Designing a topological modeler kernel: a rule-based approach

Thomas Bellet, Mathieu Poudret, Agnès Arnould, Laurent Fuchs
XLIM-SIC, UMR CNRS 6172
University of Poitiers
Poitiers, France
Email: first_name.name@univ-poitiers.fr

Pascale Le Gall
MAS
Ecole Centrale Paris
Paris, France
Email: pascale.legall@ecp.fr

***Abstract*— In this article, we present a rule-based language dedicated to topological operations and based on graph transformations. Generalized maps are described as a particular class of graphs determined by consistency constraints. Hence, topological operations over generalized maps can be specified using graph transformations. The rules we define are provided with syntactic criteria which ensure that graphs computed by applying rules on generalized maps are also generalized maps. We have developed a static analyzer of transformation rules which checks the syntactic criteria in order to ensure the preservation of generalized map consistency constraints. Based on this static analyzer, we have designed a rule-based prototype of a kernel of a topology-based modeler that is generic in dimension. Since adding a new topological operation can be reduced to write a graph transformation rule, we directly obtain an extensible prototype where handled topological objects satisfy built-in consistency. Moreover, first benchmarks show that our prototype is reasonably efficient compared to a reference implementation of 3D generalized maps which use a classical implementation style.**

***Keywords*- topology-based geometric modeling; graph transformation; topological consistency; rapid prototyping**

I. Introduction

This paper is a first investigation about the use of graph transformation [4] in the scope of topology-based geometric modeling. Indeed, previously, rule-based languages have been used in a restricted way to systematically apply predefined operations on predefined objects, like in L-system languages dedicated to plant growing [21]. We propose a rule-based dedicated language for user-friendly describing operations of topological models. This language includes some specialized variables to make easier the handling of some operations such as triangulation, cone or extrusion operations. Assuming that our chosen topological structure is the generalized maps (or G-maps for short) [16], the topological modifications are modeled as graph transformation rules. The main interest of rules stays in the fact that it becomes possible to deal with them in a systematic way: a program dedicated to rule application allows us to consider all operations in a consistent and generic manner. We have developed a prototype of a topology-based modeler as a specialized rule application engine. Our modeler presents the advantage of being generic with respect to the dimension of the considered objects. Within our framework, it is particularly easy to extend a rule-based modeler with new topological operations. Actually, for that purpose, since the design of a new rule directly provides its implementation, no additional programming step is required. The consistency of the transformations as well as their implementation is fully automated. This feature considerably improves the design of a consistent topological modeler. Indeed, in [19], we have expressed the preservation of the G-map consistency constraints by direct syntactic criteria on rules which can be statically checked. Due to these syntactic criteria, every new operation is consistent from a topological point of view. This means that a rule applied to a generalized map always produces a generalized map. We illustrate our prototype through examples demonstrating the modeling capabilities of our rule-based approach. We compare it on some data sets with another topological software, Moka [22].

This paper is organized as follows: we present graph transformations and generalized maps in section II. In section III, we present our language for G-map transformation rules. In section IV, we give some syntactic criteria on our transformation rules which ensure the preservation of G-map consistency constraints. In section V, we briefly present the main design points underlying our rule-based modeler kernel. In section VI, we give some preliminary and encouraging results of our prototype.

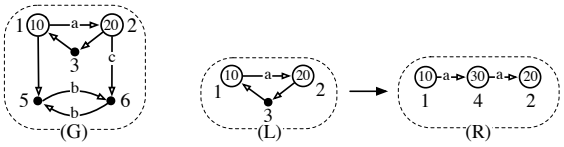
II. Context

In this section, we briefly introduce the key definitions of both graph transformation rules [4], [10] and generalized maps [16], [17].

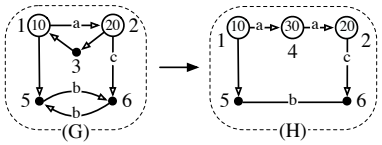
A. Graph transformation rules

As classical rewriting of terms [1], rewriting of graphs allows one to match a pattern in a graph and then to transform it into a new pattern. Several approaches of graph transformation exist. We give below a short and intuitive introduction. The interested reader can find a complete presentation in [4], [10].

We call *graph* a classical directed graph with possibly labeled nodes and arcs. For example, in Fig. 1(a), the graph G has five nodes respectively named $1,2,3,5$ and 6. The node 1 and 2 are respectively labeled by 10 and 20, they are depicted by a circle decorated with a label. The unlabeled nodes, like nodes $3,5$ and 6, are drawn with dots. Arcs are drawn with arrows possibly labeled (by a,b or c in our example). Moreover, two inversely-directed arcs, like the two arcs between nodes 6 and 5 (both labeled by b), can be drawn with a simple line (see graph H of Fig. 1(c)). Moreover, most of the time, arc names are omitted since in practice, this simplification does not raise ambiguity.



(a) A graph G (b) A rule r



(c) An application of r on G

Figure 1. Partially labeled graph and rule

A *graph transformation rule* $r : L \to R$ is defined by two graphs: L is the left-hand side and R is the right-hand side. See the rule r on Fig. 1(b) for example. The left-hand side L represents the pattern which is matched, the right-hand side R represents the pattern which replaces the pattern L.

To be workable, such a graph transformation rule should be provided with some mechanisms which explain how the rule is applied on a graph G. The first step consists in finding an instance of L in G. Rigorously, this means computing a *pattern matching* $m : L \to R$. Formally, m is a morphism[1] of graphs that preserves structure and labeling. In our examples, the matching is given by the identity of node names. For instance, in Fig. 1, the node 1 of L is matched with the node 1 of G. Let us note H the resulting graph obtained by applying a rule $r : L \to R$ on G, with the match $m : L \to G$. The graph H is computed by removing the subgraph $m(L)$ in G and then by adding $m(R)$. The construction of H is usually denoted by $G \Rightarrow_{r,m} H$.

Nevertheless, to apply the rule r on a graph G, the existence of a match $m : L \to G$ is not sufficient. The so-called *dangling condition* should also be satisfied. This condition means that no removed node of $m(L) \backslash m(R)$ is linked to an unmatched node of $G \backslash m(L)$. For example, in Fig. 2, the matched node 3 (it belongs to $m(L)$) is linked to the unmatched node 5 (it does not belong to $m(L)$). So, the removing of $m(L)$ in G creates a dangling arc[2] and the resulting structure H is not a graph. When the dangling condition is fulfilled, such situation is prevented and the resulting structure H is a graph (see Fig. 1(c)).

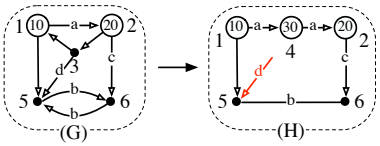


Figure 2. Dangling condition

As for classical term rewriting, to make graph transformation rules general, they may contain some variables [11]. For example, the notion of attributed variables has been introduced to deal with node labels and arc labels. In Fig. 3, the variables x and y of the left-hand side can match any labels and the expression $x+y$ of the right-hand side should be evaluated to define the label of the new node 4. To apply this rule with variables on the graph G (see Fig. 1(a)), we must first instantiate variables occurring in the rule (i.e. the variables x and y) with appropriate values (here, resp. with the values 10 and 20) in order to compute a classical rule r without variables (see Fig. 1(b)) and then apply r (see Fig. 1(c)).

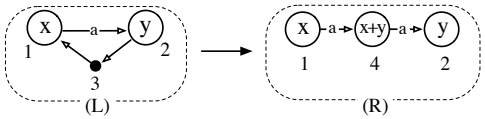


Figure 3. A rule with attributed variables

Other variables have been introduced to capture generic graph patterns to be transformed. For example, graph variables are special nodes which can be substituted by any graph provided that arcs connecting this special node are preserved in the transformation. However, we will not use directly the variables as they are defined in [11]. Indeed, to our point of view, their use is not specialized enough with regard to our needs, but [11] inspired us to introduce a new kind of variables dedicated to the modeling of topological transformations (see Section III).

B. Generalized maps

In this section, the basic definition of generalised maps is introduced. But first, as generalised maps are not so commonly employed, we discuss some motivations of their use.

In solid modeling, B-rep modelers are based on topological data structures as winged-edge, half-edge [18] or quad-edge [9] to represent the cell subdivision of geometric objects. Some data are attached to the cells in order to define the shape of the objects. These data are usually referenced as the embedding of the object.

[1] $m : L \to G$ is a morphism if for any l-labeled node v in L, $m(v)$ is an l-labeled node, and for any l-labeled arc e of L with a source node u and target node v, $m(e)$ is an l-labeled arc in G with the source node $m(u)$ and the target node $m(v)$.

[2] A dangling arc is an arc which has no source node or no target node.

It has been shown that all these data structures can be formalized and modeled by combinatorial maps or generalized maps (or G-maps, for short) [17] depending on whether they handle orientable or non orientable objects. For example, detailed conversion between the half-edge data structure and 2-maps can be found in [15].

The first main advantage of dealing with maps is the homogeneity in the handling of dimensions: subdivisions of any dimension can be manipulated in the same manner. This allows to develop efficient general algorithms. For example, the connectivity compression algorithm developed in [20] can be used to transmit large models such as volumic urban environments. For a given dimension, statistical study of the cells of 3D-manifolds [7] helps in the complexity analysis of data structures and algorithms handling these manifolds.

The second advantage is the consistency constraints: they express relations between elementary components (the darts) that compose a combinatorial map or a G-map. They also give the conditions to obtain a consistent object. These conditions have to be maintained and checked when operations are performed. For example, these consistency constraints can help to to build well-formed building interior models to simulate heat transfer or radio-wave propagation [12].

From a practical point of view, all the basic topological operations, such as Euler operators can be expressed using dart sewing and unsewing that are the atomic operations over G-maps. Moreover, high level operations such as triangulation, extrusion, rounding, etc. are generalized in dimension.

Even if combinatorial maps have demonstrated their usefulness to handle complex geometric objects such as multiresolution subdivision surfaces [15]. Different studies outside the solid modeling field have also been conducted using (combinatorial or generalized) maps. They usually turn to be very helpful to structure knowledge in fields such as image processing [5], [14], [3] or in applications such indoor wave propagation in very large building [6]. Moreover, the formalization of topological structures provided by combinatorial maps can also be useful to proof non trivial theorems [2].

However, a common need is to develop an adapted modeler kernel and then to plug high level specialized modeling operations over it. Most of the time, the modeler kernel is developed directly in a programming language and then the verification of the topological consistency becomes an hard work.

The topological structure of an object consists in both its decomposition into topological cells (vertices, edges, faces, volumes, etc.) and the neighborhood relations between these cells. The decomposition of a 2D object is shown on Fig. 4.

From this example of subdivision the notion of generalized maps can be intuitively introduced. The object represented on Fig. 4(a) is composed by two faces F_1 and F_2 *glued* along their common edge c. This edge can be split into two new edges (cf. Fig. 4(b)), in order

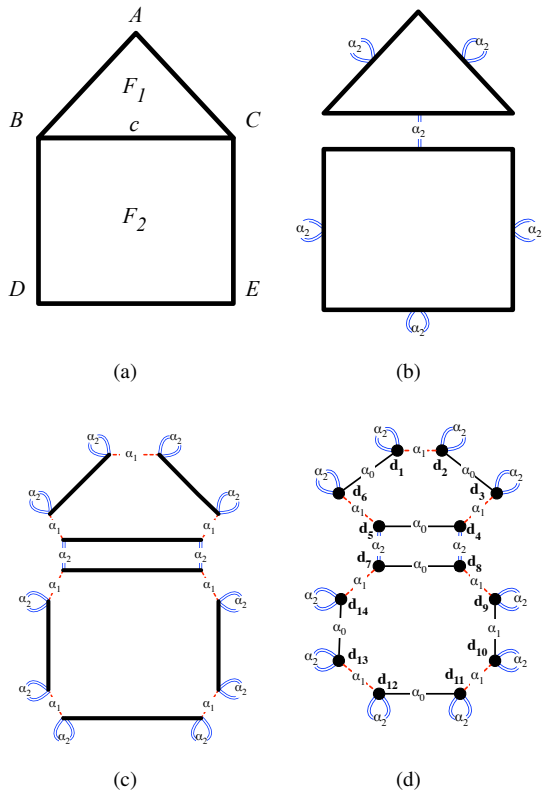


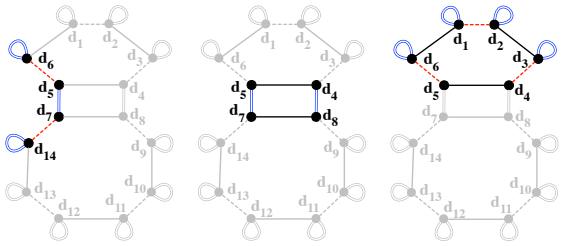
Figure 4. Cell decomposition of an object

to dissociate F_1 and F_2. These new edges correspond respectively to edge c *seen* from face F_1 and to edge c *seen* from face F_2. A 2-*dimensional relation* α_2 is added between these new edges, in order to remember that they initially correspond to a single one. This process is applied to all other edges, but no new edge is created since all edges belong to the boundary of the surface: in order to formalize this fact, each edge is linked with itself by the 2-dimensional relation (cf. Fig 4(b)). The boundary of each face F_1 or F_2 is a 1-dimensional quasi-manifold : the same process can be applied, and each vertex is split into two distinct vertices, linked by a 1-dimensional relation α_1 (cf. Fig 4(c)). The 2-dimensional relation is now defined upon these new basic elements (cf. Fig 4(c)). The boundary of each edge is now defined by two distinct vertices, linked by a 0-dimensional relation α_0 meaning that they correspond to a single edge (cf. Fig 4(d)). Split vertices obtained at the end of the process are the basic elements of the model, and they are called *darts* in the combinatorial map terminology.

Figure 5. α_i graphical codes

As the index i of the α_i links gives the the dimension of the considered neighborhood relation, the α_3 link gives volume neighborhood relation. Notice that in all figures given in the sequel, we will use the α_i graphical codes of Fig. 5 in order to be more readable.

Unlike most data structures used in solid modeling, topological cells are not explicitly manipulated in G-maps but only implicitly defined. They can be computed using traversal of nodes by the means of given neighborhood arcs. For example, the d_5 incident 0-cell (Fig. 6(a)) is the subgraph which contains d_5, nodes reachable from d_5 using arcs α_1 and α_2 (nodes d_5, d_6, d_7 and d_{14}) and the arcs themselves. This subgraph is denoted by $<\alpha_1\alpha_2>(d_5)$ and models the vertex B of Fig. 6(a). In Fig. 6(b), the d_5 incident 1-cell is the subgraph $<\alpha_0\alpha_2>(d_5)$ containing nodes d_5, d_4, d_7 and d_8, and adjacent α_0 and α_2 arcs. It represents the topological edge c. Finally, in Fig. 6(c), the d_5 incident 2-cell is the subgraph $<\alpha_0\alpha_1>(d_5)$ and represents the face F_1. More generally, an *orbit* $<\alpha_i \dots \alpha_j>(d)$ is the subgraph which contains d, all nodes reachable from d using α_i to α_j-labeled arcs, and all of these arcs.



(a) Vertex (b) Edge (c) Face

Figure 6. Reconstruction of cells adjacent to d_5

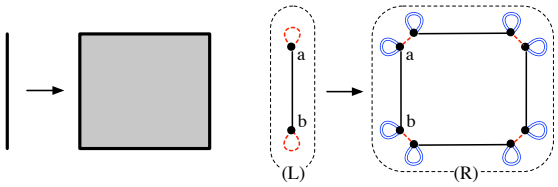
Usually, generalized maps are defined algebraically [16], but they can also be defined as graphs whose nodes are the darts and graph arcs are the neighborhood relations α_0 to α_n. Then the topological consistency constraints are expressed by three graph constraints. The following definition gives them.

Definition 1 (generalized map):
A n-dimensional G-map ($n \geq 0$) is defined as a graph in which arcs are labeled in $\{\alpha_0, \dots, \alpha_n\}$ and such that:
Non-orientation constraint: G is non-oriented.
Adjacent arcs constraint: each node is the source node of exactly $n+1$ arcs respectively labeled by α_0 to α_n.
Cycles constraint: for every α_i and α_j verifying $0 \leq i \leq i+2 \leq j \leq n$, there exists a cycle labeled by $\alpha_i\alpha_j\alpha_i\alpha_j$ starting from each node.

The cycle constraint ensures that in G-maps, two i-cells can only be adjacent along $(i-1)$-cells. For instance, in the 2-G-map of Fig. 4(d), the $\alpha_0\alpha_2\alpha_0\alpha_2$ cycle implies that faces are always stuck along topological edges. Let us notice that thanks to loops (see α_2-loops in Fig. 4(d)), these three constraints are preserved at the border of objects.

III. G-MAPS TRANSFORMATION RULES

We aim to use graph transformations (see section II) to define topological operations. For example, in Fig. 7, we define the 2D extrusion of a border edge (see Fig 7(a) for an intuitive definition of the operation) in a 2-G-map by means of a graph transformation rule (see Fig 7(b) for a rigourous definition in terms of a graph transformation).

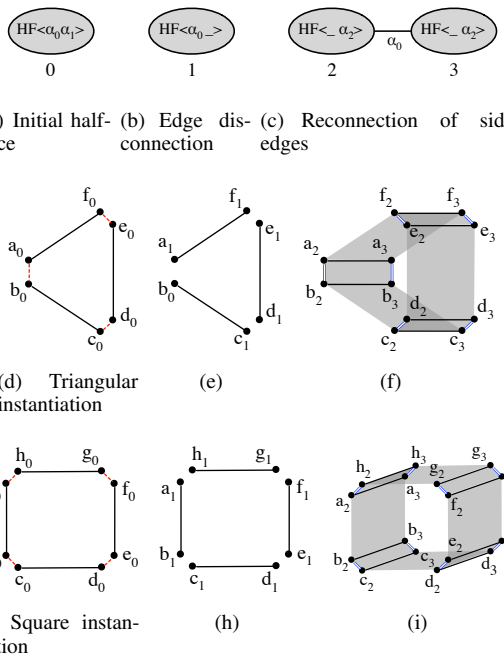


(a) Intuitive representation (b) Rule

Figure 7. 2D extrusion of an edge

In the same manner, we want to define the 3-dimensional extrusion of an half-face in a 3-G-map. Since there exist different kinds of half-faces, for instance triangular or square ones, we introduce a notion of variables (see section II) to write a generic rule which defines the extrusion of any half-face. By instantiating this variable, we are then able to specialize our generic rule for any particular half-face. We therefore define specialized variables which abstract any orbit of a generalized map.

An *orbit variable* X typed by $<\alpha_i \dots \alpha_j>$, noted $X<\alpha_i \dots \alpha_j>$, can be instantiated by any orbit $<\alpha_i \dots \alpha_j>(d)$. For example, an orbit variable $HF<\alpha_0\alpha_1>$ of the matched pattern Fig. 8(a) can be instantiated by the triangular half-face of Fig. 8(d), or by the square half-face of Fig. 8(g).



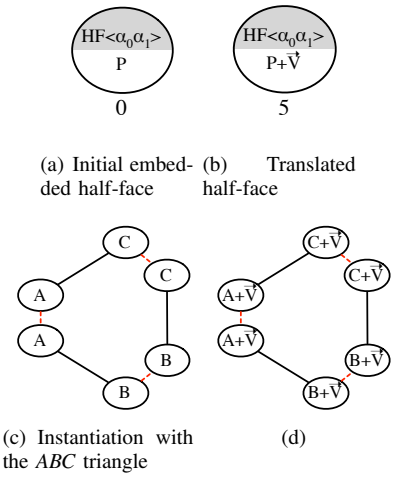
(a) Initial half-face (b) Edge disconnection (c) Reconnection of side edges

(d) Triangular instantiation (e) (f)

(g) Square instantiation (h) (i)

Figure 8. Orbit variable relabeling

Relabeling functions allow to modify orbits. First, we can remove some arcs of a given orbit. For example, by removing the α_1 arcs of an $<\alpha_0\alpha_1>$-typed half-face (for example, in the pattern Fig. 8(b)), we disconnect the border edges of the half-face (see Fig. 8(e) for the corresponding instantiated pattern with triangular half-face). We adopt the convention that the removing of α_1 arcs is denoted by the *removing label* "_" at the same position than α_1 in the initial pattern typed $<\alpha_0\alpha_1>$ associated to



(a) Initial embedded half-face (b) Translated half-face

(c) Instantiation with the *ABC* triangle (d)

Figure 9. Embedding with attributed variables

the node 0. Thus, by instanciating the variable HF with a triangular face, we get a full triangle in Fig. 8(d) and three disconnected edges in Fig. 8(e). In the same manner, by instanciating HF with a square face, we obtain the graphs of Fig. 8(g) and 8(h).

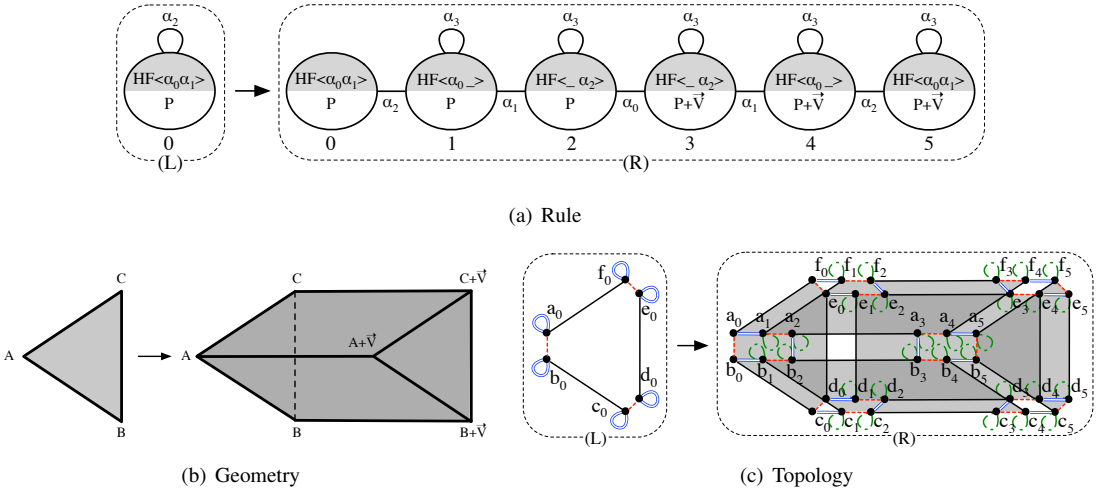
Instead of removing some arcs, we can relabel them. For example, from an half-face $HF<\alpha_0\alpha_1>$, we can compute the sides of a prism by using two copies of HF where α_0 arcs are removed and α_1 ones are relabeled to α_2. We note this modified half-face: $HF<\_\alpha_2>$. Again, the position of labels defines the modifications. Thus, the sides of a prism can be abstracted by the pattern Fig. 8(c). Instantiating the HF variable by a triangular half-face produces the three sides of the triangular-based prism of Fig. 8(f). Conversely, instantiating HF with a square half-face produces the four sides of the parallelepiped of Fig. 8(i).

At last, we use attributed variables (see section II) to handle the embedding within the patterns. By the means of attributed variables, we can match the geometry of an existing object or set a new geometry to a computed object. For example, in the pattern Fig. 9(a), the attributed variable P allows one to match the geometric points of the half-face HF. Moreover, in the pattern Fig. 9(b), the expression $P+\vec{V}$ allows to translate geometric points of an half-face. By instantiating the orbit variable HF of patterns Fig. 9(a) and Fig. 9(b) by a triangular face, we respectively obtain the graph of Fig. 9(c) (where A, B and C are three geometric points) and the translated graph of Fig. 9(d).

By mixing previous features (orbit variables, removing, relabeling and attributed variables), we can specify the 3D extrusion with the general rule of Fig. 10(a). The triangular instantiation of this general rule gives the geometry of Fig. 10(b) and the topology of Fig. 10(c). In this last figure, we can distinguish the 5 copies of the original half-face. The nodes of the original half-face are indexed by 0 (the name of the node in the general rule) while the nodes of ith copy are indexed by i (the name of their corresponding nodes in the general rule). In the left-hand side, the original node 0 is the matched half-face, that is to say the one we want to extrude. Let us notice that in the right-hand side, this half-face stays unmodified, it represents the base of the computed prism. The node 5 is a new half-face translated from the original one and represents the other base of the prism constructed by the rule. The four intermediate nodes, and the arcs which connect them to each other, are the half-faces that form the sides of the extruded prism.

Our syntax allows one to specify a large class of operations on polyhedral objects. For instance, in addition to extrusion, the rules of the sewing of two volumes, the topological triangulation, the cone operation and the rounding of a vertex are given in the annex section VIII.

From our point of view, our rule-based language is expressive enough to cope with operations on polyhedral objects whose definition only depends on the topological structure of the manipulated objects. In particular, our language does not allow us to specify operations whose definition depends on some geometric conditions on the initial objects under modifications. Thus, we cannot at the moment define the Boolean operations for which it is well-known that the resulting objects are closely related to initial geometric conditions.



(a) Rule

(b) Geometry (c) Topology

Figure 10. 3D extrusion of a face

IV. G-MAP TRANSFORMATION RULES CONSTRAINTS

Usually, the definition of a new operation implies the verification of its mathematical definition. It consists in proving that the consistency constraints of the G-maps are preserved by the operation. This proof is generally done manually for each operation. With our rule-based framework, we can provide a solution that applies to all operations defined by rules. Indeed, we intend to provide our rules with syntactic criteria ensuring the preservation of G-maps consistency constraints by rule application. These criteria are shortly presented here and their detailed definition can be found in [19]. A correct rule should satisfy the three following criteria, directly corresponding to the G-maps consistency constraints given in section II:

- *Non-orientation criterion*;
- *Adjacent arcs criterion*;
- *Cycles criterion*.

A. Non-orientation criterion

As G-maps are non-oriented graphs, the application of rules on a G-map must also produce a non-oriented graph. So, intuitively, the non-orientation criterion means that, in a rule $r : L \rightarrow R$, both L and R are non-oriented graphs (this condition remains the same whether the rule contains variables or not).

B. Adjacent arcs criterion

In n-G-maps, the adjacent arcs constraint means that each node has exactly $n+1$ adjacent arcs respectively labeled with α_0 to α_n. In order to preserve this constraint, let us first consider rules without variables, like the rule of Fig. 10(c). Such rules (without variables) have to satisfy the following properties:

- Adjacent arcs of preserved nodes (nodes that belong to both sides) have the same labels on both the left-hand side and right-hand side. For example, in Fig. 10(c), a_0 has three adjacent arcs labeled with α_0, α_1 and α_2 in the left-hand and right-hand sides. Nevertheless, these arcs are not necessarily connected to the same nodes in both sides.
- The removed nodes (nodes that belong to only left-hand side) and the added nodes (nodes that belong to only right-hand side) must have exactly $n+1$ adjacent arcs respectively labeled with α_0 to α_n. For instance, in Fig. 10(c), both the nodes a_1, a_2, a_3, a_4 and a_5 have their four α_0, α_1, α_2 and α_3-labeled adjacent arcs.

In order to extend these properties on generic rules containing variables and relabeling functions, like the one in Fig. 10(a), arc labels and relabeling functions are considered in the same manner. Indeed, the relabeling functions associated to a given node can be seen as implicitly defining adjacent arcs of this node. For example, by taking into account these implicit arcs, the node 0 of the rule of Fig. 10(a) satisfies the first property: in both side, the node 0 has two implicit α_0 and α_1-labeled adjacent arcs and an explicit α_2-labeled adjacent arc. Let us notice that explicit arcs in one side can be implicit in the other side. In the same manner, the added nodes 1, 2, 3, 4 and 5 satisfy the second property: they have their four α_0, α_1, α_2 and α_3-labeled adjacent arcs, implicitly or explicitly.

C. Cycles criterion

Intuitively, this syntactic criterion guarantees the cycle constraint of n-G-map: for $0 \leq i \leq i+2 \leq j \leq n$ every node belongs to an $\alpha_i\alpha_j\alpha_i\alpha_j$-labeled cycle. The criterion is defined by case studies of the status (added, removed or preserved) of nodes and on the way (implicit, explicit) labels are managed on the rule patterns:

- An added node must be added with all the required cycles. For example, the nodes 1, 2, 3, 4 and 5 of the rule of Fig. 10(a), belong to $\alpha_0\alpha_2\alpha_0\alpha_2$ and $\alpha_1\alpha_3\alpha_1\alpha_3$-labeled cycles. Both cycles can be derived from the given of implicit and explicit labels attached respectively to the labeling function or to the adjacent arcs.
- If a preserved node belongs to a $\alpha_i\alpha_j\alpha_i\alpha_j$-labeled cycle in the left-hand side of the rule, it must belong to an $\alpha_i\alpha_j\alpha_i\alpha_j$-labeled cycle in the right-hand side too. For example, the node 0 of the rule of Fig. 10(a), belongs to an $\alpha_0\alpha_2\alpha_0\alpha_2$ cycle in both sides.
- If a preserved node belongs to an incomplete $\alpha_i\alpha_j\alpha_i\alpha_j$-labeled cycle (i.e. at least, one of the two labels α_i and α_j does not occur on an adjacent edge), then its α_i and α_j-labeled arcs are preserved. Indeed, as some of the cycle arcs are matched in the left-hand side while the other ones are not matched, modifying the matched arcs could lead to a breaking of the cycle. For example, in Fig. 10(a), the $\alpha_1\alpha_3\alpha_1\alpha_3$-labeled cycle of node 0 is partially matched in the left-hand side (only α_1 is matched). So, in order to avoid the breaking of the corresponding cycle, this α_1-labeled arc is required to be unchanged in the right-hand side.

As these criteria are syntactically expressed on our rules, their verification can be done both automatically and statically thanks to their genericity. Indeed, they are common to every G-map transformation rules. Thus, while in the classical approach, one has to verify the preservation of the G-map consistency constraints for each operation, in our framework, it suffices to provide our rules with a static analyser for the verification of the above syntactic criteria. So, a full rule-based modeler kernel will not only handle a rule application engine but also a function to check the rule syntax. Hence, a rule-based definition of operations provides an automatic and reliable implementation of operations that preserves constraint. Indeed, supposing that the operation under design can be defined as a rule, then its implementation is immediate and requires no special effort since its application is reduced to a simple application of the corresponding rule in the engine. Such an approach is analogous to the one of rule-based languages involving term rewriting, but has never been addressed in the context of topology-based geometric modeling. The design and the implementation of such engine is precisely the subject of the two next sections.

V. A RULE APPLICATION KERNEL

In this section, we first introduce the general structure of our rule-based modeler kernel. Then, we present our G-map and rule data structures and an efficient rule application algorithm. The chosen programming language is the functional language OCaml [13] well recognized for prototyping issues and for symbolic manipulations. Nevertheless, the presented data structures and algorithms can be directly translated into any programming language.

A. Architecture of our modeler kernel

Let us introduce the main lines of our implemented kernel. It regroups both our G-map data structure and our rule application engine into an OCaml library. This library is parameterized with the objects dimension (*i. e.* the G-map dimension) and the objects embedding (*i. e.* a data type). This library allows to load from a file different sets of rules according to the application needs. Indeed, the type of the rules both depends on the chosen G-map dimension and on embedding data type. Let us notice that as rules are stored in an external file, they can be modified without recompiling the specified kernel. So, there are as many modeler instances as there are possible initial settings for the map dimension and for the considered rule set. Moreover, thanks to the rule syntactic criteria, the rule design is guided. Actually, at the file loading, the criteria checking program indicate which are the mistakes in a rule.

Let us now describe some key elements of the implementation of G-maps, rules, and rule application.

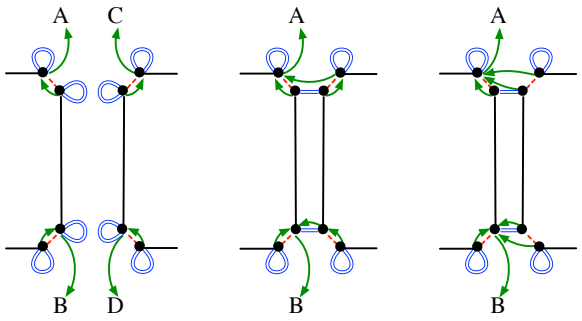
B. The G-map data structure

The G-map data structure consists in a data type associated to the following minimal set of functions:

- Creation and removal of a free node, *i. e.* a node which is not connected to another node;
- Orbit covering operation (consisting in enumerating nodes and edges of an orbit, and corresponding in practice to dedicated graph traversal algorithms) ;
- Vertex embedding setting. It defines the embedding for each node of a given vertex orbit.

It should be noticed that two classical G-map operations are not listed: the sewing and unsewing of two cells. Actually, as they can be implemented by rules, no specific function is needed.

We make our framework workable for objects of any dimension with any kind of embedding. Therefore, our G-map implementation is parametrized by both the G-map dimension and the vertex embedding data type. For example, we can have a 2-G-map embedded by *RGB* colours or a 3-G-map embedded with 3D points. In our implementation, the embedding is explicitly given for every node so no covering operation is needed to retrieve it since by construction the embedding is directly available from any node of the G-map. However, we optimize the embedding handling with a sharing strategy mainly inspired from the union-find forest [8]. Indeed, only one node per vertex orbit (we call it the carrier node) directly refers to the embedding. The other nodes only have an access to the embedding through a chain of indirections (see Fig. 11).



(a) Two face borders (b) After the sewing of the faces (c) After an embedding reading

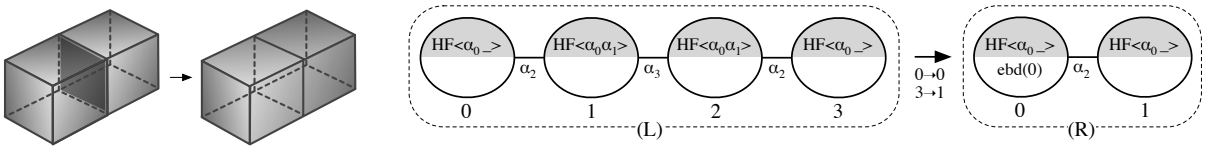
Figure 11. Chaining of the embedding

For example, on Fig. 11(a), four nodes directly refer the embeddings *A*, *B*, *C* and *D* while the others use indirections (represented by arrows on Fig 11). Hence, construction operations do not need any covering to update the embedding when vertices fuse. Let us take an example. When the two edges *AB* and *CD* are sewed (see Fig. 11), only the carrier nodes of *CD* are updated. Moreover, we take benefit of every embedding reading to reduce the length of indirection chains. For example, after an embedding reading on the structure of Fig. 11(b), the chains are advantageously reduced into the chains of length 1 of Fig. 11(c).

C. Rule data structure

In this section, we propose a data structure which implements our rules. First, we introduce some choices mostly motivated by the rule application algorithm described in section V-D.

As a first convention, the rule nodes are named using integer ranges starting from 0. Hence, we can use these node names as array indexes in our algorithm. Let us take an example. The face removal of Fig. 12(a) consists in identifying two half-faces of a same face and their respective neighbours. Then, the two half-faces are removed while their neighbours are connected together. Practically, in the left-hand side of the final rule on Fig. 12(b), nodes 1 and 2 identify the two half-faces while nodes 0 and 3 denote their neighbours. However, although the half-faces are removed by the rule, the nodes are named 0, 1, 2, 3 in the left-hand side and 0, 1 in the right-hand side. Let us notice that because of this naming convention, preserved nodes can have different names in the two sides of the rule. Thus, we have to record the update of node names. For example, in the removal rule, $3 \rightarrow 1$ denotes the fact that the node 3 is renamed into 1 while $0 \rightarrow 0$ denotes the fact that node 0 is not renamed.



(a) Face removal

(b) Final rule of face removal

Figure 12.

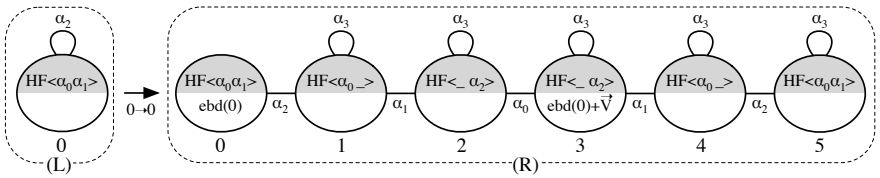


Figure 13. Concrete rule of face extrusion

The second implementation choice concerns the embedding computation. As introduced in section III, this computation is specified by expressions carrying by nodes of the right-hand side and corresponding to the new or updated values of node embedding. As our rules essentially capture topological operations, the pertinent embedding values of the initial G-map are recovered through the nodes of the rule left-hand side, while the embedding values of the transformed G-map are computed through computations from initial embedding data. More precisely, the expressions use names of left-hand side nodes to refer to knowledge about their embedding. Hence, no specific embedding variables are required. Thus, only nodes of the right-hand side can be labeled with expressions whose basic elements directly denote embedding of left-hand nodes. For example, the extrusion rule of Fig. 10(a) is translated into the concrete rule of Fig. 13 where the expression $x+\vec{V}$ becomes $ebd(0)+\vec{V}$. Moreover, the multiple embedding notations of the original rule have disappeared. Indeed, in our concrete rules, it suffices to explicitly give an embedding expression for only one node of each topological vertex (sharing by construction the same embedding). In the rule of Fig. 10(a), nodes 3, 4 and 5 belong to the same topological vertex. So, the expression $ebd(0)+\vec{V}$ defines the embedding of all of them.

At last, as we want to apply rules interactively, we need to point out where a rule has to be applied in a G-map. In practice, the two sides of a rule are not obtained in the same way. The right-hand side is literally constructed by the rule application while the left-hand side is recovered from the existing G-map[3].

For example, in Fig.14, we point out the face where we want to apply the face removal rule. We call this the instantiation of the abstract nodes of the left-hand side (the ones which handle an orbit variable) by concrete

[3]Furthermore, having two different node types in the left-hand side does not mean that edges have to be relabeled. In the same way, having a type with a label "_" does not mean that edges have to be removed. Practically, these two cases only specify the pattern where the rule can be apply.

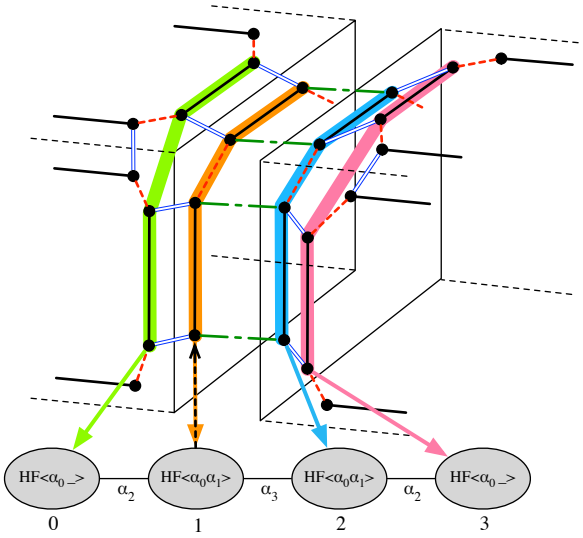


Figure 14. Instantiation of the removal rule left-hand side

orbits of the G-map. Intuitively, considering an abstract node, we associate it with one concrete node (a node of the G-map) and then retrieve the corresponding concrete orbit with a covering operation. For example in the removal rule of Fig. 14, we do not need to associate one concrete node to each abstract node. Actually, considering one abstract node per connected component is sufficient. In the example, the nodes 1 and 2 of the left-hand side are connected by α_3. Thus, the concrete orbits of both 1 and 2 can be retrieved with a single covering. Nodes that need to be associated to a concrete node for rule application are called *hooks*. In the removal example (see Fig. 14), the abstract node 1 is a hook and the instantiation of the four abstract nodes results from the concrete node associated to the abstract node 1. Let us notice that in order to retrieve the concrete orbits with a covering of the G-map, hooks must hold a complete type (a type without the label "_").

Here, we propose an OCaml rule data type. In Fig. 15, we give the source codes of the previous extrusion and removal rules. The first field, `left_hooks`, is the list of names of hooks. For the removal rule (see Fig. 12(b)) this list only contains the name 1. The field `left_nodes` is an array of the types (represented by OCaml lists) of the left-hand side nodes, such that its indexes are the nodes names. For the extrusion rule (see Fig. 13), it only contains the node 0 typed `[0;1]` (for $\alpha_0\alpha_1$). The particular removing label "_" is noted -1. The field `left_arcs` is the list of arcs of the left-hand side. Arcs are noted by a triple as *(source name, target name, label)*. For example, the α_2 arc between nodes 0 and 1 of the removal rule (see Fig. 12(b)) is noted `(0,1,2)`. The field `names_updating` is an array that defines how nodes are renamed from one side to the other (as explained in the beginning of this section). Its indexes are left-hand side names while its values are right-hand side names. For the removal rule, node 3 is renamed into 1. Thus, the third element of `names_updating` is 1. As well as arc labels, a removed node is renamed -1. For example, as node 1 of the removal rule is removed, the element stored at the index 1 is -1. The fields `right_nodes` and `right_arcs` are respectively the nodes types and the arcs of the right-hand side. At last, the field `right_ebd` is an array of embedding expressions of the right-hand side nodes.

```
let face_extrusion = {
  let_hooks = [0];
  left_nodes = [|[0;1]|];
  left_arcs = [(0,0,2)];
  names_updating = [|0|];
  right_nodes = [|[0;1];[0;-1];[-1;2];[-1;2];
    [0;-1];[0;1]|];
  right_arcs = [(0,1,2);(1,2,1);(2,3,0);(3,4,1);
    (4,5,2);(1,1,3);(2,2,3);(3,3,3);(4,4,3);(5,5,3)];
  right_ebd_fun = [ebd 0;none;none;(ebd 0)+V;
    none;none]}

let face_removal = {
  let_hooks = [1];
  left_nodes = [|[0;-1];[0;1];[0;1];[0;-1]|];
  left_arcs = [(0,1,2);(1,2,3);(2,3,2)];
  names_updating = [|0;-1;-1;1|];
  right_nodes = [|[0;-1];[0;-1]|];
  right_arcs = [(0,1,2)];
  right_ebd_fun = [ebd 0;none]}
```

Figure 15. OCaml type of rules

D. Rule application algorithm

This technical section briefly introduces our rule application algorithm. The algorithm input consists in associating one concrete G-map node to each hook. As explained in previous section, we then instantiate the abstract nodes into concrete orbits. As we have seen in section III, orbit variables allow ones to handle several relabeled copies of a same given orbit (the one abstracted by a hook). After the instantiation process, the algorithm needs to remember which concrete nodes are copies of a same node of this given orbit. For this purpose, we store the result of the instantiation process in a 2D array such that these copies are stored at the same line. In this array, columns model the abstract nodes.

In the second step an analogous 2D array is constructed for the right-hand side. For this purpose, the preserved columns (the abstract nodes) of left-hand side are reused and stored at their new array index according to the node names updating. Concerning abstract nodes added by the rule (the ones wich only appear in the right-hand side), additional columns are filled with names of concrete nodes newly added to the G-map. For now, in the G-map, these new concrete nodes are disconnected and not embedded. These new connections are computed in the two following steps.

In the third step, we create the arcs within the different copies according to the orbit variable relabelings. Thanks to our arrays (let us recall that copies of one concrete node are stored at the same line), the new arcs are created by considering only the concrete orbit of a hook. For example, let us consider an abstract node a which has a relabeling $\alpha_h \rightarrow \alpha_k$ and a concrete node d stored at line i of the a column. To compute the α_k-neighbour of d, we consider the hook column and look for the α_h-neighbour of the concrete node of line i. This way, we obtain a new line index j. At last, we get the α_k-neighbour of d by looking at the j line in the a column.

In the next step, we connect the orbit copies to each other according to the arcs of the right-hand side. Hence, for an arc (s,t,α_k) in the right-hand side, all the concrete nodes of the s column are α_k-linked, line by line, to the concrete nodes of the t column. Let us notice that for now, the connections of the preserved concrete nodes have not been modified.

The next step is the embedding computation according to the embedding expressions of the right-hand side. As embedding expressions refer existing embeddings (see section V-C), the transformations of these existing embeddings are performed at last. Let us notice that the number of embedding computations can be minimized with our previously described embedding sharing strategy (see section V-B): in practice, before the embedding is computed, new concrete nodes share a default embedding.

In the final step, we first remove the concrete nodes (and their adjacent arcs) associated to the abstract nodes which only belong to the left-hand side. Then, if it is required, we then relabel the preserved orbits. Finally, added concrete nodes are linked to preserved concrete nodes according to the arcs of the right-hand side.

VI. Results

A. Prototyping either a 2D modeler or a 3D modeler

In the previous section, we have proposed a kernel which implements our rule application engine. In this section, two case studies are presented. The first one illustrates the fact that modeler instances are parameterized by the objects dimension. For this purpose, we have studied the computation of 2D fractal objects. Here, the kernel is parametrized for 2-G-maps embedded by 2D points. Let us notice that as fractals are usually defined by rules, the design of fractals is straightforward in our framework. For instance, the Sierpinski's carpet illustrated

in Fig. 16 is easily computed. First, the external square is obtained by inserting a vertex, then by extruding it into an edge and finally by extruding this edge into a face. The carpet is then computed from the square by the means of four successive applications of a dedicated rule.

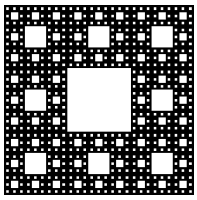
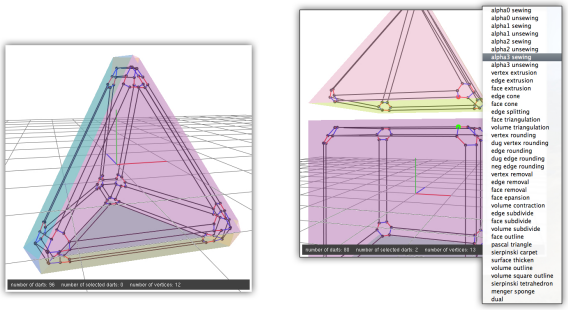


Figure 16. Sierpinski's carpet fractal computed with rules on a 2-G-map

The second case study of our framework is a prototype of a 3D extendable modeler. Here, our kernel is parametrized to be used with 3-G-maps embedded by 3D points. All the modeling operations are defined with our transformation rules. Let us recall that rules are not hardcoded in the program but loaded from an external file at the startup of the application. Thus this modeler is easily extendible. Any new operation can be added with a rule without recompiling the modeler.



(a) A rounded tetrahedron (b) $\alpha_3 sewing$

Figure 17. The 3D modeler prototype

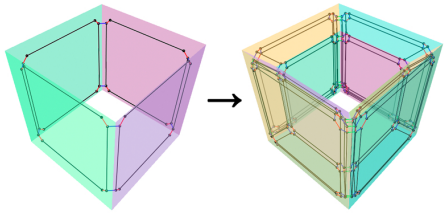
B. Prototyping operations

The modeler main window is an OpenGL 3D view of the embedded 3-G-map, see Fig. 17. All the loaded rules are listed in a menu. Choosing one rule in the list applies it to a list of selected nodes. The node selection is done with mouse picking allowing to indicate nodes which are associated to each hook of a rule. The order of the selection gives the position in the list of hooks.

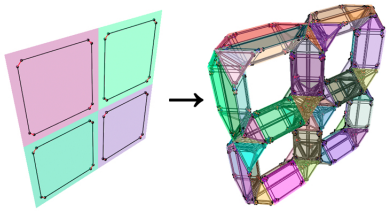
Our rules provide an easy and efficient way to design and simultaneously implement operations. We have designed all the usual topology-based operations with rules: sewing, unsewing, cone operations, extrusions, triangulations, roundings, subdivisions, removals, expansions and contractions. Moreover, some other simple operations have been prototyped. For example, on Fig. 18(a), an operation that thicks the surfaces of an object[4]. Another example is an operation that computes a volume outline of an object[5].

[4]This operation can be used to quickly design walls of a building.

[5]This operation can be used to quickly design a windows from a regular grid.



(a) Thicken of a surface



(b) Volume outline of an object

Figure 18. Easily prototyped operations

We also re-use our 2D fractal rule on 3-G-maps. These rules generate a lot of nodes and thus provide a relevant robustness test on real size objects. For example, by applying our Sierpinsky's carpet rule on the Standford Bunny, its number of nodes is multiplied by 8. This operation take[6] 0.41 s on a medium resolution model of 23106 nodes (see Fig. 19) and 1.97 s on an high resolution model of 97806 nodes.

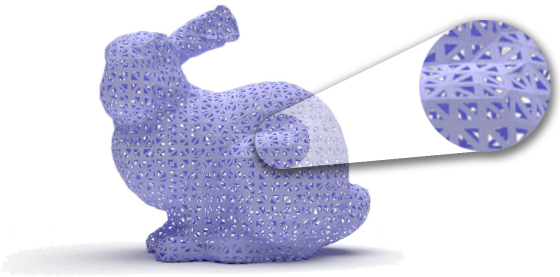


Figure 19. Sierpinski's carpet rules applied on the Stanford Bunny

We have also computed some 3D fractal objects with 3D dedicated rules analogous to the 2D fractal ones. For example, we computed the Menger sponge (which is the 3D extension of the Sierpinski's carpet) of Fig. 20. The corresponding rule is quite complex. In particular, its right-hand side is a grid of 20 interconnected nodes. Nevertheless, the constraints checking guides us for its design and only about forty minutes were required to design it. As the design of a new rule stands for both the definition of a new operation and its implementation, this design cost is low. Indeed, there is no implementation cost.

[6]For all presented computation times, we used a 2 GHz Intel Core 2 Duo MacBook with 2 GB RAM.

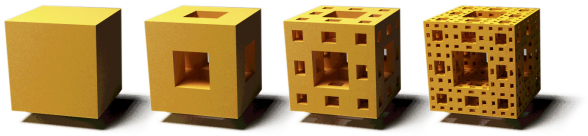


Figure 20. Menger sponge computed with rules on a 3-G-maps

C. Efficiency

Here, in order to validate our approach, we present some results obtained by our modeler kernel. These results are compared with the kernel of a topology-based modeler called Moka [22] that uses 3-G-maps. As Moka is a 3D dedicated application written in C++, its operations have been developed according to conventional methods and have been carefully optimized. From another side, our kernel is generic in dimension (i.e. parametrized by the G-map dimension) and the operations are not optimized because their implementation results from a unique function that applies all rules. Considering this comparison, it can be noticed that our important gains in development time have been obtained with an acceptable loss of performances. In the study below, computing times and length of operation programs are both compared.

Here, we first discuss results about computing times. The table I presents computing times of operations performed on objects of different sizes (*i.e.* number of nodes). Test objects have realistic sizes as they are made of several thousands of nodes. An analysis of the results shows that computation times obtained by our approach are of the same complexity order than Moka. For example, the volume triangulation have been tested with volumes made of both various numbers of faces and different face degrees (triangles or squares). Our results show that its computing times are in a constant ratio (approximately 3.5) with the operation implemented in Moka. In the same manner, the face triangulation has been tested with various face degrees and is also in a constant ratio (approximately 2.1) with Moka.

Table I
TIME EFFICIENCY COMPARISON

Operation	Number of nodes	Moka	Prot.	Ratio
Face triangulation	32768 → 98304	0.09 s	0.19 s	×2.11
	65536 → 196608	0.19 s	0.37 s	×1.95
	262144 → 786432	0.71 s	1.58 s	×2.23
Volume triangulation	12288 → 49152	0.03 s	0.10 s	×3.33
	49152 → 196608	0.13 s	0.48 s	×3.69
	196608 → 786432	0.60 s	2.18 s	×3.63
Dug edges and vertices rounding	3072 → 12288	0.08 s	0.05 s	×0.63
	24576 → 98304	0.72 s	0.46 s	×0.64
	196608 → 786432	5.79 s	3.96 s	×0.68
Full edges and vertices rounding	3072 → 12288	0.14 s	0.06 s	×0.42
	24576 → 98304	1.17 s	0.61 s	×0.52
	196608 → 786432	9.01 s	5.63 s	×0.62
α_3-sewing	65536	0.22 s	0.19 s	×0.86
	131072	0.43 s	0.39 s	×0.91
	262144	0.91 s	0.83 s	×0.91
α_3-unsewing	65536	0.22 s	0.19 s	×1.16
	131072	0.43 s	0.51 s	×1.19
	262144	0.86 s	1.00 s	×1.16

Comparisons have also been made for more complex operations like topological rounding of edges and vertices. For this last operation, objects containing various numbers of volumes were used. Its computation times are also in the same complexity as those obtained with Moka. Actually, computation times of our kernel are better. This can be explained by the geometric preconditions of the rounding operation which are computed in Moka. Future works have to extend the rules to take this geometric part into account.

Last considered operations are α_3-sewing and unsewing. Again, these operations have been tested with various face degrees. Like in the previous comparisons, the complexity is the same in both applications.

We notice that our sewing is more efficient than the Moka's one, while our unsewing is less efficient. This difference is explained by the different ways the applications deal with the embedding. In our kernel, every node refers to the embedding using the strategy previously described in section V-B. Conversely, in Moka, the embedding is referenced by only one node per vertex orbit. The sewing operation is faster in our kernel because we optimize the embedding sharing. Indeed, in the two applications, when two vertices are sewed, only one of the two carrier nodes need to have its embedding reference updated. Nevertheless, in our case, the carriers nodes are directly known through the references chain while they have to be found with a covering operation in Moka. As a dual consequence, the unsewing operation is faster in Moka. To unsew, vertex orbits must be split in two and the embeddings have to be duplicated. One of the two new vertex orbits keeps the original embedding reference while the other gets a new one. This implies to define this reference for all nodes in our kernel and only for the carrier node in Moka.

Let us notice that these computing times comparisons should be related to the program lengths (in terms of number of code lines). Our kernel is 700 OCaml lines long. This includes both data structures and covering operations of G-maps, data structures of rules, rule syntax checking and the 200 lines of rules application engine. In the loaded file that contains operation rules, one rule definition takes few lines. In Moka's topological kernel, each operation is 100 to 300 lines long and the whole kernel is about 30000 lines. Moreover, in a classical development approach, every single line must be tested and validated. Thus, applications like Moka usually take several years to be developed. In contrast, our prototype had been developed during approximately seven weeks. The evaluation of the amount of efforts required to develop software is not straightforward. However, with no doubt, our first estimations show that the rule approach provides a very convenient and quick way to develop a kernel modeler. Indeed, once the rule syntax is learned, the design of a new operation has a low cost.

VII. CONCLUSION

In this article we have proposed a rule-based language for specifying topological operations on n-G-maps. Based

on graph transformation, our rules include variables to generically denote orbits and relabeling functions which allow us to relabel orbits. These features are useful for most of the topological operations, for instance triangulation, cone, extrusion or even rounding operation. Our rules are defined according to a formal and graphical syntax which makes our specifications both clear, concise and easy to write. Moreover, we give syntactic criteria on rules which ensure that rules application preserves the topological consistency constraints of the G-maps.

We have designed a prototype which consists in a rule-based kernel of a topology-based modeler. Our tool can be seen as a rule-application engine dedicated to our G-map transformation rules. Thanks to syntactic criteria of rules, we ensure the topological consistency of designed G-maps. We have shown that the benefits of a rule-based approach are unquestionable. First of all, the efficiency of our prototype is comparable to other topology-based modelers based on G-maps. Moreover, operations are quickly designed and implemented and last but not least, the prototype is easily and safely extensible.

From a pure topological point of view, the scale of objects handled in our prototype is reasonably close to those manipulated in real world modelers. For now, the geometry is not fully integrated to our framework. For instance, while our syntax of rules allows one to match particular topological patterns as orbits, we cannot consider geometric conditions to restrict the scope of a transformation rule. In particular, this limitation does not allow us to specify the Boolean operations. Indeed, it is obvious that the computation of edge intersections mainly depends on their geometric positions. We plan to extend our rule-based language by incorporating such geometric conditions. We also plan to develop a graphical editor for our rules. Such a graphical interface will be useful to help designers in the writing process of new rules.

REFERENCES

[1] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. 1990.

[2] J.-F. Dufourd. An intuitionistic proof of a discrete form of the Jordan Curve Theorem formalized in coq with combinatorial hypermaps. *Journal of Automated Reasoning*, 43(1):19–52, 2009.

[3] A. Dupas and G. Damiand. Region merging with topological control. *Discrete Applied Mathematics (DAM)*, 157(16):3435–3446, August 2009.

[4] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2006.

[5] S. Fourey and L. Brun. A first step toward combinatorial pyramids in n-D spaces. In *Graph-based Representations in Pattern Recognition*, volume 5534 of *Lecture Notes in Computer Sciences*, pages 304–313, Venice, Italy, 2009.

[6] D. Fradin, D. Meneveaux, and P. Lienhardt. A hierarchical topology-based model for handling complex indoor scenes. *Computer Graphics Forum*, 25(2):149–162, June 2006.

[7] J. Françon and Y. Bertrand. Topological 3D-manifolds: a statistical study of the cells. *Theoretical Computer Science*, 234(1–2):233–254, 2000.

[8] B. A. Galler and M. J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7:301–303, 1964.

[9] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM Transactions on Graphics (TOG)*, 4(2):123, 1985.

[10] A. Habel and D. Plump. Relabelling in graph transformation. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *ICGT*, volume 2505 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 2002.

[11] B. Hoffmann. Graph transformation with variables. *Formal Methods in Software and System Modeling*, 3393:101–115, 2005.

[12] S. Horna, D. Meneveaux, G. Damiand, and B. Bertrand. Consistency constraints and 3D building reconstruction. *Computer Aided Design*, 41(1):13–27, January 2009.

[13] INRIA. The OCaml Language. http://www.ocaml.org.

[14] W. Kopatsch, Y. Haximusa, and P. Lienhardt. Hierarchies relating topology and geometry. In I. C. Henrik and H.-H. Nagel, editors, *Cognitive Vision Systems: Sampling the Spectrum of Approaches*, volume 3948 of *Lecture Notes in Computer Sciences*, pages 199–220, Dagstuhl, Germany, September 2006.

[15] P. Kraemer, D. Cazier, and D. Bechmann. Extension of half-edges for the representation of multiresolution subdivision surfaces. *The Visual Computer*, 25(2):149–163, 2009.

[16] P. Lienhardt. Subdivision of n-dimensional spaces and n-dimensional generalized maps. In *Annual Symposium on Computational Geometry SCG'89*, pages 228–236, Saarbruchen, Germany, Juin 1989. ACM Press.

[17] P. Lienhardt. Topological models for boundary representation: a comparison with n-dimensional generalized maps. *Computer-Aided Design*, 23(1):59–82, jan 1991.

[18] M. Mantyla. *Introduction to Solid Modeling*. W. H. Freeman & Co., New York, NY, USA, 1988.

[19] M. Poudret. *Transformations de graphes pour les opérations topologiques en modélisation géométrique, Application à l'étude de la dynamique de l'appareil de Golgi*. Thèse, Université d'Évry val d'Essonne, Programme Epigénomique, October 2009.

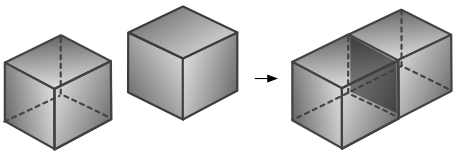
[20] S. Prat, P. Gioia, Y. Bertrand, and D. Meneveaux. Connectivity compression in arbitrary dimension. *The Visual Computer*, 21(8–10):876–885, 2005.

[21] O. Terraz, G. Guimberteau, S. Mérillou, D. Plemenos, and D. Ghazanfarpour. 3Gmap L-systems: an application to the modelling of wood. *The Visual Computer*, 25(2):165–180, 2009.

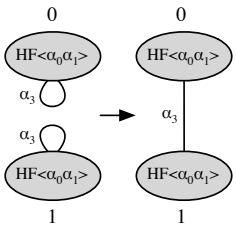
[22] F. Vidil and G. Damiand. Moka - a Topology-based 3D Geometric Modeler. http://www.sic.sp2mi.univ-poitiers.fr/moka/.

VIII. ANNEX

In order to convince the reader of the expressiveness of our rule-based language, we specify four classical topological operations by illustrating them with an intuitive representation and by giving the corresponding rules according the syntax previously explained.

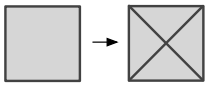


(a) Intuitive representation

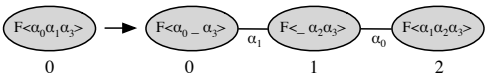


(b) Rule

Figure 21. Sewing two volumes along two isomorphic half-faces

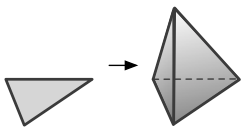


(a) Intuitive representation

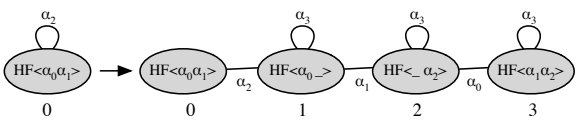


(b) Rule

Figure 22. Barycentric triangulation

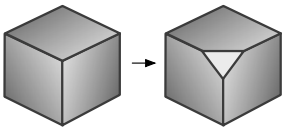


(a) Intuitive representation

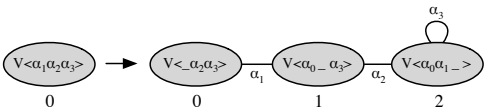


(b) Rule

Figure 23. Cone from a face



(a) Intuitive representation



(b) Rule

Figure 24. Rounding a vertex