# Algorithmic Self-assembly by Accretion and by Carving in MGS

Antoine Spicher, Olivier Michel, and Jean-Louis Giavitto

LaMI UMR 8042 CNRS – Université d'Evry, Genopole,
523 place des Terrasses de l'Agora, 91000 Evry, France
{aspicher, michel, giavitto}@lami.univ-evry.fr,
http://mgs.lami.univ-evry.fr

**Abstract.** We report the use of MGS, a declarative and rule-based language, for the modeling of various self-assembly processes. The approach is illustrated on the fabrication of a fractal pattern, a Sierpinsky triangle, using two approaches: by *accretive growth* and by *carving*. The notion of topological collections available in MGS enables the easy and concise modeling of self-assembly processes 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. 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 behaviour.

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* [7].

The emergence of the global structure of self-assembled systems cannot be deduced from the individual composing elements. To obtain a deeper insight of these complex systems, simulation models are often the only available option. 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.

E. Talbi et al. (Eds.): EA 2005, LNCS 3871, pp. 189–200, 2006.

*Self-Assembly by Accretive Growth.* One of the most fundamental kind of self-assembly is certainly processes where basic elements are united into a structure during a *growth process*. A growth process can be described as an iteration process. In such a process the output of an iteration step is used again as input for the next iteration step. In a growth process the form of a growing object in a certain growth stage is also determined by the form of the object in the preceding growth stage. In each growth stage, new basic entities (e.g., material) are added to this 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* [9]. 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 [8], 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.

### 1.2 DSL 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].

### 1.3 Organization of the Paper

The rest of this paper is organized as follows. The next section provides a quick introduction to MGS. Two kinds of topological collections are sketched: group-based data fields which are used to define various lattices used in the modeling of accretive growth, and abstract cellular complexes used to model arbitrary shape for carving. Section 3 presents three short and well-known examples of growth by aggregation processes in MGS. Section 4 shows the self-assembly of Sierpinsky triangles and section 5 build the same shape but using a carving process. The conclusion reviews some previous, related and future work.

## 2 A Short MGS Presentation

### 2.1 Transformations of Topological Collections

In this section, we present the notions needed to understand the MGS coding of the previous 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 the notions 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 the *pattern* $\beta$,
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.

*Path Patterns.* Path patterns match paths in a collection. A path pattern is a sequence of elements separated by a comma. The 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 5.

### 2.2 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 elements 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 regular Von Neuman neighborhood in a classical array (a cell above, below, left or right – not diagonal). The two names `north` and `east` (together with their inverses `-north` and `-east`, always provided in a group structure) 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 1. Each cell has six neighbors (following the three generators and their inverses). 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.

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 word, the group elements act as indices of a generalized array. An empty GBF is the everywhere undefined function.

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 $G$

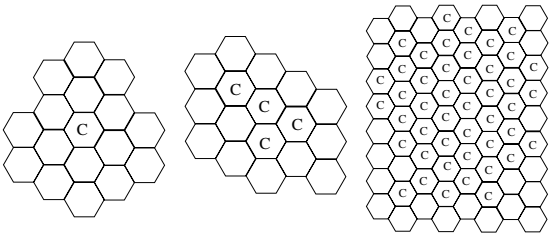

**Fig. 1.** Eden's model on an hexagonal mesh (initial state, and states after 3 and 7 time steps). This shape corresponds to the Cayley graph of `Hex2` with the following conventions: a vertex is represented as a face and two neighbors in the Cayley graphs share an edge in this representation. An empty cell has an undefined value. Only a part of the infinite domain is figured.

and vertex $x$ and $y$ are linked if there is a generator `u` in the presentation of $G$ such that $x + \mathtt{u} = y$. A word (a sum of generators) is a path. Path composition corresponds to group addition. A closed path (a cycle) is a word equal to $e$ (the identity of the group). 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 $\mathtt{east} + \mathtt{north} - \mathtt{north} - \mathtt{east}$) and closed paths specific to the own group equations (e.g.: $\mathtt{east} - \mathtt{north} - \mathtt{east} + \mathtt{north}$). 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 $P + x = Q$.

## 3 Growth 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 use the value `true` for an occupied cell and left undefined the unoccupied cells). We start with only one occupied cell. At each step, occupied cells with an empty neighbor are selected, and the corresponding empty cell is made occupied.

The Eden's aggregation process is simply described as the following MGS global transformation: `trans Eden = { x, <undef> => x, true }`.

*The Growth of a Snowflake.* A crystal forms when a liquid is cooled below its freezing point. Crystals start from a seed and then grow 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 [16]: 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:

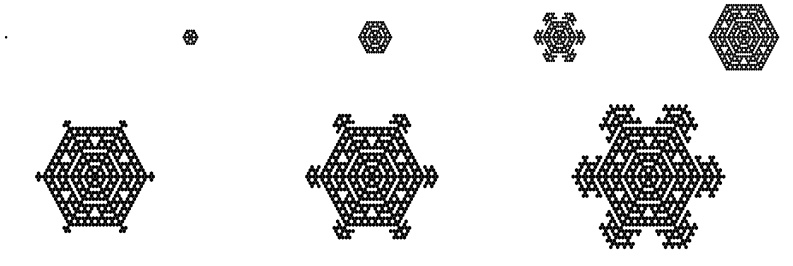

**Fig. 2.** Formation of a snowflake. The pictured states are the steps at time steps 1, 4, 8, 12, 16, 18, 20 and 23.

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

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 (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 2.

*Diffusion Limited Aggregation.* In a *diffusion limited aggregation* process, or DLA [15], 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 a topological collection 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. Figure 3 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

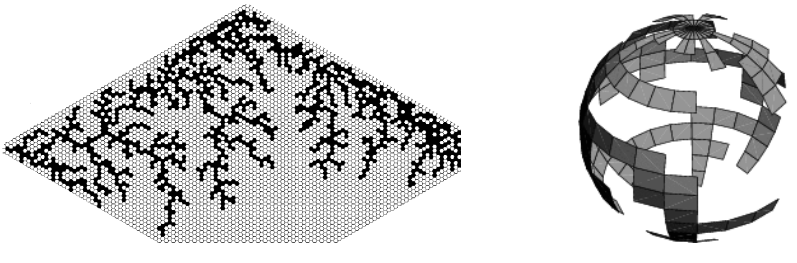

**Fig. 3.** Example of DLA on two different topologies: an hexagonal mesh and a sphere. The plain hexagons and facets represent fixed particles. On the sphere, the empty positions are not drawn. The same transformation is used on the two collections.

on a meshed sphere. The elements are the facets, and two facets are neighbors if they share an edge. For more details, refer to [13].

## 4 Accretive Growth of Sierpinski Triangles

The Sierpinski triangles (ST from now on) is a fractal described by Sierpinski in 1915 and appearing in Italian art from the 13th century. It is also called the Sierpinski gasket or Sierpinski sieve [14]. The ST can be produced by taking the Pascal's triangle modulo 2 (see figure 4), or equivalently by iterating the bidimensional morphism defined on $\{0,1\}$ by $0 \longrightarrow \begin{smallmatrix}0&0\\0&0\end{smallmatrix}$ and $1 \longrightarrow \begin{smallmatrix}1&0\\1&1\end{smallmatrix}$. Starting from 1, we obtain:

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

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. 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 must be iterated on an initial lattice where the position $(0,j)$ are filled with `1` and positions $(i,0)$ are filled with `0` for $i > 0$.

However, this transformation uses arithmetic operators (the `+` and `mod`). A more elementary computation is possible, turning the formula modulo 2 into a tiling process. Following [11] we consider 4 tiles corresponding to the two boolean

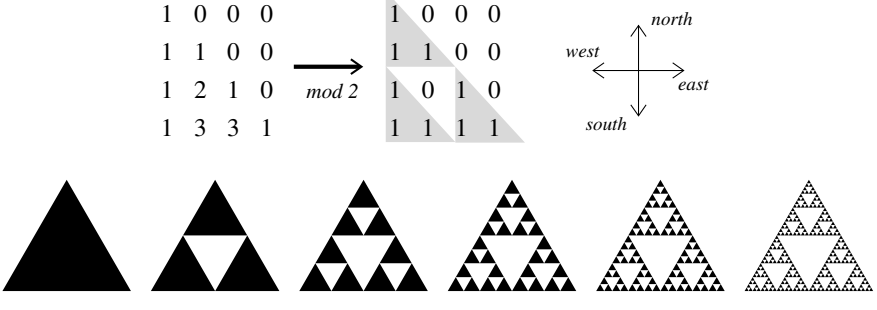


**Fig. 4.** *Upper line:* taking the binomial coefficients modulo 2 produces the shape of the ST. *Lower line:* ST can also be produced by iterating the carving of a triangle inside another triangle.

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 4 symbols `‘T00`, `‘T10`, `‘T01` and `‘T11` to represents the 4 types of tiles: tile `‘Txy` 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 ST2 = {
    <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.

## 5 Carving Sierpinski Triangles

Building a ST by carving is illustrated in figure 4. This process is also easily coded in MGS using *patch patterns* on *abstract cellular complexes*.

An abstract cellular complex is composed of elements of various dimensions (vertices, edges, surfaces, ...) called *topological cells* of dimension $n$ or $n$-cells [10]. These basic elements are organized following the *incidence relationship* that relies on the notion of boundary: let $c_1$ and $c_2$ be respectively a $n_1$-cell and an $n_2$-cell with $n_1 < n_2$, $c_1$ is incident to $c_2$ if $c_1$ belongs to the border of $c_2$. More especially, if $n_1 = n_2 - 1$, $c_1$ is called a *face* of $c_2$, and $c_2$ is a *coface* of $c_1$.

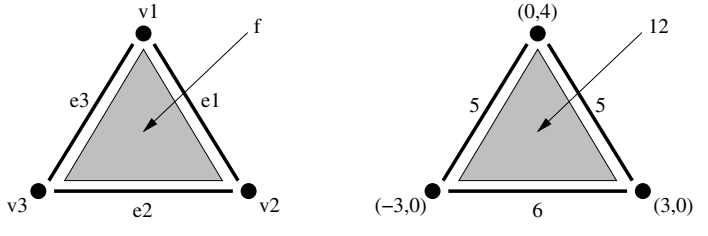


**Fig. 5.** On the left is an example of a cellular complex: it is composed of 3 0-cells ($v_1$, $v_2$, $v_3$), 3 1-cells ($e_1$, $e_2$, $e_3$), and a 2-cell $f$. The boundary of $f$ is formed by its incident cells $v_1$, $v_2$, $v_3$, $e_1$, $e_2$ and $e_3$. Especially, the 3 edges are the faces of $f$, and therefore, $f$ is the coface of $e_1$, $e_2$ and $e_3$. On the right, data are associated with the topological cells: positions are associated with vertices, lengths with edges and area with $f$.

This data structure generalizes the idea of graph, that is a complex composed of 0-cells and 1-cells. As the definition of a GBF collection uses the elements of a mathematical group as indexes, here $n$-cells are used as indexes to define a cellular complex based topological collection. Basically, a value is associated with each topological cell. This corresponds to the concept of *topological chain* in algebraic topology. This notion won't be detailled in the paper. An example of such a collection is given on figure 5.

*Patch transformations* have been created to handle any arbitrary cellular subcomplex. The main advantage of using these 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".

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.

There are two transformations used to carve the ST. The first one, `AV`, adds a vertex in the middle of each edges (see figure 6):

```
patch AV = {
  ~v1 < e:[dim = 1] > ~v2
  => ‘v:[dim = 0, cofaces = (‘e1,‘e2),
         val = { x=(v1.x+v2.x)/2, y=(v1.y+v2.y)/2, new=true }]
     ‘e1:[dim = 1, faces = (v1,‘v)]
     ‘e2:[dim = 1, faces = (v2,‘v)]
}
```

The keyword `patch` is used instead of the keyword `trans` to outline that the defined transformation uses patch patterns in its rules. In this patch transformation, `v1` and `v2` are not consumed (the `~` qualifier in front of an identifier) to allow the matching of all the edges incident to a same vertex. Indeed, if an element is matched by a pattern, it can't be matched in another one: two subcollections 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*. Here, if a vertex was matched and consumed together with one of its incident edges, no any other incident edges could be matched by the rule. A clause `c1 < c2` means that cell `c1` is incident to cell `c2` and of lower dimension. The right hand side of the rule is a special form used to transform the matched edge `e` into two edges `‘e1` and `‘e2` incident to a new vertex `‘v`. A flag `new` distinguishes the newly created vertices.

The next step looks for all the hexagons and replaces them with three triangles (see figure 6):

```
patch RF = {
  f:[dim=2, faces = (e1,e2,e3,e4,e5,e6)]
  ~v1 < ~e1 > ~v2:[? v2.new] < ~e2 >
  ~v3 < ~e3 > ~v4:[? v4.new] < ~e4 >
  ~v5 < ~e5 > ~v6:[? v6.new] < ~e6 > ~v1
```

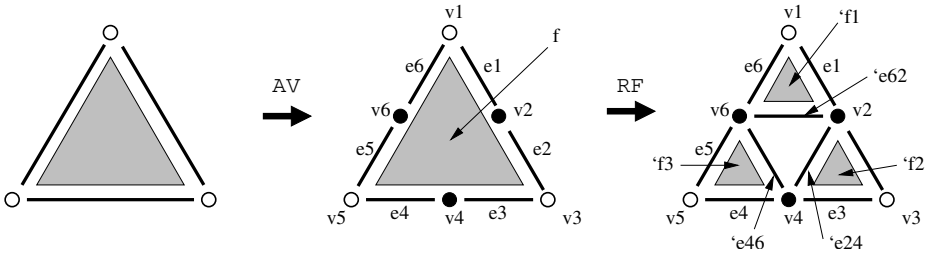


**Fig. 6.** Carving a triangle. The first transformation `AV` adds vertex in the middle of an every edge. The second transformation `RV` refines the central hexagonal face into three triangles.

```
    => '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)]
}
```

In this patch, only the hexagon `f` is matched and consumed. We select its boundary without consuming it. Note the guards in the specification of the matched vertices: a flag is used to match only newly created hexagons.

## 6 Discussion and Conclusion

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.

Most of the examples described in this paper relie on chemical processes. The sierpinsky gasket pattern has been really implemented using DNA molecules. Previously, the process has been designed and simulated using the *kinetic Tile Assembling Model* (kTAM) [11]. kTAM provides a complete framework for the description of such chemical reactions where a lot of physical parameters (like temperature, error rates, ...) are taken into account to allow accurate studies of crystallization processes. The DNA assembly of tridimensional fractal has been proposed and studied in [2], based on DNA trigonal tiles. Compared to this work, the MGS modelings presented in this work are much more abstract: the purpose is not to study the physical implementation using a DNA computing paradigm but to investigate the shape produced by some families of abstract self-assembly processes.

Obviously, the mechanisms provided by MGS allow the specifications of more complex and abstract operations, that could be very difficult to implement using polymerization and depolymerization reactions of kTAM for instance. These higher level features can be used in the domain of robotics self-assembly. For instance, [17] presents the elaboration of a self-reproducing machine. This

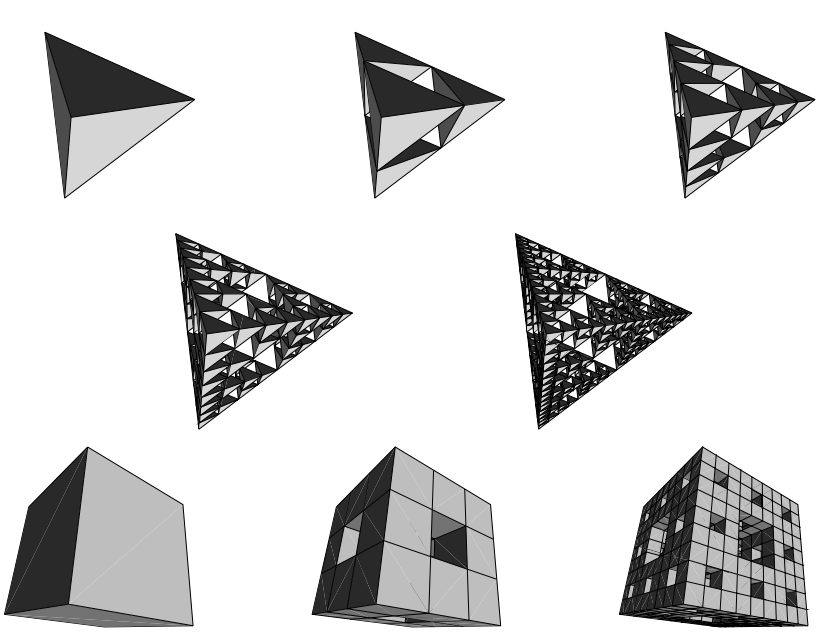

**Fig. 7.** On top, Sierpinski sponge building process: initial state and steps 1, 2, 3 and 4. At bottom, Menger sponge building process: initial state and steps 1 and 2.

machine is composed of elementary cubic modules. Each module is able to behave in different ways: pivoting, connecting or disconnecting with other modules, transfering data and power to its connected neighbors. The organization and the complex behaviors of the whole machine could be captured by a MGS modeling using topological collections and transformations. The modeling in MGS of such complex self-assembly processes, where we must specify the complex interaction of a few complex entities, is a part of our current work.

We insist on the expressivity brought by the notions of topological collections and their transformations. For example, the patch language used in section 5 is powerful enough to produce Sierpinski and Menger sponge (a generalization of carving a tetrahedron and a cube in 3D), see figure 7. MGS has also been succesfully used to model several biological growth processes, like the development of an epithelial sheet or a neurulation process [12], as well as the flock of birds or the subdivision of a triangulated surface.

## References

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

[2] A. Carbone, C. Mao, P. E. Constantinou, B. Ding, J. Kopatsch, W. B. Sherman, and N. C. Seeman. 3D fractal DNA assembly from coding, geometry and protection. *Natural Computing*, 3(3):235–252, 2004.

[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] 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`.
[8] 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.
[9] 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.
[10] J. Munkres. *Elements of Algebraic Topology*. Addison-Wesley, 1984.
[11] 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.
[12] A. Spicher and O. Michel. Declarative modeling of a neurulation-like process. In *Sixth International Workshop on Information Processing in Cells and Tissues (IPCAT'05)*, pages 304–317, York, August 2005.
[13] 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.
[14] I. Stewart. Four encounters with sierpinski's gasket. *Mathematical Intelligencer*, 17:52–64, 1995.
[15] T. A. Witten and L. M. Sander. Diffusion-limited aggregation, a kinetic critical phenomenon. *Phys. Rev. Lett.*, 47:1400–1403, 1981.
[16] S. Wolfram. *A new kind of science*. Wolfram Media, 2002.
[17] V. Zykov, E. Mytilinaios, B. Adams, and H. Lipson. Self-reproducing machines. *Nature*, 435(7038):163–164, 2005.