a fixed one, they stick together and stay fixed. This process leads to a simple lattice gas automata that could be easily done in MGS using topological collections and transformation:

```
trans dla = {
    ‘mobile, ‘fixed   =>  ‘fixed, ‘fixed
    ‘mobile, <undef>  =>  <undef>, ‘mobile
}
```

We use two symbols `‘mobile` and `‘fixed` to represent respectively a mobile and a fixed particle (MGS's symbols are like Lisp's atoms). The two rules of the transformation deal with:

(1) the aggregation: the first rule specifies that if a diffusing particle is the neighbor of a fixed one, then it becomes fixed (at the current position);
(2) the diffusion: if a mobile particle is neighbor of an empty place (position), then it may leave its current position to occupy the empty neighbor (and its current position is made empty).

Note that the order of the rules is important because, following the rule application semantics of MGS, the first one has priority over the second. The figure 4 presents the final state of the application of the transformation `dla` on two kinds of topological collections: on the left, the neighborhood relationship is homogeneous and a GBF is used. On the right, the `dla` transformation is applied on a meshed sphere modeled as an abtract complex, cf. section 5 below. The elements are the facets, and two facets are neighbors if they share an edge. For more details, refer to (16).

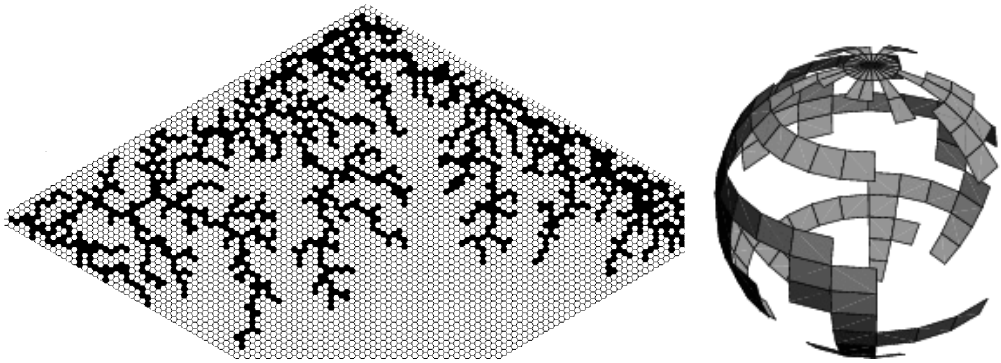


Fig. 4. Examples of DLA on two different topologies: an hexagonal mesh and a sphere. The plain hexagons and facets represent fixed particles. On the sphere, the empty positions are not drawn. Exactly the same transformation is used on the two collections.

4 Accretive Growth of Sierpinski Triangles

The Sierpinski triangles (ST from now on) is a fractal described by Waclaw Sierpinski (polish mathematican, 1882-1969) in 1915 but appearing at least back from the 13th century (e.g. in Cosmati mosaics on the Anagni cathedral in Italy). It is also called the Sierpinski gasket or Sierpinski sieve (17). The ST can be produced by iterating the 2-dimensional morphism defined on $\{0,1\}$ by $0 \to \begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}$ and $1 \to \begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}$. Starting from 1, we obtain:

$$1 \longrightarrow \begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix} \longrightarrow \begin{smallmatrix} 1&0&0&0 \\ 1&1&0&0 \\ 1&0&1&0 \\ 1&1&1&1 \end{smallmatrix} \longrightarrow \begin{smallmatrix} 1&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 1&0&1&0&0&0&0&0 \\ 1&1&1&1&0&0&0&0 \\ 1&0&0&0&1&0&0&0 \\ 1&1&0&0&1&1&0&0 \\ 1&0&1&0&1&0&1&0 \\ 1&1&1&1&1&1&1&1 \end{smallmatrix} \longrightarrow \ldots$$

Equivalently, it can be produced by taking the Pascal's triangle modulo 2 (see Fig. 5). The formula for the binomial coefficient in Pascal's triangle is: $P(0,j) = 1$, $P(i,j) = 0$ for $i > j$ and $P(i,j) = P(i-1,j-1) + P(i-1,j)$ for the remaining cases.

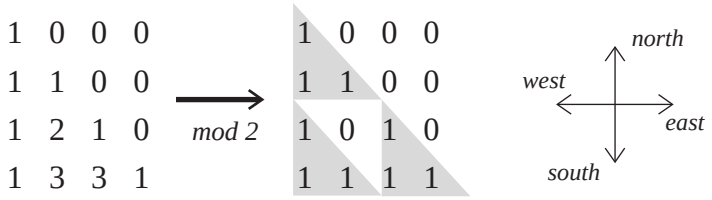


Fig. 5. Taking the binomial coefficients modulo 2 produces the shape of the ST.

Considered modulo 2, this formula gives raise to the transformation below acting on a lattice `Grid2`:

```
trans ST1 = { <undef> |south> x |west> y => (x+y) mod 2, x, y }
```

In this rule, the comma is refined using a GBF generator: `a |south> b` means that `b` is a neighbor of `a` following the `south` direction. The transformation could be iterated on an initial lattice where the positions $(0, j)$ are filled with `1` and positions $(i, 0)$ are filled with `0` for $i > 0$. The result is shown in figure 6.

However, this transformation uses arithmetic operators (the `+` and `mod`). This is not acceptable if we want model more closely a physical growth process: then, the values `0` and `1` must be considered as arbitrary symbols and not as integers amenable to arithmetic operations. To avoid the arithmetic operations, the rule can be split into 4 rules where all the different cases are enumerated. Therefore, we obtain:

```
trans ST2 = {
    <undef> |south> 0 |west> 0  =>  0, 0, 0
    <undef> |south> 0 |west> 1  =>  1, 0, 1
    <undef> |south> 1 |west> 0  =>  1, 1, 0
    <undef> |south> 1 |west> 1  =>  0, 1, 1
}
```

This more elementary computation is close to a specification of the ST growth using a tiling process where the tiles are DNA fragments, as in (15). Following this work, we consider 4 tiles corresponding to the two boolean values a cell (i, j) receives from the cells $(i-1, j-1)$ and $(i-1, j)$. This tiling is easily coded and then simulated in MGS. We use the four symbols `‘T00`, `‘T10`, `‘T01` and `‘T11` to represents the 4 types of tiles: tile `‘T`xy at position (i, j) means that x is the value of $P(i-1, j)$ and y is the value of $P(i-1, j-1)$. So the value `0` is represented by either `‘T00` or `‘T11` and the value `1` by `‘T10` or `‘T01`. Finally, we use a transformation with 4 rules to specify the placement of the tiles:

```
trans ST3 = {
    <undef> |south> (‘T00|‘T11) as x |west> (‘T01|‘T10) as y
    => ‘T01, x, y

    <undef> |south> (‘T00|‘T11) as x |west> (‘T00|‘T11) as y
    => ‘T11, x, y

    ... two additional symmetric rules ...
}
```

The path pattern works as follow: the `|` operator in a pattern denotes an alternative: `‘T00 | ‘T11` matches the symbol `‘T00` *or* the symbol `‘T11`; the `as` construct is used to bind the value of a pattern fragment to a variable: in `(‘T00 | ‘T11) as x` the pattern variable is bound to the actual value matched by the pattern.

In (15), the tile corresponds to molecules and the self-assembly process to a crystallization. In this kind of process, at a given time, only a fraction of the available binding sites effectively accept a new molecule. This is not correctly specified using the standard rule application strategy in MGS. The standard rule application strategy is called *maximal parallel*: in this strategy, *all distinct instances of the subcollections matched by the l.h.s. pattern* are "simultaneously replaced" by the r.h.s. This matching strategy ensures a maximal rule application over a collection. In other words, if a rule is not triggered, then there is no instance of a possible path that fulfills the pattern. Such strategy is for example used in the application of the production rules of a Lindemayer system (14) and may give an account for the fact that the laws of physics apply everywhere.

The non-deterministic character of the process described in (15) is due to errors and the kinetics of chemical reactions and can be taken into account using a *stochastic application strategy*. As a matter of fact, there are several kinds of application strategy in MGS.

With the stochastic application strategy, only one rule is applied (asynchronous mode) and the rule is selected according to a given probability. A pseudo-probability p can be associated to a rule using a new kind of arrow:

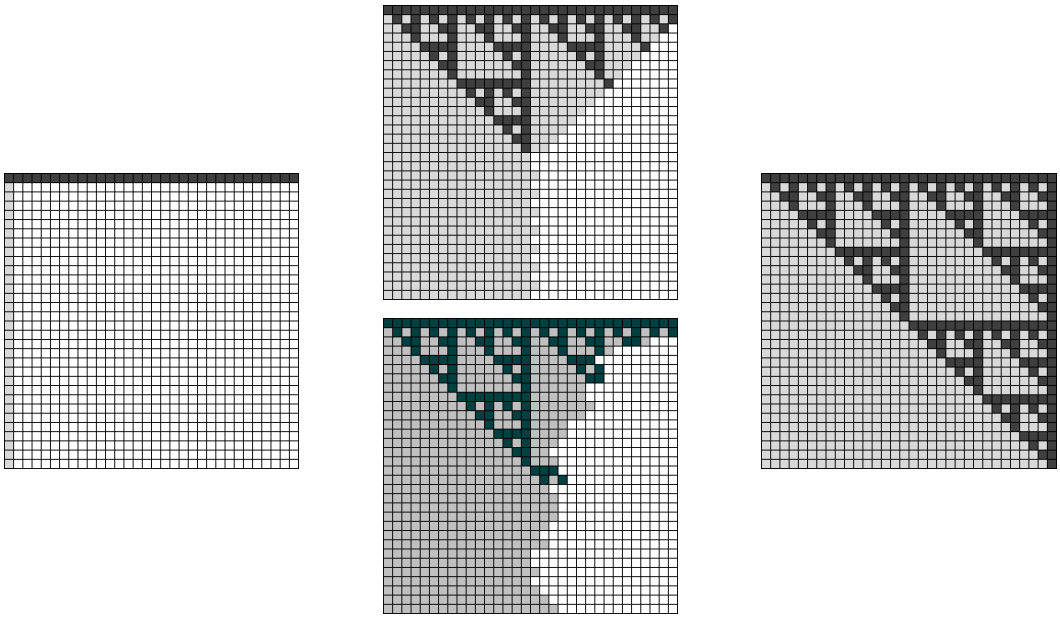


Fig. 6. Growth of ST on GBF: the four figures present the growth of the ST on a `Grid2` at different steps. Left: the initial state; rigth: the final state; on top middle: an intermediate step with the standard rule application strategy (maximal parallel); on bottom middle an intermediate step using a stochastic application strategy (note the irregular boundary). On the grid, blank squares corresponds to the special value `<undef>`, light and dark gray squares are respectively the digits 0 and 1.

lhs ={probability=p}=> *rhs*

(the pseudo-probabilities of the rules within a transformation are summed up and normalised to give a probability; an ordinary rule has implicitly a pseudo-probability of 1).

The use of the stochastic application strategy is specified in the transformation application, not in the transformation definition. The expression:

```
ST3[strategy='stochastic](c)
```

applies the transformation `ST3` using the stochastic application strategy to collection `c`. Because the four rules in `ST3` do not define any explicit probability, each rule as an equal chance to be chosen.

In the building of a ST by the previous transformations, the final result is not affected by the choice of a stochastic application strategy instead of the maximal parallel one, because each rule application is exclusive from the other ones. However, intermediate results may differ, cf. figure 6.

5 Handling arbitrary shapes

Using a group presentation to define the underlying lattice of a DLA process is a powerful mechanism. However, the sphere in the right of figure 4 cannot be specified as the Cayley graph of a group. Indeed any Cayley graph is a regular graph (i.e.: each vertex has the same number of neighbors) and this is not the case for the sphere.

In addition, the self-assembly processes we have described so far take place in a predefined discretized space. This is a shortcomming in several situations, for example when there is a need to handle arbitrary rotation.

In the rest of this chapter, we will use *abstract cellular complexes* to represent and build incrementally arbitrary shapes in arbitrary dimensions.

5.1 Cellular complexes

The notion of abstract cellular complex comes from the combinatorial algebraic topology theory. An abstract cellular complex represents a space build by gluing together cells of various dimension (13). This data structure generalizes the idea of a graph: cells of dimension 0 are vertices, cells of dimension 1 are edges, cells of dimension 2 are faces, etc.

More formally, a cellular complex is a set K of abstract elements together with an antisymmetric, irreflexive and transitive binary relation $B \subset K \times K$ and a function $dim : K \rightarrow \mathbb{N}$. The relation B is called the *boundary relationship*, and the function dim, *dimension*. Moreover, B and dim verify: $\forall (c_1, c_2) \in B, dim(c_1) < dim(c_2)$. The elements of K are called *topological cells*. If $c \in K$ and $dim(c) = n$, then we say that c is an *n-cell*.

The structure of cellular complex can be easily handled using a lattice based on the boundary relationship B. The following notion will be used in the rest of the paper:

(1) **incidence:** let c_1 and c_2 be two cells, c_1 and c_2 are *incident* if $(c_1, c_2) \in B$ or $(c_2, c_1) \in B$. Therefore, the lattice based on B is also called *incidence graph*.
(2) **notion of face:** let c be an n-cell, the *faces* of c, are the set of the $(n-1)$-cells c_f such that $(c_f, c) \in B$. As an example, if c is a triangle (a 2-cell), the faces of c are the three edges of c.
(3) **notion of coface:** let c be an n-cell, the *cofaces* of c, are the set of the $(n+1)$-cells c_{cf} such that $(c, c_{cf}) \in B$. Of course, if c_1 is a coface of c_2, c_2 is a face of c_1.

In this context, the corresponding MGS topological collection associates a value with each topological cell. This association corresponds to the notion of *chain* developed in algebraic topology.

5.2 Patch transformations

Path patterns, previously used to operate on GBF, are not well fitted to handle arbitrary cellular complex: an arbitrary sub-collection cannot be easely described by a path. *Patch transformations* have been created to handle any arbitrary cellular sub-complex. As any kind of transformations, patch transformations acts on topological collections using rewriting techniques, and are defined by case where each case is a rewriting rule. The rest of this section describes the patch rules syntax.

Patch patterns. In patch patterns, a subcollection is described through the elements that compose it (*i.e.*, the n-cells) and their topological organization. We use two new operators $<$ and $>$ to represent the incidence relationship: $a < b$ (or $b > a$) means a is a face of b (and obviously b is a coface of a).

So we define a new syntax for pattern: a patch pattern `PPat` is a sequence of basic filters `PBF` separated by an incidence operator `POp`. Each basic filter matches one n-cell. The new grammar is defined by:

$$
\begin{array}{rcl}
\texttt{PPat} & ::= & \texttt{PBF} \mid \texttt{PBF POp Pat} \\
\texttt{POp} & ::= & \texttt{<} \mid \texttt{>} \mid \texttt{' '} \\
\texttt{PBF} & ::= & \texttt{PVar} \mid \texttt{\~{}PVar} \\
\texttt{PVar} & ::= & \text{id} \mid \text{id} : [\texttt{dim=}exp, \ldots, clause_i, \ldots]
\end{array}
$$

where id ranges over the pattern variables and *exp* is an expression. A systematic interpretation for these patterns is given below.

Variable. A pattern variable `a:[dim=`*exp*`, ...]` matches exactly one element with a well defined value. The pattern variable `a` denotes the matched element.

For each variable, some constraints can be given between square brackets. The constraint `dim =` *exp* specifies the dimension of the filtered n-cell (the expression *exp* has to be evaluated into an integer). Other constraints can be given and some are detailed below.

Contrary to what was required with path patterns, a pattern variable `a` may occur several times elsewhere in the whole rule. If the same variable is defined several times, both guards are grouped together in a conjunction. For example, `a:[`$clause_1$`]` `a:[`$clause_2$`]` is equivalent to `a:[`$clause_1$`,` $clause_2$`]`.

Neighborhood. The expression e *op* p matches an element e that respect the incidence property denoted by *op* with the first element e' matched by p. The incidence relation *op* is either: "`<`" which means that e is a face of e', "`>`" which means that e is a coface of e' or " " (i.e. a white space) if e and e' don't have to respect any relationship.

Consuming. Usually, if an element is matched by a pattern, it cannot be matched in another pattern matching. Indeed, two sub-collections matched by the l.h.s. of some rules of a transformation cannot overlap. We say that the elements matched by a pattern are "consumed".

Nevertheless, we need sometimes to refer to some particular elements (e.g. for the sake of the reconstruction in the r.h.s.), but *without* consuming those elements (that is: leaving this element free for another possible matching). This is the role of the tilda qualifier: `~v` is a basic pattern that leaves the element `v` free to be matched in a forthcoming rule application. To avoid any overspecification, any matched and unconsumed element can be matched but not consumed in another occurrence of a patch pattern.

Note that using this syntax, the specification of a path of n-cells is easy: the path pattern `a,b,c` is equivalent to the patch pattern

```
a < _ > b < _ > c
```

the `_` is used to match a cell without giving it a name.

As an example, the following pattern matches a triangle `f` with its boundary and without consuming edges `e1`, `e2` and `e3`:

```
f:[dim=2]
v1 < ~e1 > v2 < ~e2 > v3 < ~e3 > v1
~e1 < f
~e2 < f
~e3 < f
```

In order to have a more concise syntax, syntactic sugar is used to refer to a part of the neighborhood. The previous example can be rewritten as:

```
f:[dim=2, (e1,e2,e3) in faces]
v1 < ~e1 > v2 < ~e2 > v3 < ~e3 > v1
```

The clause "`(e1,e2,e3) in faces`" means that a matched cell must have at least 3 faces named `e1`, `e2` and `e3`. These elements are not consumed.

Specifying the right hand side. Basically, the left hand side specifies a subpart of the incidence graph of the cellular complex. So, the reconstruction can be described by an incidence graph that will replace the one matched by the patch pattern. The syntax to specify this graph is very close to the patch pattern specification: it is a sequence of two kinds of elements. An element can be either

`id:[`*exp*`]`

which refers to an existing matched and consumed element named `id` in the patch pattern and whose value has to be updated by the value of *exp*, or

`‘id:[dim=`exp_d`,`
`faces=(`id_1`,`id_2`,...,‘`id_1`,‘`id_2`,...),`
`cofaces=(`id_1`,`id_2`,...,‘`id_1`,‘`id_2`,...),`
`val=`exp_v`]`

which refers to a new cell of dimension exp_d, with the associated value exp_v. If the `val` clause is missing, there is no value associated to the cell. The symbol `‘id` can be used elsewhere in another clause to refer to this newly created element.

Two clauses `faces` and `cofaces` are given to specify the list of faces and cofaces of the new cell `‘id`. In these lists, the variables refer to the matched elements (i.e. they are pattern variables) while the symbols refer to new elements created elsewhere during the reconstruction step. The default value for `faces` and `cofaces` is the empty list. Note that the list of faces and cofaces do not need to be complete: all the faces and cofaces specifications are collected and normalized (that is, if x appears in the face list of y, then y is added if necessary in the coface list of x and vice-versa).

Note that a pattern variable in an expression may refer to two different values: the topological cell corresponding to the matched element *or* the value associated with the topological cell. The denoted value depends on the context: for instance, in a `face` clause, a pattern variable refers to the matched cell. In an ordinary expression, a pattern variable refers to the value associated to the matched cell. The operator ˆ applied on pattern variable can be used to force the denotation of the matched cell is necessary.

A simple Example. Here is a short and simple example of a patch rule that matches an edge and adds a new vertex on it:

```
v1 < e:[dim=1] > v2
=> ‘e1:[dim=1,faces=(v1, ‘v)]
   ‘v:[dim=0,cofaces=(‘e1, ‘e2)]
   ‘e2:[dim=1,faces=(v2, ‘v)]
```

In this example, three new elements are created `‘v`, `‘e1` and `‘e2`, and `e` is removed. Figure 7 gives an illustration of this rewriting step.

Fig. 7. Insertion of a vertex on an edge.

6 Self-assembled Polymers

In this section we will use abstract cellular complexes and patch transformation to model a polymerization process. The model does not refer to a real example. Its only purpose is to illustrate the ability of modeling arbitrary shapes using abstract complexes. In the next section we will use abstract complexes to build Sierpinsky triangle and tetrahedra by carving.

Polymers are long-chained molecules with repeating units. They are formed by reacting monomers (the repeating part of the chain) in a process known as polymerization. These reactions fall into 2 main groups – addition polymerization and condensation polymerization. Addition polymerization is adding monomers and polymers together to form the chain. Thus, it is an example of accretive growth. In condensation polymerization, a small molecule, often water, is removed as part of the process.

Here we suppose that a monomer is represented by a rectangle, that is, a 2-cell. When a monomer is joined to another monomer or a chain, the junction may form an angle that depends of the monomer constituent. For the sake of the

simplicity, we suppose that each monomer is associated with its corresponding angle, see figure 8.

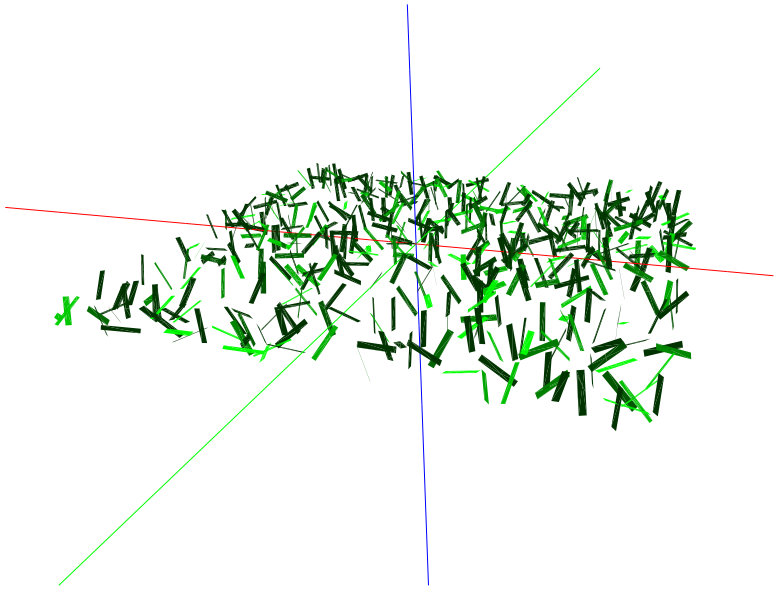


Fig. 8. The initial population of monomers. They are represented with their corresponding joining angle.

The self-assembly process consists in taking arbitrarily two monomers (or intermediate polymers) and gluing them together, as sketched in figure 9. We assume that a monomer or a polymer has only one active binding site which correspond to an edge. The functions `mark` and `unmark` are used to activate and inhibit a binding site.

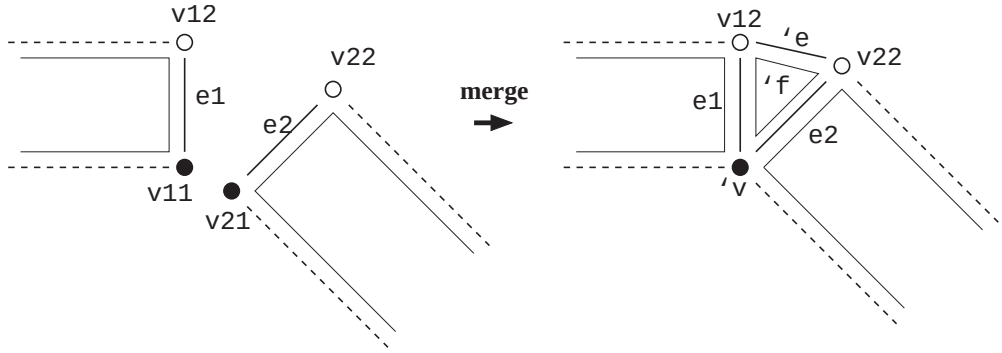


Fig. 9. Gluing of two monomers: a triangular 2-dimensional cell is added between the two rectangles. Beware that the rectangles are in a general position in a tridimensional space (see figure 8) and consequently, the three cells are not in the same plane.

Polymerization is a patch transformation that takes two elements in the previous multiset and joins them. This operation is iterated until there is only one polymer left in the solution. The transformation is applied following a stochastic strategy. Here is the code of the `reaction` transformation:

```
trans reaction = {
    a,b => let cc = mark(merge(a + b)) in translate(cc)
} ;;
```

This transformation implements the chemical reaction in the chemical soup: two complexes are selected and merged using the patch `merge`. The expression `a + b` creates a complex made of the two disjoint complexes `a` and `b`. After the merging, an edge of the boundary is selected to become the new active site using the function `mark`.

The only rules in the `merge` patch is looking for two active edges `e1` and `e2`. They have to belong to two disconnected complexes. The guards (the clause that starts with a "`?`") check these properties. We only match and consume the vertices `v11` and `v21` because they are merged to create `‘v` (see figure 9). The other cells remain after the application of the rule. The patch `merge` must be applied in an asynchronous mode to link the molecules by a unique edge.

```
patch merge = {
    v11 < ~e1:[dim=1, ?(...) ] > ~v12
    v21 < ~e2:[dim=1, ?(...) ] > ~v22
    =>
      ‘v:[dim=0,cofaces=join(cofaces(v11),cofaces(v21)),...]
      ‘e:[dim=1,faces=(v12,v22),cofaces=(‘f),...]
      ‘f:[dim=2,faces=(‘e,e1,e2),...]
} ;;
```

This patch searches for two edges `e1` and `e2`. They have to belong to two distinguished complexes and correspond to a possible link between the molecules. The guards check these properties. We only match and consume the vertices `v11` and `v21` because they are merged to create `‘v`. The other cells remain unchanged after the application of the rule. The patch `merge` must be applied in an asynchronous mode to link the molecules by a unique edge. Figure 10 illustrates a final result.

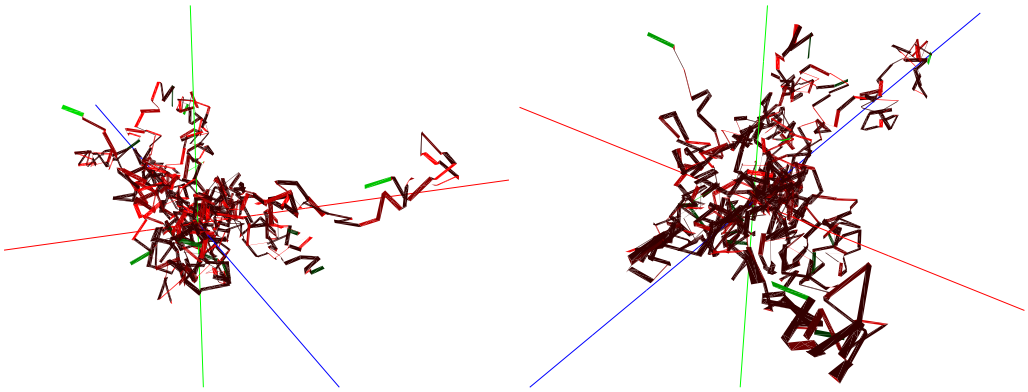


Fig. 10. A final polymer, from two points of view, generated by the `reaction` and `merge` transformations.

7 Carving Sierpinski Triangles

In this last section, we will use abstract complexes to introduce a new kind of self-assembly: self-assembly by carving. We will use the shape already introduced in section 4: the Sierpinski triangles and its tridimensional generalization. Building a ST by carving is illustrated in figure 11. This process is easily coded in MGS using patch patterns on abstract cellular complexes.

7.1 Carving transformation

To represent the ST, we use an abstract cellular complex where the value of a vertex represents the coordinate of an embedding of the ST in the plane. The main advantage of using cellular complexes is that we can handle cells of various dimensions to represent all the elements that compose the ST. In fact, in the previous representation, the ST were patterns appearing on a matrix of digits, that is, on a predefined space. Here the concrete geometric structure of the ST is specified and the building of the ST also builds "its own embeding space". Building ST by carving means iterating the removing of the central part of a triangle. The limit of this process corresponds to the ST fractal (see fig. 11).

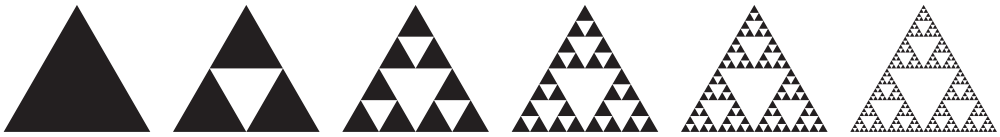
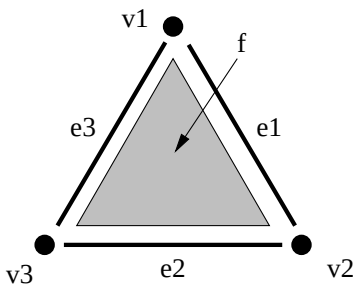


Fig. 11. ST can also be produced by iterating the carving of a triangle inside another triangle.

The initial step consists in a unique triangle. It corresponds to the digit 1 of the previous representation at the beginning of the series of transformations given in section 4. It is composed of three 0-cells (vertices), three 1-cells (the edges that link the vertices pairwise) and a unique 2-cell (the unique coface of the edges):

```
let rec v1 = new_cell(0,[],[e1,e3])
and     v2 = new_cell(0,[],[e1,e2])
and     v3 = new_cell(0,[],[e2,e3])
and     e1 = new_cell(1,[v1,v2],[f])
and     e2 = new_cell(1,[v2,v3],[f])
and     e3 = new_cell(1,[v3,v1],[f])
and     f  = new_cell(2,[e1,e2,e3],[]);;
```



The primitive `new_cell(`d`,`ℓ`,`ℓ'`)` creates a new topological d-cell whose faces are given as the list ℓ and cofaces are ℓ'.

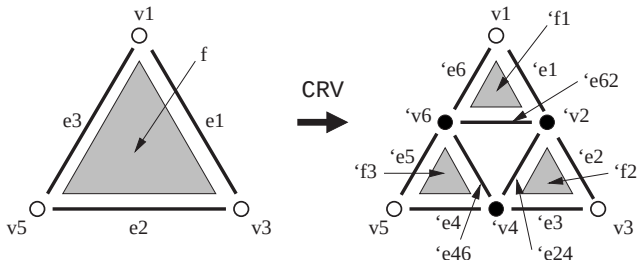


Fig. 12. Local topological transformation: the patch transformation `CRV` divides a triangle into 3 smaller ones, leaving a triangular hole in the middle.

The carving, which is the unique step of the process, is the topological surgery described on figure 12. The MGS patch transformation follows:

```
patch CRV = {
  ~v1 < e1:[dim = 1] >
  ~v3 < e2:[dim = 1] >
  ~v5 < e3:[dim = 1] > ~v1
  f:[dim = 2, (e1,e2,e3) in faces]
  => ‘v2[dim=0, val=average(v1,v3)]
     ‘v4[dim=0, val=average(v3,v5)]
     ‘v6[dim=0, val=average(v5,v1)]
     ‘e1[dim=1, faces=(v1,‘v2), val=‘edge]
     ‘e2[dim=1, faces=(‘v2,v3), val=‘edge]
     ‘e3[dim=1, faces=(v3,‘v4), val=‘edge]
     ‘e4[dim=1, faces=(‘v4,v5), val=‘edge]
     ‘e5[dim=1, faces=(v5,‘v6), val=‘edge]
     ‘e6[dim=1, faces=(‘v6,v1), val=‘edge]
     ‘e24:[dim=1, faces=(‘v2,‘v4), val=‘face]
     ‘e46:[dim=1, faces=(‘v4,‘v6), val=‘face]
     ‘e62:[dim=1, faces=(‘v6,‘v2), val=‘face]
     ‘f1:[dim=2, faces=(‘e6,‘e1,‘e62), val=‘face]
     ‘f2:[dim=2, faces=(‘e2,‘e3,‘e24), val=‘face]
     ‘f3:[dim=2, faces=(‘e4,‘e5,‘e46), val=‘face]
}
```

The function `average` that is not detailed here, computes coordinates of the middle point between its 2 arguments. In this patch, all the elements of a triangle are matched. They are all consumed except the vertices that remain after the application of the rule. Moreover, they may be shared by other triangles, and therefore they can't be destroyed and have to be matchable in a parallel application of the rule. Note that each edge has only one coface at any step of the building process. This is easy to prove considering the initial state and making an inductive reasoning at each step: in fact, the rule creates 9 new edges that have a unique coface. This kind of proofs on rewriting systems is usual and allows easy verifications of properties on self-assembly processes specified using rewriting techniques.

The patch `CRV` is a little long and may be difficult to understand. Moreover, `CRV` depends on the property of having one coface per edge. Indeed, if we take another initial state that does not assume this property, some aberrations can appear as shown on figure 13.

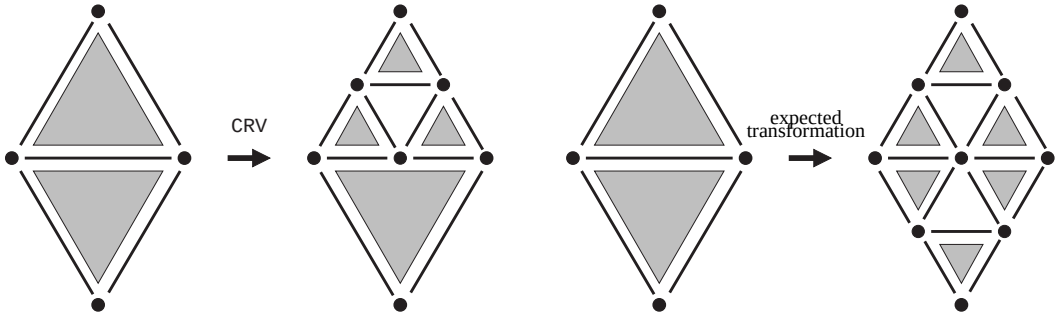


Fig. 13. On the left, application of the `CRV` patch on a different initial state. On the right, the expected result.

In this example, two triangles share an edge. If we apply `CRV` on one of the two triangles, the topology of the other one is modified. It is not a triangle anymore, but a square.

To reach the expected result irrelevantly of the initial shape, the building process can be divided into two distinguished transformations. The first one, `AV`, adds a vertex in the middle of each edges (see Fig. 14):

```
patch AV [gen] = {
  ~v1 < e:[dim = 1] > ~v2
  => ‘v:[dim=0, cofaces=(‘e1,‘e2), val=average(v1,v2,gen)]
     ‘e1:[dim=1, faces=(v1,‘v), val=‘edge]
     ‘e2:[dim=1, faces=(v2,‘v), val=‘edge]
}
```

An additional argument, `gen`, is required to distinguish the new and the old vertices and is used in the next computation step. Conveniently `gen` can be an integer representing the generation number of the vertex. The function average is modified to compute and store the position and the age of the vertex.

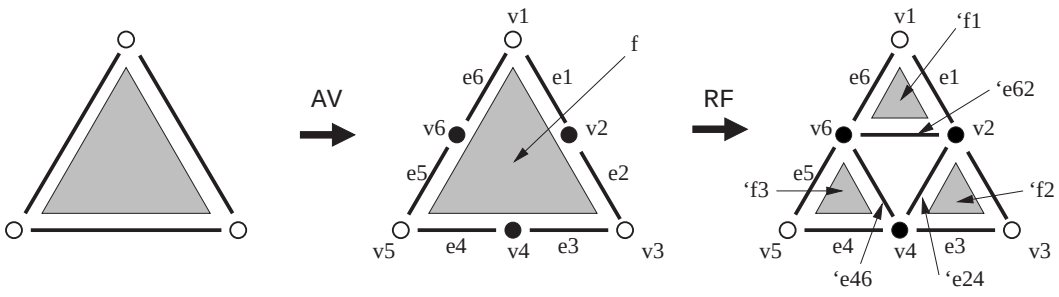


Fig. 14. Carving a triangle. The first transformation `AV` adds vertices in the middle of every edge. The second transformation `RV` refines the central hexagonal face into three triangles.

The next step looks for all the hexagons and replaces them with three triangles (see figure 14):

```
patch RF [gen] = {
  f:[dim=2, faces = (e1,e2,e3,e4,e5,e6)]
  ~v1 < ~e1 > ~v2:[? ok(gen,v2)] < ~e2 >
  ~v3 < ~e3 > ~v4:[? ok(gen,v4)] < ~e4 >
  ~v5 < ~e5 > ~v6:[? ok(gen,v6)] < ~e6 > ~v1
  => ‘e24:[dim=1, faces=(v2,v4)]
     ‘e46:[dim=1, faces=(v4,v6)]
     ‘e62:[dim=1, faces=(v6,v2)]
     ‘f1:[dim=2, faces=(e6,e1,‘e62)]
     ‘f2:[dim=2, faces=(e2,e3,‘e24)]
     ‘f3:[dim=2, faces=(e4,e5,‘e46)]
}
```

Note the guards in the specification of the matched vertices: the predicate `ok` checks the vertices `v2`, `v4` and `v6` to have the right generation number `gen`.

The ST is 2-dimensional. The corresponding 3-dimensional fractal is called the *Sierpinsky sponge*, see figure 15. These pictures have been generated using the current implementation of the MGS language. The 3D program is the same as the 2-dimensional one. In fact, we use the patches `AV` and `RV` to perforate the faces of a tetrahedron. A last patch (not detailed in this chapter) is used to create the 4 smaller tetrahedrons associated with the 4 original vertices. It is interesting to note that the Sierpinsky process is recursive within the dimension: we first applied an iteration of the 2D process (with `AV` and `RV`), and a last patch finalized the 3D construction.

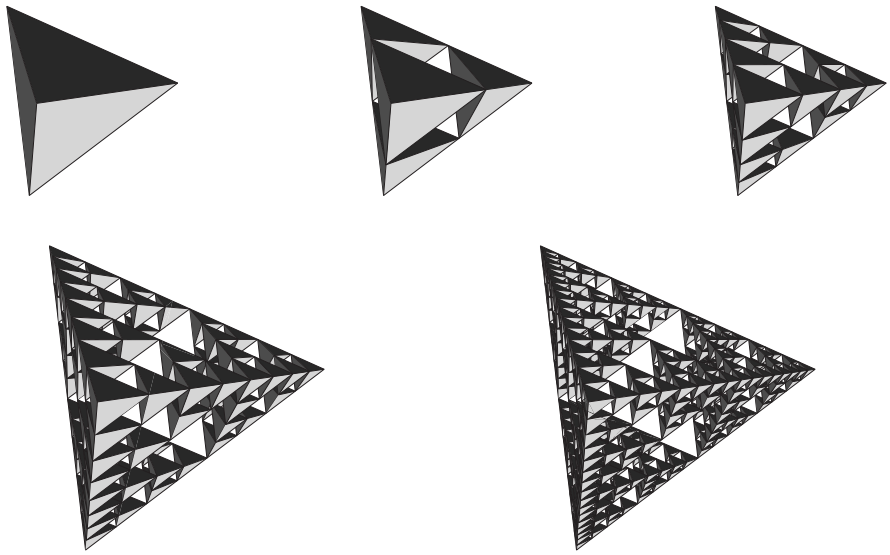


Fig. 15. Sierpinski sponge building process: initial state and steps 1, 2, 3 and 4.

8 Conclusions

In this paper we have presented the use of a DSL language for the modeling and the simulation of two kinds of self-assembly processes: by accretive growth and by space carving. Despite their specificities, we are convinced that they are paradigmatic of a full class of self-assembly processes. For instance, arbitrary Wang tiling (8) could be coded in MGS following the scheme presented in section 4. The self-assembly of ST has been really implemented using DNA molecules (15). Compared to the TAM and kTAM simulations used in this last work, the MGS modeling is more abstract: the purpose is not to study the implementation in DNA but to investigate the shape produced by some families of self-assembly processes. We insist on the expressiveness brought by the notions of topological collections and their transformation. For example, the patch language used in section 7 is powerful enough to produce Sierpinski and Menger sponge (a generalization of carving a tetrahedron and a cube in 3D), see fig. 16.

The MGS programming style corresponds to the rule-based programming paradigm. Rule-based programming is currently experiencing a renewed period of growth with the emergence of new concepts and systems that allow a better understanding and better usability. However, the vast majority of rule-based languages (like expert-systems) are funded on a logical approach (computation is a logical deduction and a deduction step is specified by a rule) which is not adequate to describe various spatio-temporal processes. We hope that the previous section have demonstrated the ability of MGS to express easely and concisely the building of sophisticated spatial structures, like the ones needed to model self-assembly processes.

Our topological approach is motivated by some considerations internal to computer science, and also by the needs expressed by some application domains. A target application domain for MGS is the modeling and simulation of dynamical systems and especially dynamical systems that exhibit a dynamic structure (4). This kind of dynamical systems is very challenging to model and simulate. New programming concepts could be developed to ease their modeling and simulation. Another target application that should be investigated is robotics self-assembly.

All the examples presented in this chapter have been developped and processed using the MGS interpreter. The results of the various programs have been stored in data files that are later used by separate external viewers for graphic representation.

The perspectives opened by this work are numerous. We want to develop several complementary approaches to define new topological collection types.

One approach to extend the GBF applicability is to consider monoids instead of groups, especially automatic monoids which exhibit good algorithmic properties. Another direction is to handle other kind of combinatorial spatial structures. At the language level, the study of the topological collections concepts must continue with a finer study of transformation types. Several kinds of restrictions can be put on the transformations, leading to various kinds of pattern languages and rules. The complexity of matching such patterns has to be investigated. The efficient compilation of a MGS program is a long-term research plan. We have considered in this paper only one dimensional paths, but a general n-dimensional notion of path exists and can be used to generalize the substitution mechanism of MGS. From the applications point of view, we are targeted by the simulation of more complex developmental processes (7).

Acknowledgments

We are grateful to Natalio Krasnogor that give us the initial motivation to write this chapter. We are also grateful to the anonymous referrees for comments and suggestions. A big thank is due to Olivier Michel at the University of Evry which is one of the designer of MGS. Further acknowledgments are also due to M. Gheorghe at the University of Sheffield, F. Delaplace and J. Cohen at the University of Evry, C. Godin and P. Barbier de Reuille at CIRAD-Montpellier, the members of the Epigenomic group at GENOPOLE-Evry, P. Prusinkiewicz at the University of Calgary, and the participants of the CellCom FET proposal for helpful discussions, biological motivations, fruitful examples and challenging questions. This research is supported in part by the CNRS, GENOPOLE-Evry, the GDR ALP, IMPG, and the University of Evry.

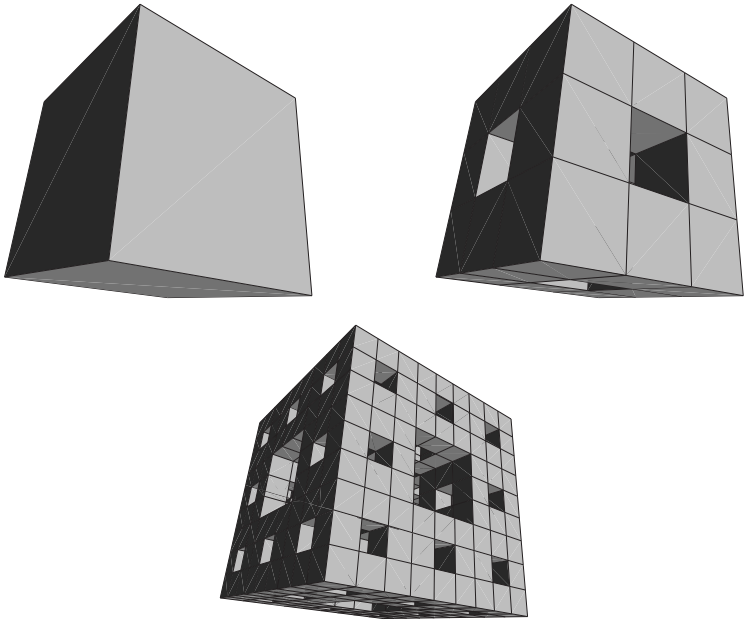


Fig. 16. Menger sponge building process: initial state and steps 1 and 2.

References

[1] Abelson, Allen, Coore, Hanson, Homsy, Knight, Nagpal, Rauch, Sussman, and Weiss. Amorphous computing. *CACM: Communications of the ACM*, 43, 2000.

[2] J.-P. Banâtre, P. Fradet, and D. L. Métayer. Gamma and the chemical reaction model: Fifteen years after. *Lecture Notes in Computer Science*, 2235:17–44, 2001.

[3] M. Eden. In H. P. Yockey, editor, *Symposium on Information Theory in Biology*, page 359, New York, 1958. Pergamon Press.

[4] J.-L. Giavitto. Invited talk: Topological collections, transformations and their application to the modeling and the simulation of dynamical systems. In *Rewriting Technics and Applications (RTA'03)*, volume LNCS 2706 of *LNCS*, pages 208 – 233, Valencia, June 2003. Springer.

[5] J.-L. Giavitto and O. Michel. Declarative definition of group indexed data structures and approximation of their domains. In *Proceedings of the 3nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-01)*. ACM Press, Sept. 2001.

[6] J.-L. Giavitto and O. Michel. Modeling the topological organization of cellular processes. *BioSystems*, 70(2):149–163, 2003.

[7] J.-L. Giavitto and O. Michel. *Molecular Computational Models: Unconventional Approaches*, chapter Modeling Developmental Processes in MGS, pages 1–46. Idea Group, 2004.

[8] B. Grünbaum and G. C. Shephard. *Tilings and patterns*. W. H. Freeman & Co., 1986.

[9] P. Horn. Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM Research, Oct. 2001. `http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf%`.

[10] J. A. Kaandorp. *Modelling growth forms of biological objects using fractals*. PhD thesis, University of Amsterdam, 1992.

[11] K. N. Kutulakos and S. M. Seitz. A theory of shape by space carving. *International Journal of Computer Vision*, 38(3):199–218, July 2000.

[12] V. Manca, C. Martin-Vide, and G. Paun. New computing paradigms suggested by dna computing: computing by carving. *Biosystems*, 52(1-3):47–54, Oct. 1999.

[13] J. Munkres. *Elements of Algebraic Topology*. Addison-Wesley, 1984.

[14] P. Prusinkiewicz, A. Lindenmayer, J. S. Hanan, et al. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[15] P. W. K. Rothemund, N. Papadakis, and E. Winfree. Algorithmic self-assembly of dna sierpinski triangles. *PLoS Biol*, 2(12):e424, 2004. www.plosbiology.org.

[16] A. Spicher, O. Michel, and J.-L. Giavitto. A topological framework for the specification and the simulation of discrete dynamical systems. In *Sixth International conference on Cellular Automata for Research and*

Industry (ACRI'04), volume 3305 of *LNCS*, Amsterdam, October 2004. Springer.

[17] I. Stewart. Four encounters with sierpinski's gasket. *Mathematical Intelligencer*, 17:52–64, 1995.

[18] T. A. Witten and L. M. Sander. Diffusion-limited aggregation, a kinetic critical phenomenon. *Phys. Rev. Lett.*, 47:1400–1403, 1981.

[19] S. Wolfram. *A new kind of science*. Wolfram Media, 2002.