Designing a topological modeler kernel: a rule-based approach

Thomas Bellet[1], Mathieu Poudret[1], Agnès Arnould[1] , Laurent Fuchs[1], Pascale Le Gall[2]
[1]XLIM-SIC, UMR CNRS 6172, University of Poitiers, France
[2]MAS, Ecole Centrale Paris, France

***Keywords*—topological base modeling; graph transformation; topological coherence ; rapid prototyping**

***Abstract*—In this article, we present a rule-based language dedicated to topological operations, 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 equipped with syntactic criteria ensuring that graphs obtained by applying the rules to generalized maps are also generalized maps. We have developed a static analyzer of transformation rules that checks the syntactic criteria 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. 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 have built-in consistency. Moreover, first benchmarks show that our prototype is reasonably efficient compared to a reference implementation of 3D generalized maps using classical implementation style.**

1. Introduction

Currently, most of the modelers are topology-based modelers. The geometric objects are implemented using their topological structure, *i. e.* their division into vertices, edges, faces, volumes, *etc*. Geometric components are associated to topological cells. For example, points may be associated to vertices. This introduction of geometric data is called the geometric embedding. Thus, operation definitions are split into a topological part, computed on the topological structure, and a geometric part, defined for each geometric component.

Most of the time, the operations are manually implemented with as many *ad hoc* programs as there are operations handled by the modeler. In previous works [10], [9], we have showed that topological operations of the generalized maps can be represented by graph transformation rules. The main interest of rules is 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.

The main contribution of this paper is to provide a first insight about how to implement a topology-based modeler by the means of a rule application engine. Thus, a topological operation will now be automatically implemented as a rule interpretation by such an engine. Moreover, in [8], we have proposed a manner to automatically verify topological consistency of the designed objects by checking syntactic criteria on rules. Consequently, our rule-based kernel can be easily extended with new topological operations. Due to these syntactic criteria, every new operations are consistent from a topological point of view. This means that a rule applied to a generalized map always produces a generalized map.

This paper is organized as follows: we present graph transformations and generalized maps (or G-maps, for short) in section 2. In section 3, we present our language for G-map transformation rules. In section 4, we give syntactic criteria on rules ensuring the preservation of topological consistency. In section 5, we briefly present the main design points underlying our rule-based modeler kernel. In section 6, we give some preliminary and encouraging results of our prototype.

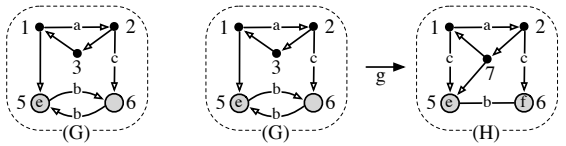
2. Context

In this section, we shortly introduce the key definitions of both graph transformation rules [2], [3] and generalized maps [6], [7].

2.1 Graph transformation rules

Let us introduce the basic notations concerning the so-called *double-pushout approach* for graph transformation rules. A partially labeled graph is a classical oriented graph with possibly labeled nodes and edges. For example, in Fig. 1(a), G has five nodes respectively named $1,2,3,5$ and 6. The unlabeled nodes are drawn with dots (like nodes $1,2$ and 3) or with an empty circle (like node 6). Edges are drawn using arrows possibly labeled (by a or b in our example). Most of the time, without ambiguity, edge names are omitted.

Moreover, two reverse edges $n \xrightarrow{l} n'$ and $n' \xrightarrow{l} n$ (like the two edges between nodes 6 and 7 both labeled by b) can be drawn with a single non-oriented link $n \overset{l}{\text{—}} n'$.



(a) A graph G (b) A graph morphism

Fig. 1. Partially labeled graph and morphism

(see graph H of Fig. 1(b)). More formally, a *partially labeled graph* G is a tuple $(V_G, E_G, s_G, t_G, l_{G,V}, l_{G,E})$ composed of two finite sets V_G and E_G of nodes and edges, two source and target functions $s_G, t_G : E_G \to V_G$ and two partial labeling functions $l_{G,V} : V_G \to Label_{G,V}$ and $l_{G,E} : E_G \to Label_{G,E}$ where $Label_{G,V}$ and $Label_{G,E}$ are two label sets respectively of nodes and of edges.

All constructions of category theory are based on *morphisms*. A morphism carries nodes and edges (with possibly some renamings) and labels (with no relabeling). For example, in Fig. 1(b), the morphism $g : G \to H$ renames the node 3 in the node 7 and preserves the other nodes. Morphisms can add nodes, edges (like the edge from 7 to 6) and labels (like the label f of the node 6 and the label c of edge from 1 to 5), but morphisms cannot remove either nodes, edges or labels. Formally, a graph morphism $g : G \to H$ between two graphs consists in two functions $g_V : V_G \to V_H$ and $g_E : E_G \to E_H$ that preserve sources, targets and labeling functions. Such a morphism is called an inclusion, and is noted $g : G \hookrightarrow H$, when g_V and g_E are inclusion functions.

A *graph transformation rule* $r : L \hookleftarrow K \hookrightarrow R$ is defined by two inclusion morphisms $K \hookrightarrow L$ and $K \hookrightarrow R$ between three graphs L the left-hand side, R the right-hand side and K the kernel. See the rule r on top of Fig. 2 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 and the kernel graph K represents the interface between the patterns L or R and the rest of the graph. Roughly speaking, the transformation is simply defined by the removed part $L \backslash K$ and the added part $R \backslash K$.

To become workable, such a transformation graph rule should be provided with some mechanisms explaining how the rule is applied on a graph G. The first step consists in making sure that the pattern L is present in G. This can be represented by a graph morphism $m : L \to G$, called a *match morphism*. In general, match morphisms are not inclusions. However, in Fig. 2 and in following examples, for simplicity's sake, we often choose inclusions as match morphisms. Let us note H the resulting graph obtained by applying a rule $r : L \hookleftarrow K \hookrightarrow R$ on G, with the match morphism $m : L \to G$. The graph H is built by means of two constructions of category theory, called *pushout*. It consists in a commutative diagram made of four morphisms. A diagram $K \to L$, $L \to G$, $K \to D$ and $D \to G$ is a pushout if $K \to L \to G = K \to D \to G$ (where the morphism composition is defined component-wise) and if G is the least graph verifying this property. The construction of H, ilustrated in Fig. 2, is called a *direct transformation* and is usually denoted by $G \Rightarrow_{r,m} H$.

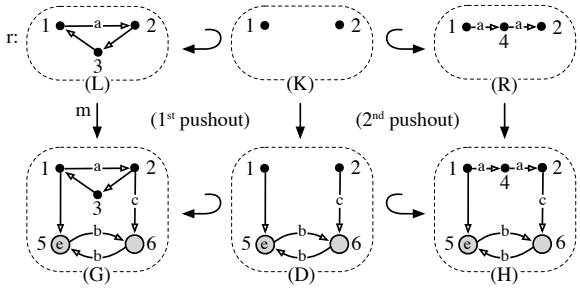


Fig. 2. A double pushout

Though, to apply the rule r on a graph G, it does not suffice that the match morphism $m : L \to G$ exists. The so called *dangling condition* should also be verified, it says that no node of $m(L) \backslash m(K)$ is linked to a node of $G \backslash m(L)$. For example, in Fig. 3, the node 3 occurring in the image of L by the morphism m is the source of an edge whose target node 5 does not belong to $m(L)$. So, the remaining graph D obtained by removing $m(L)$ from G gets a dangling edge. As a direct consequence, the resulting structure H inherits this dangling edge and is not a graph. Thanks to the dangling condition, such situations are precisely prevented and the resulting graph H can be computed as follows (see Fig. 2). First, D is obtained by removing all nodes and edges of $m(L) \backslash m(K)$ from G. Then H is built as the disjoint union of $m(R)$ and D along $m(K)$.

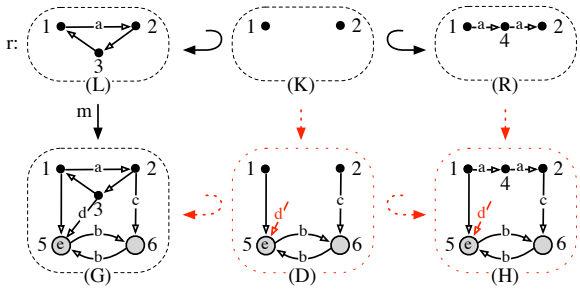


Fig. 3. Dangling condition

In the sequel, we will mainly discuss about graph transformation rules dedicated to manage operations on generalized maps. By lack of space, we will skip the details of the rules application and thus, the double pushout diagrams and the other theory category constructions will be left implicit henceforth. For more details on these aspects, the reader can consult [8].

2.2 G-maps

Generalized maps or G-maps characterize a topological model which presents the advantage of homo-

geneously dealing with all dimensions. Indeed, a G-map is composed of basic elements, also called *darts*. The G-maps represent objects of the quasi-manifolds's class. Classical *topological cells* as vertices, topological edges or faces can be retrieved by splitting these objects according to neighboring relations connecting darts: an α_i connection between two darts represents an i-dimensional neighboring relation. The 2D object on Fig. 4(a) is first split into faces (b) connected along edges by α_2 connections. Then these faces are split into edges (c) connected by α_1 connections. At last, these topological edges are split into couples of darts (d) linked by α_0. Classically, G-maps are defined as an algebraic structure [7], but G-maps are also particular graphs whose nodes are darts and α_i-labeled edges (or α_i edges, for short) models neighboring relations, with $0 \leq i \leq n$ in the case of an n-dimensional object.

Let us notice that some loops appears in the examples of Fig. 4(d). For instance, an α_2 loop $d \overset{\alpha_2}{—} d$ means that the topological edge carried out by the dart d has no proper neighbor of dimension 2 and thus represents a border of the object which is not linked to another face. Such a dart is said to be α_2-free.

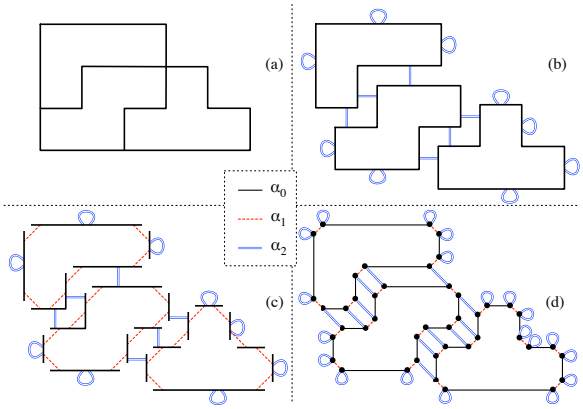


Fig. 4. Cell decomposition of an object

Given a dart d of a G-map, we can compute the topological cells from which it belongs as particular subgraphs. For example, the face adjacent to a dart d on a 2-G-map, is the subgraph which contains d, all darts reachable from d using α_0 and α_1 edges, and all corresponding edges themselves. More generally, an *orbit* $< \alpha_i, ..., \alpha_j > (d)$ is the subgraph which contains d, all darts reachable from d using α_i to α_j-labeled edges, and all of these edges. Thus, in 2-G-maps, vertices are $< \alpha_1, \alpha_2 >$-typed orbits (or $< \alpha_1, \alpha_2 >$ orbits, for short). Formally, a graph is said to be $< \alpha_{i_1}, \ldots, \alpha_{i_k} >$-typed when all of its edges are labeled in$\{\alpha_{i_1}, \ldots, \alpha_{i_k}\}$. In the same manner, topological edges are $< \alpha_0, \alpha_2 >$ orbits and faces are $< \alpha_0, \alpha_1 >$ orbits. More generally, i-dimensional cells of an n-G-maps are modeled with $< \alpha_0, ..., \alpha_{i-1}, \alpha_{i+1}, ..., \alpha_n >$ orbits.

In the algebraic approach, consistency constraints over generalized maps ensure the topological consistency of modeled objects. As we choose to interpret G-maps as a particular class of graphs, we naturally translate these constraints in terms of graph constraints. Thus, we define the G-maps this way :

Definition 1 (generalized map): A n-dimensional G-map ($n \geq 0$) is defined as a graph in which nodes represent darts (and are not labeled), edges are labeled in $\{\alpha_0, \ldots, \alpha_n\}$ and such that:
Non oriented constraint: G is non oriented.
Adjacent edge constraint: each node is the source node of exactly $n+1$ edges respectively labeled by $\alpha_0, \ldots, \alpha_n$.
Cycle 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 G-maps precisely define the class of quasi-manifolds. Indeed, this constraint guarantees that two i-cells can only be adjacent along $(i-1)$-cells. For instance, in the 2-G-maps of Fig. 4(d), the $\alpha_0\alpha_2\alpha_0\alpha_2$ cycle implies that faces are necessarily stuck along topological edges.

The G-maps can be manipulated by the means of operations. Considering that an half-edge consists in two α_2-free darts connected by an α_0 edge, let us first consider a simple operation consisting in splitting an half-edge in a 2-G-map. We can easily express this operation by means of a graph transformation rule (given in Fig. 5). Let us point out that the left-hand side L is the matched pattern (two darts connected by an α_0 edge, and provided with α_2 loops). The right-hand side R is the new pattern introducing two new darts so that a new vertex is created between the two darts at the extremities. We can also remark that nodes of $R \backslash K$, wich are named respectively c and d, have corresponding nodes in the resulting graph H (also named c and d by inclusion matching). Let us remark that on Fig. 5, the match morphism m satisfies the dangling condition. Indeed, there is no node in $m(L) \backslash m(K)$.

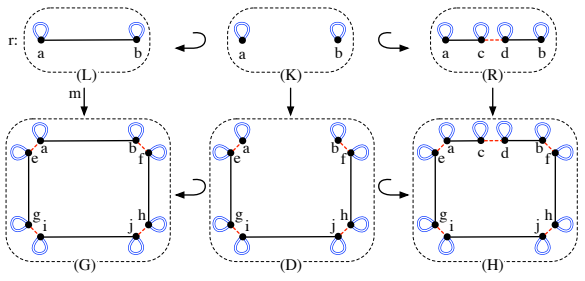


Fig. 5. Rule for the splitting of half-edges and its application

3. G-MAPS TRANSFORMATION RULES

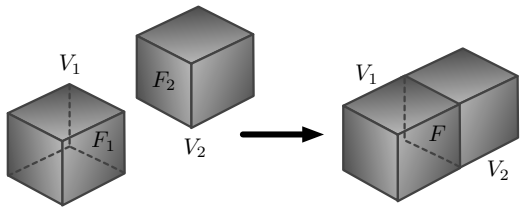
3.1 Orbit variables

In Section 2, we have seen that the double pushout approach of graph transformations allows us to write rules

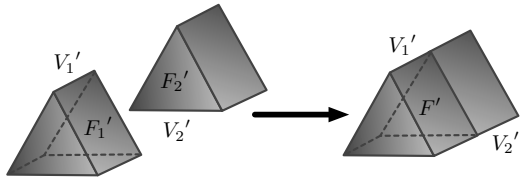
which transform G-maps. Nevertheless, graph transformation rules are not expressive enough to translate usual topological operations of G-maps into rules. Indeed, for many operations, orbits play a decisive role in the sense that their definition involves orbits in a generic way. They are defined in terms of a given arbitrary typed orbit, regardless of their structure (for instance their number of nodes). Thus, in order to increase the expressiveness of transformation rules in the context of topological operations, we introduce specialised variables which abstract the orbits and parameterize our rules. We call them *orbit variables* [10], [9], [8]. Obviously, they will be typed accordingly to the type of the required orbit. This introduction of dedicated variables has been motivated by some previous works: for example, in [4], it has been stressed that variables can make rule-based systems more expressive and abstract.

1) Introduction of orbit variables: Let us illustrate the notion of orbit variables by means of a classical topological operation: the *sewing* operation. Intuitively, the i-sewing operation sticks two i-dimensional topological cells along a common $(i-1)$-dimensional cell. In Fig. 6(a), we stick two cubes V_1 and V_1 along their respective square faces F_1 and F_2. We obtain two cubes which are stuck along a common square face F. Let us remark that in order to apply this sewing operation, F_1 and F_2 must be isomorphic. Another example is given in Fig. 6(b). Here, the same operation is used to stick two prisms $V_1{}'$ and $V_2{}'$ along their respective triangular faces $F_1{}'$ and $F_2{}'$. The resulting object consists in two prisms which are stuck along a common triangular face F'. In this second example, $F_1{}'$ and $F_2{}'$ are also isomorphic.

The particular case of the 3-sewing along a square face



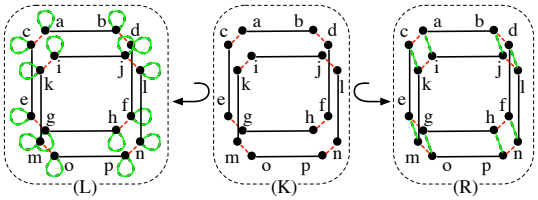
(a) Square faces



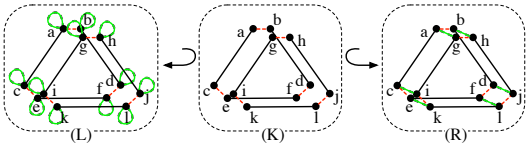
(b) Triangular faces

Fig. 6. 3-sewing of two volumes

in a 3-G-map may be translated into the rule illustrated Fig. 7(a). In the left-hand side, two square half-faces[1] are matched. They represent the half-faces F_1 and F_2 of Fig. 6(a). Let us remark that both of them are α_3-free since their darts have an adjacent α_3-loop. It means that in Fig. 6(a), V_1 and V_2 are not connected to another volume, respectively along F_1 and F_2. In the right-hand side of the rule, the two $< \alpha_0, \alpha_1 >$-typed orbits are linked with α_3. Thus, the right-hand side represents the common face F of Fig. 6(a). Let us remark that this first rule allows one to stick any two volumes, but only along square faces. A new rule is required for the 3-sewing along a triangular face. This second rule, which is analogous to the first one, is illustrated in Fig. 7(b) with triangular $< \alpha_0\alpha_1 >$ orbits.



(a) Square faces



(b) Triangular faces

Fig. 7. 3-sewing rules

2) Transformation rules with orbit variables: We use orbit variables in order to abstract the different isomorphic half-faces involved in the 3-sewing rules. Indeed, in Fig. 7, each 3-sewing rule, respectively dedicated to the square and triangular faces, contains six occurrences of isomorphic half-faces: two in the left-hand side, two in the interface and two in the right-hand side. We factorize these two rules by replacing all occurrences of isomorphic half-faces by a node labeled by a variable orbit. The variable orbit is typed by $< \alpha_0\alpha_1 >$ to indicate that the variable should only be replaced by graphs representing $< \alpha_0\alpha_1 >$ orbits. Thanks to this variable-based mechanism, we get a single graph transformation rule for specifying the topological 3-sewing operation, whatever the structure of the half-face along which the two volumes are stuck. The rule with an $< \alpha_0\alpha_1 >$-typed orbit variable F is given in Fig. 8. Orbit variables label the nodes. Graphically, we represent them with a grey-coloured region decorated with the variable and its type. Let us remark that in the left-hand side of the rule, as nodes 0 and 1 are labeled with the same variable F, they abstract two isomorphic copies of a same orbit. For practical purposes, the instanciation of the rule consists in substituting F by an $< \alpha_0\alpha_1 >$-typed orbit. Indeed, the typing of orbit variables restricts the set of graphs substituted to variables, to the ones generated by labels of a given type (α_0 and α_1 in our example). From a technical point of view, the interface can be either two nodes labeled by the typed variable $F < \alpha_0\alpha_1 >$ or

[1] In a 3-G-map, the $< \alpha_0\alpha_1 >$-typed orbits are called half-faces.

simply two unlabeled nodes, provided that names 0 and 1 are used to explicit the inclusion morphism. In the next paragraph, we will emphasize that the application of a generic rule involving orbit variables requires that some nodes labeled by an orbit variable points on a dart of the aimed G-map in order to be used as starting point to build the match morphism between the left-hand side of the rule and the G-map. We call these particular nodes *hooks* and denote them within rules by a grey-coloured region surrounded by an additional black ring. Thus, we can see that in Fig. 8, both nodes of the left-hand side are hooks.

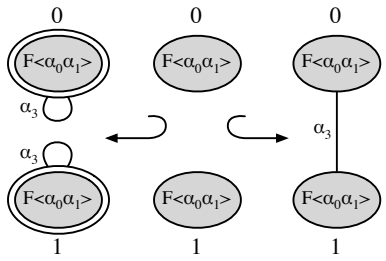


Fig. 8. The 3-sewing rule with an $< \alpha_0\alpha_1 >$-typed orbit variable

3) Application of generic rules: The application of a graph transformation rule (without variables) first requires to find a match morphism from the left-hand side L of the rule towards the graph under transformation G. Intuitively, the application of a generic rule (with variables) requires two main steps. First, the construction of the isomorphic orbits present in the graph G and intended to replace the nodes labeled by a variable orbit. Then, the construction of the concrete graph transformation rule (that is to say without variables) derived by the replacement by the identified orbit of the node labeled by an orbit variable in the generic rule. To guide the construction of the concrete rule from the generic rule and the match morphism, some nodes are hooks which point on darts so that the orbits will be searched only from these pointed darts by typical graph-coverage algorithms[2]. In other words, hooks allow us to restrict the search of both the orbit which replace the variable orbit and the match morphism. Let us consider Fig. 9 to illustrate this idea. On the figure, the node a (resp. i) of the graph G corresponds to the hook 0 (resp. 1) of the left-hand side of the generic rule. In practice, such a connexion will be initiated by the user of the modeler by selecting particular darts on the initial G-map (See Section 5 for more details). Based on this starting point, the generic rule can be concretized by considering the orbits $< \alpha_0\alpha_1 > (a)$ and $< \alpha_0\alpha_1 > (b)$ belonging to the two volumes V_1 and V_2 present in the G-map under transformation. Then, the $< \alpha_0\alpha_1 >$ orbit which is substituted to F is constructed starting from a and covering its neighbor darts using α_0 and α_1 edges. Let us notice that as 0 and 1 are labeled with the same orbit variable F, the constructed orbits issued from a and i have to be isomorphic. Finally, the derived concrete rule is isomorphic to the one of Fig. 7(a) since the orbit variable F is substituted by a square half-face. Hooks are convenient to define both concrete rule and match morphism, and thus application of generic rules.The selection of hooks in a generic rule is left to the expertise of the designer.

[2]Let us notice that in the particular case of G-maps, the search for particular subgraphs (as the orbits) in a graph is simpler than in the general case. Indeed, the particular labeling of edges allows us to optimize the coverage algorithms.

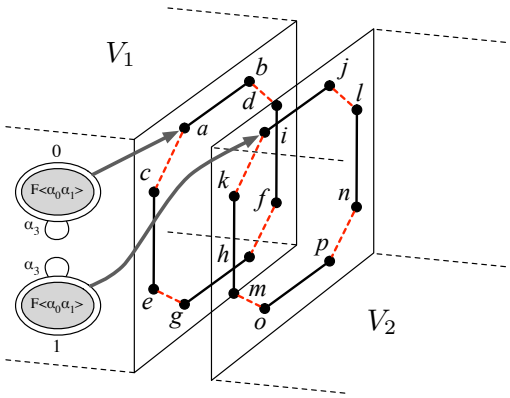


Fig. 9. The pointing system

3.2 Relabeling of orbit variables

The orbit variables allow us to translate the operations in which several copies of a same orbit are handled into a unique rule. However, in many classical topological operations such as the cone, the extrusion, the triangulation or even the rounding operation, we do not handle direct copies of orbits, but orbits modified by relabeling or removing edges. So, in order to cope with these operations, we introduce *relabeling of orbit variables* in our rules.

1) Requirement of orbit variables relabeling: Let us illustrate the relabeling of orbit variables with the triangulation of a face. For instance, in Fig. 10(a), starting from a face F, the application of the triangulation produces four triangles T_1, T_2, T_3 and T_4. In Fig. 10(b), this example is illustrated in the case of 2-G-maps. The starting face is represented on the left-hand side by the grey-coloured $< \alpha_0\alpha_1 >$-typed orbit F. On the right-hand side, F is duplicated into three grey-coloured subgraphs F_0, F_1 and F_2. In F_0, the α_1 edges of F are removed while the α_0 edges remain. In F_1, the α_0 edges are removed and the α_1 edges are relabeled into α_2 ones. In the last copy F_2, α_0 and α_1 are respectively relabeled into α_1 and α_2. Finally, the triangles T_1, T_2, T_3 and T_4 of Fig. 10(a) are modelled by, respectively, orbits $< \alpha_0\alpha_1 > (a_1)$, $< \alpha_0\alpha_1 > (d_1)$, $< \alpha_0\alpha_1 > (h_1)$ and $< \alpha_0\alpha_1 > (e_1)$. We denote the relabeling of orbit variables as follows. The function $< \gamma_1 ... \gamma_k > : < \beta_1 ... \beta_k > \rightarrow < \gamma_1 ... \gamma_k >$ renames the labels β_i into γ_i (with $1 \leq i \leq k$) in $< \beta_1 ... \beta_k >$-typed orbits.

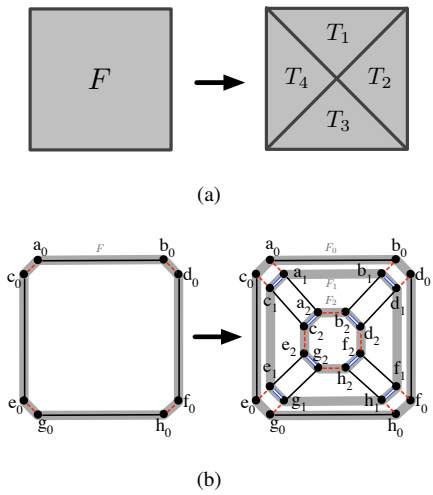


Fig. 10. A triangulation

Moreover, if γ_i is the *removing label* "_", the edges labeled with β_i are removed. In our example, with F a $< \alpha_0\alpha_1 >$-typed variable, we have $F_0 =< \alpha_{0\_} > (F)$ with $< \alpha_{0\_} >:< \alpha_0\alpha_1 > \rightarrow < \alpha_{0\_} >$. Similarly, $F_1 =< \_\alpha_2 > (F)$ with $< \_\alpha_2 >:< \alpha_0\alpha_1 > \rightarrow < \_\alpha_2 >$ and finally $F_2 =< \alpha_1\alpha_2 > (F)$ with $< \alpha_1\alpha_2 >:< \alpha_0\alpha_1 > \rightarrow < \alpha_1\alpha_2 >$.

2) Transformation rules with relabeling of orbit variable : The introduction of relabeling functions in our rules allows us to translate the triangulation into the rule of Fig. 11. This rule is general in the sense that it can be used in order to triangulate any face in a 2-G-map. The rule involves an $< \alpha_0\alpha_1 >$ orbit variable F. In the left-hand side of the rule, the hook 0 abstracts the $< \alpha_0\alpha_1 >$ orbit F of Fig 10(b). In the right-hand side of the rule, the nodes 0, 1 and 2 respectively abstract the three copies F_0, F_1 and F_2, respectively labeled with the previously introduced relabeling functions $< \alpha_{0\_} >$, $< \_\alpha_2 >$ and $< \alpha_1\alpha_2 >$. Thus, the triangulation rule matches an $< \alpha_0\alpha_1 >$ orbit (the face of the pointed dart in the G-map associated to the hook 0) and replaces it by three relabeled copies pairwise connected with α_1 and α_0 edges. Indeed, let us remark that in Fig. 10(b), the nodes of F_0 (respectively F_1) are connected to corresponding nodes in F_1 (respectively F_2) with α_1 (respectively α_0) edges. Thus, in the right-hand side of the rule, the nodes 0 and 1 are linked by an α_1 edge. Similarly, the nodes 1 and 2 are linked by an α_0 edge. Finally, by associating to the hook a dart which carries a square face, we get the concrete rule depicted in Fig. 12.

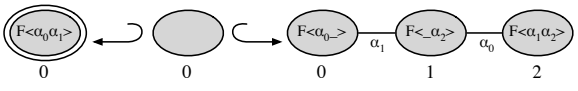


Fig. 11. The triangulation rule with relabelings of an orbit variable F

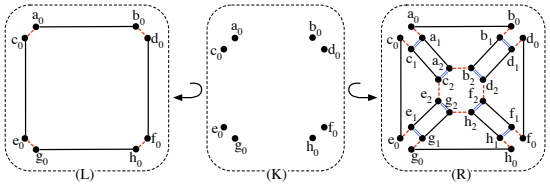


Fig. 12. Instanciation of the triangulation rule with a square face

3.3 Additionnal operations

We illustrate our language with four operations: the cone, the extrusion, the rounding and the removal.

1) The cone and the extrusion: Both cone and extrusion operations consist in creating an $(i+1)$-dimensional cell from an i-dimensional one. In Fig. 13(a), the application of a cone operation on a triangular face F produces a tetrahedron. The corresponding rule in the 3-G-maps is illustrated in Fig. 13(c). It matches an $< \alpha_0\alpha_1 >$-typed α_3-free half-face and replaces it with one isomorphic copy (see node 0 in the right-hand side) and three relabeled copies (see nodes 1 to 3). Intuitively, the node 0 of the right-hand side abstracts the bottom face F of Fig. 13(a) while the node 3 abstracts the top vertex F_3. The principle of the extrusion operation is analogous. In Fig. 13(b), a prism is produced from a triangular face F'. The corresponding rule for 3-G-maps is illustrated in Fig. 13(d).

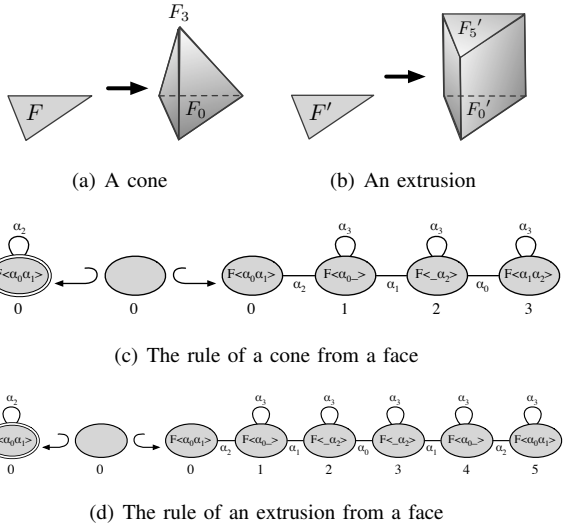
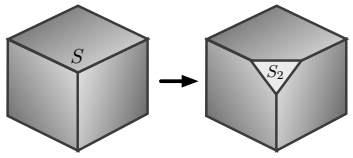


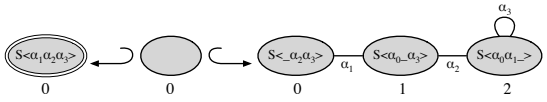
Fig. 13. The cone and the extrusion

2) The rounding: Intuitively, the rouding operation produces a soft object from an angular one. In Fig. 14(a), the application of the rouding operation on a vertex S of a cube produces a face S_2 in the place of S. In the case of 3-G-maps, the rounding of a vertex can be translated in the rule of Fig. 14(b). An $< \alpha_1\alpha_2\alpha_3 >$ orbit, like the vertex S of Fig. 14(a) can be associated to the orbite variable labeling the hook of the left-hand side of the rule. This orbit is replaced with three relabeled copies. Intuitively, the node 2 abstracts the face S_2 created

during the rounding operation on S.



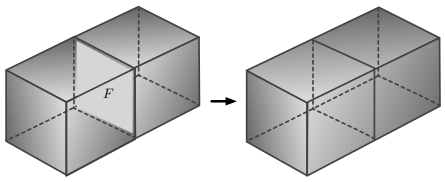
(a) The rounding of a vertex



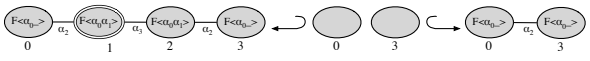
(b) The rounding of a vertex rule

Fig. 14. The rounding

3) The removal: The removal operation allows ones to remove an $(i-1)$-cell which connects two i-cells. In Fig. 15(a), the removal operation is applied on a face F which connects two cubes. The rule of the removal of a face in a 3-G-map is illustrated in Fig. 15(b). Intuitively, in the left-hand side of the rule, the nodes 1 and 2 connected with an α_3 edge abstract the face F. Indeed, a face is constituted by two half-faces (one is abstracted by 1 while the other one is abstracted by 2) connected with α_3 edges. The node 0 (resp. 3) abstracts the darts of one of the cubes connected to F (resp. the other cube connected to F). In the right-hand side of the rule, the nodes 1 and 2, and so the face F, have been removed. Thus, the darts which in the cubes, were initially connected to F, are now pairwise connected. Let us remark that hook must be labeled with a full orbit, without any "remove" label _.



(a) The removal of a face



(b) The rule of a face removal

Fig. 15. The face removal

4. G-MAP TRANSFORMATION RULES CONSTRAINTS

Usually, the definition of a new operation implies two mandatory verifications. The first one is 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. The second verification deals with the implementation of the operation. It consists in testing that the program correctly implements the operation without any side effect. Generally, many tests are performed on a large set of entries objects. The main advantage of our rule approach is that these verifications become immediate. As a rule definition is also an implementation of an operation, we only need to check that rules preserve the G-maps constraints.

For this prupose, [8] introduces syntactic criteria on rules. These criteria are shortly presented here and their full definition can be consulted in [8]. A correct rule must satisfy the three following criteria, corresponding to the G-maps consistency constraints :

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

As these criteria are syntactic, their verification can be done both automatically and statically. So, a full modeler kernel based on rules should not only provide a function to apply rules but also a function to check the rules syntax, according to these three previous criteria. The main advantage of these syntactic criteria stays in their genericity. Indeed, they are common to every G-map transformation rules. Thus, while in the classical approach a proof was required for every operations, here, the only program that must be proved or tested is the one which checks and applies our rules.

4.1 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, on a rule $r : L \hookleftarrow K \hookrightarrow R$, both L, K and R are non-oriented graphs.

4.2 Adjacent edges criterion

In n-G-maps, the adjacent edges constraint means that each node has exactly $n+1$ adjacent edges respectively labeled with α_0 to α_n. In order to preserve this constraint, rules must satisfy the following properties:

- Nodes of the kernel K must have adjacent edges with the same labels on both the left-hand side and right-hand side of the rule. For example, in Fig. 12, a_0 has two adjacent edges labeled with α_0 and α_1 in the left-hand and right-hand sides.
- The removed nodes of $L\backslash K$ and the added nodes of $R\backslash K$ must have exactly $n+1$ adjacent edges respectively labeled with α_0 to α_n. For instance, in Fig. 12, the nodes a_1 and a_2 have their three α_0, α_1 and α_2-labeled adjacent edges.

In order to extend these properties on generic rules with relabeling functions, like the one in Fig. 11, edges labels and relabeling functions labels must be considered in the same manner. Indeed, the relabeling functions labels of a given node can be seen as implicit adjacent edges of this node. Let us notice that considering this extension, the node 0 of the rule of Fig. 11 satisfies the first property: in the left-hand side, the node 0 has

two implicit α_0 and α_1-labeled adjacent edges while in the right-hand side, he has one implicit α_0-labeled adjacent edge and one explicit α_1-labeled edge. In the same manner, the added nodes 1 and 2 satisfy the second property: they have their three α_0, α_1 and α_2-labeled adjacent edges, implicitly or explicitly.

4.3 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 dart belongs to an $\alpha_i\alpha_j\alpha_i\alpha_j$-labeled cycle. As the adjacent edges constraint, it is divided into properties depending on the considered sets of darts:

- A added dart of $R\backslash K$ must be added with all of its cycles. For example, the nodes a_1 and a_2 of the rule of Fig. 12 belong to $\alpha_0\alpha_2\alpha_0\alpha_2$-labeled cycles.
- If a preserved dart of K belongs to a fully matched $\alpha_i\alpha_j\alpha_i\alpha_j$-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.
- If a preserved dart of K belong to an $\alpha_i\alpha_j\alpha_i\alpha_j$-labeled cycle which is not fully matched, its α_i and α_j-labeled edges are preserved. Intuitively, this means that some of the cycle edges are matched in the left-hand side while the other ones are not matched. Thus, modifying the matched edges could lead to a break of the cycle. For example, in Fig. 12, the $\alpha_0\alpha_2\alpha_0\alpha_2$-labeled cycle of node a_0 is partially matched in the left-hand side (only α_0 is matched). So, in order to avoid the breaking of the existing cycle, this α_0-labeled edge must not be modified in the right-hand side.

As previously, this properties are extended to generic rules with relabeling functions by considering in the same manner both explicit and implicit edges.

5. A RULE APPLICATION KERNEL

One of our objectives about rule-based transformations is to propose a new approach to develop geometric modeler software. As topological operations can be formally defined by rules, a program that can apply any rule can compute operations. In this section, a such program is presented. The chosen implementation language is OCaml [5] mainly because our formalism is well adapted to this language. Nevertheless, the presented data structures and algorithms can be easily translated into any programming language.

5.1 Rules syntax

The classical rule representation $r : L \hookleftarrow K \hookrightarrow R$ formally defines a transformation. Thus, it is useful for proofs writing. In the sequel, we use a simplified notation by replacing $L \hookleftarrow K \hookrightarrow R$ by $L \rightarrow R$. As we represent inclusions morphisms by the means of nodes names, the interface K is implicitly defined by the intersection $L \cap R$ of L and R. Thus, local transformations and preserved nodes are constructively specified by the partial function $L \rightarrow R$ which includes relabeling. The darts in the set $L\backslash R$ are removed and those in $R\backslash L$ are added. For example, a face triangulation in a 3-G-map is defined by the rule given on Fig. 16. The used syntax remains similar, the hooks are again represented by a double border node.

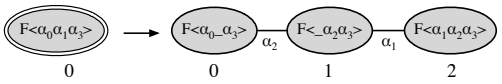


Fig. 16. Triangulation of a face rule ($L \rightarrow R$ notation)

As a convention, the nodes are named using successive integers starting from 0 in the two hand-sides of the rule. Hence, we get a useful way to use this node indexing as indexes of arrays in the algorithm that implement node application. So, as the partial function $L \rightarrow R$ specifies dart preservation, it also indicates how nodes are renamed. This is noted under the function arrow, as it is the case for the face removal rule (see Fig. 17) where it is indicated that node 3 is renamed into node 1.

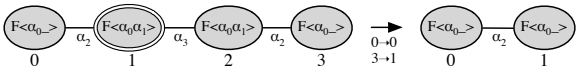


Fig. 17. Removal of a face ($L \rightarrow R$ notation)

The graphical syntax used to present the rules is translated into an OCaml data type, see Fig. 18. The first field, `left_hooks`, is the list of indexes of nodes of the left-hand side of the rule that are hooks. For example, for the triangulation rule (see Fig. 16) this list contains the index 0 and for the removal rule (see Fig. 17) this list contains the index 1. Another example is the sewing rule that have two hooks and then the list contains two indexes. The field `left_nodes` is an array such that its indexed by names of the nodes. It is the array of the orbit types of left abstract nodes. For example, for the triangulation rule (see Fig. 16), this array should has only one element that is a list of the link labels 0, 1 and 3. For the particular "remove" label (denoted _ previously), we use the value -1. The field `left_edges` is the list of the explicit edges between the nodes. These edges are noted by a triple as *(source name, target name, label)*. For example, the α_2 connection between the nodes 0 and 1 of the removal rule (Fig. 17) is defined by `(0,1,2)`. The field `nodes_matching` defines the partial function that maps node names between the left-hand side and the right hand side of the rule. This is done using an array indexed by names of nodes in the left-hand side of the rule and where values are the new node names in the right hand side. The field `right_nodes` is the orbit types that label nodes in the right-hand side of the rule. If the type of a preserved node is different than

the type it have in the `left_ nodes` array field, this means that it is relabeled. The field `right_edges` is the list of the explicit edges in the right-hand side of the rule with the same notation as previously. Using this rule presentation, the algorithm described in section 5.3 computes the result of a rule application.

```
type t_rule = {
   left_hooks : int list;
   left_nodes : (int list) array;
   left_edges : (int * int * int) list;
   nodes_matching : int array;
   right_nodes : (int list) array;
   right_edges : (int * int * int) list;
   right_ebd_fun : t_ebd_fun array;}
```

Fig. 18. OCaml type of rules

In order to get a visualization of objects before and after an operation application, basic pieces of geometric information are needed. Hence, a G-map must have an embedding. The simplest way to define a G-map embedding is to associate points to vertexes (i.e. 0-cell orbits). Then, to visualize a 2-G-map or a 3-G-map, 3D points can be associated to darts. In order to manage this simple embedding, expressions are given for each node of the rule right hand side. These expressions define embeddings of vertexes obtained after rule application from the original embeddings. Embedding expressions can be seen as added labels to nodes of the right hand-side. For example, the face extrusion rule on Fig. 19 is presented with embedding expressions.

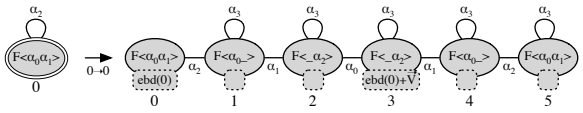


Fig. 19. Extrusion of a face with embeddings

In this rule, the expression `ebd(0)` specify the embedding of each concrete copie of the node 0. And the embeddings of the created darts (concrete copies of node 3) are computed as matched points translated by a vector $\vec{V}$. The Fig. 19 also shows that the embedding expression is not repeated when several abstract nodes share a same vertex orbit. Hence, the expression `ebd(0)` defines the embedding of nodes 0, 1 and 2 and the expression `ebd(0)+V` defines the embedding of nodes 3, 4 and 5. Actually, this syntax provides a good way to optimize computation of the embedding. To complete the data type description, the extrusion rule of a face (Fig. 19) is written in Fig. 20 with the OCaml data type defined in Fig. 18.

From a practical point of view the expressions language is independent from the topological rules. The operators in these expressions depends of the chosen embedding and the needs of the application. For example, in the face triangulation rule (see Fig. 16), the barycenter of the face must be computed. To do that, two operators are defined. The first one collects the embeddings of a specified orbit from a given node. The second one

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

Fig. 20. OCaml rule of the extrusion of a face

computes the barycenter of a set of points. Then the barycenter of the face is defined by the expression `mean(collect_edb(face_orbit,0))`.

5.2 *Implementation of G-map*

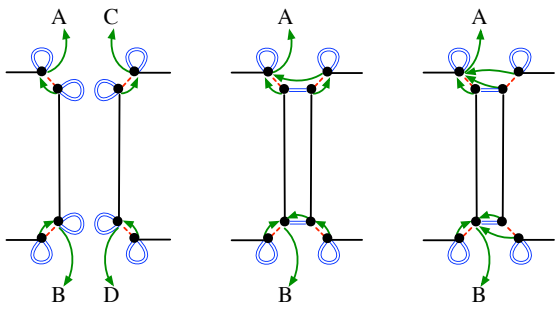
Before describing the algorithm which applies our rules, let us introduce our G-map implementation. It consists in a G-map data type associated to the following minimal set of functions:

- Creation and removal of a free dart, *i. e.* a dart which is not connected to another dart;
- Orbit covering operation;
- Vertex embedding setting. It defines the embedding for each dart of a given vertex orbit.

It should be noticed that two classical G-map building operations are not listed, cell sewing and cell unsewing operations. Actually, as seen previously, they are defined by rules and then they are implemented as all other operations provided with a rule-based definition.

As rules can be defined for G-maps of any dimension, our implementation is parameterized by G-map dimension. Then, the embedding type must also be a parameter. For example, 2-G-map embeddings could be 2D points or 3D ones.

Vertex embedding is implemented in such a way that only one dart (the carrier) of the vertex orbit refers to the embedding. The other darts have access to the embedding through a chain of indirections (see Fig. 21).



(a) Two face borders (b) After the sewing of the faces (c) After getting darts's embedding

Fig. 21. Chaining of the embedding

For example, on Fig. 21(b), two darts directly refer to their embedding and the others use indirections to get the embedding. Hence, construction operations do not have to traverse the vertex orbit to update the embedding when vertexes are unified. The example

on Fig. 21 shows embeddings management when two edges are sewed, only the carrier darts of one edge must be updated. Moreover, the chain of indirections to an embedding can be lazily reduced during reading operations. It is sufficient to affect the dart resulting from a reading operation to the dart from where the reading operation was called. Hence, direct references to embedding could be obtained after some reading operations (see Fig. 21(c)).

5.3 Rule application function

In this section, we introduce the algorithm which applies the rules to a given G-map. Some steps of this algorithm are illustrated Fig. 22 in the case of the face triangulation rule Fig. 16 applied to the square Fig. 10(b). The input of a rule application consists in an association of each hook of the left-hand side to a dart of the start G-map. Then, we use the covering function of G-maps in order to reach every darts matched by the hook. The other darts matched by the rule are reach from previous darts using the edges of the left-hand side which connect hooks to the other nodes. For example, in the face removal rule (Fig. 17), the darts matched by the hook 1 are first reached by a coverage of the orbit $< \alpha_0\alpha 1 >$ from an input dart given to the rule application. The darts matched by nodes 0,2 and 3 are reached using the edges of the rule. For example, the darts matched by the node 0 are the α_2 neighbors of the darts identified with the coverage of hook 1. In the case of the triangulation Fig. 10(b), the result of this identification process is depicted by the 2D table of dart labels Fig. 22(a). Each column of the table represents an abstract node and two darts of the same line are copies of a same dart of the orbit.

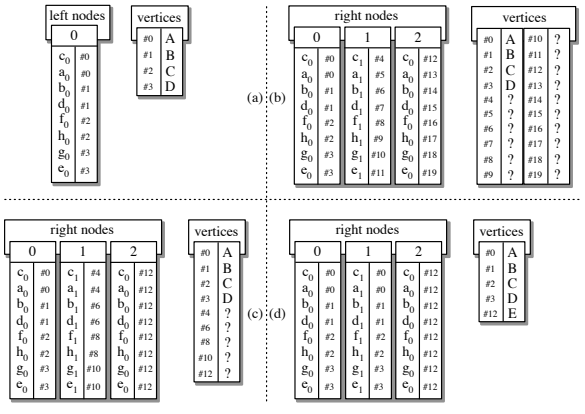


Fig. 22. Application of the triangulation rule

In the second step the equivalent array for the right hand side of the rule is constructed. To do that, columns (abstract nodes) that are preserved by the rule application are reused and placed at the correct new array index using the array of morphism's node matching. For the added nodes, columns corresponding to new darts are created using new labels. At this point, these new darts are disconnected from the others and have no embedding. The darts that are reused continue to have their connections and their embedding. For example, using the triangulation rule from the array of darts of Fig. 22(a), the array corresponding to the right-hand side is showed on Fig. 22(b). We can see that every new dart refers to an undefined embedding.

Now, the new darts are relabeled using the links determined by the first hook in the following way. For a dart d at the line i of node that has a relabeling $\alpha_h \rightarrow \alpha_k$, we get the neighbor by α_k of the dart at the same line i of the hook. Then we get the index j of this neighbor. To avoid combinatorial explosion, this is done with a hash table that associates each dart of the hook with its line index. At last, the new neighbor of d is the dart in the same column at the line j. As a consequence of these new links, some embeddings are shared. On Fig. 22(c), paired darts belonging to the node 1 share their embedding, and all the darts belonging to the node 2 share the same embedding. Then, darts are linked with the explicitly given edges that appear in the right-hand side of the rule. Hence, for an edge (i, j, k), all the darts of the column i are linked, line by line, to the darts of column j by the link α_k. Hence, some other darts share their embedding.

In the next step new vertexes are computed. If the embedding of a dart is not defined, it is computed using the embedding function of the corresponding node named by index. The sharing of the embedding minimizes the number of new vertexes computations. At last, we compute the transformations of already existent vertices. Finishing by these already existent vertices, we prevent to distort the computation of the new vertices. Now, all edges explicitly given by the left-hand side of the rule are removed and darts belonging to removed nodes are deleted. Then, darts belonging to preserved nodes are relabeled the same way that the new darts before. Finally, the preserved darts are connected to the new darts using the edges explicitly given in the right-hand side of the rule. Again, embeddings are shared. For example, in Fig. 22(d), the darts $a_1, \ldots, h_1$ at the border of the created part get their embedding from the darts of the original face $a_0, \ldots, h_0$.

6. Results

6.1 Prototyping a 3D modeler

In this section, two case studies of our rule-based kernel are presented. The first one illustrates the framework independence of the G-map dimension. To show that, a fractal object computation with a 2-G-map is done. As we use G-maps, 2D fractals that can be modeled with quasi-manifolds have been chosen. As fractals are usually defined by rules, the design of fractals is straightforward in our framework. Thus, for instance, Sierpinski's carpet depicted in Fig. 23 can be easily computed. First, the external square is obtained by inserting a vertex whose orbit is a single dart. This vertex is then extruded to an edge and after to a face using

extrusion rules. And, the carpet is computed from the square with four successive applications of a dedicated rule.

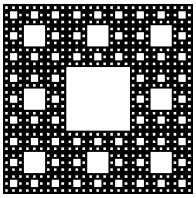
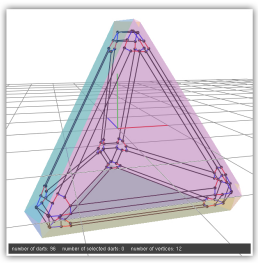


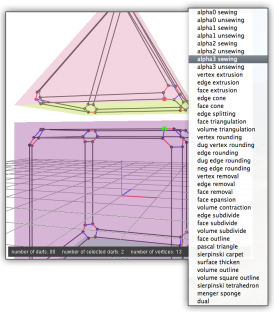
Fig. 23. A 2D fractal computed with rules on a 2-G-maps

The second case study of our kernel prototypes a 3D modeler. Now, our kernel is parametrized to be used with 3-G-maps embedded by 3D points. This modeler is easily extendible because all its operations are defined by rules. When the applications starts, the rules are loaded and the consistency constraints are checked for each rule. So, operations (defined by rules) can be added to the modeler without recompiling it. The syntax to define rules in the file is similar to the OCaml rule syntax described previously. For example, the type of an orbit $< \alpha_0, \alpha_2, \_ >$ is noted by `02_` instead of the full OCaml syntax `[0;2;-1]`.

The modeler main window is an OpenGL view of the embedded G-map. All the loaded rules are listed in a Glut menu. Choosing one rule of the list applies it to the list of selected darts. For example, the object on Fig. 24(a) was obtained in the following way: add a single dart vertex; extrude it to an edge; create a triangle using the cone operation from edge; create a tetrahedron using the cone operation from face; round the edges and vertexes. The dart selection is done with mouse picking in the OpenGL view. This is a convenient way to indicate darts that are associated to each hook of a rule. The order of the selection gives the position in the list of hooks. For example, on Fig. 24(b), the two volumes are sewed in the following: a first left click give the dart for the first hook; a second left click give the dart for the second hook; a right click open the menu where the sewing rule by α_3 can be chosen.



(a) A rounded tetrahedron

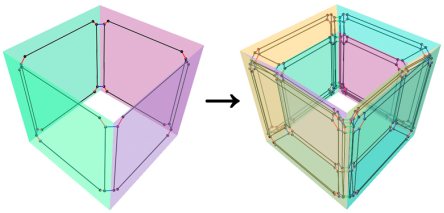


(b) α_3*sewing*

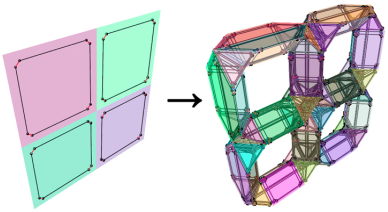
Fig. 24. The 3D modeler prototype

6.2 *Prototyping operations*

The main objective of the rules was to provide an easy and efficient way to define and implement operations. All the usual topologically-based operations on G-maps were defined with rules: sewing, unsewing, cone operations, extrusions, triangulations, full and dug vertex rounding, full and dug generalized rounding, subdivisions, removals, expansions and contractions. Moreover, some simple more involved operations were also prototyped. For example, on Fig. 25(a), an operation that thicks the surfaces of an object was defined by a rule[3]. Another example is an operation that compute a volume outline of an object[4].



(a) Thicken of a surface



(b) Volume outline of an object

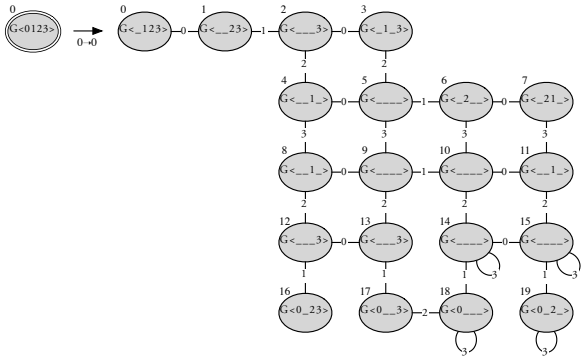
Fig. 25. Easily prototyped operations

Rules that computes 2D fractal objects can also be extended to compute 3D fractal objects. For example, the rule to compute the Menger sponge is given on figure 26(a) (the labels α_i are noted i to be short). Application of this rule is shown on figure 26(b). Even if this operation is complex only about fourty minutes were needed to define it. This information is an indication of the ”designing cost”. Such information could be used to estimate the difficulty to design operations using rules.

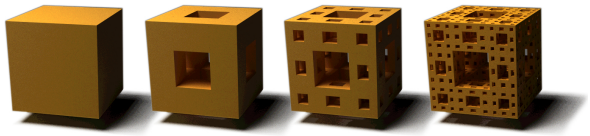
6.3 *Efficiency*

To validate our approach, some results obtained by our modeler kernel are presented. These results are compared with a topology-based modeler called Moka [11] that uses 3-G-maps. As Moka is 3D dedicated application written in C++, its operations have been developed following 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 could not be optimized because their implementation results from a unique function

[3]This operation can be used to quickly design walls of a building.
[4]This operation can be used to quickly design a windows from a regular grid.



(a) Menger sponge rule



(b) Menger sponge operation result

Fig. 26. A complex operation: Menger sponge

TABLE 1
TIME EFFICIENCY COMPARISON

Operation	Number of darts	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

that applies all rules. Considering this comparison, it could be noticed that important gains in development time have been obtained with an acceptable loss of performances. In the study below, computing times and length of program operation codes are both compared.

Results about computing times are first discussed. The table 1 presents computing times of operations on objects of different sizes. The object size is computed in number of darts. Test objects have realistic sizes as they are made of several thousands of darts, even several thousand hundreds of darts. Observation of the results shows that computation times obtained by our approach are in the same order of complexity than Moka. For example, the volume triangulation operation that have been tested with volume made of various numbers of faces and also different face degrees (triangles or squares) are in a constant ratio (approximately 3.5) with the operation implemented in Moka. The face triangulation operation that have been tested with objects made of faces with various degrees is also in a constant ratio (approximately 2.1) with the operation implemented in Moka.

Comparison has also be made for more complex operations as topological rounding of edges and vertexes. Objects containing various numbers of volumes were used. Computation times have also the same complexity as those obtained with Moka. The obtained results show that our kernel is faster but this is because the rounding operations in Moka have a geometrical part that is not considered in our kernel. Future works have to extend the rules to take this geometric part into account.

Last operations that are considered are sewing and unsewing by α_3. Again, these operations have been tested with various face degrees. In that case, Moka and our kernel have different strategies to deal with the embedding. In our kernel, every dart refers to the embedding using the previously described strategy. Conversely, in Moka, the embedding is referenced by only one dart per vertex orbit. For the sewing operation, the better results are obtained by our kernel. This is because embedding sharing is quicker. In the two applications, when two vertexes are sewed, only one of the two carrier darts need to have its embedding reference updated. After the operation, in our kernel, updated carrier dart refers to the sewed dart and, in Moka, updated carrier dart refers to empty embedding. But in our case, the carriers darts are directly known through the references chain and they have to be found with a covering operation in Moka. For the unsewing operation, our kernel takes more time. This is a dual consequence of the embedding sharing strategy. To unsew, vertex orbits must be split in two. Two embedding references must be defined. One is kept from original vertex orbit and the other must be defined. In our kernel, the embedding reference of all the darts of the new vertex orbit has to be changed. In Moka, only one dart in the new vertex orbit has to be updated as a carrier dart.

These preliminary results are very encouraging and studies must be conducted to better understand the complexity of computation times obtained by our approach. These performances should be related to the small number of code lines that are needed to program our kernel. Actually, our kernel is 700 OCaml lines long, including the 200 lines that program the application of a rule. In the loaded file that contains operation definitions, one operation definition takes one or two lines. In Moka's kernel, each of the operations is 100 to 300 lines long and the whole kernel is about 30000 lines. Moreover, with the classical development approach, all these lines must be tested and validated. Applications like Moka generally take several years to be developed. From another side, our prototype had been developed during approximately seven weeks, and the major part of time was dedicated to deal with the OpenGL library. The amount of efforts needed to develop software is

always evaluated with difficulty but with no doubt our first estimations show that the rule approach provides a very convenient and quick way to develop a kernel modeler and to prototype new operations, even if our kernel is not a complete modeler.

7. 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 extendible.

Our work will be extended by taking into account geometric embeddings in our rule-based framework. First of all, we are going to formally associate a polyhedric geometry to vertices (*i. e.* 0-dimensional cells of objects). Such a feature will allow us to define geometric operations, such as Boolean operations, which require geometric preconditions to be applied. 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] T. Bellet, M. Poudret, A. Arnould, L. Fuchs, and P. Le Gall. Designing a topological modeler kernel: a rule-based approach. Research Notes 2009-1, XLIM-SIC, UMR CNRS 6172, University of Poitiers, December 2009.

[2] 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.

[3] A. Habel and D. Plump. Relabelling in graph transformation. *Lecture notes in computer science*, pages 135–147, 2002.

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

[5] INRIA. The Ocaml Language. http://www.ocaml.org.

[6] 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.

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

[8] 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.

[9] M. Poudret, A. Arnould, J.-P. Comet, and P. Le Gall. Graph transformation for topology modelling. In *4th International Conference on Graph Transformation (ICGT'08)*, volume 5214 of *LNCS*, pages 147–161, Leicester, United Kingdom, September 2008. Springer.

[10] M. Poudret, J.-P. Comet, P. Le Gall, A. Arnould, and P. Meseure. Topology-based geometric modelling for biological cellular processes. In *1st International Conference on Language and Automata Theory and Applications (LATA 2007)*, Tarragona, Spain, March 29 - April 4 2007. http://grammars.grlmc.com/LATA2007/proc.html.

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