Jerboa: A Graph Transformation Library for Topology-Based Geometric Modeling

Hakim Belhaouari[1], Agnès Arnould[1], Pascale Le Gall[2], and Thomas Bellet[1]

[1] University of Poitiers, Laboratory Xlim-SIC UMR CNRS 7262,
Bd Marie et Pierre Curie, BP 30179, 86962 Futuroscope Cedex
{hakim.belhaouari,agnes.arnould,thomas.bellet}@univ-poitiers.fr
[2] Laboratoire MAS, Ecole Centrale Paris
Grande Voie des Vignes, 92295 Chatenay-Malabry, France
pascale.legall@ecp.fr
Website: https://sicforge.sp2mi.univ-poitiers.fr/jerboa

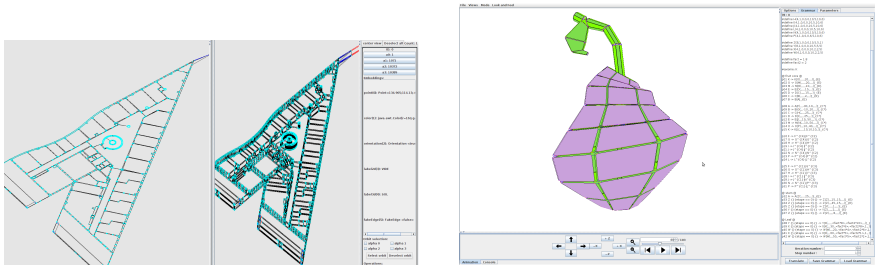
Abstract. Many software systems have to deal with the representation and the manipulation of geometric objects: video games, CGI movie effects, computer-aided design, computer simulations... All these softwares are usually implemented with ad-hoc geometric modelers. In the paper, we present a library, called Jerboa, that allows to generate new modelers dedicated to any application domains. Jerboa is a topological-based modeler: geometric objects are defined by a graph-based topological data structure and by an embedding that associates each topological element (vertex, edge, face, etc.) with relevant data as their geometric shape. Unlike other modelers, modeling operations are not implemented in a low-level programming language, but implemented as particular graph transformation rules so they can be graphically edited as simple and concise rules. Moreover, Jerboa's modeler editor is equipped with many static verification mechanisms that ensure that the generated modelers only handle consistent geometric objects.

Keywords: topology-based geometric modeling, labelled graph transformation, rule-based modeler tool-set, static rule verification, generalized map.

1 Introduction

Context. Geometric modeling is the branch of computer science that focuses on modeling, manipulation and visualization of physical and virtual objects. Over the past decade, numerous generic tools have been developed to assist the conception of dedicated modelers (3D modeler for game design, CAD for mechanical design, and so on). Even if such tools usually offer a wide set of modeling operations ready to use, they may be not sufficient to answer new requirements that are outside their scope or involve complex transformations. Fig. 1 gives two examples of such specific modelers. Fig. 1(a) shows a modeler dedicated to architecture which provides a specific operation of extrusion. From a 2D map, the operation constructs a floor by distinguishing walls, doors and

windows. The right figure (Fig. 1(b)) presents a modeler devoted to the modeling of plant growth (here a pear) and based on L-systems [TGM+09]. Both modelers have in common that the consistency of manipulated objects is ensured by the use of an underlying structure acting as a skeleton, called the topological structure. Other pieces of informations attached to objects, as position, color, density, etc. are of geometric or physical nature, and called *embedding*.



(a) Architectural modeler with a dedicated extrusion operation

(b) L-System modeler simulating plant growth

Fig. 1. Two geometric modelers with different application domains

Contribution. Graph transformations are already often used as key ingredients of dedicated software applications [BH02]. We present a java library, called Jerboa, based on graph transformations and designed to assist the development of new modelers whose objects and operations are specific to their application domains. All applications developed by means of the Jerboa library share the same topological model, the one of *generalized maps* (*G-map* for short) [Lie91], that can be viewed as a particular class of labelled graphs. Operations on geometric objects are specified by the developer as *rules*. These rules fall within the general framework of graph transformations, more precisely of the DPO approach on labelled graphs [EEPT06,HP02] and of rules with variables [Hof05]. The two modelers glimpsed in Fig. 1 have been developed using the Jerboa library. While classically data structures and operations of modelers are hand-coded in a low-level programming language, modelers built over the Jerboa library inherit from a predefined data structure implementing G-maps embedded (and being generic with respect to object dimensions). Once data structures associated to the embedding are given, operations are then defined as simple and concise rules. The workflow of a Jerboa-aided modeler development can be briefly described as follows: first, the user develops the data structures needed to represent embedding data, then, using the JerboaModelerEditor, he/she implements operations that will be used in the final application by writing one rule per operation. The rule editor comes with some static analysis mechanisms that verify both topological and geometric consistencies of objects. Finally, the Jerboa rule application engine ensures that rules are correctly applied. Note that rules manipulated in the

Jerboa library have been studied in some of our previous works. We first introduced in [PACLG08] special variables to generically denote topological cells. As a proof of concept, [BPA+10] presented a first prototype of a rule-based modeler. But unlike Jerboa, this prototype was not generic with respect to embedding data, but designed with a single predefined embedding (position of 2D or 3D points). We introduced in [BALG11] rules built over embedding variables and provided with syntactic conditions related to geometric constraints. The new Jerboa library combines all these previous contributions with some additional efficiency concerns.

Related work. Rule-based languages have previously been used for twenty years in the context of geometric modeling. In particular, L-systems [PLH90] have been introduced to model plant growth. As L-systems are based on iterated applications of a set of rules until a stop condition is satisfied, they are suited to represent arborescent patterns, like flowers or trees [MP96,KBHK07]. Moreover, L-systems have already been used in a topological-based context in [TGM+09] to model internal structure of wood, or to model leaves growth. Inspired by L-systems, grammars were introduced to model buildings or to generate them from aerial pictures [VAB10] in order to be displayed in navigation applications. All these applications built over L-systems or grammars are defined by a limited set of specialized high level operations. To our knowledge, while the latter are often abstractly specified by means of some kinds of rules, they are mostly hand-coded in a classical way. On the contrary, our rule-based approach remains independent from the application domain and avoids any hand-coding, except the step of rule writing.

Outline of the article. In Section 2, we briefly present the topological model of generalized maps, and the way geometric elements are attached. We then introduce the first elements of JerboaModelerEditor. In Section 3, we explain by means of examples how an operation is defined as a rule in Jerboa and focus on graph transformation techniques involved in the rule application engine. We then present the different verification mechanisms that assist the design of correct rules. Lastly, before concluding the paper in Section 5, we discuss about the efficiency of the Jerboa library in Section 4.

2 Object Data Structure : Embedded Generalized Maps

2.1 Generalized maps

As already stated in the introduction, we choose the topological model of generalized maps (or G-maps) [Lie91] because they provide an homogeneous way to represent objects of any dimension. This allows us to use rules for denoting operations defined on G-maps in an uniform way [PACLG08]. Moreover, the G-map model comes with consistency constraints characterising topological structures. Obviously, these constraints have to be maintained when operations are applied to build new objects from existing objects.

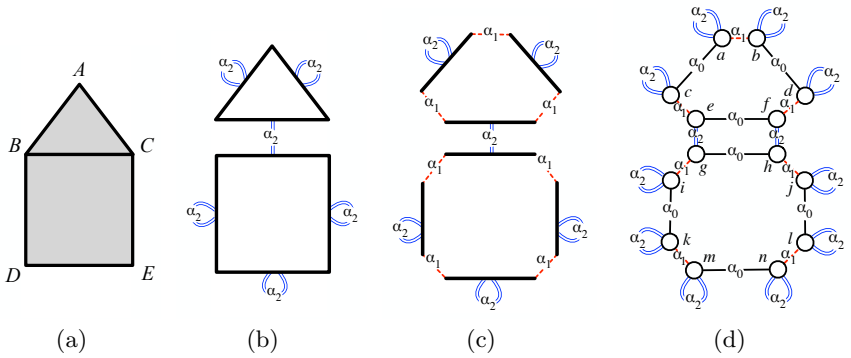
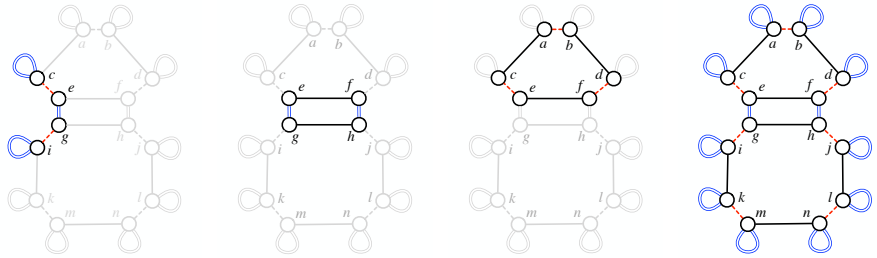


Fig. 2. Cell decomposition of a geometric 2D object

The representation of an object as a G-map comes intuitively from its decomposition into topological cells (vertices, edges, faces, volumes, etc.). For example, the 2D topological object of Fig. 2(a) can be decomposed into a 2-dimensional G-map. The object is first decomposed into faces on Fig. 2(b). These faces are *linked* along their common edge with an α_2 relation: the index 2 indicates that two cells of dimension 2 (faces) share an edge. In the same way, faces are split into edges connected with the α_1 relation on Fig 2(c). At last, edges are split into vertices by the α_0 relation to obtain the 2-G-map of Fig 2(d). Vertices obtained at the end of the process are the nodes of the G-map graph and the α_i relations become labelled arcs: for a 2-dimensional G-map, i belongs to $\{0, 1, 2\}$. According to the notation commonly used in geometric modeling, the labelling function is denoted α and an arc will be qualified as α_i-labelled. However, for simplicity purpose, when representing G-maps as particular graphs, arcs will be directly labelled by an integer. Hence, for a dimension n, n-G-maps are particular labelled graphs such that arcs are labelled in the $[0, n]$ interval of integers.

In fact, G-maps are non-oriented graphs as illustrated in Fig. 2(d) : labelled non-oriented arcs represent a pair of reversed oriented arcs that are identically labelled. Notice that in order to be more readable, in all figures given in the sequel, we will use the α_i graphical codes introduced in Fig. 2(d) (simple line for α_0, dashed line for α_1 and double line for α_2) instead of placing a label name (α_i or i) near the corresponding arc. So, in the following, the way non-oriented arcs are drawn will implicitly indicate the arc label values.

Topological cells are not explicitly represented in G-maps but only implicitly defined as subgraphs. They can be computed using graph traversals defined by an originating node and by a given set of arc labels. For example, on Fig. 3(a), the 0-cell adjacent to e (or object vertex attached to the node e) is the subgraph which contains e, nodes reachable from the node e using arcs labelled by α_1 or α_2 (nodes c, e, g and i) and the arcs themselves. This subgraph is denoted by $G\langle\alpha_1\alpha_2\rangle(e)$, or simply $\langle\alpha_1\alpha_2\rangle(e)$ if the context (graph G) is obvious, and models the vertex B of Fig. 2(a). On Fig. 3(b), the 1-cell adjacent to e (or object edge



(a) Vertex (b) Edge (c) Face (d) Connected component

Fig. 3. Reconstruction of orbits adjacent to e

attached to the node e) is the subgraph $G\langle\alpha_0\alpha_2\rangle(e)$ that contains the node e and all nodes that are reachable by using arcs labelled by α_0 or α_2 (nodes e, f, g and h) and the corresponding arcs. It represents the topological edge BC. Finally, on Fig. 3(c), the 2-cell adjacent to e (or object face attached to e) is the subgraph denoted by $\langle\alpha_0\alpha_1\rangle(e)$ and built from the node e and arcs labelled by α_0 or α_1 and represents the face ABC. In fact, topological cells (face, edge or vertex) are particular cases of *orbits* denoting subgraphs built from an originating node and a set of labels. We will use an ordered sequence of labels, encoded as a simple word o and placed in brackets $\langle\rangle$, to denote an orbit type $\langle o\rangle$. In addition to the orbit types already mentioned, we can mention the orbit $\langle\alpha_0\alpha_1\alpha_2\rangle(e)$ on Fig. 3(d) representing the whole connected component.

For a graph G with arc labels on $[0, n]$ to represent an n-G-map, it has to satisfy the following topological consistency constraints:

- *Non-orientation constraint*: G is non-oriented, i.e. for each arc e of G, there exists a reversed arc e' of G, such that the source of e' is target of e, target of e' is source of e, and e and e' have the same α label,
- *Adjacent arc constraint*: each node is the source of exactly $n+1$ arcs respectively α_0 to α_n-labelled,
- *Cycle constraint*: for every i and j verifying $0 \leq i \leq i+2 \leq j \leq n$, there exists a cycle labelled by $ijij$ starting from each node.

These constraints ensure that objects represented by G-maps are consistent manifolds [Lie91]. In particular, 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. 2(d), the $\alpha_0\alpha_2\alpha_0\alpha_2$ cycle constraint implies that faces are stuck along edges. Let us notice that thanks to loops (see α_2-loops in Fig. 2(d)), these three constraints also hold at the border of objects.

2.2 Embedding

Topological structures of n-G-maps have been defined as labelled graphs whose arc label set is $[0, n]$. We complete now this definition by using node labels to

represent the embedding data. Actually, each kind of embedding has its own data type and is defined on a particular type of orbit. For example, a point can be attached to a vertex, and a color to a face. Thus, a node labelling function defining an embedding has to be typed by both a (topological) orbit and a data type. For example, the embedding of the 2D object of Fig. 4(a) is twofold: geometric points attached to topological vertices and colors attached to faces. On the embedded G-map of Fig. 4(b), each node is labelled in this way by both a point and a color. For example, the node g is labelled by both the point B and the light color.

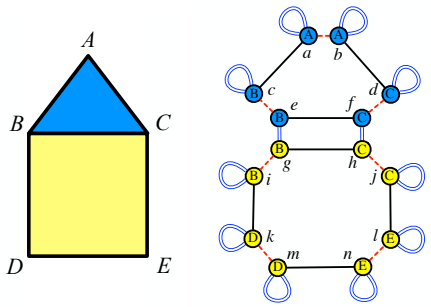


Fig. 4. Representation of 2D object with multiple embedding functions

Actually, for embedded G-maps, we characterize a node labelling function as an *embedding operation* $\pi : \langle o \rangle \rightarrow \tau$ where π is an operation name, τ is a data type and $\langle o \rangle$ is a domain given as an n-dimensional orbit type. For an embedded G-map G equipped with an embedding $\pi : \langle o \rangle \rightarrow \tau$, we generically denote $\lfloor \tau \rfloor$ the set of values associated to the data type τ and $g.\pi$ the value associated to a node v by π. For the object of Fig. 4, the embedding operation $point : \langle \alpha_1 \alpha_2 \rangle \rightarrow point\_2D$ associates the values A, B, C, D, E of type $point\_2D$, suggesting some 2-dimensional coordinates, to the topological vertices. Similarly, an embedding operation $color : \langle \alpha_0 \alpha_1 \rangle \rightarrow color\_RGB$ associates RGB coordinates to the topological faces.

For an embedding operation $\pi : \langle o \rangle \rightarrow \tau$, it is expected that an embedded G-map G verifies that all nodes of any $\langle o \rangle$-orbit share the same value by π, also called π-label or π-embedding. This defines the *embedding constraint* that an embedded G-map G defined on a family $\Pi = (\pi : \langle o \rangle \rightarrow \tau)$ of embedding operations has to satisfy [BALG11]: for each π in Π, each node is π-labelled on $\lfloor \tau \rfloor$ and for all nodes v and w of G if v and w belong to the same orbit of type $\langle o \rangle$, then v and w are labelled with the same π-label (i.e. $v.\pi = w.\pi$).

2.3 Creation of a new modeler

The previous subsections highlight that a modeler is statically defined by its topological dimension and by the profile of considered embedding operations. These static data will intuitively be the first inputs that should be entered by using our editor, named *JerboaModelerEditor*, whose main function is the creation of a new modeler. The first step for creating a new modeler is thus to give its name and its dimension. The second step is the specification of all embeddings through a *pop-up*, that requires informations such as the embedding name, the associated orbit type and the data type. For execution issues, data types are given in terms of built-in or user-defined data types of the underlying programming language (in our case, Java).

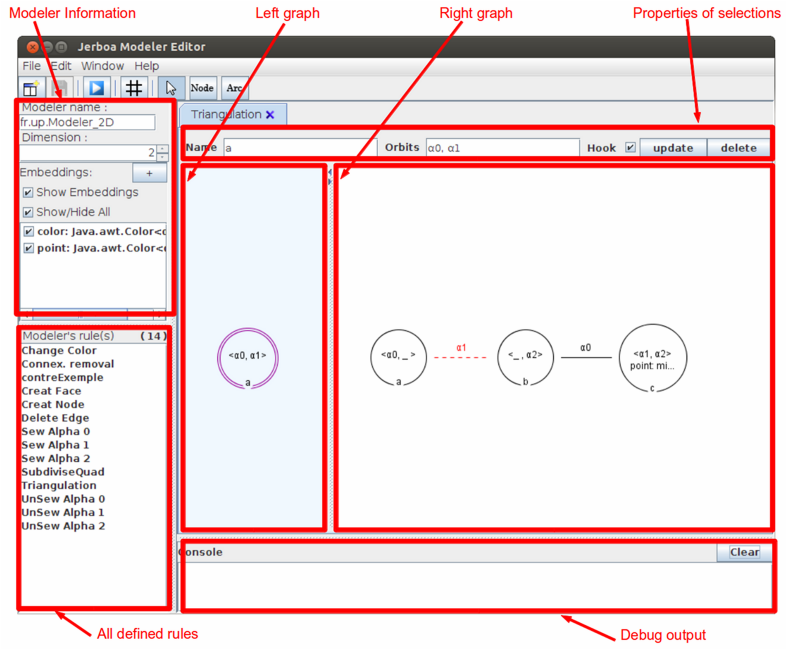


Fig. 5. Main interface of JerboaModelerEditor

The main *JerboaModelerEditor* window is organized in several parts (see Fig. 5). The upper leftmost box presents the previously described core informations as the modeler name, its dimension and the list of embeddings. The other parts of the window are dedicated to the description of operations. The lower leftmost box contains the current list of available operations (specified as rules) for the modeler under construction. The central boxes are used for the edition of a rule. More details are given in the next section devoted to the treatment (edition, verification, application) of rules.

3 Operations on G-Maps Defined as DPO Rule Schemes

3.1 Formal framework

The formal background of Jerboa rules is the DPO approach [EEPT06], more precisely, the DPO approach of [HP02] devoted to labelled graphs, extended with variables in the style of [Hof05]. Roughly speaking, a DPO rule is in the form of a span of inclusions $L \hookleftarrow I \hookrightarrow R$, where L is the pattern to be matched in a graph G under modification and R is the new pattern to be inserted in the transformed graph, I being the rule interface, the common part of L and R. Our concrete syntax (see Fig. 6) contains only two graphs: the left-hand (resp. right-hand) side corresponds to the graph L (resp. R), the graph interface I being implicitly defined as the intersection of left-hand and right-hand sides. The mapping of L in the graph G to be rewritten is usually called the *match morphism*.

Roughly speaking, the application of rule schemes with variables can be sketched as follows: the user gives first a match morphism from the left graph structure, i.e. the underlying unlabelled graph, of the rule towards to the graph structure of G. From there, the variables are instantiated in order to compute a classical DPO rule without variable and a match morphism applicable on G. To define topological-based geometric operations, two kinds of variables are used: *topological variables* to match topological orbits [PACLG08], and *embedding variables* to match geometric or physical embedded informations [BALG11]. Thus the application of Jerboa rules is defined in three passes: first, instantiation of topological variables, then, instantiation of embedding variables, and finally application of a basic rule. However, we will see that Jerboa rules are applied in a one-pass process.

3.2 Editing topological rule schemes

To enable the design of operations generic in both terms of size and of orbit nature, rules include *topological variables* [PACLG08]. The latter are denoted by an orbit type and are instantiated as particular orbits of same type. In Fig. 6, the left node of the rule is labelled with the face orbit type $\langle \alpha_0 \alpha_1 \rangle$, and thus allows the user to match any face of an object. Its instantiation by the a node (resp. g node) of the G-map of Fig. 4 gives rise to a triangular (resp. quadrilateral) face.

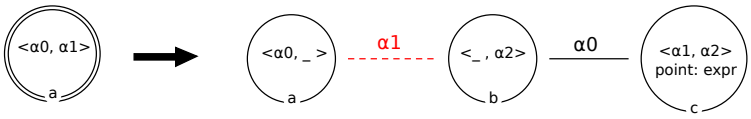
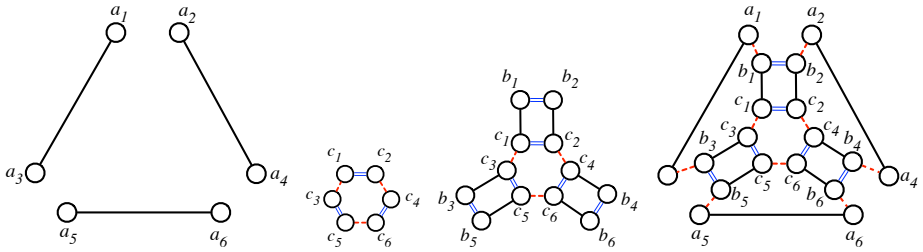


Fig. 6. Jerboa Rule of the triangulation for 2G-map



(a) a node (b) c node (c) b and c nodes (d) Right nodes

Fig. 7. Topological instantiation

All nodes of the right-hand side carry an orbit type of the same length, but that can differ by deleting or relabelling arcs. Thus, all considered patterns are isomorphic up to some orbit correspondence. More precisely, by using the special character '_', topological variables allow us to delete arcs. The left hand-side node $a : \langle \alpha_0 \_ \rangle$ combined with the face attached to the node a of the G-map of Fig. 4 allows us to build the G-map on Fig. 7(a): α_0-labelled arcs are preserved while α_1 ones are deleted. Thus, edges of the matched face are disconnected. Similarly, arcs can be relabelled. The instantiation of the node $c : \langle \alpha_1 \alpha_2 \rangle$ with the face attached to the node a of the G-map of Fig. 4 leads to the G-map of Fig. 7(b), where α_0-labelled arcs are relabelled to α_1 ones, and α_1 arcs to α_2 ones. Thus, a dual vertex of the matched face is added. Afterward, once each node is instantiated by an orbit, all these orbits are linked together. More precisely, each arc of the right-hand side graph is duplicated in several arcs linking instantiated nodes sharing the same index. For example, the instantiation of the α_0-arc linking nodes b and c of the right-hand side graph leads to 6 α_0-arcs linking b_i and c_i nodes in G-map on Fig. 7(c). Thus, edges are added around the dual vertex. Finally, the instantiation of the right-hand side of the rule of Fig. 6 on a triangle face gives rise to the G-map of Fig. 7(d).

Some left nodes of Jerboa rules are denoted with a double circle, and called *hook* nodes. Thus, in Fig. 6, the left node a is an hook. To apply a Jerboa rule, each hook node of the rule scheme must be map to a node of the target object. From an association between hook nodes and target graph nodes, the Jerboa library automatically computes the match morphism (if it exists).

3.3 Editing geometrical rule schemes

The second rule edition step concerns the embedding counterpart of operations. In Fig. 6, since *point* is an embedding operation, the left node a of the rule is implicitly labelled with the *a.point* embedding variable. When instantiating the topological variable by a particular orbit, embedding variables are duplicated as many times as the size of the considered orbit. Thus for example on Fig. 7, all instantiated nodes a_1 to a_6 are implicitly labelled with embedded variables $a_1.point$ to $a_6.point$.

On the right-hand side of rules, embedding variables are put together in expressions built upon user-defined operations on embedding data types and some predefined iterators on the orbits. In Fig. 6, the full expression of point embedding of c node is not detailed, but it can be defined with the following expression $\Phi(point_{\langle\alpha_0,\alpha 1\rangle}(a))$ where $point_{\langle\alpha_0,\alpha 1\rangle}(a)$ collects in a set all geometric points associated to the nodes belonging to the $\langle\alpha_0,\alpha 1\rangle$ face orbit of a and Φ simply computes the barycenter of a set of geometric points. c_1 to c_6 nodes are labelled with embedded expressions $\Phi(point_{\langle\alpha_0,\alpha 1\rangle}(a_1))$ to $\Phi(point_{\langle\alpha_0,\alpha 1\rangle}(a_6))$. Thus, c_1 to c_6 nodes are labelled with a common value, i.e. the barycenter of the matched face. Moreover, right nodes without any associated embedding expressions (like nodes a and b of Fig. 6) either preserve matched embedding values or inherit from their embedding orbit. For instance, the right hand side a node inherits from the point embedding of left hand side a node: each a_i node of Fig. 7(d) keeps its initial point embedding (as collected by the match morphism). Since the b node is α_1-linked with the a node in Fig. 6, it belongs to the same $\langle\alpha_1\alpha_2\rangle$ vertex orbit and inherits from the point embedding of a node. More precisely, each b_i node of Fig. 7(d) inherits from the value of the point embedding of the corresponding a_i node. Thus, as a result, point embeddings of the matched face are preserved and the point embedding of the new vertex added by c is set to the barycenter of the face.

As previously explained, the Jerboa editor illustrated in Fig. 5, allows the user to graphically edit left and right rule scheme patterns. In addition, an informative toolbar summarizes the current selection and offers buttons to create/modify the rule scheme, especially topological and embedding labels. Nonetheless, it allows to change many settings like hide/draw the alpha link name, color convention and so on. Finally, the editor can generate an image from the current rule (Jerboa rules of this article were created with the SVG export).

3.4 Syntax checking

The main advantage offered by this editor is an on-the-fly verification of the rule sheme, avoiding compilation errors and debugging. At each rule modification, the editor checks simple syntactic properties numbered from (i) to (v) thereafter. (i) usual lexical analyses are applied to identifiers and expressions annotating rules. (ii) all labels carried by arcs are of form α_i with i an integer less than or equal to the modeler dimension. (iii) all orbit types labelling nodes are of the same size, that is, contain exactly the same number of (α or _)-elements; thus, all nodes instantiated patterns are isomorphic only up to relabelling and deletion of arcs. (iv) each hook node should be labelled by a full orbit type, that is, by $\langle\alpha_i \dots \alpha_j\rangle$ without any '_' occurring in it. Indeed, at the instantiation step, the targeted orbit is built by performing a graph traversal from a selected node, and guided by the arc labels given in the full orbit type. (v) lastly, each connected component of the left hand side of the rule should contain exactly one single hook. This last condition allows us to compute an unique match morphism (if it exists) or to trigger a warning message.

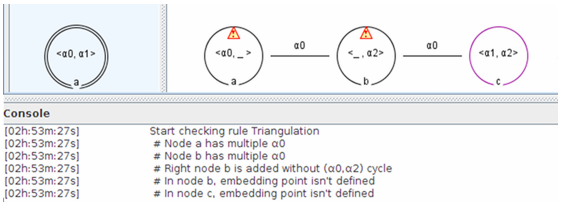


Fig. 8. Topology verification by JerboaModelerEditor

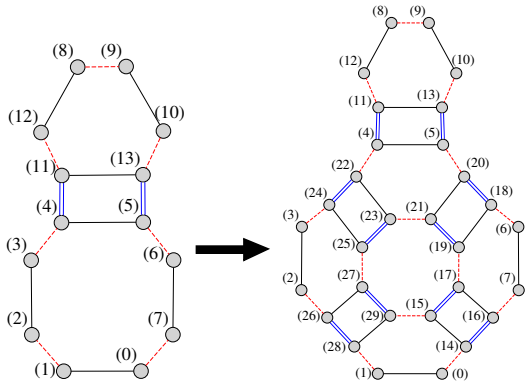
Beyond these basic syntactic properties, we showed in [PACLG08,BALG11] that some additional syntactic properties on rules can be considered to ensure object consistency preservation through rule application, that is, to ensure that resulting objects satisfy both the topological consistency and embedding constraints given in Sections 2.1 and 2.2. For example, the preservation of the adjacent arc constraint can be ensured by checking for each node how many arcs are linked to. First, all removed and added nodes should have exactly $n+1$ arcs respectively α_0 to α_n-labelled, including arcs that are implicitly handled by orbit types attached to them. In Fig. 6, the b node has two explicit arcs linked to it and labelled resp. by α_0 and α_1 and one implicit arc linked to it and labelled by α_2, provided by its topological type ($\langle \_\alpha_2 \rangle$). Secondly, preserved nodes must have the same links on both sides. In Fig.6, on the left, the a node, typed by $\langle \alpha_0\alpha_1 \rangle$, is given with two implicit arcs, resp. labelled by α_0 and α_1, and on the right, the a node, being typed by $\langle \alpha_0 \_ \rangle$, keeps an implicit α_0-arc and is linked to a new α_1-arc.

To preserve the embedding constraint, Jerboa checks that each embedding orbit of Jerboa rule carries one and only one embedding expression. But the instantiation of topological variables produces several embedding expressions which do not necessarily have the same value. Jerboa computes the first one and ignores the following ones: by default, the computed value is attached to all nodes belonging to the embedding orbit.

By lack of space, other advanced syntactic conditions are not more detailed. All syntactic conditions are on-the-fly checked by the JerboaModelerEditor. The editor shows the encountered errors directly in the graph and identifies the incriminated nodes. In Fig. 8, nodes `a` and `b` are decorated by warning pictograms. All errors are detailed in the console: the right-hand side a node has two α_0 links and loses an α_1 link, and so on.

3.5 Jerboa rule application

The Jerboa rule application has been sketched by means of the three following steps: the instantiation of topological and embedding variables, and the application itself. In practice, these steps are done at once by the rule application engine encompassed in the Jerboa library. Let us emphasize that this rule application engine is unique and is able to apply any rule of any generated modeler.



(a) Application

#	a	→	a	b	c
1	0	→	0	14	15
2	7	→	7	16	17
3	6	→	6	18	19
4	5	→	5	20	21
5	4	→	4	22	23
6	3	→	3	24	25
7	2	→	2	26	27
8	1	→	1	28	29

(b) Inner matrix

Fig. 9. Rule application and inner structure of the triangulation

First the engine instantiates the topological variables and computes the match morphism if possible. For example the application of the Jerboa rule of Fig. 6 on the left object[3] of Fig. 9(a) allows to complete the first column of the inner matrix given on Fig. 9(b). For that, once the user has mapped the hook node a of the rule scheme to the node `(0)` of the object, one line is created in the matrix for each node of the orbit $\langle\alpha_0\alpha_1\rangle$ of `(0)`. For a rule with multiple hooks, this step can fails and the engine triggers an exception.

Secondly, the right part of the inner matrix is completed in accordance with the right-hand side of the rule. As shown on Fig. 9(b), names of preserved nodes are copied, and names of added nodes are created. Then, the created part of the rewritten object is computed. This part corresponds to the two last columns of nodes b and c on Fig. 9(b). Each arc of the rule is duplicated for each inner matrix line. Thus, the α_0-arc between b and c nodes produces α_0-arcs between `(14)` and `(15)` nodes, ..., between `(28)` and `(29)` nodes. First, each arc of the instantiated orbit of the hook is translated for each inner matrix columns up to relabelling or removing on added nodes. The α_1-arc between `(0)` and `(7)` is α_2 relabelled between `(14)` and `(16)`, and between `(15)` and `(17)`, the α_0-arc between `(7)` and `(6)` is not added between `(16)` and `(18)`, and α_1 relabelled between `(17)` and `(19)`, ..., the α_0-arc between `(1)` and `(0)` is not added between `(28)` and `(14)`, and α_1 relabelled between `(29)` and `(15)`.

Before further topological modification, the new embedding values are computed. Indeed, evaluations of embedded expressions depend on initial embedding values, but also on the topological links in the initial object. We use the fact that all orbits of same embedding type necessarily contain nodes sharing the same embedding value (with respect to this considered embedding function). The engine computes new embedding values but does not yet replace them in the nodes

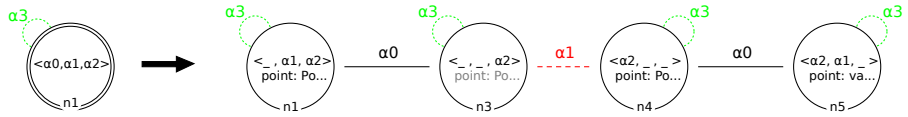
[3] Note that *graphviz* exportation is used to generate from our generic viewer that uses barycentric coordinates as exploded view.

to avoid any corruption of the next embedding value computations. Instead, those values are memorized in a private buffer of the engine. The treatment of topological variables is completed by relabelling the preserved nodes with new embedding values and connecting them to the added nodes with appropriate embedding values.

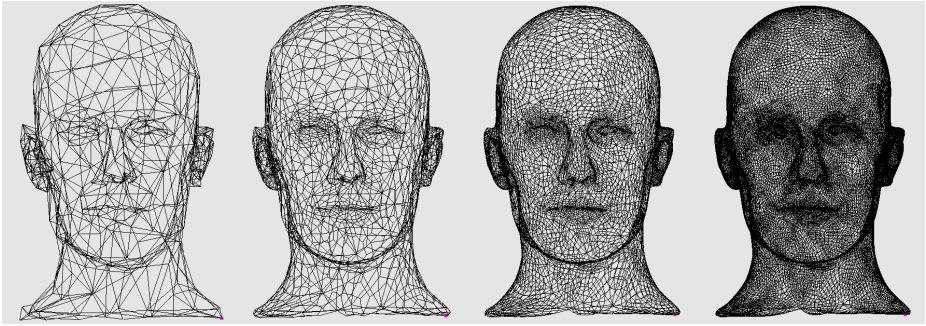
This way of processing Jerboa rule application calls for a comment on the management of the embedding values. The case of the merge of two embedding orbits could lead to a non-deterministic choice between the two original embedding values. To avoid such a situation, the editor asks for a unique embedding expression, that is sufficient to ensure the embedding expression constraint.

4 Discussion

Examples. We briefly introduce some non-trivial operations. First, we propose the Catmull-Clark operation used to smooth face by performing some face subdivision mechanisms. The rule for a 3D modeler is described on[4] Fig. 10(a) and Fig. 10(b) illustrates the successive applications of this rule on a mesh.



(a) Rule obtained

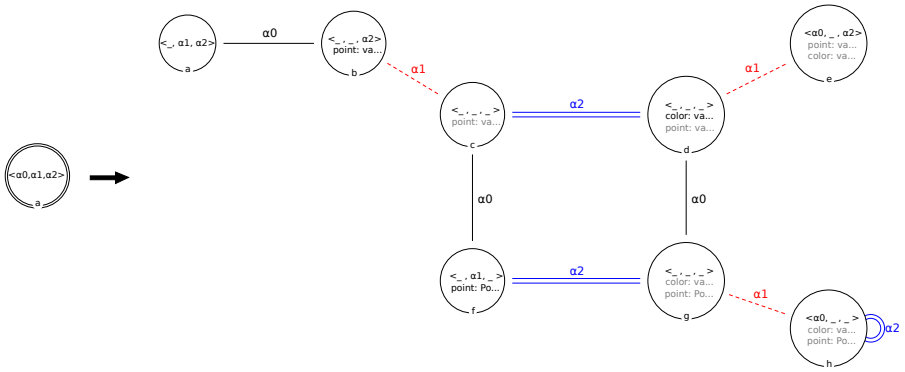


(b) Successive application of the rule

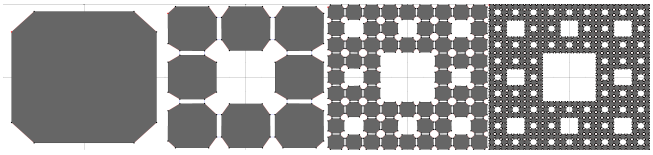
Fig. 10. Illustration of the Catmull-Clark smoothing operation

Second, the *Sierpinski carpet* is a fractal on a 2D surface (in higher dimension, this fractal is called Menger sponge). The rule, presented on Fig. 11(a), matches a face and produces eight faces from it. Fig. 11(b) shows successive applications of this rule on a simple 2D face.

[4] By lack of space, complete embedding expressions are omitted.



(a) Obtained rule



(b) Successive application of the rule

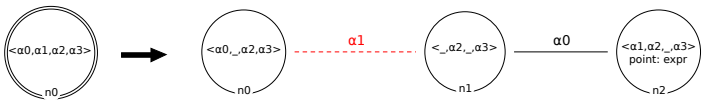
Fig. 11. Illustration of the Sierpinski carpet

Performance and evaluation. In this section, we present a comparison between Jerboa and another similar library. As far as we know, only few libraries offer the ability to manipulate G-map (Moka [Mod] and CGoGN [KUJ+14]), and most of them are designed in a similar way. We choose the well-known and established Moka modeler and compare the performances of both Jerboa and Moka by benchmarking them on two operations that are already fully optimized in Moka.

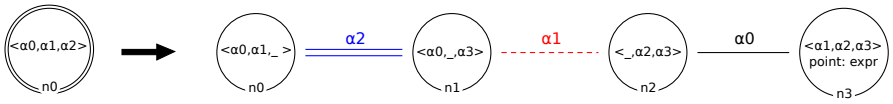
The first operation (see Fig. 12(a)) is a generalization of the basic face triangulation in which all faces of a connected component are triangulated at once. The main difference with the previous version (Fig. 6) is that the left-hand side matches more dimension and the right-hand side manages the third dimension. The second operation is volume triangulation, i.e. the subdivision of any (convex) volume into linked pyramids whose common vertex is the volume barycenter. Fig. 12(b) presents the Jerboa rule in which the left-hand side matches a full volume. In the right-hand side, the node `n3` represents the center of the subdivision while nodes `n2` and `n3` represent sides of inserted faces. Let us notice that the border faces remain unmodified.

Tests have been executed on a Core i7-2600 – 3.4GHz with 8GB of RAM, under Linux/Ubuntu 12.04 LTS system with JDK 7 of Oracle. We computed the average time of operation execution in both modelers with a single embedding of geometric points[5] to represent 3D objects. We used the same objects for both libraries, with various sizes (from 4 nodes to 3 millions nodes). To summarize,

[5] By default, a Moka modeler is only defined by this unique embedding



(a) Triangulation of all faces at once



(b) Triangulation of volume

Fig. 12. Two Jerboa rules of triangulation operations

Jerboa shows better performances than Moka for the triangulation of all faces at once whereas Moka is better considering the volume triangulation. There is no better optimization than tuned optimization carefully performed by a developer, and Jerboa is implemented in Java language that is recognized to be slower than C++: this can explained why generally, Moka has still better performances than Jerboa. The better performances of Jerboa for the face triangulation operation is due to the fact that the Moka developer has intensively reused existing codes.

To conclude this section, we wish to emphasize the ease of developing new operations using Jerboa. Classically, the development of a modeler's operation is done by hand-coding with all induced problems: debugging, verification steps on customized objects, and so on. While developers usually mix inside the same code topological and embedding considerations, developing with Jerboa imposes a clear separation of topological and embedding aspects, and even more, requires that static parts are declared before designing operations. Thus, as regards the code development, using Jerboa brings two clear advantages: a significant gain in time and a high level of code quality. For instance, for the two operations discussed above and given in Fig. 12, we only took half a day.

5 Conclusion and Future Works

This article introduces a novel tool set dedicated to rule-based geometric modeling based on G-maps. This tool set includes the JerboaModelerEditor, that allows a fast characterization of any new dedicated modeler and a fast design of its operations in a graphical manner, assisted by static verification steps. When the design is over, the Jerboa library produces a full featured modeler kernel that can be used in a final application. Moreover, the produced modeler is highly reliable as generated rules take benefit from graph transformation techniques ensuring key consistency properties.

Jerboa has been successfully used in other works, especially for the adaptation of L-Systems with G-map or in an architecture context (see Fig. 1). These experimentations allowed us to identify some required features for the next version of Jerboa, as the need of stronger verification mechanisms of the embedding

expressions or the need of a *rule script* language in order to apply several rule schemes accordingly to some strategies.

References

[BALG11] T. Bellet, A. Arnould, and P. Le Gall. Rule-based transformations for geometric modeling. In *6th International Workshop on Computing with Terms and Graphs (TERMGRAPH)*, Saarbrucken, Germany, 2011.

[BH02] L. Baresi and R. Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *First International Conference on Graph Transformation (ICGT)*, volume 2505 of *LNCS*. Springer, 2002.

[BPA+10] T. Bellet, M. Poudret, A. Arnould, L. Fuchs, and P. Le Gall. Designing a topological modeler kernel: a rule-based approach. In *Shape Modeling International (SMI'10)*. IEEE Computer Society, 2010.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series Springer, 2006.

[Hof05] B. Hoffmann. Graph transformation with variables. *Formal Methods in Software and System Modeling*, 3393:101–115, 2005.

[HP02] A. Habel and D. Plump. Relabelling in graph transformation. In *First International Conference on Graph Transformation (ICGT)*, volume 2505 of *LNCS*. Springer, 2002.

[KBHK07] O. Kniemeyer, G. Barczik, R. Hemmerling, and W. Kurth. Relational growth grammars - a parallel graph transformation approach with applications in biology and architecture. In *Application of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 5088 of *LNCS*, pages 152–167, 2007.

[KUJ+14] P. Kraemer, L. Untereiner, T. Jund, S. Thery, and D. Cazier. CGoGN: n-dimensional meshes with combinatorial maps. In *22nd International Meshing Roundtable*. Springer, 2014.

[Lie91] P. Lienhardt. Topological models for boundary representation: a comparison with n-dimensional generalized maps. *Computer-Aided Design*, 23(1), 1991.

[Mod] Moka Modeler. XLim-SIC. http://moka-modeller.sourceforge.net/.

[MP96] R. Měch and P. Prusinkiewicz. Visual models of plants interacting with their environment. In *23rd Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH. ACM, 1996.

[PACLG08] M. Poudret, A. Arnould, J.-P. Comet, and P. Le Gall. Graph transformation for topology modelling. In *4th International Conference on Graph Transformation (ICGT)*, volume 5214 of *LNCS*. Springer, 2008.

[PLH90] P. Prusinkiewicz, A. Lindenmayer, and J. Hanan. *The algorithmic beauty of plants*. Virtual laboratory. Springer-Verlag, 1990.

[TGM+09] O. Terraz, G. Guimberteau, S. Merillou, D. Plemenos, and D. Ghazanfarpour. 3Gmap L-systems: an application to the modelling of wood. *The Visual Computer*, 25(2), February 2009.

[VAB10] C.-A. Vanegas, D.-G. Aliaga, and B. Benes. Building reconstruction using manhattan-world grammars. In *Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2010.