HAL open science

Inferring topological operations on generalized maps: application to subdivision schemes

Romain Pascual, Hakim Belhaouari, Agnès Arnould, Pascale Le Gall

▶ To cite this version:

Romain Pascual, Hakim Belhaouari, Agnès Arnould, Pascale Le Gall. Inferring topological operations on generalized maps: application to subdivision schemes. Graphics and Visual Computing, 2022, 10.1016/j.gvc.2022.200049 . hal-03491856v2

HAL Id: hal-03491856
https://hal.science/hal-03491856v2

Submitted on 27 May 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

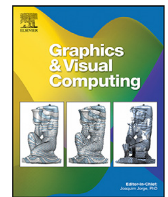
L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Graphics and Visual Computing 6 (2022) 200049

Contents lists available at ScienceDirect

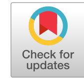
Graphics and Visual Computing

journal homepage: www.elsevier.com/locate/gvc



Technical section

Inferring topological operations on generalized maps: Application to subdivision schemes☆



Romain Pascual [a,*], Hakim Belhaouari [b], Agnès Arnould [b], Pascale Le Gall [a]

[a] Laboratoire Mathématiques et Informatique pour la Complexité et les Systèmes (MICS), CentraleSupélec, Université Paris Saclay, France
[b] Laboratoire XLIM UMR CNRS 7252, Université de Poitiers, France

ARTICLE INFO

Article history:
Received 3 December 2021
Received in revised form 9 May 2022
Accepted 10 May 2022
Available online 14 May 2022

Keywords:
Topology-based geometric modeling
Subdivision schemes
Operation inference
Inference from examples
Topological operation
Computational topology

ABSTRACT

The design of correct topological modeling operations is known to be a time-consuming and challenging task. However, these operations are intuitively understood via simple drawings of a representative object before and after modification. We propose to infer topological modeling operations from an application example. Our algorithm exploits a compact and expressive graph-based language. In this framework, topological modeling operations on generalized maps are represented as rules from the theory of graph transformations. Most of the time, operations are generic up to a topological cell (vertex, face, volume). Thus, the rules are parameterized with orbit types indicating which kind of cell is involved. Our main idea is to infer a generic rule by folding a graph comprising a copy of the object before modification, a copy after modification, and information about the modification. We fold this graph according to the cell parametrization of the operation under design. We illustrate our approach with some subdivision schemes because their symmetry simplifies the operation inference.
© 2022 The Author(s). Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

In interactive modeling, the possibility to effortlessly create dedicated operations is a long-craved ambition. These operations aim at simplifying the production of domain-specific objects. Geometric modelers [1–3] usually enable the user to hand-code new operations through an API, adapting a generic tool into a dedicated one. Such solutions allow constructing geometric objects across many application domains such as computer-aided design, architecture, or animation movies.

Our ambition is to deduce the general operation from a single representative instance. Indeed, domain experts usually experiment on simple instances when the target object is complex, to the point that they often can characterize an operation using a well-chosen use case. Besides, inferring operations from an instance reduces the cumbrous nature of implementing new operations, coping with the unfamiliarity of domain experts with the tool's implementation. We aspire to exploit the intuition experts can provide to infer operation in the specific case of topological modifications on meshes.

Our approach lies in the field of topology-based geometric modeling [4]. An object consists of a topological structure, i.e., its *topological cells* (volumes, faces, edges, and vertices) and geometrical aspects. All the non-topological information is called embedding information and may encode the position of vertices, the curvature of edges, or the texture mapped onto faces. We exploit the formalism of *generalized maps*, or G-maps [4–6]. Generalized maps are similar to (combinatorial) maps, equivalent to graph rotation systems [7,8] in 2D. This model has the main benefit of being homogeneously defined in all dimensions. The standard construction of G-maps exploits permutations on a set of darts. Here, we will represent them as graphs, similarly to the approach of [9,10].

Since objects are formalized using graphs, modeling operations can be studied as rules in the framework of graph transformations [11–13]. Intuitively, a rule is written $L \to R$ and allows to transform object L into object R within a more general context. Rule-based languages have already been studied to apply predefined operations on predefined objects, such as in L-system languages [14,15]. In [16], the authors used a graph-transformation-like approach to define surface subdivision algorithms. Rule-based dedicated languages allow for a user-friendly description of modeling operations and a generic manipulation of these operations via a dedicated rule application engine. In [10], the authors showed how to design a modeler kernel as a rule application engine in the context of graph transformation rules applied to generalized maps. These rules separately handle the topological modification and their geometric counterpart. Besides, as operations should produce a well-formed object when applied to a well-formed object, rules are subject to syntactic conditions that ensure the preservation of the topological [17,18]

☆ This article was recommended for publication by L. Barthe.
* Corresponding author.
E-mail address: romain.pascual@centralesupelec.fr (R. Pascual).

https://doi.org/10.1016/j.gvc.2022.200049
2666-6294/© 2022 The Author(s). Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

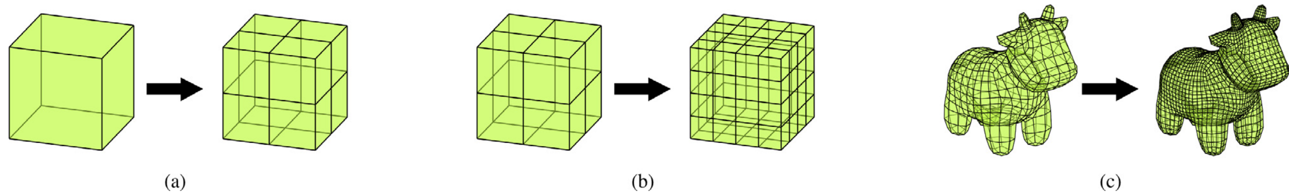


Fig. 1. Operation of quad subdivision: application to a cube (a), iterated application to the cube (b), and application to a quadmesh (c).

and the geometric consistency [19]. Operations can be parameterized by topological cells. Topological cells, and more generally orbits, encode rule parameters to offer a compact and expressive graph-based language to design modeling operations. These extended rules, called rule schemes, express transformations valid for all possible shapes of a given orbit type, providing the desired generalization. Applying a modeling operation is achieved by instantiating a rule scheme to a concrete rule based on the object under modification.

We wish to provide a tool that can infer the topological part of an operation from a representative example. The operation should be applicable to similar objects. The user specifies an initial object A, modifies it, and provides the final object B. We offer to deduce the operation that transforms A directly into B. For example, the subdivision of the cube, illustrated in Fig. 1(a), is a transformation where object A is the cube before subdivision and object B is the object after subdivision. Since we want an operation applicable in a broader context than simply on the object A, we infer operations generic up to a user-specified topological cell. For instance, we can require that the inferred operation for the subdivision described in Fig. 1(a) is general up to a surface. The inferred operation can be applied again on the resulting object (see Fig. 1(b)). The operation also allows modifying different objects, such as the quad mesh of a cow depicted in Fig. 1(c). The present paper focuses on inferring operations for subdivision schemes to exploit their regularity. Indeed such transformations rely on local modifications applied similarly on the whole object, creating many symmetries. These symmetries simplify the inference of the operation.

Our main contribution is an algorithm that infers topological operations described as rules. The algorithm takes as input two objects described as G-maps, a mapping of the element preserved by the operation and an orbit describing the generality of the operation to be inferred. We prove the algorithm's correctness in the sense that applying the inferred operation on the before instance yields the after instance. We also illustrate our algorithm by inferring standard subdivision schemes and applying the rules to various examples.

This paper presents an algorithm to conceive topological modeling operations without any knowledge of generalized maps or graph rewriting. We will illustrate our approach with the help of subdivision schemes. We provide context for operations inference by presenting other approaches that deduce modeling operations in Section 2. We recall the formalism of generalized maps in Section 3 and give some insights on graph transformation rules used for modeling operations in Section 4. We present the topological folding algorithm used for reconstructing rule schemes in Section 5 and illustrate it with examples and counter-examples in Section 6. We explain how to gather both instances in a single graph used as input for our algorithm in Section 7. We analyze the topological folding algorithm in Section 8. We present a validation of our approach in Section 9 with the illustration of several subdivision schemes. We discuss some practical side-effects of our inference mechanism in Section 10. The complete proof of correctness of the presented algorithm is provided in Appendix A.

2. Related works

There are several categories of previous works that relate to our contribution. They mainly belong to procedural modeling and, more generally, the inference of rules, scenes, and modeling operations.

Procedural modeling and inverse procedural modeling. Procedural modeling refers to techniques used in computer graphics to derive a model from a ruleset. These techniques avoid manually editing objects, proving fruitful to model regular objects, i.e., objects with many repetitions of sub-patterns. Procedural modeling techniques have been exploited to generate plants [14], terrain [20], buildings [21], or cities [22]. Since these techniques could not guarantee an output faithful to the designer's idea, they were extended to inverse procedural modeling. Thanks to machine learning, these new approaches try to discover the correct parameterized rules and values. Inverse procedural modeling techniques have proven successful in most of the domains where procedural modeling was used, namely trees [23], building facades [24], weather simulation on urban models [25], virtual worlds [26], and texture modeling [27]. In such approaches, the set of possible rules describes a specific domain, and the operation inference is tailored to this domain. Conversely, the approach that we will present in this article builds topological operations that allow for editing objects regardless of the application field. Indeed, our rule-based approach discovers the correct rule parameterized by topological information and not by domain-specific values.

L-systems. In [28], the authors present the definition of rules to be the "key challenge of procedural modeling" and use a clustering approach to construct the rules and parameters of a parametric context-free L-system, given a vectorial 2D image. They capitalize on L-systems, introduced to describe plant cells' behavior and, in particular, their growth processes [29]. Such rewriting systems build objects by recursively applying production rules from an axiom word until a stop condition is met. The produced string is transformed into a geometric object via an interpretation [30, II.6, III.5]. More recently, [31] extended the work of [28] with deep learning techniques for the detection of elements and a derivation tree for the retrieval of the rules. In these works, the emphasis is put on the generation of scenes while we focus on the inference of operations. L-systems have been used to represent the refinement operation for subdivision curves [15], where the authors show how to infer an L-system from the subdivision matrix. However, L-systems are essentially geometric interpretations of words, thus inherently equivalent to string rewriting. Therefore, they are ill-suited for working on surfaces and volumes. Graph rewriting extends string rewriting, allowing to find an occurrence of a graph pattern and replace it with another pattern. In a sense, G-map rewriting represents a more general approach than L-systems. Therefore, the ambition of the present article can be understood as a generalization of [32] from L-systems to graph rewriting.

Reevaluation. To avoid cumbersome implementation, some modelers support the definition of modeling operations through the recording of a sequence of already existing operations. The reevaluation [33,34] of the sequence provides a solution to apply this new operation via a specific naming of the modified entities [35]. Thus the reevaluation method allows for modifying similar objects, i.e., objects with the same naming of entities. However, the constructed operation is often not an optimized solution to define the modification because every step of the sequence is reproduced faithfully.

Neural networks and geometrical approaches. Recently, [36] offered automatic generation of geometric operations. This approach takes the Loop subdivision scheme [37] as the atomic operation for refinement. A neural network is then used to learn the geometric values for the subdivision. Therefore, the approach produces the result from an initial object through the direct computation of the targeted geometry. This construction is dual to ours as we focus only on the topology. Nonetheless, our generalization power is broader as the inferred operations do not assume a fixed topology. For instance, we can reconstruct any subdivision scheme.

Inference of graph transformation rules. Our approach exploits a domain-specific language within graph transformations for topology-based geometric modeling. We offer an algorithm to reconstruct modeling operations from an example. Similar ideas were used in [38] to reconstruct a graphical modeling environment for domain-specific language using yED. In [39], the authors used graph transformations as a learning mechanism to detect and fix bugs in Javascript programs. A rule-based definition of geometric modeling operation has been used in [40] to predict operations. Here, we use a particular formalism: generalized maps together with a domain-specific language. Thus, we infer graph transformation rules dedicated to geometric modeling. However, we do not assume a given set of rules and retrieve more general operations.

Inference of modeling sequences in constructive solid geometry. In [41], the authors infer the sequence of modeling operations as a sequence of sketches, extrusions, and boolean operations. It extends previous works such as [42,43] or [44] using constructive solid geometry (CSG). These works retrieve a CSG tree to obtain a specific object. The deduced tree yields an exact object construction but does not endow a definition of operations. One could easily use these techniques to build the first iteration of a subdivision scheme. However, the obtained tree would not give a solution to build the successive iterations of the subdivision scheme.

3. Generalized maps

Boundary representation of objects relies on topological data structures called edge-based models [45] that encode a cell subdivision of the geometric object. Some data, usually called embedding, are attached to the object's cells to define its geometry. Generalized maps are defined as a formalization of these edge-based models to represent non-oriented (quasi-)manifolds in any dimension [4–6]. In [10], the model of generalized maps was mainly chosen for its homogeneity in manipulating dimensions, i.e., because the manipulation of a subdivision does not depend on its dimension. Here, generalized maps homogeneity allows searching for regularities along all dimensions via a simple graph traversal.

In this section, we will first give a graph-based definition of generalized maps and provide some intuition about how it describes the topological structure of an object (Section 3.2). Then, we will describe how to retrieve cells using orbits in Section 3.3. Finally, in Section 3.4 we will sketch the construction of embedded G-maps.

3.1. Vocabulary

In this section, we will introduce several notations. Please note that some elements may have the same name in different communities but we will take care of using distinct words.

(a) We call *vertex* a 0-cell and *edge* a 1-cell. For instance, we can talk about the vertices and edges of the stacked cube given in Fig. 3(a).
(b) We reserve the terms *node* and *arc* to refer to the constitutive elements of a graph, in agreement with algebraic graph rewriting [11,12] used to model operations.
(c) We call *darts* and *links* the elements of a generalized maps, following the combinatorial definition of [4]. Therefore we say that the representation of the stacked cubes with a G-map in Fig. 3(e) contains 96 darts, 48 0-links, 48 1-links, 48 2-links, and 88 3-links.

The combinatorial (c) and graph-based (b) approaches to G-maps are strictly equivalent, but we use graphs to exploit graph rewriting techniques. We will preserve this vocabulary throughout the article to better indicate which point of view should be considered. Thus, we will be referring to the geometric object when using the words 'vertex' and 'edge'; its representation as a G-map with the terms 'dart' and 'link'; or the elements of a rewriting rule with the expressions 'node' and 'arc'.

3.2. Topological structure

We rely on a graph-based definition of generalized maps (see [18]), enabling the expression of modeling operations as rules. Rules simplify the design of operations and alleviate their implementation, provided that a rule application engine exists. Moreover, our algorithm relies on a traversal of the G-map, expressed as a graph algorithm.

In this article, we call *graph* a classical undirected graph, possibly with parallel arcs and loops. We write $G = (V, E)$ for the graph with V as the set of nodes and E as the set of arcs. More precisely, we will use arc-labeled graphs. An arc labeled with i will also be called an i-arc. When two nodes u and v are linked by an i-arc, we write $u \overset{i}{\bullet\!\!-\!\!\bullet} v$. We will label arcs with dimension to encode neighboring relations between the object's sub-parts.

The combinatorial definition of G-maps [4] exploits a set of involutions $I_1, \ldots, I_n$ on a set of darts D. We consider each involution I_i as a symmetric relation over D, i.e., a subset of $D \times D$. Therefore, we can translate the structure $\langle D, I_1, \ldots, I_n \rangle$ into a graph $(D, I_1 \cup \ldots \cup I_n)$. Each dart is considered as a node of the graph, while each involution I_i yields the set of i-labeled (undirected) arcs linking these nodes. The final set of edges consists of the union of the I_i's. An illustration of this construction is provided in Fig. 2. Both graphs 2(a) and 2(b) share $D = \{a, b, c, d, e\}$ for the set of nodes have a set of arcs deduced from an involution, respectively drawn in red and blue. The final graph 2(c) has the same nodes and the union of the arcs from the two graphs. Since the combinatorial definition [4] of G-maps sub-scripts involutions by the adequate dimension, we label each arc with the suitable dimension.

Definition 3.1 (*Generalized Map [18]*)**.** A *generalized map* of dimension n, or n-G-map, is a graph $G = (D, \alpha)$ whose arcs in α are labeled on $[\![0, n]\!]$. The nodes of the graph are called *darts*, and its arcs are called *links*.

A graph $G = (D, \alpha)$ labeled on $[\![0, n]\!]$ is a n-G-map if it satisfies the following topological constraints:

Incidence constraint any dart d of G is the source of a unique i-link $d \overset{i}{\bullet\!\!-\!\!\bullet} d'$, for each dimension i in $[\![0, n]\!]$.

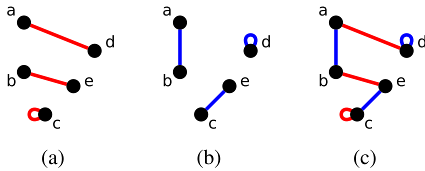


Fig. 2. Representation of the set of the involutions $\{I_1, I_2\}$ over the set $D = \{a, b, c, d, e\}$ as graphs: (a) $I_1 = \{(a, d), (b, e), (c, c), (d, a), (e, b)\}$ represented as the graph (D, I_1); (b) $I_2 = \{(a, b), (b, a), (c, e), (d, d), (e, c)\}$ represented as the graph (D, I_2); (c) $\langle D, I_1, I_2\rangle$ represented as the graph $(D, I_1 \cup I_2)$. Similar examples with permutations can be found in [4, Chap. 2.5.2].

Cycle constraint for all dimensions i and j in $[\![0, n]\!]$ such that $i + 2 \leq j$, any path of length 4 labeled by $ijij$ is a cycle, i.e., the source u and the target v are equal in the path $u \bullet\overset{i}{-}\bullet\overset{j}{-}\bullet\overset{i}{-}\bullet\overset{j}{-}\bullet\, v$.

Definition 3.1 faithfully transforms the involution-based notion of generalized maps [4] in terms of graph vocabulary but retain the combinatorial terminology for the constitutive elements, namely *darts* and *links*. In particular, the graph is subject to two local constraints that translate the involution properties used in the combinatorial definition. These constraints ensure that a $(i-1)$-cell can split at most two i-cells and guarantee that any two i-cells can only be glued along a cell of dimension $i - 1$. For instance, in a 2D object, three faces cannot share an edge, and faces cannot be glued along a vertex. The formalism of G-maps imposes that the only way to connect two faces is to glue them along a unique edge.

The representation of a geometric object as a G-map can be reconstructed from its decomposition into topological cells of decreasing dimensions. The final elements of the cell decomposition are the darts of the G-map, while the decomposition provides the arcs describing the neighboring relations between the cells. For instance, the recursive subdivision of two stacked cubes (3D object) is illustrated in Fig. 3, from the cubes in Fig. 3(a) to the 3-G-map in Fig. 3(e). In the latter, arcs are colored according to their label: — for $\bullet\overset{3}{-}\bullet$, — for $\bullet\overset{2}{-}\bullet$, — for $\bullet\overset{1}{-}\bullet$, and — for $\bullet\overset{0}{-}\bullet$.

Let us detail the reconstruction of the G-map from two stacked cubes with the help of Fig. 3:

- The object of Fig. 3(a) consists of two volumes sharing a face: a green cube glued to a blue cube via a purple square. Therefore, the object is first split into two volumes as in Fig. 3(b). The topological cells of dimension 3 or 3-cells (the volumes) are now linked along their shared face with green arcs (—). These 3-links encode the neighboring relation between the two entities: they are adjacent volumes.
- From the representation of Fig. 3(b), we iterate the decomposition by splitting the faces in each volume. Each cube yields 8 faces, as illustrated in Fig. 3(c). Any two faces adjacent within the same volume are 2-linked along their common edge with blue arcs (—).
- By iteration on decreasing dimensions, we obtain Fig. 3(d) after the decomposition of dimension 1. In each face of each volume, the edges are disconnected and linked with red arcs (—) for 1-links.
- Finally, splitting the 0-cells, i.e., the vertices, with 0-links drawn as black arcs (—) ends the subdivision process. The atomic elements obtained after this last decomposition correspond to the darts of the G-map. Loops are added on each dart that misses a link (for each possible dimension) to obtain the proper G-map. For the object of Fig. 3, only the darts not belonging to the shared purple face are missing some links, and 3-loops are added to all of them. The graph displayed in Fig. 3(e) provides the 3-G-map representation of the two stacked cubes. In the sequel, loops will sometimes be omitted to simplify the figures.

Note that splitting the cubes yields two faces for the initial purple face. The top one corresponds to the purple face from within the green cube, while the bottom one represents the purple face from within the blue cube. The 3-links characterize that the two faces are actually the same face in the global object. If we extend this intuition to all dimensions, we understand that a dart encodes part of a vertex, an edge, a face, and a volume simultaneously. For instance, let the pointed purple dart of Fig. 4 be called e. This dart is part of the green volume. In the green volume, it corresponds to the purple face. In this face, it belongs to the right-side edge. Within this edge, it encodes the front vertex. Each of these topological cells is respectively illustrated in Figs. 4(a) to 4(d) and retrieved via orbits that we present next.

3.3. Cells and orbits

In the sequel, we consider an n-G-map G. As outlined in Section 3.2, the topological cells (vertices, edges, faces, volumes) of the represented geometric object are not explicitly defined in G but can be implicitly retrieved. Cells can be computed via graph traversals restricted to a subset of dimensions. For instance, the cells incident to the pointed purple dart e are depicted in Fig. 4.

- The 0-cell (the vertex) incident to dart e is given in Fig. 4(a). This cell contains the dart, and every dart reachable by all links except 0-links (i.e., 1, 2, and 3-links), along with the links themselves. This subgraph is written $G\langle 1, 2, 3\rangle(e)$.
- The 1-cell incident to dart e, representing the edge incident to the purple dart is displayed in Fig. 4(b). This cell contains all darts and links gathered in the traversal of G with the all dimensions but 1, when starting from e. The subgraph $G\langle 0, 2, 3\rangle(e)$ corresponds to this edge.
- The 2-cell $G\langle 0, 1, 3\rangle(e)$ incident to e is shown in Fig. 4(c) and represents a face.
- The 3-cell (the volume) incident to e is the subgraph $G\langle 0, 1, 2\rangle(e)$ illustrated in Fig. 4(d).

More generally, the cells correspond to specific cases of orbits.

Definition 3.2 (*Orbit*)**.** An *orbit* of G consists of a subgraph induced by all the darts reachable from an initial dart, only using links from a subset of $[\![0, n]\!]$.

The orbit is written $G\langle o\rangle(v)$ where o is a subset of $[\![0, n]\!]$, or $\langle o\rangle(v)$ when there is no ambiguity on the graph. Such an orbit is said to be of type $\langle o\rangle$ or referred to as an $\langle o\rangle$-orbit.

Retrieving an orbit intuitively provides all darts of G that correspond to a common topological element. Such topological elements encompass cells, such as vertices or volumes, or more restricted elements, such as half faces, half edges, or corners of faces. For example, the orbit $G\langle 0, 1\rangle(e)$ described in Fig. 4(e) represents the face of one volume (also called half face) incident to e. Indeed the face incident to e is the orbit $G\langle 0, 1, 3\rangle(e)$, thus removing the dimension 3 splits the two half faces (the one in the green cube and the one in the blue cube. Since the full graph G of Fig. 3(e) contains a single connected component, it corresponds to the orbit $G\langle 0, 1, 2, 3\rangle(e)$.

3.4. Embedded G-maps

This presentation of generalized maps is only of topological content. Although this article aims at inferring the topological part of a modeling operation, we present briefly how the geometric data are handled on G-maps. More details can be found in [19].

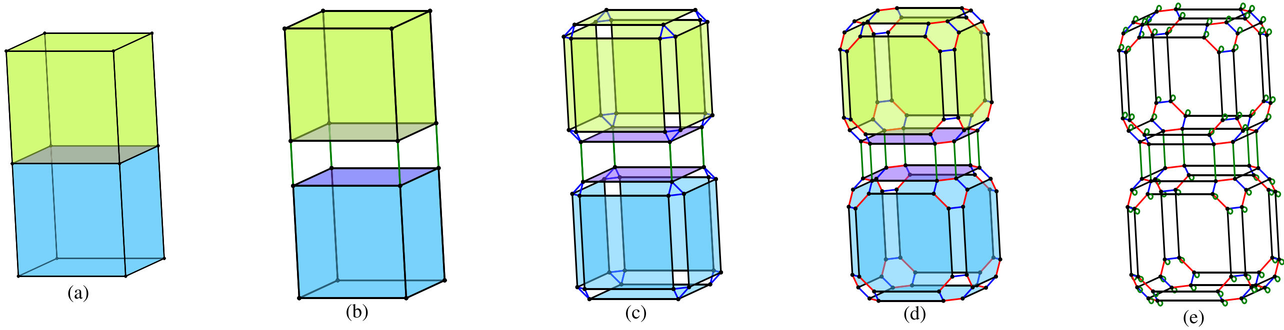


Fig. 3. Topological decomposition of a geometric object in dimension 3: (a) two cubes sharing a face, (b) split on dimension 3, (c) dimension 2, (d) dimension 1, and (e) dimension 0. The graph G is the corresponding G-map (e). Arc color legend: ▬ 3, ▬ 2, ▬ 1, and ▬ 0.

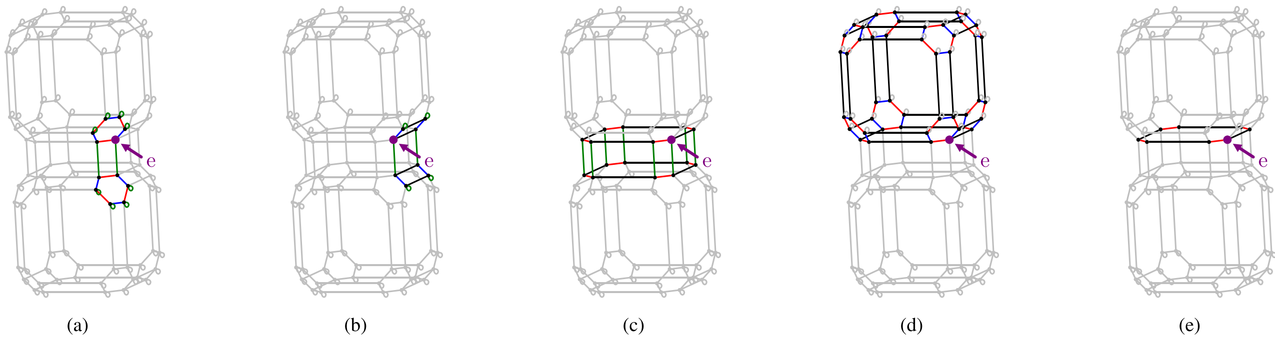


Fig. 4. Orbits incident to the purple dart e in the G-map G from Fig. 3(e): (a) vertex $G\langle 1, 2, 3\rangle(e)$, (b) edge $G\langle 0, 2, 3\rangle(e)$, (c) face $G\langle 0, 1, 3\rangle(e)$, (d) volume $G\langle 0, 1, 2\rangle(e)$, and (e) face of volume $G\langle 0, 1\rangle(e)$.

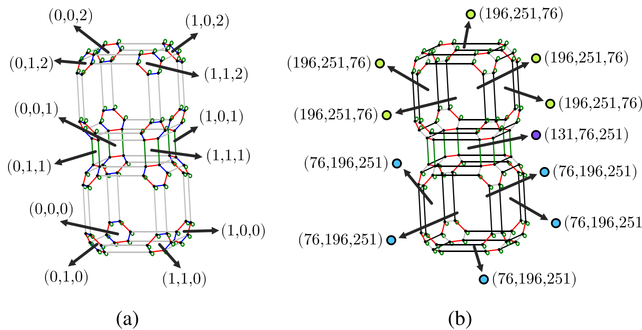


Fig. 5. Embedded representation of the stacked cubes: (a) embedding of the vertices position with $pos : \langle 1, 2, 3\rangle \to Point3D$, both cubes are of length 1; (b) embedding of the faces color with $color : \langle 0, 1, 3\rangle \to RGB$, the colors described by the RGB values are displayed next to the values. Arc color legend: ▬ 3, ▬ 2, ▬ 1, and ▬ 0.

All non-topological information such as position or color is represented via an embedding function that maps each topological cell to its relevant data. In this paper, each vertex (0-cell) has a position, and each face (2-cell) has a color. These are minimal embedding information that allows for a simple display of objects, with edges represented as straight segments. More formally, an embedding is described via a function $\pi : \langle o\rangle \to \tau$ where π is the operation name, τ is the data type, and $\langle o\rangle$ the domain described as an orbit type. For instance, the embedding $color : \langle 0, 1, 3\rangle \to RGB$ provides RGB coordinates to the topological faces while $pos : \langle 1, 2, 3\rangle \to Point3D$ maps each topological vertex to some 3D coordinates. These two embedding data support the representation of the G-map in Fig. 3 and are the only ones that we manipulate in this article. An intuitive description of the two cubes' embedding is provided in Fig. 5.

4. Modeling operations

Since topological structures for representing geometric objects rely on graphs, modeling operations can be expressed through graph transformations. Graph rewriting is an extension of term rewriting to non-linear structures. We advise the reading of [46] for a quick introduction to graph rewriting, or [13] for a more in-depth explanation with an emphasis on software engineering. Graph rewriting allows to match a pattern within a graph and replace it with a new pattern. In such a framework, both patterns are graphs.

4.1. Graph rewriting

A *graph transformation rule* $r : L \to R$ is simply a rule where the left-hand side L and right-hand side R are graphs. For our needs, a graph transformation rule can be understood as a pair (L, R) where the nodes are accessed through a unique identifier. A node of L is preserved if a node in R has the same identifier. It is deleted if no such node exists. Symmetrically, a node in R is an added node if no node from L shares the same identifier. Throughout this article, we will use node names as identifiers. An example of graph transformation rule is provided in Fig. 6(a). The rule removes the arc between nodes x and y, adds a new node z, an arc between the nodes x and z, and an arc between z and y. The elements preserved by the transformation are identified by their name. Here only the nodes x and y are preserved.

The application of a rule $r : L \to R$ on a graph G relies on the specification of an occurrence, or *match*, of the left pattern L in the transformed graph G. For our concern, a match of L within a graph G corresponds to a subgraph of G similar to L. More precisely, 'similar to' means isomorphic, i.e., there is a subgraph G_L of G for which there is a bijection f from L to G_L that preserves node adjacency. From a given match, we can apply the rule. The preserved elements that appear in L and R are used to

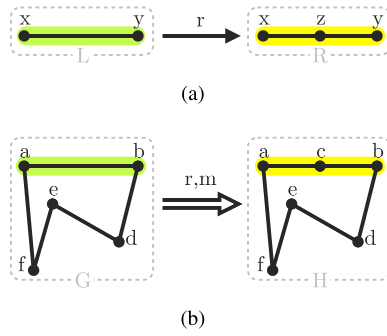


Fig. 6. A graph transformation rule (a) and its application (b) via the match $x \mapsto a$, $y \mapsto b$, and $(x \bullet\!\!-\!\!\bullet\, y) \mapsto (a \bullet\!\!-\!\!\bullet\, b)$. The codomain of the match, i.e., nodes a, b and the arc between them is highlighted for readability purposes.

connect the added elements of R to the surrounding context. This context consists of the graph G where elements of G_L have been removed. The formal construction exploits a monic morphism $m : L \to G$, i.e., an injective mapping from L to G that preserves graph structure (nodes, arcs, and labels). Once the match m is provided, the application of r to the graph G results in a graph H by deleting the subgraph $G_L = m(L)$ and adding R. The derivation is denoted $G \Rightarrow^{r,m} H$ and achieved via categorical constructions (see [12]).

The intuition that the operation is defined as a rule applied by removing and adding elements is sufficient to understand the discussions in this document. An application of the rule from Fig. 6(a) is illustrated in Fig. 6(b). The match from L to G maps x to a, y to b, and the arc between x and y to the arc between a and b. Applying the rule to G keeps unmodified any element that is not within the match and propagates the modification described by the rule. It removes the arc between nodes a and b, adds an arc between the nodes a and c, and adds an arc between c and b. The occurrence of the rule is highlighted in green, while the modification is highlighted in yellow.

4.2. G-map rewriting

As a first step, we illustrate the use of graph transformations to represent topological operations with simple examples. Fig. 7 provides two possibilities for the vertex insertion in a 2-G-map, based on the freedom of the edge. A free edge is a $\langle 0, 2\rangle$-orbit where the 2-links are loops while a sewn edge is a $\langle 0, 2\rangle$-orbit where the 2-links are non-loop arcs. This distinction gives rise to two configurations for the vertex insertion operation: one for a free edge (see Fig. 7(a) for the rule and Fig. 7(b) for an example of application) and one for a sewn edge (see Figs. 7(c) and 7(d)).

Applying a graph transformation rule relies on a mapping from the left pattern to the rewritten graph. The standard approach to graph rewriting requires a complete mapping of the nodes and the edges that preserves node adjacency and labels. Here, rules are not intended for any graph but only generalized maps, defined as graphs with regularity properties. In particular, each dart of a G-map is required to have exactly one incident arc per dimension, meaning we do not need the mapping of all elements from the left pattern of the rule. The choice of a single node is sufficient as we can build the complete mapping by a joint traversal of the left pattern and the rewritten graph. We illustrated the construction with the rule application of Fig. 7(b). This application assumes the match maps node x onto node b. Because a match has to preserve the node adjacency and the labels, the only possible match maps the arcs incident to x onto the arcs incident to b. Thus $x \overset{2}{\bullet\!\!-\!\!\bullet}\, x$ is mapped onto $b \overset{2}{\bullet\!\!-\!\!\bullet}\, b$, while $x \overset{0}{\bullet\!\!-\!\!\bullet}\, y$ is mapped onto $b \overset{0}{\bullet\!\!-\!\!\bullet}\, d$. Now, mapping the arcs incident to x provides information on how the match should map the nodes adjacent to x. Indeed, a valid match that maps $x \overset{0}{\bullet\!\!-\!\!\bullet}\, y$ onto $b \overset{0}{\bullet\!\!-\!\!\bullet}\, d$ must map y onto d. By recursively exploring the arcs incident to the newly found nodes, we can unambiguously recreate the match from the sole information that x is mapped onto b.

Intuitively, the incident arcs constraint, and, therefore, generalized maps, allows for the reconstruction of the match from a partial mapping. The minimal information required is the mapping of one node per connected component of L. Therefore, [10] specifies one particular node per connected component, called *hook*, rather than explicitly providing the entire match mapping.

4.3. Relabeling functions

Were modeling operations to be described as graph transformations, each possible configuration should be individually specified. However, modeling operations are usually defined for a specific topological cell. For instance, we can define the extrusion of a face without accounting for its arity, i.e., its number of edges. To this end, [9,17] introduced labels on nodes of the rule. These labels are in fact generalized orbit types and allow the rewriting of generalized maps regardless of a specific layout. These extended rules are called *rule schemes*, where the left and right patterns are called graph schemes. The simplest construction to understand the application of an operation described with a rule scheme is probably that of [19], using relabeling functions. The basic idea is that a rule scheme is parameterized by an orbit type, while all orbit labels of the rule scheme describe relabeling functions obtained by following the position of the dimension in the orbit type parameter. These relabeling functions allow the rewriting of orbits, given an initial orbit graph of the same type parameterizing the rule scheme. This process is called the instantiation of a node.

We will now provide the formal constructions of the relabeling functions and the instantiation of graph schemes. The remaining part of this subsection might be skipped as Section 4.4 provides some informal explanation. The informal explanation of Section 4.4 should be enough to understand the algorithm presented in Section 5 which is the main contribution of our work. However, the details below are needed to understand the proof of correctness of the algorithm.

Cells support the definition of modeling operations. Some examples of operations would be the subdivision of a face and the volume's edge-rounding. These operations are defined regardless of a specific topology, i.e., without considering the number of edges in a face or the arrangement of faces in a volume. Therefore, in [9,17] graph transformation rules were extended with orbit type parametrization. The rule is assigned an orbit type as a parameter, intuitively describing the rule's modified orbit. Each node is also assigned an orbit type but as a label used to compute the operation. These extended rules allow writing generic rules regardless of a specific layout.

Their instantiation specializes a generic rule into a particular configuration, i.e., a basic graph transformation rule. Intuitively, a rule scheme describes a folded representation of a transformation, which must be unfolded to obtain the actual graph transformation that can modify an object. For instance, the two configurations of the vertex insertion can be unified by folding the edge along its 2-links. We obtain the rule of Fig. 8(a). This rule is parameterized with the orbit type $\langle 2\rangle$. To obtain a graph-level rule, we choose a graph that consists of an orbit $\langle 2\rangle$. This graph is then used to unfold all the node labels. If unfolded as a 2-loop, the rule scheme provides the graph transformation rule of Fig. 7(a) while unfolding with two nodes sharing a 2-link yields the rule of Fig. 7(c). We will see that we can even further fold the graph transformation rule to obtain the rule scheme of Fig. 8(b). Formally, folding and unfolding are defined via *relabeling functions*.

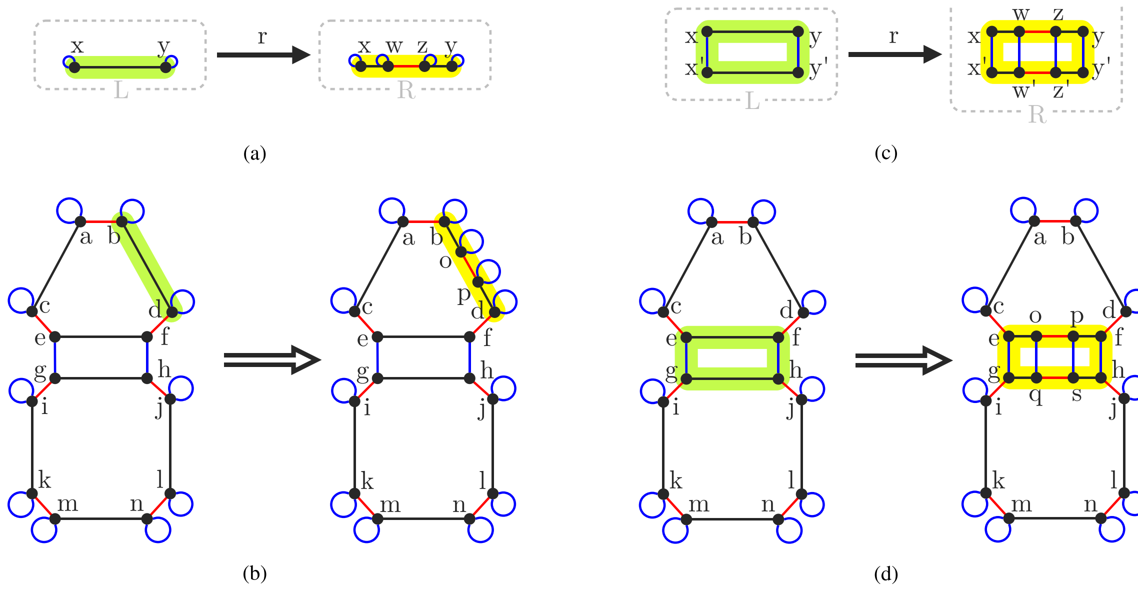


Fig. 7. Graph transformation rule for the vertex insertion in a free edge (a) and its application on a 2-G-map on a outer edge (b) via the match deduced from $x \mapsto b$. Graph transformation rule for the vertex insertion in a sewn edge (c) and its application on a 2-G-map on an inner edge (d) via the match deduced from $x \mapsto e$. Arc color legend: ▬ 3, ▬ 2, ▬ 1, and ▬ 0.

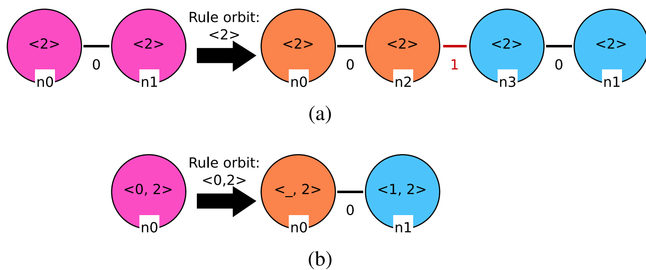


Fig. 8. Rule schemes for the vertex insertion: (a) by folding the 2-links and (b) both the 0 and 2-links.

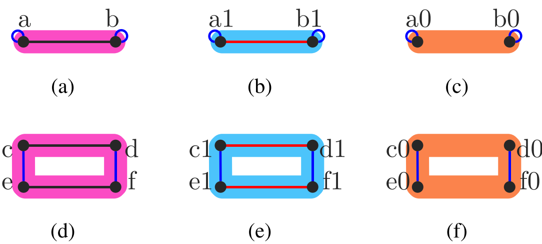


Fig. 9. Orbits (a) and (d) of type $\langle 0, 2\rangle$, label modification (b) and (e) via the relabeling function $\langle 0, 2\rangle \mapsto \langle 1, 2\rangle$, and label deletion (c) and (f) via the relabeling function $\langle 0, 2\rangle \mapsto \langle _, 2\rangle$. Arc color legend: ▬ 3, ▬ 2, ▬ 1, and ▬ 0.

4.3.1. Relabeling function

Introduced in [19], relabeling functions allow rewriting orbit types.

Definition 4.1 (*Relabeling Function*)**.** A *relabeling function* is a partial function $f : [\![0, n]\!] \to [\![0, n]\!] \cup \{_\}$, injective on $[\![0, n]\!]$, where '_' is a special symbol called *removing symbol*.

The application of a relabeling function to an orbit type is its application to each dimension within the orbit. For example, $\{0 \mapsto 1, 2 \mapsto 2\}(\langle 0, 2\rangle) = \langle 1, 2\rangle$. If we assume a reference orbit type $\langle o\rangle$ and a relabeled orbit type $\langle o'\rangle$, then the position of the dimensions within the orbit type entirely describes the relabeling function. For instance, given $\langle 0, 2\rangle \mapsto \langle 1, 2\rangle$, one can unambiguously reconstruct the relabeling function $\{0 \mapsto 1, 2 \mapsto 2\}$. The motivation to use relabeling functions is to encode orbit rewriting. Therefore, we usually denote such functions as relabeling of orbit types. Let $\langle o\rangle = (o_i)_{i \leq k}$ be the set of dimensions for which f is defined (ordered by increasing value), then f is written $\langle o\rangle \mapsto \langle (f(o_i))_{i \leq k}\rangle$. The injectivity property simply means that a dimension d cannot appear twice in $\langle o'\rangle = \langle (f(o_i))_{i \leq k}\rangle$. Let us remark that $\langle o'\rangle$ is not strictly speaking an orbit type, as it may contain the symbol '_'. In this sense, it is a *generalized* orbit type, which we will also call orbit type for convenience. Please note that the domain $\langle o\rangle$ of the relabeling function must not contain the removing symbol.

A relabeling function naturally extends from orbit type rewriting to orbit rewriting. Given a relabeling function $\langle o\rangle \mapsto \langle o'\rangle$ and an orbit $\langle o\rangle(v)$, one can build the orbit $\langle o'\rangle(v)$ by relabeling all arcs according to the function. For instance, the relabeling function $\{0 \mapsto 1, 2 \mapsto 2\}$ applied on the graphs of Fig. 9(a) yields the graphs of Fig. 9(b). The highlighting on the graphs of Fig. 9 will be exploited later on. Here we are only interested in the arc relabeling, i.e., the modification of the arc color. The 2-loops incident to nodes a and b yields 2-loops incident to nodes $a1$ and $b1$, as described by the relabeling $2 \mapsto 2$. Similarly, the relabeling $0 \mapsto 1$ transforms the arc $a \overset{0}{\bullet\!-\!\bullet} b$ into $a1 \overset{1}{\bullet\!-\!\bullet} b1$. The application of the same function on the graphs of Fig. 9(d) yields the graphs of Fig. 9(e).

The removing symbol '_' extends orbit types to generalized orbit types. This special symbol extends the definition of relabeling functions and their application on orbits. For instance, $\{0 \mapsto _, 2 \mapsto 2\}$ denotes the removal of the 0 label while preserving 2. Similar to any dimension, the removing symbol may appear in the orbit type of a node label. For example, $\langle _, 2\rangle$ is a valid node label. Assuming a reference orbit type $\langle 0, 2\rangle$, one can unambiguously reconstruct the relabeling function $\{0 \mapsto _, 2 \mapsto 2\}$. When applied to an orbit, all arcs that are relabeled with _ are in fact deleted. Two examples are provided in Figs. 9(c) and 9(f), using the graphs of Fig. 9(a) and 9(d) as references. In these examples, the arcs $a \overset{0}{\bullet\!-\!\bullet} b$, $c \overset{0}{\bullet\!-\!\bullet} e$, and $d \overset{0}{\bullet\!-\!\bullet} f$ do not have corresponding arcs between $a0$ and $b0$, $c0$ and $e0$, and $d0$ and $f0$.

Relabeling functions allow encoding folded representation of graphs called *graph schemes*.

Definition 4.2 (*Graph Scheme and Rule Scheme*)**.** Let $\langle o \rangle$ be an orbit type defined on $[\![0, n]\!]$.

A *graph scheme* of dimension n on $\langle o \rangle$, $(n, \langle o \rangle)$-graph scheme, or simply graph scheme consists of a graph $\mathcal{S}$ whose arcs are labeled on $[\![0, n]\!]$ and nodes are labeled with generalized orbit types of the same length as $\langle o \rangle$. For each node v of $\mathcal{S}$, we write $\langle o^v \rangle$ the orbit type labeling v.

A *rule scheme on* $\langle o \rangle$, or simply rule scheme, is a graph transformation rule $\mathcal{L} \xrightarrow{\langle o \rangle} \mathcal{R}$ where $\mathcal{L}$ and $\mathcal{R}$ are graph schemes defined on $\langle o \rangle$.

The orbit type of a rule scheme is also said to be its parameter and might be omitted when the context is clear. Both graph schemes of a rule scheme must share the same orbit type.

The node labels of a graph scheme are used as placeholders to encode any orbit of the given orbit type. More precisely, each node of a graph scheme is intended to be substituted by an orbit whose type matches its label. For example, a graph scheme reduced to a unique node labeled by the orbit type $\langle 0, 2 \rangle$ may be instantiated with a free or sewn edge, as depicted in Figs. 9(a) and 9(d). As soon as the graph scheme $\mathcal{S}$ contains several nodes, an additional condition comes into play. The size condition on the orbit types labeling the nodes of $\mathcal{S}$ means that they all share the same number of symbols as $\langle o \rangle$. Therefore the node substitutions can be obtained from relabeling functions built with $\langle o \rangle$. All the nodes of $\mathcal{S}$ will be substituted by the same orbit typed by $\langle o \rangle$, whose arcs will be relabeled by the relabeling function attached to the nodes.

Graph schemes can be used to define rules with the requirement that both left and right patterns are graph schemes defined on the same orbit type.

4.3.2. Instantiation

Unfolded graphs that correspond to a given graph scheme are reconstructed through relabeling functions. This process is defined inductively and called the instantiation of a graph scheme. We provide a step-by-step construction of the instantiation process to ease its understanding.

For the following definitions, we consider a graph scheme $\mathcal{S}$ defined on the orbit type $\langle o \rangle$ and a graph $\mathbf{O}$ that consist of an orbit typed by $\langle o \rangle$.

Isolated node. The first component is the instantiation of an isolated node s, i.e., a node without arc. In such a case, the instantiation process is reduced to applying the relabeling function $\langle o \rangle \mapsto \langle o^s \rangle$ to the orbit graph chosen for the instantiation. An example of such a graph scheme is provided in Fig. 10, some of its instantiation are depicted in Figs. 9(b) and 9(e) (when considered as a graph scheme on the orbit $\langle 0, 2 \rangle$).

Definition 4.3 (*Instantiation of an Isolated Node*)**.** If $\mathcal{S}$ is of the form $(\{s\}, \emptyset)$, its instantiation with $\mathbf{O}$ is the graph obtained by the application of the relabeling function $\langle o \rangle \mapsto \langle o^s \rangle$ to $\mathbf{O}$.

$$\iota^{\langle o \rangle}((\{s\}, \emptyset), \mathbf{O}) = [\langle o \rangle \mapsto \langle o^s \rangle](\mathbf{O})$$

Discrete graph. The instantiation of a discrete graph, i.e., a graph without arc, consists of the union of the instantiation of its labeled nodes. An illustration is given in Fig. 11.

Definition 4.4 (*Instantiation of a Discrete Scheme*)**.** If $\mathcal{S}$ is of the form $(V, \emptyset)$, its instantiation with $\mathbf{O}$ is the disjoint union of the instantiation of each node of S with $\mathbf{O}$.

$$\iota^{\langle o \rangle}((V, \emptyset), \mathbf{O}) = \bigsqcup_{v \in V} \iota^{\langle o \rangle}((\{v\}, \emptyset), \mathbf{O})$$

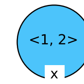


Fig. 10. Isolated node extracted from the right pattern of Fig. 8(b). When considered as graph scheme on the orbit $\langle 0, 2 \rangle$, its instantiation on the orbit graph of Fig. 9(a) yields the graphs of Fig. 9(b). Similarly, its instantiation on the orbit graph of Fig. 9(d) yields the graph of Fig. 9(e).

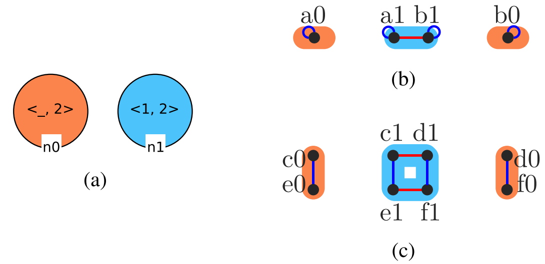


Fig. 11. Discrete graph scheme (a) extracted from the right pattern of Fig. 8(b), instantiation (b) with the orbit graph of Fig. 9(a), and instantiation (c) with the orbit graph of Fig. 9(d).

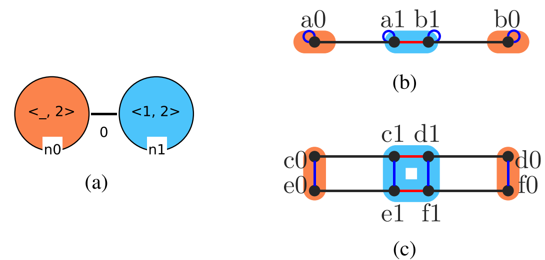


Fig. 12. Graph scheme (a) with an arc between two nodes, and (b) and (c) two instantiations by addition of the 0-links. Arc color legend: — 3, — 2, — 1, and — 0.

Arc. The instantiation of an arc between two nodes adds links between darts image of the same initial dart via the two relabeling functions. An example is provided in Fig. 12.

Definition 4.5 (*Instantiation of an Arc*)**.** For s a node of $\mathcal{S}$ and u a node of $\mathbf{O}$, we write (u, s) for the image of u in $\iota^{\langle o \rangle}((\{s\}, \emptyset), \mathbf{O})$.

If $\mathcal{S}$ is of the form $(\{s, t\}, \{s \stackrel{i}{\bullet\!\!-\!\!\bullet} t\})$, its instantiation with $\mathbf{O}$ extends $\iota^{\langle o \rangle}((\{s, t\}, \emptyset), \mathbf{O})$ to link copies of the same node from $\mathbf{O}$:

$$\iota^{\langle o \rangle}((\{s, t\}, \{s \stackrel{i}{\bullet\!\!-\!\!\bullet} t\}), \mathbf{O}) = \iota^{\langle o \rangle}((\{s, t\}, \emptyset), \mathbf{O}) \cup \bigcup_{u \in \mathbf{O}} (u, s) \stackrel{i}{\bullet\!\!-\!\!\bullet} (u, t)$$

We write $\iota^{\langle o \rangle}(s \stackrel{i}{\bullet\!\!-\!\!\bullet} t, \mathbf{O})$ for $\bigcup_{u \in \mathbf{O}} (u, s) \stackrel{i}{\bullet\!\!-\!\!\bullet} (u, t)$.

Complete graph scheme. From the instantiation of a discrete graph, we obtain the complete instantiation of a graph scheme by instantiating each arc.

Definition 4.6 (*Instantiation of a Graph Scheme*)**.** If $\mathcal{S}$ is of the form (V, E), its instantiation with $\mathbf{O}$ extends $\iota^{\langle o \rangle}((V, \emptyset), \mathbf{O})$ to link copies according to all the arcs of E:

$$\iota^{\langle o \rangle}((V, E), \mathbf{O}) = \iota^{\langle o \rangle}((V, \emptyset), \mathbf{O}) \cup \bigcup_{v \stackrel{i}{\bullet\!\!-\!\!\bullet} v' \in E} \iota^{\langle o \rangle}(v \stackrel{i}{\bullet\!\!-\!\!\bullet} v', \mathbf{O})$$

Instantiating a graph scheme intuitively corresponds to:

1. Unifying the application of the relabeling functions (encoded by the orbit types on the node).

2. Adding a link between the images of a node whenever there is an arc in the graph scheme.

Rule scheme. The instantiation of a rule scheme $\mathcal{L} \xrightarrow{\langle o \rangle} \mathcal{R}$ is defined as the instantiation of both $\mathcal{L}$ and $\mathcal{R}$ with the same orbit $\mathbf{O}$ of type $\langle o \rangle$. The resulting instantiations directly yield to a graph transformation $\iota^{\langle o \rangle}(\mathcal{L}, \mathbf{O}) \to \iota^{\langle o \rangle}(\mathcal{R}, \mathbf{O})$ as discussed in Section 4.2.

Let us illustrate with the reconstruction of the two rules of Figs. 7(a) and 7(c) via the rule scheme of Fig. 8(b). We consider the left-hand side and the right-hand side of the rule separately. Each graph is a graph scheme defined on the orbit type $\langle 0, 2 \rangle$. The left pattern consists of a single node $n0$ without any arc. The label of $n0$ has the type $\langle 0, 2 \rangle$. Thus, the relabeling function is $\langle 0, 2 \rangle \mapsto \langle 0, 2 \rangle$, i.e., the identity function. In other words, the instantiations of the graph scheme with the graphs of Fig. 9(a) and 9(d) yield these exact graphs. The right pattern is the graph scheme of Fig. 12(a). Therefore, the complete instantiations of the right pattern correspond to graphs of Figs. 12(b) and 12(c):

- The instantiation of the left pattern on the graph of Fig. 9(a) is the graph of Fig. 9(a), isomorphic to the graph L in the rule of Fig. 7(a).
- The instantiation of the left pattern on the graph of Fig. 9(d) is the graph of Fig. 9(d), isomorphic to the graph L in the rule of Fig. 7(c)
- The instantiation of the right pattern on the graph of Fig. 9(a) is the graph of Fig. 12(b), isomorphic to the graph R in the rule of Fig. 7(a).
- The instantiation of the left pattern on the graph of Fig. 9(d) is the graph of Fig. 12(c), isomorphic to the graph R in the rule of Fig. 7(c)

In practice, the orbit type $\langle o \rangle$ for the rule parameter is given through the *hook*. We choose a node with no removing symbol '_' in its node label and use it as a reference to construct all relabeling functions. Thus, the hook serves a double purpose. On top of specifying where the modeling operation occurs in the object, it is used to build the relabeling functions. The application of a rule scheme to a G-map starts with the selection of a dart a. From this dart, a graph traversal builds the orbit $\langle o \rangle(a)$ where the orbit type $\langle o \rangle$ is the one carried by the hook. The rule scheme is instantiated with the orbit $\langle o \rangle(a)$ via instantiation of both its left-hand side and right-hand side. The instantiation provides a graph transformation rule, applied on the initial G-map.

4.4. Example of rule scheme for the quad subdivision

We provide an intuitive explanation of a rule scheme instantiation closer to the actual implementation with the subdivision illustrated in Fig. 1. This subdivision, called the quad subdivision, is mainly used to refine the topological structure of meshes [47]. We can consider this operation as purely topological, i.e., without considering the geometric refinement. A new vertex is added to the center of the face and the middle of each edge. An edge then attaches the center of each face to the midpoints of each of the initial edges. When applied to a cube, the subdivision splits each face into four new faces (see Fig. 1(a)). From the decomposition of the cube before and after the subdivision, we can reconstruct the operation as a rule on G-maps, which subdivides any hexahedron (see Fig. 13). Rule schemes allow to write this operation for an entire surface, i.e., on the volume cell $\langle 0, 1, 2 \rangle$. The rule is provided in Fig. 14, nodes are colored to ease the explanation of Fig. 15. Nodes are generically named n plus an integer in a graph scheme, e.g., $n0$, $n1$.

In the rule scheme of Fig. 14, the left-hand side contains a single node. The double line around the node means that it is a *hook*. The hook's label is the parameter of the rule scheme and

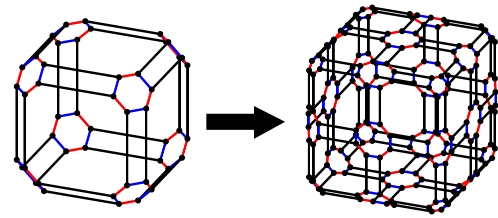


Fig. 13. Quad subdivision represented on G-maps associated with the objects of Fig. 1(a).

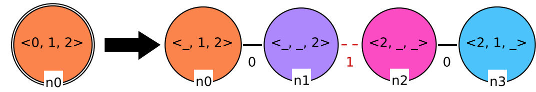


Fig. 14. Rule scheme generalizing the operation from Fig. 13.

encodes the valid possibilities to unfold the rule, i.e., to build a legal concrete rule. In our example, the node $n0$ of the left-hand side is labeled by $\langle 0, 1, 2 \rangle$. Thus, the unfolding is realized using a $\langle 0, 1, 2 \rangle$-orbit graph. The graph of Fig. 15(a) is a $\langle 0, 1, 2 \rangle$-orbit and can be used to unfold the rule scheme. The relabeling function associated with the hook is the identity function, and its application on a graph will result in the same graph. Therefore, the graph of Fig. 15(a) is a possible unfolding of the node $n0$ from the left-hand side of the rule scheme.

All other nodes from the left-hand side or the right-hand side represent copies of the graph built through relabeling functions. Each relabeling function is encoded by the mapping of dimensions in the node label via their position in the orbit type (see 4.3). For instance, the label of $n0$ is $\langle _, 1, 2 \rangle$ in the right-hand side. Since the orbit parameter is $\langle 0, 1, 2 \rangle$, the relabeling function is $0 \mapsto _$, $1 \mapsto 1$, and $2 \mapsto 2$. This substitution specifies that the 0-links are deleted (removing symbol '_'), while the 1 and 2-links are preserved. All darts are always copied. The darts corresponding to the same node have been drawn with the same color in Figs. 14 and 15. From the graph of Fig. 15(a), we obtain the graph of Fig. 15(b) by removing all the 0-links.

The dimensions in the node labels of the graph scheme are called *implicit arcs* since they implicitly represent relabeled links. On the other hand, the arcs between nodes of the rule are called *explicit arcs*. They allow linking copies of the same dart of the graph used for the unfolding. In the right-hand side of the rule, node $n0$ has an incident explicit 0-arc. The unfolding adds 0-links to all the darts associated with the node $n0$, as illustrated in Fig. 15(c). Note that unfolding an explicit arc means adding exactly one link incident to each dart unfolded from the node. For example, the 0-arc between nodes $n0$ and $n1$ means that each copy associated with node $n0$ is 0-linked to its counterpart in the copy associated with node $n1$. The complete unfolding of the rule scheme is obtained from a graph traversal, iterating additions of darts with links corresponding to the node label and additions of links corresponding to the arcs between the nodes.

The copy $n1$ removes the 0 and 1-links and keeps only the 2-links, as indicated by the label $\langle _, _, 2 \rangle$ with two "_". In Fig. 15(d), purple darts are added at the dangling extremity of the 0-links of Fig. 15(c), together with 2-links from the node label. The incident 1-arc from the rule scheme provides 1-links dangling from the purple darts in Fig. 15(e). Pink darts corresponding to the copy $n2$ are attached to the dangling extremities in Fig. 15(f). The label of $n2$ in Fig. 14 renames all 0-links to 2-links and deletes the 1 and 2-links, as indicated by the label $\langle 2, _, _ \rangle$. Therefore, the pink darts share 2-links where darts from the initial copy were sharing 0-links. Finally, 0-arc incident to $n2$ is extended in Fig. 15(g) and

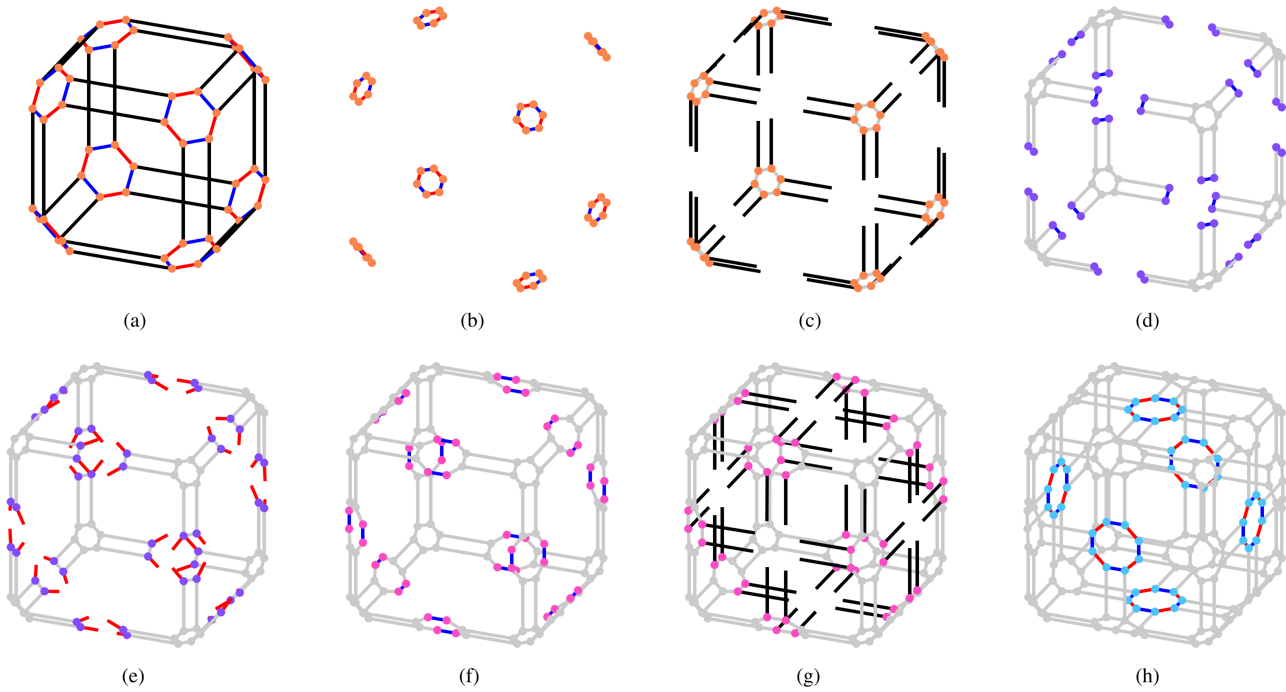


Fig. 15. Unfolding the rule scheme of Fig. 14: (a) $n0$ from the left hand-side, (b) $n0$ from the right hand-side, (c) 0-arc between $n0$ and $n1$, (d) $n1$, (e) 1-arc between $n1$ and $n2$, (f) $n2$, (g) 0-arc between $n2$ and $n3$, and (h) $n3$. The colors match the node coloring of Fig. 14 to ease the reading, already built elements are displayed in gray. Arc color legend: — 3, — 2, — 1, and — 0.

blue darts are added to unfold the node $n3$ in Fig. 15(h). The relabeling function from node $n3$ renames the 0-links into 2-links, preserves the 1-links, and removes the 2-links. The first graph (Fig. 15(a)) is the cube in left part of the rule from Fig. 13, while the last graph (Fig. 15(h)) is the complete subdivided cube in the right part of the rule.

Applying a rule scheme to a specific object requires finding an unfolded rule compatible with the G-map representation of the object. This process, called the instantiation of the rule, relies on the hook node. First, the user selects a dart within the G-map. Then, a graph traversal gathers all darts and arcs corresponding to the orbit type labeling the hook associated with the selected dart. In other words, if the user selects a dart v in a G-map G and tries to apply a rule scheme with a $\langle o \rangle$-labeled hook, the retrieved graph is $G\langle o \rangle(v)$. This graph is then used for unfolding the rule scheme.

4.5. Embedded operations

This subsection describes how to handle geometric computations at the rule level. Although these computations are needed to display the objects, they are not required to understand our contribution, namely the topological folding algorithm. More details about the addition of embedding variables to rule schemes can be found in [19].

The geometric properties of objects consist of values shared by all darts among a topological cell, i.e, among a given orbit. Therefore, applying an operation might lead to an update of the geometric values for all the orbits within the scope of the operation. When directly modifying G-maps, the embedding computations should be carried out for each dart. At the level of rule schemes, these computations are extended with terms for better expressivity through references that yield distinct embedding values for darts instantiated from the same node.

Fig. 16 shows the embedded versions of the rule schemes used for the splitting of an edge into two edges by the insertion of a vertex. Recall that a rule scheme consists of a folded representation of graph rules that generalize operation up to an orbit variable. The rule scheme from Fig. 16(a) depicts the case where the graph rule for the edge split is folded along the 2-links. In this rule scheme, nodes $n0$ and $n1$ represent each endpoint of the edge. The addition of the new vertex is supported by the nodes $n2$ and $n3$. All darts that correspond to these two nodes are necessarily created by the operation and will not have a position value. The usual solution is to add the new vertex at the barycenter position of the two endpoints of the initial edge. Therefore, we embed the operation with a position computation for nodes $n2$ and $n3$, displayed above the rule in Fig. 16(a). The geometric computation is realized during the instantiation process via the substitution of $n0$ and $n1$ by the associated darts. This substitution concerns each dart associated with the nodes $n2$ and $n3$. Similarly, the ruling scheme from Fig. 16(b) represents the folding of the graph rule along both the 0 and 2-arcs. Therefore, node $n0$ represents both endpoints of the edge simultaneously. The splitting vertex now only corresponds to node $n1$, and the geometric computation requires accessing both the darts associated with $n0$ and their neighbor by 0, written $n0@0$.

In [19], such variables are subject to consistency conditions to ensure that the embedding of a generalized map is well-defined, i.e., all darts within the same orbit whose type represents the domain of an embedding should have the same value. We also want to point out that rule schemes allow considering the topological computations and the geometric computations separately. In this article, we aim at deducing the topological part of an operation from a representative example. We leave aside all geometrical considerations. Therefore, the inferred rule scheme will have no embedding computation. In the illustrative section (Section 9), we manually added these embedding computations to the rule such that we obtain operations applicable to geometric objects.

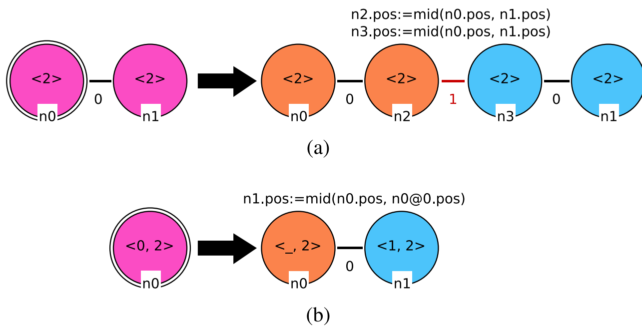


Fig. 16. Embedded rule schemes for the vertex insertion: (a) by folding the 2-links and (b) both the 0 and 2-links.

4.6. Consistency preservation

We wish to emphasize that applying a rule to a G-map will not necessarily provide a G-map. Intuitively, we are simply saying that modifying a (quasi-)manifold will not result in a (quasi-) manifold. Therefore, the rules are equipped with conditions that preserve the (quasi-)manifold property. Prosaically, these conditions [17] guarantee the preservation of the topological constraints of Definition 3.1. These conditions can be extended [18] to rule schemes, resulting in one of our principal motivations to use graph rewriting: we can design operations that guarantee to preserve the model's consistency via conditions on the rule. A simple syntactic verification of the rule scheme allows discriminating between well-formed and ill-formed rules. We will not provide more details on these conditions but highlight that we can easily distinguish the well-formed rules. Therefore, if our algorithm builds an ill-formed rule, we discard it.

5. Topological folding algorithm

Rule schemes are a compact format to describe modeling operations. However, their authoring requires learning this specific language and a good knowledge of generalized maps. Therefore, we propose to infer them automatically based on an instance of the operation, hence providing a solution that simplifies the design of new operations and hides the technical elements.

We now present our main contribution, which is a mechanism to infer a generic modeling operation from an application example. In other words, we seek to reconstruct the rule scheme that corresponds to a modification of the object performed by a user who is not an expert in the underlying theory. For instance, we wish to infer the rule of Fig. 14 from the objects of Fig. 1 when considering the orbit $\langle 0, 1, 2\rangle$, i.e., a surface.

The user describes a starting object, modifies it, and asks the corresponding operation for a given orbit. Therefore, we aim at designing a mechanism for reconstructing a rule scheme given by an orbit (provided it exists), i.e., having an instantiation corresponding to the rule provided by the user. We call *operation inference* this mechanism.

The objective is to reconstruct the rule scheme that generalizes a specific modeling operation for a given orbit. Rather than directly reconstructing the rule scheme, we first design an algorithm to reconstruct a single *graph scheme*. We will then explain how to exploit the algorithm to build the complete rule scheme. Intuitively, a graph scheme corresponds to the left (or right) pattern of a rule scheme. It corresponds to a graph labeled on each node with an orbit type.

5.1. Notations

Our algorithm reconstructs a graph scheme $\mathcal{S}$ from a given G-map G and a given orbit type $\langle o\rangle$ (with $o \subseteq [\![0, n]\!]$). The algorithm works as follows. First, we choose a dart a in the G-map. We assume that the orbit graph incident to a corresponds to the orbit used for instantiation, i.e., an orbit with the same type as the label of the hook h from the graph scheme $\mathcal{S}$. By local exploration, we then try to reconstruct the scheme. Intuitively the construction of the scheme $\mathcal{S}$ consists of folding the graph G along its i-links, for all dimensions i of the orbit type $\langle o\rangle$. If the construction fails, we start again with another initial dart until a scheme is found. The new initial dart is chosen in a pool of unseen darts (see discussion in Section 10.3). If all darts have been tried, then no scheme exists for the orbit type $\langle o\rangle$.

In the explanation of the algorithm, we will write:

- h for the hook in the graph scheme $\mathcal{S}$, m for the node of interest, and v for the other nodes.
- Letters from the beginning of the alphabet for the darts in $G\langle o\rangle(a)$, e.g., a, b, c, with a being reserved for the initial dart.
- Letters from the end of the alphabet for generic darts in G, e.g., x.
- d, i and j for the dimensions in $[\![0, n]\!]$.
- $\langle o^v\rangle$ for the orbit type used to label the node v of $\mathcal{S}$ such that the associated relabeling function is $\langle o\rangle \mapsto \langle o^v\rangle$.
- (b, v) for the dart in G that corresponds to the dart b from the orbit $G\langle o\rangle(a)$ and the node v from the scheme $\mathcal{S}$.

5.2. Algorithm

We detail the algorithm that constructs a graph scheme for a given connected G-map G, a given orbit type $\langle o\rangle$, and a chosen dart a of G. The graph scheme $\mathcal{S}$ we seek to construct will have by construction a hook node h labeled by $\langle o\rangle$ if it exists.

Step 1 (Orbit graph and construction of the hook)
We build the orbit graph $G\langle o\rangle(a)$ on a in G and initialize the graph scheme $\mathcal{S}$ as a single hook node h labeled by the chosen orbit type $\langle o\rangle$. $G\langle o\rangle(a)$ is by construction a possible instantiation of $\mathcal{S}$.

Step 2 (Traversal)
Using a breadth-first traversal of G, we reconstruct a graph scheme $\mathcal{S}$ that yields G by instantiation when (a, h) is mapped onto a. First, the algorithm constructs the explicit arcs incident to the next node (Step 2.1), starting from h. Then, it immediately adds the node label (Step 2.2) of each newly created node. Finally, it iterates on a new node. If the algorithm terminates without failure, we obtain a topologically correct graph scheme $\mathcal{S}$ for the chosen dart.

Step 2.1 (Construction of the explicit arcs incident to a node)
Given a node m in $\mathcal{S}$, already labeled with an orbit type, we construct its incident arcs. These explicit arcs are labeled by dimensions d not belonging to the orbit type $\langle o^m\rangle$. Thus, their instantiation provides the remaining links incident to the darts of the instance of m.

The arcs construction starts with finding an extension of the scheme based on the information we can retrieve from (a, m). Then, the extension is verified to be compatible with all the (b, m) for b in $G\langle o\rangle(a)$. The verification is straightforward as the addition of a node m to $\mathcal{S}$, ensures that we can retrieve all the darts (b, m) for b in $G\langle o\rangle(a)$.

Since the algorithm runs on an n-G-map G, the dart (a, m) has exactly one incident d-link e_d per dimension d in $[\![0, n]\!] \setminus \langle o^m\rangle$. There are three possibilities for each e_d:

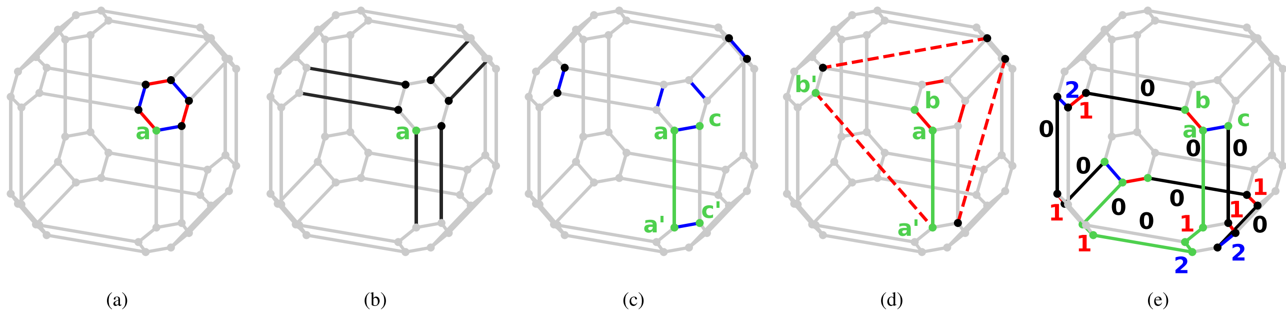


Fig. 17. Steps of the folding algorithm on the object. Step 1 (a) - construction of $G\langle 1, 2\rangle(a)$. Step 2.1 (b) - construction of the hook's explicit arcs. Step 2.2 (c) and (d) - construction of a node and its implicit arcs. The algorithm terminates (e). Arc color legend: ▬ 3, ▬ 2, ▬ 1, and ▬ 0.

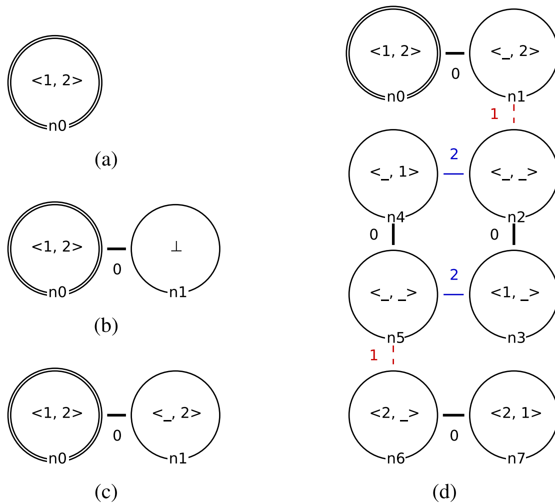


Fig. 18. Different steps of the graph scheme in the folding algorithm: Step 1 (a) - construction of the hook, Step 2.1 (b) - construction of the hook's explicit arcs and Step 2.2 (c) - construction of a node, (d) - result of the algorithm.

Arc addition e_d is a d-link $(a, m) \overset{d}{\bullet\!\!-\!\!\bullet} (a, m')$ where m' is a node of $\mathcal{S}$. In this case we add a d-arc $m \overset{d}{\bullet\!\!-\!\!\bullet} m'$ in $\mathcal{S}$ (if not already in $\mathcal{S}$ by the extension of m').

Arc failure e_d is a d-link $(a, m) \overset{d}{\bullet\!\!-\!\!\bullet} (b, m')$ where b is a dart of $G\langle o\rangle(a)$ different from a, and m' is a node of $\mathcal{S}$. In this case the algorithm stops in a failure state because such a link is not constructible using the instantiation mechanism.

Arc extension e_d is a d-link $(a, m) \overset{d}{\bullet\!\!-\!\!\bullet} x$ where x has not yet been reached by the algorithm. In this case, we add a node v with the label $\langle o^v\rangle = \perp$ and a d-arc $m \overset{d}{\bullet\!\!-\!\!\bullet} v$ in $\mathcal{S}$. The symbol $\perp$ serves as a placeholder to signify that the node v has no label yet, with the convention that $\perp$ deletes all arcs when instantiated.

Let $\mathcal{S}'$ be the graph scheme obtained after the addition and extension of all the d-arcs for d in $[\![0, n]\!] \setminus \langle o^m\rangle$ (assuming no failure). Recall that a d-arc in the graph scheme gives rise to one d-link per dart b in $G\langle o\rangle(a)$, each one incident to a dart (b, m). Therefore, we still need to check that the decision made for each dimension is coherent with the links incident to the other darts. In other words, check that each dart (b, m) for b in $G\langle o\rangle(a)$ has an incident d-link with the other extremity corresponding to the same node in $\mathcal{S}'$.

Instantiation failure For each d-arc $m \overset{d}{\bullet\!\!-\!\!\bullet} m'$ in $\mathcal{S}$, if a dart b exists in $G\langle o\rangle(a)$ such that there is no link $(b, m) \overset{d}{\bullet\!\!-\!\!\bullet} (b, m')$ in G, then the algorithm stops in a failure state because such an arc would be constructed using the instantiation mechanism.

Note that the case of 'arc addition' where $m' = m$ covers the addition of loops to the graph scheme $\mathcal{S}$.

Step 2.2 (Construction of a node label)

A node m can be added with a fake label in the 'arc extension' case from Step 2.1. The construction of the explicit d-arc assumes that the target darts of the corresponding d-links are the instantiation of the new node m in the graph scheme. The proper label associated with this node needs to be reconstructed. We look for the existence of a label $\langle o^m\rangle$ whose instantiation provides the links that join the darts of the instance of m.

Here again, we first extend the graph scheme $\mathcal{S}$ before checking if the extension is correct. We consider, for each dimension i of $\langle o\rangle$, the i-neighbor b_i of a, i.e., the dart such that $a \overset{i}{\bullet\!\!-\!\!\bullet} b_i$ is in $G\langle o\rangle(a)$. We now distinguish between two possibilities:

Relabeling There is a dimension j in $[\![0, n]\!]$ such that a link $(a, m) \overset{j}{\bullet\!\!-\!\!\bullet} (b_i, m)$ exists in G. In this case, we fix the relabeling $i \mapsto j$, i.e., j appears in $\langle o^m\rangle$ at the position of i in $\langle o\rangle$.

Deletion The previous possibility is not fulfilled, i.e., (a, m) and (b_i, m) are not linked. We then fix $i \mapsto$ _, i.e., the symbol "_" appears in $\langle o^m\rangle$ at the position of i in $\langle o\rangle$.

Recall that the orbit type used to label m gives rise to arcs based on the relabeling function ($\langle o\rangle \mapsto \langle o^m\rangle$). Therefore one still needs to check that the decision made for each implicit arc is coherent with the other darts related to the instantiation of m. In other word, check that each i-link between b and c in $G\langle o\rangle(a)$ can be mapped onto a link between (b, m) and (c, m) in G.

Relabeling failure If the decided relabeling is $i \mapsto j$ and there is an i-link $b \overset{i}{\bullet\!\!-\!\!\bullet} c$ in the orbit $G\langle o\rangle(a)$ such that no link $(b, m) \overset{j}{\bullet\!\!-\!\!\bullet} (c, m)$ exists in G, then the algorithm stops in a failure state. Indeed, such a link would be constructed by the instantiation mechanism.

We are identifying the implicit arcs that are preserved, deleted, or renamed by the new node m. Note that the preservation of an arc is covered by the case 'relabeling', where $j = i$.

In general, two darts b and c of $G\langle o\rangle(a)$ can be connected by several i-links (i.e., with different i of $\langle o\rangle$). One can then construct several different orbit labels. Our algorithm uses a heuristic to construct a plausible relabeling function among all possible ones. To ensure the injectivity of the relabeling function, we ensure that two dimensions i and i' are not mapped onto the same dimension j. The possible plurality of inferred graph schemes (and therefore rule schemes) is further discussed in Section 10.3.

Note that we do not need a failure possibility for the 'deletion' case. Indeed, if the decided relabeling is $i \mapsto _$, but there is a non-loop i-link $b \overset{i}{\bullet\!\!-\!\!\bullet} c$ in $G\langle o\rangle(a)$ and a dimension j of $[\![0, n]\!]$ such that $(b, m) \overset{j}{\bullet\!\!-\!\!\bullet} (c, m)$ is a link of G, then j does not appear in $\langle o^m\rangle$. Since G is a G-map, the incident arcs constraint ensures the existence of a j-link incident to (a, m). This j-link will bring the addition or creation of a j-arc in the construction of the explicit arcs incident to m (if not directly a failure state). This j-arc will result in an instantiation failure on the darts (b, m) and (c, m).

6. Illustrative examples

Before dealing with the analysis of the algorithm, we first provide some examples to illustrate its possible behaviors. First, we give an example where the algorithm does not enter a failure state and outputs a valid graph scheme. Secondly, we present the two prominent cases where the algorithm halts in a failure state. Throughout all examples, nodes of graph schemes and rule schemes are generically named n plus an integer, e.g., $n0$, $n1$.

6.1. Folding of the cube

We now illustrate the algorithm using the orbit type $\langle 1, 2\rangle$ (the vertex of volume) on the G-map of a cube, i.e., the left-hand side of the rule given in Fig. 13. The cube is considered a closed surface and represented with a 2-G-map.

Step 1 (Orbit graph and construction of the hook). The orbit graph $G\langle 1, 2\rangle(a)$ on the cube is illustrated in Fig. 17(a). Irrelevant links and darts have been dimmed. Thus, the orbit graph contains six darts and links: three 1-links drawn in red and three 2-links in blue. We initialize $\mathcal{S}$ as in Fig. 18(a): a graph scheme containing a simple hook $n0$ labeled $\langle 1, 2\rangle$. The subgraph of Fig. 17(a) is indeed one of its possible instantiations.

Step 2.1 (Construction of the explicit arcs incident to a node). The 6 darts of the vertex $G\langle 1, 2\rangle(a)$ exhibit a 0-link to 6 distinct darts outside the vertex matched by the hook $n0$, as shown in Fig. 17(b). We deduce the beginning of the graph scheme described in Fig. 18(b). The scheme is incomplete at this point since the node $n1$ has a fake label $\perp$.

Step 2.2 (Construction of a node label). We added a node $n1$ in the scheme and now have to construct its label. We try to construct a renaming of the 1 and 2-links of the initial orbit graph on this set of darts. As illustrated in Fig. 17(c), we obtain a plausible preservation of the 2-link between a and c (since there is a 2-link between a' and c'). Conversely, as seen in Fig. 17(d), there is no renaming of the 1-link between a and b; this link is deleted. We check that the renaming (the replacement of 2 by 2 and the deletion of 1) is correct for all darts. This renaming is valid, and we deduce the partial graph scheme given in Fig. 18(c).

Termination. On the cube, the algorithm ends by the Construction of the label of the last node $n7$. This node corresponds to the vertex opposite to the initial vertex, as shown in Fig. 17(e). Following the exploration path 012010, we reach the darts of the opposite vertex, renaming the orbit $G\langle 1, 2\rangle(a)$ by the function $\langle 1, 2\rangle \mapsto \langle 2, 1\rangle$. The resulting scheme is given in Fig. 18(d).

6.2. Counterexamples

We now provide examples that result in a state of failure. Step 1 corresponds to an unrestricted graph traversal and will always succeed. Step 2.1 concerns the construction of arcs incident to a node, while Step 2.2 deals with the label associated with a node. The instantiation of a node with an orbit corresponds to applying a relabeling function to this orbit. For an arc, the instantiation corresponds to its transformation in as many links as darts in the orbit.

For the construction of the relabeling function associated with a node, the conditions ensure that the relabeling found is consistent for all darts associated with the node. The algorithm will fail to fold a pyramid from a corner of the base, i.e., not the apex. Figs. 19(a), 19(b), 19(c), and 19(d) display the same steps as in Fig. 17, meaning that the partial schemes of Figs. 18(a), 18(b) and 18(c) will correctly be built by the algorithm. However, as illustrated in Fig. 19(e), the algorithm will fail. The partial scheme at this moment is provided in Fig. 21. When building the relabeling, we identify a 0-arc between darts a' and b'. We can thus check if the relabeling $1 \mapsto 0$ is coherent with the other darts. As we can see, the 1-arc between d and e is mapped onto a non-arc between d' and e'. Therefore, the algorithm fails, meaning no folding can be obtained from a as initial dart and $\langle 0, 1\rangle$ as orbit type.

For the construction of an arc, the condition ensures that all links folded into the arc have the same status. These links can either be loops or links to darts that all correspond to the same node (coherently with the image from the initial orbit graph) or links to darts not seen yet (checked for relabeling afterward). Whenever the incident d-links for a given dimension d outside the orbit type fall into a different category, an explicit arc cannot be added, and the algorithm stops. For instance, let us consider the object of Fig. 20(a). The object consists of a cube where the top face has been removed. Its G-map representation is depicted in Fig. 20(b). If we fold it along one of the vertical edges, the algorithm fails. Indeed, with the orbit type $\langle 0, 2\rangle$ and the dart a in Fig. 20(c), we reach the state of Fig. 20(d) after traversing and folding along the 1-links. However, dart b' has a 2-loop while dart a' has a non-loop 2-link. The algorithm cannot proceed, meaning no folding is obtainable with a as initial dart and $\langle 0, 2\rangle$ as orbit type.

7. Generalization to a rule scheme

The algorithm presented in the previous sections allows for deducing a folded representation of a G-map from an orbit and a dart. To infer a rule scheme from a given concrete rule using the folding algorithm, we propose to consider the G-maps (left and right) of the concrete rule as part of the same graph. From two G-maps L and R, we construct the graph $\kappa(L, R)$ as the disjoint union of L and R, plus arcs labeled by κ (a fresh symbol) connecting both copies of the preserved nodes. The graph $\kappa(L, R)$ constructed from the rule of Fig. 13 is given in Fig. 22(a). The κ-arcs are drawn in pink; the original cube is enlarged while the modified cube is shrunk. The κ-arcs state which nodes are unmodified by the operation. These arcs are computed from a mapping of some elements from the L to R, further discussed in Section 9.

To determine the scheme for a rule $r = L \to R$, we construct a graph scheme of the related graph $\kappa(L, R)$. We run the algorithm on $\kappa(L, R)$ by memorizing whether the darts belonged to L or R. The algorithm is identical on the graph $\kappa(L, R)$. We only need to add some additional conditions:

- The graph $\kappa(L, R)$ is connected;
- The dart chosen as input of the algorithm belongs to L;
- At each identification of a renaming function from $\langle o\rangle$ to $\langle o^m\rangle$, κ is not in $\langle o^m\rangle$. Therefore, κ-arcs can only occur as explicit arcs in the reconstructed graph scheme.

The desired rule scheme is obtained by removing the κ-arcs from the output graph scheme. This removal yields two graph schemes corresponding respectively to the left and right parts of the rule scheme.

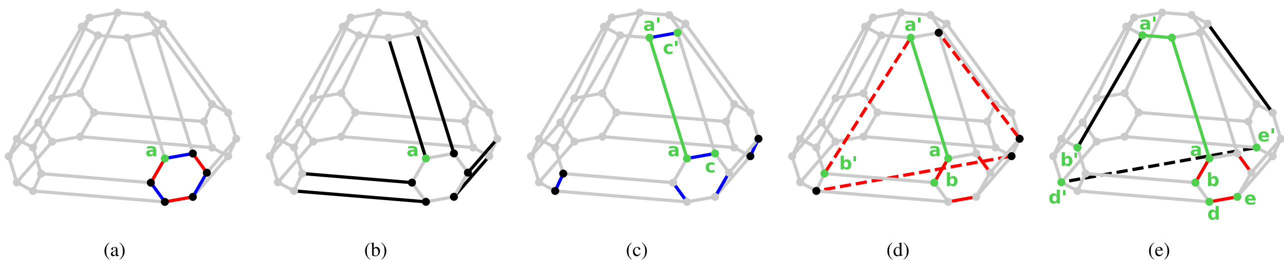


Fig. 19. Steps of the folding algorithm from a non-apex vertex of the pyramid. Step 1 (a) - construction of $G\langle 1,2\rangle(a)$. Step 2.1 (b) - construction of the hook's explicit arcs. Step 2.2 (c) and (d) - construction of a node and its implicit arcs. The algorithm stops on an inconclusive output (f). Arc color legend: — 3, — 2, — 1, and — 0.

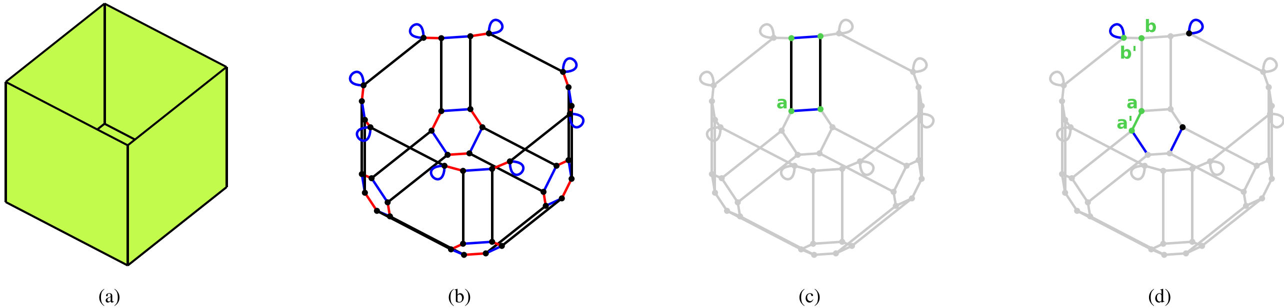


Fig. 20. Folding algorithm on a vertical edge of the box: (a) the object, (b) its G-map representation, (c) darts and links associated to the hook, (d) the algorithm stops on an inconclusive output.

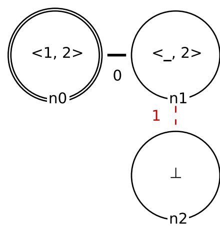


Fig. 21. Partial scheme before failure of the algorithm of Fig. 19(e).

Finally, the inferred rule scheme is checked against the conditions for the preservation of the topological constraints [17,18]. This verification ensures that the application of the rule scheme on a G-map can only produce a G-map (see Section 4.6).

If we carry on the example of Figs. 17 and 18, i.e., do the algorithm on the graph of Fig. 22(a) with the orbit $\langle 1,2\rangle$, we obtain the rule scheme of Fig. 22(b). The example was given for pedagogical purposes and the rule scheme reconstruction of the rule scheme presented in Fig. 14 will be discussed in Section 9.

8. Algorithm analysis

We now provide some analysis of correctness and complexity for the presented algorithm.

8.1. Correctness analysis

The proof of correctness essentially results from one fact. At each step of the algorithm, the partial graph pattern correctly folds all links incident to darts associated with a node with a defined label and expanded arcs. The cases of failures for the algorithm are either the impossibility to add an arc to the partial graph pattern or the impossibility to determine a legal relabeling function to add to a node. In the case of the node failure, the issue translates into an asymmetry in the operation. Conversely, an arc failure means a discontinuity in the topology. The issue on an arc is actually twofold. Either all other endpoints of the links do not share the same association with a node in the graph scheme (instantiation failure), or the other endpoints do not correspond to the image of the same dart from the orbit used as a reference for the instantiation (arc failure). The issue on the node's label is relatively less complicated as the only real issue is the hopelessness to build a joint relabeling value for a given dimension. The complete proof of correctness is presented in Appendix A.

8.2. Topological consistency

The folding algorithm provides a solution to generalize an operation from a representative example. This generalization is essentially obtained by finding symmetries (up to relabeling) in the operation. Such a process tries to reverse the instantiation of a graph scheme. However, the generalization of an operation might be topologically incorrect. Previous works have already studied the preservation of the topological consistency [9,18]. They provide topological conditions to ensure that applying a rule scheme to a G-map always provides a G-map. Therefore, every rule scheme obtained by our algorithm is checked to discard ill-formed ones. A discarded rule is an inconsistent rule with respect to the (quasi-)manifold property of G-maps.

Since the two instances used to infer the operation should be well-formed G-maps, the rule on the empty orbit is always a plausible operation and naturally satisfies the topological conditions. Intuitively, the inconsistent rules come from incoherent generalization, i.e., from the orbit used for the inference or mapping. However, the folding algorithm might produce a folded representation of the transformation for the user-provided orbit and matching. We illustrate such a possibility in Fig. 23. The before and after instances respectively consist of triangular and hexagonal faces. The complete topological structures are given. These are 2D objects, and we can try to infer the associated operation with the most general orbit, i.e., the orbit $\langle 0,1,2\rangle$. We now consider two possible mappings.

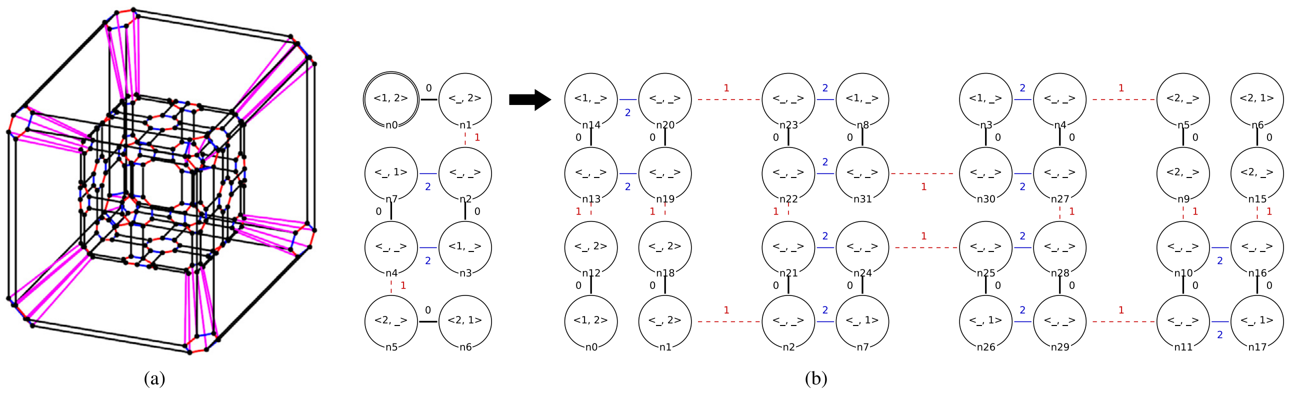


Fig. 22. The graph $\kappa(\mathbf{L}, \mathbf{R})$ (a) for quad subdivision of the cube and the rule scheme (b) obtained by resuming the algorithm after the folding of the cube displayed in Fig. 18. Arc color legend: ▬ 3, ▬ 2, ▬ 1, and ▬ 0.

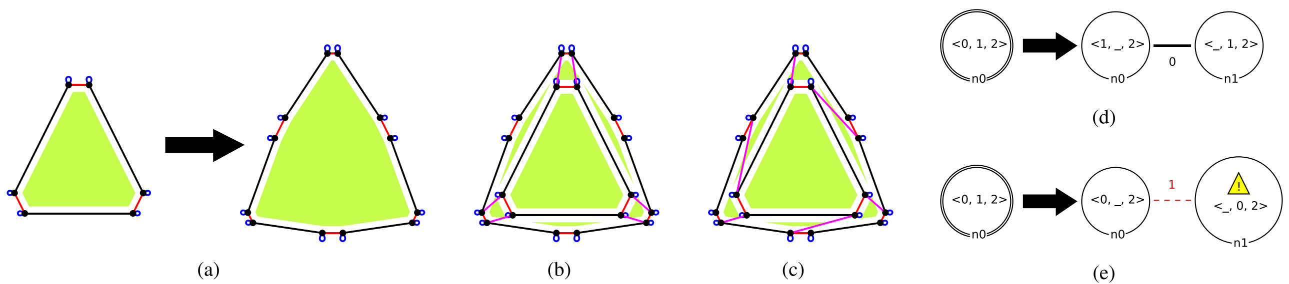


Fig. 23. Topological consistency: (a) before instance is a triangle and after instance is an hexagon, (b) the mapping induces an edge split, (c) the mapping induces a vertex split, (d) folded rule for the mapping of Figure (b), (e) incorrect rule obtained with the mapping of Figure (c).

In Fig. 23(b), the mapping (indicated by the κ-arcs in pink) induces an edge split. Each vertex, i.e., the orbits $\langle 1, 2\rangle$, is preserved by the operation. The new darts split the edges by inserting new vertices. This mapping yields the rule of Fig. 23(d), detected as well-formed. This operation inserts a vertex to split each edge when applied on a surface.

In Fig. 23(c), the mapping induces a vertex split by preserving the orbits $\langle 0, 2\rangle$, i.e., the edges. With this mapping, the operation inserts edges to split the vertices. Although valid on a curve (see Fig. 23(c)), this operation becomes topologically incorrect when applied on a surface where 2-links are not loops. For instance, inserting an edge in each vertex of a cube is impossible. However, the topological folding algorithm, which is essentially a graph traversal algorithm, can produce a folded representation of this operation from the two faces. The rule is given in Fig. 23(e). Via a syntax checker, we find that this operation would break the topological consistency, as illustrated by the flag on node $n1$.

Therefore, this rule is discarded, and the algorithm outputs that no operation can be inferred.

8.3. Complexity analysis

The algorithm runs in time $\mathcal{O}(|G|)$ for a G-maps G. The complexity does not depend on the size of the orbit type $\langle o\rangle$ nor on the size of the orbit graph $G\langle o\rangle(a)$. Step 1 is achieved as a breadth-first search on a in G and takes time $\mathcal{O}(|G\langle o\rangle(a)|)$. Step 2.1 requires checking for each dimension not in $\langle o^m\rangle$ whether all darts in the instantiation of m have a coherent incident arc for that dimension. The number of dimensions is bounded by n, and there are $|G\langle o\rangle(a)|$ darts for which the incident arcs should be checked. Thus Step 2.1 requires $\mathcal{O}(|G\langle o\rangle(a)|)$ time. Step 2.2 requires building the relabeling function. When we build a node in the scheme, we store, for each dart, its associated dart in $G\langle o\rangle(a)$. Therefore, building a relabeling means comparing the dimensions of the arcs incident to a with the arcs incident to its associated dart in the current node. Since there are at most n arcs incident to a, building the relabeling takes time $\mathcal{O}(1)$. The consistency of the relabeling must be checked for each dart associated with the current node. By construction, there are $|G\langle o\rangle(a)|$ such darts. Again, the number of possible arcs to check is bounded by n, resulting in a complexity of $\mathcal{O}(|G\langle o\rangle(a)|)$ for Step 2.2. Step 2 requires to iterate Steps 2.1 and 2.2 until the whole graph has been traversed. The number of iterations is directly equal to the number of nodes in the resulting rule scheme. Each node in the rule scheme corresponds to $|G\langle o\rangle(a)|$ darts in the initial G-map. Therefore the folded representation will have $\frac{|G|}{|G\langle o\rangle(a)|}$ nodes and the overall complexity is $\mathcal{O}(|G|)$. On two instances that compose a rule, the complexity is the same for both the left-hand side and the right-hand side, unified using the κ-arcs. Thus the full complexity to build the rule scheme from a before G-map G and an after G-map H is $\mathcal{O}(|G| + |H|)$, i.e., a linear complexity.

9. Application to subdivision schemes

All the rules and objects presented in this article have been realized within the Jerboa framework [48]. Jerboa is a topology-based geometric modeling platform that allows for the edition and application of rule schemes on objects represented with G-maps. We implemented our algorithm in Jerboa.

Jerboa contains a rule editor, which enables the design of modeling operations. For instance, the rule from Fig. 25(a) was exported from Jerboa's editor. This editor realizes a syntactic verification of the rule [19,48] to detect both topological and geometric inconsistencies. For example, the warning pictogram in Fig. 23(e) highlights that Jerboa's syntax checker found an inconsistency in the rule. This syntax checker can also be used to find the missing geometric computations on the inferred rule. We can manually write the missing computation using the embedding expressions (see Section 4.5 and [19]).

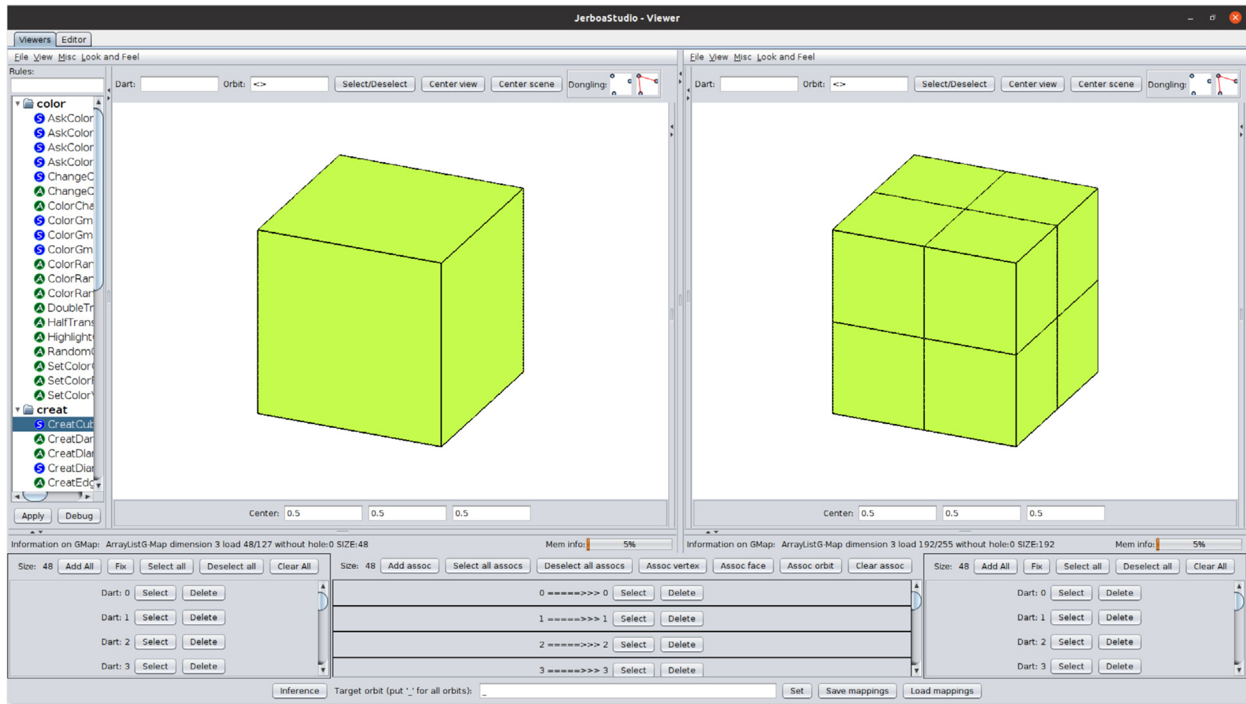


Fig. 24. Jerboa's interface for the inference of operations.

In practice, the user can manipulate the object with Jerboa's default rules or import existing objects into our tool. The user must provide the mapping of the preserved darts from the left pattern to the right pattern, either explicitly or implicitly. One solution is to modify the object in our dedicated viewer directly. In this case, the user takes two snapshots before and after its modification. We then build the mapping from IDs of the darts present in both snapshots. The user can also create two instances and manually provide the mapping. We allow the mapping of orbits (and thus cells) to simplify the process. Fig. 24 shows the interface we developed with Jerboa. It contains two viewers to display and modify the before instance (on the left) and the after instance (on the right). At the bottom, the user provides the mapping of the darts via orbit matching or leaves the association section empty to use the identity function on the dart IDs as mapping. Our tool is freely available online[1].

We reconstructed several subdivision schemes as a validation of our algorithm. Note that we only inferred the topological modifications and added the missing geometry manually. We first present applications to surfaces (Section 9.1). Afterward, we illustrate our approach with operations on volumes (Section 9.2).

9.1. Subdivision schemes for surface refinement

We reconstructed several subdivision schemes for surfaces, i.e., with the orbit $\langle 0, 1, 2\rangle$ as inference parameter.

9.1.1. Quad subdivision and Catmull–Clark

We first inferred the quad subdivision scheme presented previously. Using the two generalized maps of Fig. 13 and the orbit $\langle 0, 1, 2\rangle$, we obtain the rule given in Fig. 25(a). This rule corresponds to the one presented in Fig. 14 made manually with Jerboa's rule editor. We can now apply the operation on another quadrangulation, for instance, on Suzanne (see Fig. 25(b)).

[1] Link to website (last consulted on May, 18th 2022): http://xlim-sic.labo.univ-poitiers.fr/jerboa/doc/topological-inference-for-subdivision-schemes/.

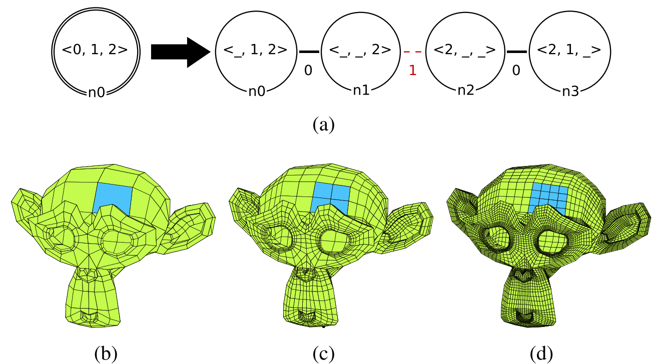


Fig. 25. Quad subdivision operation: inferred rule (a) from the objects of Figs. 1. Suzanne (b), first (c) and second (d) iterations of the quad subdivision.

We obtain the first and second iterations of the subdivision scheme, as presented in Figs. 25(c) and 25(d). We manually added an expression computing the position of the new vertices respectively as the barycenter of the faces and edges. This geometric computation is illustrated in the iterative sequence of Fig. 26(a) to 26(d). This subdivision is topologically equivalent to the Catmull–Clark subdivision [49]. Therefore, the addition of a smoothing computation on the inferred operation yields the desired refinement scheme, illustrated in Figs. 26(e) to 26(h). The inferred rules are identical. The only difference concerns the geometric computations added. For instance, the vertex added to split the edge corresponds to the rule's nodes $n1$ and $n2$. For the quad subdivision, the new position is computed as the middle of the edge's vertices with the expression:

```
// mid point of the edge
return Point3::middle(<0>_position(n0));
```

The smoothing for the Catmull–Clark subdivision requires displacing the vertex. Using the standard geometric refinement from [49], we obtain the expression:

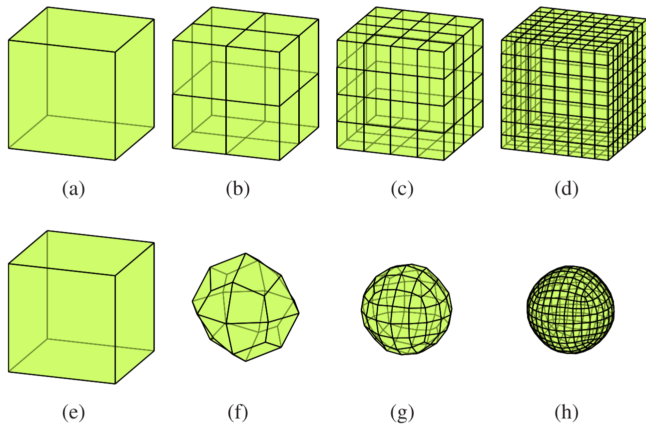


Fig. 26. The quad subdivision operation is topologically equivalent to Catmull–Clark. Iterative sequence for the quad subdivision (a), (b), (c), and (d). Iterative sequence for Catmull–Clark (e), (f), (g), and (h).

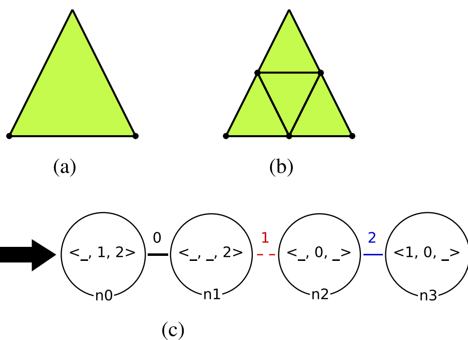


Fig. 27. Loop subdivision operation: the initial object (a), the first (b) iterations of the subdivision, and the inferred operation (c).

```
// mid point of the incident face
Point3 face1Mid = Point3::middle(<0,1>_position(n0));
// mid point of the adjacent face
Point3 face2Mid = Point3::middle(<0,1>_position(n0@2));
// average of the face points
Point3 faceMid = Point3::middle(face1Mid, face2Mid);
// mid point of the edge
Point3 edgeMid = Point3::middle(<0>_position(n0));
// average of the edge and face points
return Point3::middle(faceMid, edgeMid);
```

9.1.2. Loop and butterfly

One standard refinement of triangulated surfaces is the Loop subdivision scheme [37]. This scheme splits each edge by inserting its midpoint vertex. The new vertices are linked with edges in each face, dividing each triangle into four new triangles. One iteration on a triangle is illustrated in Figs. 27(a) and 27(b). From these two instances, we infer the rule scheme of Fig. 27(c). We added the standard masks to compute the position of the new vertices and Warren's simplified weights [50] for the smoothing of the old vertices. This subdivision approximates the initial surface. Choosing different geometric computations yields the interpolation scheme known as the butterfly subdivision [51]. Iterations of the subdivisions are provided in Fig. 28.

9.1.3. Powell–Sabin

Powell and Sabin studied various subdivision schemes on triangles [52] that ensure given values for the first derivatives on vertices. Such subdivisions are still studied to construct smooth

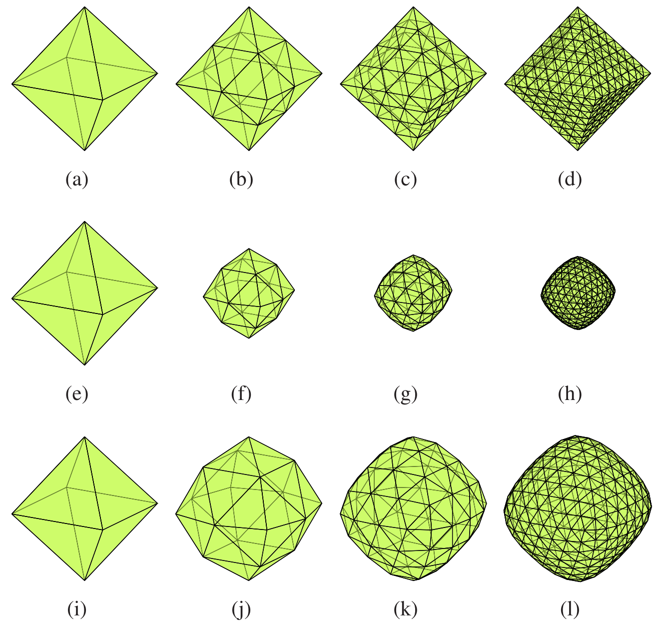


Fig. 28. Triangular subdivisions with topologically equivalent refinements. Iterative sequence for the non-refined subdivision (a), (b), (c), and (d). Iterative sequence for Loop algorithm (e), (f), (g), and (h). Iterative sequence for Butterfly subdivision (i), (j), (k), and (l).

finite element spaces [53]. The Powell–Sabin 6-split refines a triangle into six new triangles. The operation adds a vertex at the center of the face and links it with all vertices of the triangle as well as all midpoints of the initial edges. Assuming we only have three basic operations: split an edge, add a vertex, and link two vertices by an edge, we can reconstruct the Powell–Sabin 6-split refinement as described in Fig. 29(a) to 29(d). Inferring the rule scheme of Fig. 29(e), for the connected component (orbit $\langle 0, 1, 2, 3 \rangle$), takes around 5 ms. We apply the inferred operation on the triangulation of the unit square illustrated in [53], see Figs. 29(f) to 29(h).

9.1.4. Kobbelt's $\sqrt{3}$

The $\sqrt{3}$-algorithm developed by Kobbelt is topologically distinct from the previous subdivision schemes in the sense that it removes (or rather flips) edges. The algorithm is usually described in a two-step process. First, vertices are added at the center of each triangle and linked with edges to the original face vertices. From the triangles of Fig. 30(a), we obtain the mesh of Fig. 30(b). Then every edge of the initial mesh is flipped, as illustrated in Fig. 30(c). We inferred this operation on a tetrahedron. The initial and final objects are provided in Figs. 31(a) and 31(c). The associated 2-G-maps are given in Figs. 31(b) and 31(d), while the inferred operation is illustrated in Fig. 31(e). With the addition of the vertices geometric smoothing, we obtain the subdivision scheme defined by Kobbelt. An illustration of the final operation on the Stanford Bunny is provided in Figs. 31(f) and 31(g).

9.1.5. Doo-Sabin

The Doo-Sabin subdivision works with any surface [54] by recursively splitting the vertices. Fig. 33(a) presents the initial object, while Fig. 33(b) displays the first iteration of the subdivision. From these two objects, we infer the rule of Fig. 33(c). We now have a standalone rule, applicable to any isolated surface, which refines objects as many times as desired. For instance, we can further subdivide the object of Fig. 33(b) to obtain the second and third iterations, respectively illustrated in Figs. 33(d), and 33(e).

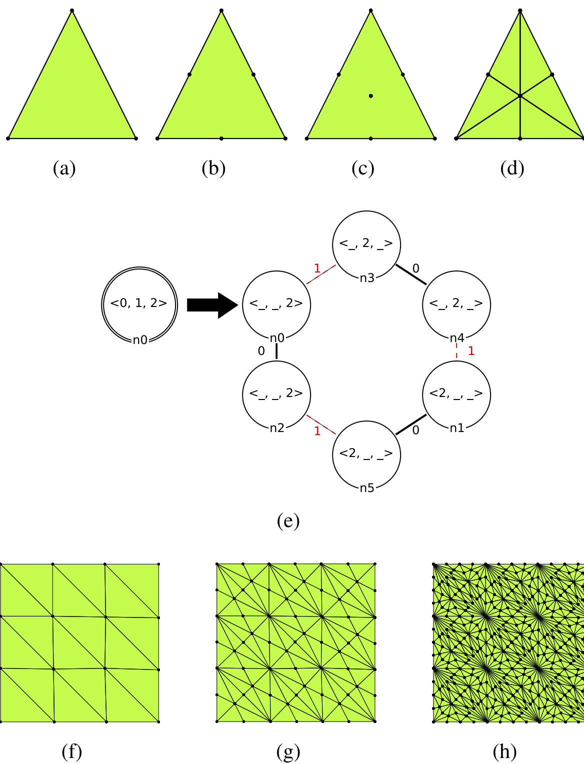


Fig. 29. Powell–Sabin 6-split refinement: given a triangle (a), split the three edges (b), add a vertex (c), and link it with all other vertices (d). The inferred rule (e) is recursively applied on a triangulation of the unit square (f) (from [53]) to obtain more refined triangulations (g) and (h).

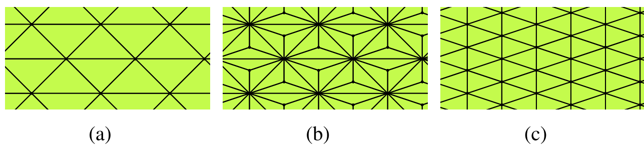


Fig. 30. Two-step illustration of the $\sqrt{3}$ algorithm: from an initial triangular mesh (a), faces are triangulated with a vertex at the center (b), and old edges are flipped (c).

From these iterated applications, we can infer new operations directly producing the second or third iteration of the Doo-Sabin subdivision. The inferred rules are illustrated in Figs. 32(a) and 32(b). In other words, our mechanism allows for straightforward self-composition of operations. Inferring the rule of the third iteration takes around 40 ms, which means our approach is usable in practice.

9.2. Subdivision schemes for volume refinement

We reconstructed two subdivision schemes for volume refinement, i.e., with the orbit $\langle 0, 1, 2, 3 \rangle$ as inference parameter.

9.2.1. Menger sponge

The Menger sponge is a 3D extension of the Cantor set (1D) or the Sierpinski carpet (2D). We can compute it as a fractal and now describe the construction of a refinement step. Take a cube and split each face into 9 squares to obtain 27 cubes. Remove all middle cubes (middle of faces and center of the initial cube). The 20 remaining cubes correspond to the iteration of the refinement step. This step is iterated on the newly obtained cubes. From the cube of Fig. 34(a), we construct the first iteration of the Menger

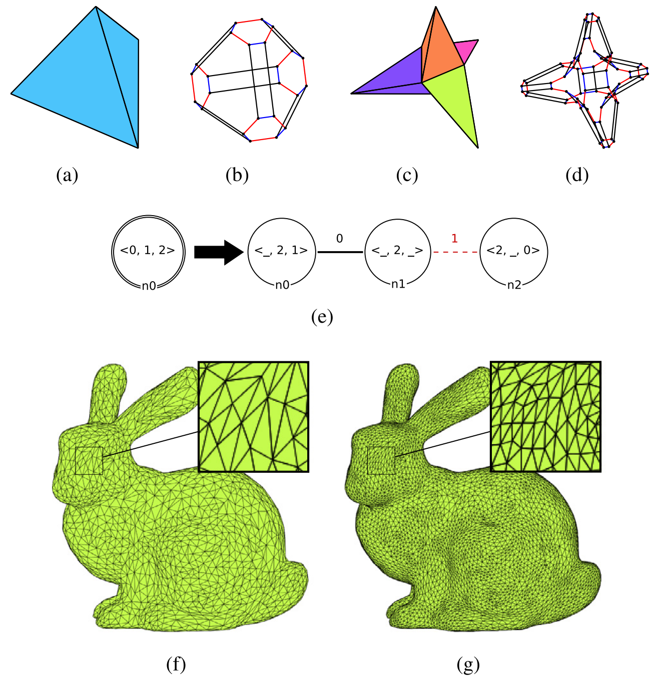


Fig. 31. Inference and application of the $\sqrt{3}$ operation: (a) tetrahedron as the "before" instance, (b) G-map of the "before" instance, (c) refined object as the "after" instance, (d) G-map of the "after" instance, and (e) inferred rule scheme for the orbit $\langle 0, 1, 2 \rangle$. After the addition of the missing geometry: (f) initial object with 4968 faces, (g) result of one subdivision (object with 14,904 faces).

sponge illustrated in Fig. 34(b). We can infer the operation by specifying that the operation occurs on the orbit $\langle 0, 1, 2, 3 \rangle$ to obtain the rule of Fig. 34(e). Note that this inferred rule has 20 nodes on its right-hand side, which might already prove challenging to write (or read). Our inference mechanism alleviates the fastidious task of manually designing such complex operations. We can now iterate the inferred operation and obtain the following iterations of the Menger sponge (second and third iterations in Figs. 34(c) and 34(d)). The inferred operation directly produces the Menger sponge's second iteration. The rule has more than 400 nodes and is too large to draw properly.

9.2.2. (2,2,2)-Menger sponge

In [55], the authors proposed a generalization of the Menger sponge to Menger polycube. One iteration of the (L, M, N)-Menger operation transforms a polycube into a polycube with L holes along the x-axis, M holes along the y-axis, and N holes along the z-axis. Each hole has the same size as a one-unit cube and is separated from the nearest holes by a one-unit cube. From a cube, we built the first iteration of the $(2, 2, 2)$-Menger operation (see Fig. 35(b)). The polycube is of genus 28 and consists of 81 volumes, 270 faces, 216 vertices. To our knowledge, there is no definition (either algorithmic or with a rule) of this operation. We built the polycube and used our algorithm to infer the operation. From the objects of Figs. 35(a) and 35(b), we inferred the rule of Fig. 35(e). We used this operation to build the second and third iterations, respectively illustrated in Figs. 35(c) and 35(d). For information, it takes around 100 ms to infer the $(2, 2, 2)$-Menger operation.

The inferred operations for both Menger and $(2, 2, 2)$-Menger allows modifying any volume. Fig. 36 shows the first iteration of the subdivision on three solids. We added inside lights to better show the holes. The modified objects are all topologically correct, although our geometric computations inherited from the cube might seem off as the holes are much more prominent for Menger than for $(2, 2, 2)$-Menger.

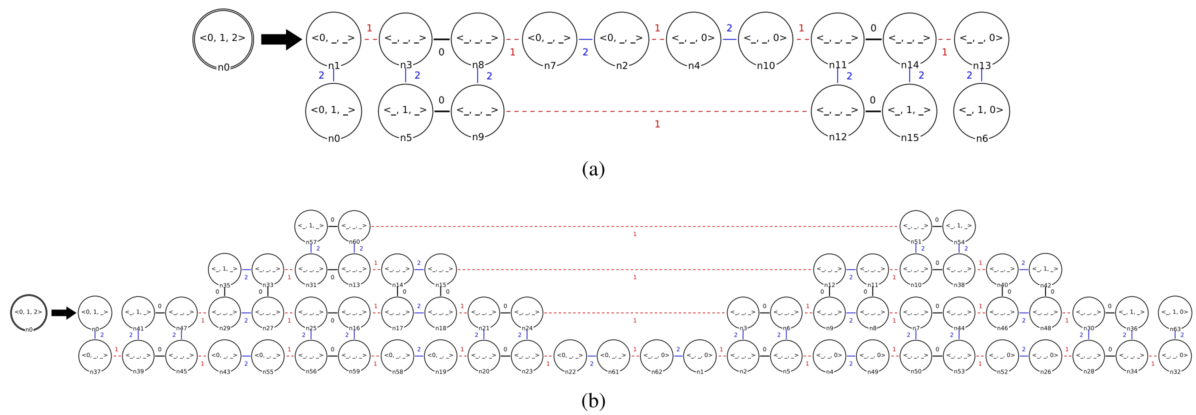


Fig. 32. Inferred rules for the second (a) and third (b) iterations of the Doo-Sabin subdivision.

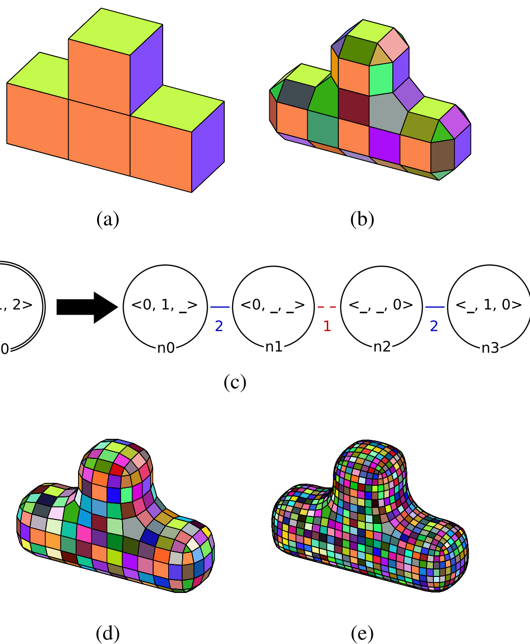


Fig. 33. Doo-Sabin subdivision operation: the initial object (a), the first (b), second (d) and third (e) iterations of the subdivision. The inferred operation (c) from the objects (a) and (b).

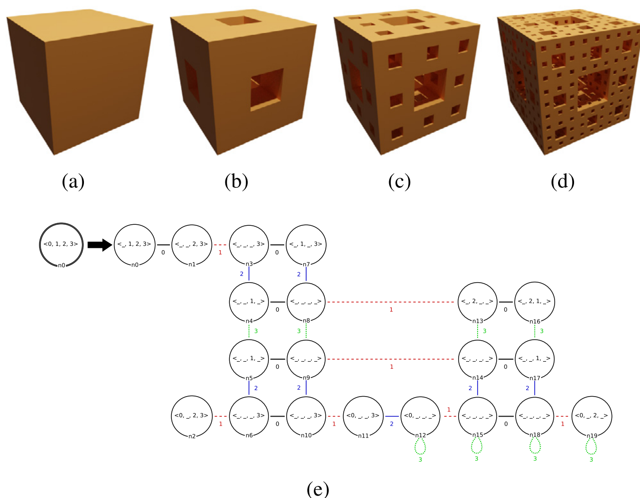


Fig. 34. A cube (a), the first (b), second (c) and third (d) iterations of the Menger sponge. The inferred operation (e) from the objects of Figures (a) and (b).

10. Advanced exploitation

We discuss certain practical side-effects of our inference mechanism. The benefits and limitations are mainly related to Jerboa's formalism to handle modeling operations on generalized maps.

10.1. Target cell parameter

Operations are inferred for a given orbit. Intuitively, an orbit is an abstraction of a cell (or a subcell), described by a subset of dimensions. Therefore, the inferred rule for Powell–Sabin 6-split does not specify that it subdivides triangles: we can apply it to any surface. For instance, we can use this operation to triangulate the quad mesh of Fig. 37(a). We obtain the triangulated mesh of Fig. 37(b). We can iterate the subdivision to obtain the mesh of Fig. 37(c). Similarly, the representation of Suzanne in Fig. 25(b) is a surface that only consists of quads. We can use the inferred operation of Fig. 29(e) to obtain a mesh triangulation, as illustrated in Fig. 37(d).

The proposed algorithm folds the graph correctly to obtain a rule by traversing all darts of the application example. The inferred operation is always valid for the provided example but might be too sensitive in some instances. In particular, the external, unmodified parts of the object may be captured by the generated operation, hindering its applicability (see Section 8.2).

10.2. Inference of other operations

We presented several subdivision schemes in Section 9. However, our approach is not limited to these specific operations. Indeed, the algorithm presented in Section 5 takes as input two generalized maps and an orbit type. Any topological operation can be inferred. For instance, if we transform the square face of Fig. 38(a) into the cube of Fig. 38(d), we infer the rule presented in Fig. 38(g) when specifying the face as the input cell. This operation corresponds to the face extrusion. We can visualize the operation by applying it on the triangle of Fig. 38(b) and the octagon of Fig. 38(c) to obtain the prisms of Figs. 38(e) and 38(f).

10.3. Plurality of inferred operations

We explained in Section 5 our algorithm for inferring topological operations from an example for a given orbit. Intuitively, the algorithm tries to fold the $\kappa(L, R)$ graph for the provided orbit by choosing an initial dart. Throughout the paper, we illustrated our approach with iterated function systems where the left-hand side of the rule consists of only one node. Therefore, any initial dart yields the same rule. It could be the case that several rules

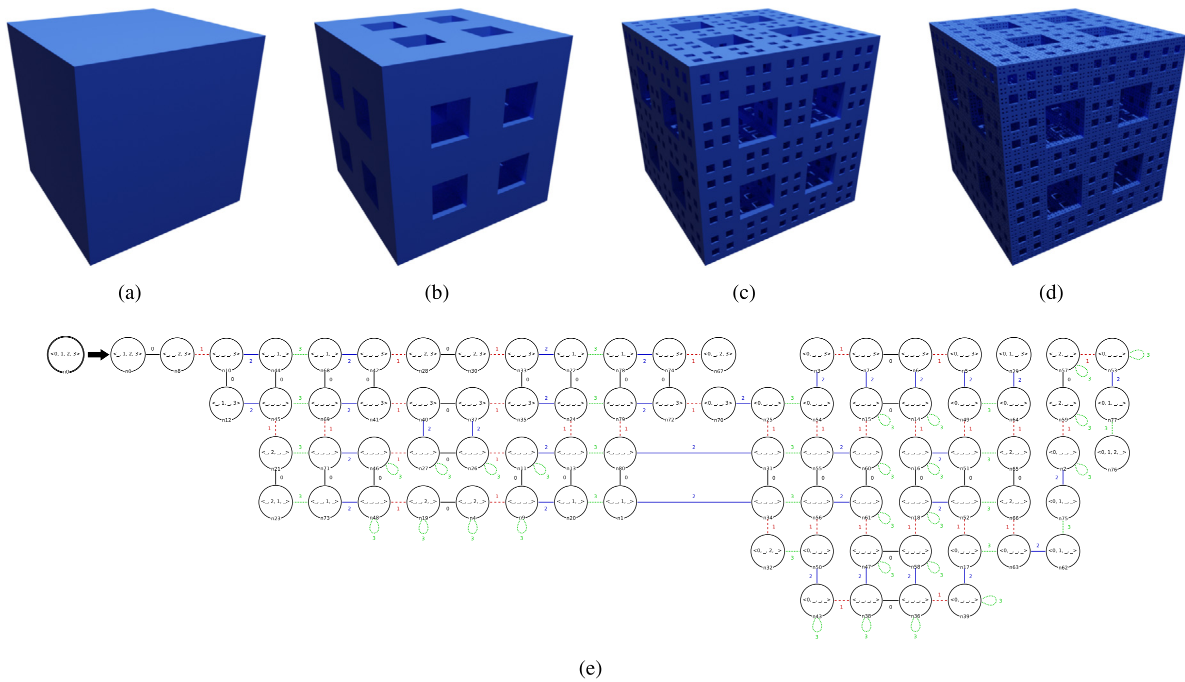


Fig. 35. A cube (a), the first (b), second (c) and third (d) iterations of the (2, 2, 2)-Menger operation. The inferred operation (e) from the objects of Figures (a) and (b).

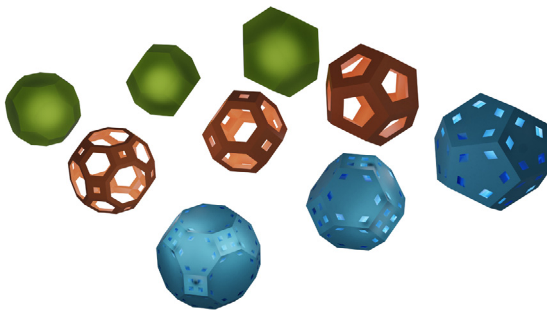


Fig. 36. Application of the inferred operations for the Menger operation on various solids: (green) initial volumes, (orange) first Menger refinement, (blue) first (2, 2, 2)-Menger refinement.

exist, e.g., when the left-hand side of the rule has more than one node. Besides, there are possibilities for failure when either the implicit or explicit arcs cannot be reconstructed. Thus, we implemented a mechanism marking the darts that correspond to the hook node during the computation of the algorithm. This mechanism avoids exploring darts that would yield an already found rule or fail to result in a rule. We can try the algorithm again with an unmarked dart until all darts are marked. Based on the symmetry of the initial and target objects, we might obtain the same rule several times. To discard duplicated rules, we use a vertex invariant algorithm similar to [56] with the addition of the node labels to speed up the computation [57].

11. Conclusion and perspectives

We presented an automated method to infer topological modeling operations from a representative instance. Our algorithm takes as input an instance of the operation application (two G-maps and a partial mapping between them) and the parametrization orbit. It produces as output a graph transformation rule parameterized by the input orbit type provided that such a graph transformation exists. Implementation was done in the Jerboa platform to exploit its formal language, ensuring the well-formedness of the inferred rules. Therefore, any inferred rule preserves the topological consistency of generalized maps, defined as edge-labeled graphs. Our approach for inferring topological operations exploits the orbit-based definition of rule schemes to fold objects along a given orbit. We experimented with our algorithm on various 2D and 3D objects for several orbit types, mainly for subdivision schemes. The inferred operation is directly applicable to any object by matching the rule's hook into

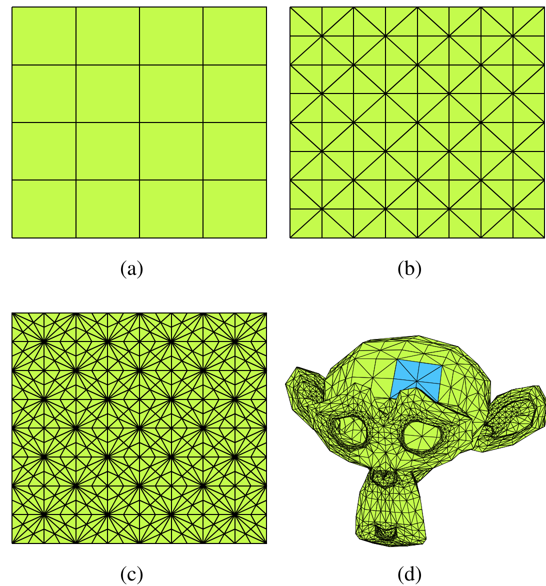


Fig. 37. Powell–Sabin 6-split on quad meshes: (a) a quad subdivision of the unit square, (b) its triangulation with Powell–Sabin 6-split, (c) further refinement of the triangulation, and (d) the triangulation of Suzanne.

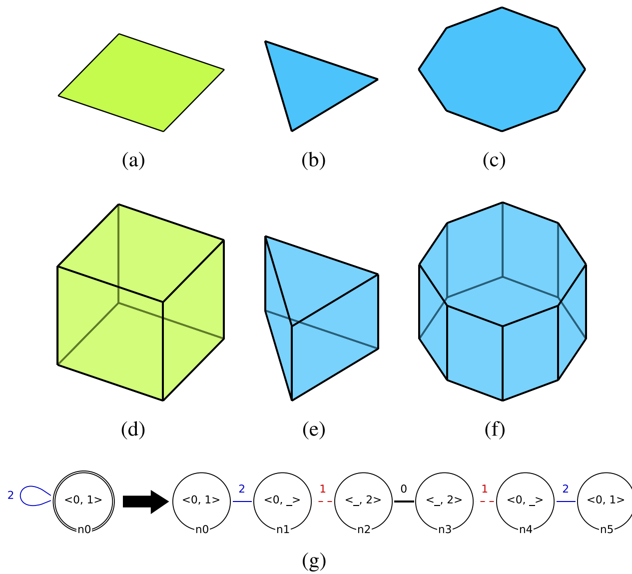


Fig. 38. Face extrusions: (a) a square face and (d) its face extrusion as a cube, (g) the inferred rule and its application to faces (b) and (e) to obtain prisms (c) and (f).

an orbit of the appropriated type in the object. Our approach offers the following two main benefits:

- First, a user unaccustomed to either topological models, i.e., generalized maps, or (graph) transformation rules, can design operations exclusively from examples.
- Secondly, a sequence of elementary operations can be optimized to generate a direct transformation that can speed up the design of complex scenes.

Our approach ensures that the topological part of modeling operations can be inferred without writing a single line of code in a standard programming language or designing a rule in Jerboa's expert language. It also hides the intern structure of Jerboa's rules which may be cumbersome, if not impracticable, to write or read with too many nodes. Finally, an automated and reliable inference mechanism offers an alternative approach for developing topology-based geometric operations.

Our work allows the inference of the topological part of modeling operations. Once the two instances, the mapping, and the target orbit type parameter are given, the inference is deterministic because there is essentially one solution. The inferred rules were manually edited to add the missing geometric computations before applying them again. Our work opens a new venue for the generation of modeling operations without programming knowledge. Indeed, inferring the geometric computations would lead to a low-code development platform for topology-based geometric modeling. Compared to its topological counterpart, the inference of the geometry is highly non-deterministic as several computations may lead to the same values for a given input. In any case, the inference of the missing geometry will need to be generic to benefit from the orbit-based generalization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Correctness analysis

Given a rule in Jerboa, i.e., a rule scheme, its application is formally described as a graph transformation using results from category theory. Among these results, the uniqueness (up to isomorphisms) of the result G-map is guaranteed. Similarly, the inferred rule's uniqueness (up to equivalent relabeling) from its instantiation is also guaranteed. Intuitively, the mechanism to infer an operation is the reserve of the application mechanism.

Let us assume that the algorithm provides a rule scheme $r : L \to R$ from two instances G and H of an object on a given orbit type $\langle o \rangle$ and a given dart a. It is sufficient to show that the instantiation of L and R on $G\langle o \rangle(a)$ are respectively isomorphic to G and H.

We aim at proving the correctness of the algorithm for a G-map G with orbit type $\langle o \rangle$ and initial dart a, i.e.,

- If the algorithm provides a graph scheme S, then its instantiation on $\langle o \rangle$ yields the initial graph G.
- The algorithm fails if there exists no scheme that instantiates in the initial graph G for the type $\langle o \rangle$ with preservation of a.

Before going into the proof, we introduce some notations:

- P denotes the partial graph scheme during the execution of the algorithm,
- V is the set of nodes of P,
- $V^{\perp}$ is the subset of V whose elements are labeled $\perp$, i.e., have not yet seen the construction of their orbit type.
- $V^{\odot}$ is the subset of V whose elements have a suitably defined label but have not yet seen the construction of their incident arcs.
- $V^{\circledast}$ is the subset of V whose elements have been assigned a label and whose incident arcs have been constructed.

Note that $V^{\perp}$, $V^{\odot}$, and $V^{\circledast}$ forms a partition of V. We can describe the algorithm through its operations on $V^{\perp}$, $V^{\odot}$, and $V^{\circledast}$. Step 1 initializes P as a graph with a single node h labeled $\langle o \rangle$. Thus at the end of this step, $V^{\perp} = \emptyset$, $V^{\odot} = \{h\}$, and $V^{\circledast} = \emptyset$. Step 2.1 moves a node m from $V^{\odot}$ to $V^{\circledast}$ and may add new elements to $V^{\perp}$. Similarly, Step 2.2 moves a node m from $V^{\perp}$ to $V^{\odot}$ without any other modification.

We also use the shorter notation $R_v^{\langle o \rangle}$ to denote the relabeling function $\langle o \rangle \mapsto \langle o^v \rangle$ for a node v in a graph scheme S.

By induction on the steps of the algorithm, we show that the partial graph P manipulated by the algorithm satisfies the following property $\mathcal{P}(P)$:

1. $\iota^{\langle o \rangle}(P, G\langle o \rangle(a))$ is the unique subgraph of G where (a, h) is mapped onto a.
2. For all nodes v in $V^{\circledast}$, no arc of the form $(b, v)\overset{i}{\bullet\!-\!\bullet}(c, v')$ exists in G that does not belong to $\iota^{\langle o \rangle}(P, G\langle o \rangle(a))$, where b and c are darts of $G\langle o \rangle(a)$, v' is a node of P, and i a dimension of $[\![0, n]\!]$.

Step 1 (Orbit graph and construction of the hook). Step 1 adds a unique node h labeled $\langle o \rangle$ and builds $G\langle o \rangle(a)$ through a graph traversal. Let P_h be the graph that only contains h. By construction $\iota^{\langle o \rangle}(P_h, G\langle o \rangle(a))$ is obtained via the identity function and provides a graph isomorphic to $G\langle o \rangle(a)$. Mapping (a, h) onto a ensures, via the incident ars property on G that $G\langle o \rangle(a)$ is essentially mapped onto itself. Besides, we have $V^{\circledast} = \emptyset$ at the end of the step. Thereafter $\mathcal{P}(P_h)$ holds.

Step 2.1 (Construction of the explicit arcs incident to a node). We consider a partial graph scheme P that satisfies $\mathcal{P}(P)$ and assume that the next action is the construction of the explicit arcs incident to a node m that belongs to $V^{\odot}$. We write $P \circledast m$ for the

graph scheme obtained by the addition or extension of arcs for all dimensions not in $\langle o^m \rangle$, assuming the algorithm does not halt on the 'arc failure' case.

Assume that the algorithm does not fail when constructing the explicit arcs incident to the node m. Let i be a dimension of $[\![0, n]\!] \setminus \langle o^m \rangle$ and e be the i-arc incident to m added to P. Assume that the addition of e violates the property $\mathcal{P}$. Since all arcs incident to m have not yet been extended, the second condition of $\mathcal{P}$ holds from P. Thus $\iota^{\langle o \rangle}(P \cup \{e\}, G\langle o \rangle(a))$ is not a subgraph of G when mapping (a, h) onto a. By induction hypothesis, one arc that results from $\iota^{\langle o \rangle}(e, G\langle o \rangle(a))$ does not belong to G. This contradicts the 'instantiation failure' condition from Step 2.1 and $\mathcal{P}(P \cup \{e\})$ holds. Once all the d-links incident to (a, m) have been considered (without failure), m belongs to $V^{\circledast}$. Assume that $\mathcal{P}(P \circledast m)$ does not hold. Since $\mathcal{P}(P \cup \{e\})$ holds for each e arc incident to m, the second condition of $\mathcal{P}(P \circledast m)$ is violated. An arc $(b, m) \overset{i}{\bullet\!-\!\bullet} (c, m)$ of G is not in the instantiation $\iota^{\langle o \rangle}(P \circledast m, G\langle o \rangle(a))$, where b and c are darts of $G\langle o \rangle(a)$, and i is a dimension. By induction hypothesis, $\mathcal{P}(P)$ held, and i cannot be a dimension of $\langle o^m \rangle$. Otherwise the construction of $\langle o^m \rangle$ would have resulted in a failure state. Thus, an i-link is incident to (a, m) in G. Because the algorithm did not stop at the 'arc failure' case, the link is of the form $(a, m) \overset{i}{\bullet\!-\!\bullet} (a, m')$ and lead to the creation of an arc $m \overset{i}{\bullet\!-\!\bullet} m'$. By the incident arcs property in G, no link $(b, m) \overset{i}{\bullet\!-\!\bullet} (b, m')$ can exist. Therefore the algorithm went into failure state for 'instantiation failure'. By contradiction, $\mathcal{P}(P \circledast m)$ holds.

Assume that the algorithm fails to run Step 2.1. In the case of 'arc failure', there is a dimension i in $[\![0, n]\!] \setminus \langle o^m \rangle$ such that the i-link incident to (a, m) is of the form $(a, m) \overset{i}{\bullet\!-\!\bullet} (b, m')$ with b $(\neq a)$ a dart of $G\langle o \rangle(a)$ and m' $(\neq m)$ a node of P. Such a link cannot be obtained by either an explicit or an implicit arc. An implicit arc would instantiate into a link $(a, m) \overset{i}{\bullet\!-\!\bullet} (b, m)$, which would not belong to G by the incident arcs constraint. An explicit arc would instantiate into a link $(a, m) \overset{i}{\bullet\!-\!\bullet} (a, m')$, which would not belong to G either, by the incident arcs constraint. Both cases would result in $\iota^{\langle o \rangle}(P \circledast m, G\langle o \rangle(a))$ not being a subgraph of G, once (a, h) is mapped onto a. In the case of 'instantiation failure', there are a dimension i in $[\![0, n]\!] \setminus \langle o^m \rangle$ and a dart c in $G\langle o \rangle(a)$ such that the i-link incident to (a, m) gave rise to an arc $m \overset{i}{\bullet\!-\!\bullet} m'$, but no link $(c, m) \overset{i}{\bullet\!-\!\bullet} (c, m')$ exists in G. The instantiation of the arc $m \overset{i}{\bullet\!-\!\bullet} m'$ would create an arc $(c, m) \overset{i}{\bullet\!-\!\bullet} (c, m')$ in $\iota^{\langle o \rangle}(P \circledast m, G\langle o \rangle(a))$. Therefore $\iota^{\langle o \rangle}(P \circledast m, G\langle o \rangle(a))$ would not be a subgraph of G, once (a, h) is mapped onto a.

Step 2.2 (Construction of a node label). We consider a partial graph scheme P that satisfies $\mathcal{P}(P)$ and assume that the next action is the construction of the label associated with a node m that belongs to $V^{\perp}$. Let $\mathcal{P} \odot m$ be the graph scheme obtained after the addition of the label $\langle o^m \rangle$ to m.

Assume that the algorithm does not fail when constructing the orbit type $\langle o^m \rangle$. The addition of $\langle o^m \rangle$ to m moves m from $V^{\perp}$ to $V^{\odot}$ without modifying $V^{\circledast}$. Thus the second condition of $\mathcal{P}(P \odot m)$ holds by induction hypothesis on P. Since the algorithm did not fail, for all i in $\langle o \rangle$ such that $R_m^{\langle o \rangle}(i) = j$ and $j \neq _$, for all b, c in $G\langle o \rangle(a)$ such that a link $b \overset{i}{\bullet\!-\!\bullet} c$ exists in $G\langle o \rangle(a)$, a link $(b, m) \overset{j}{\bullet\!-\!\bullet} (c, m)$ exists in $\iota^{\langle o \rangle}(P \odot m, G\langle o \rangle(a))$. In other words, the extension from P to $P \odot m$ results, through the instantiation mechanism in the addition of all links of $R_m^{\langle o \rangle}(G\langle o \rangle(a))$. Assume that the addition of these links violates $\mathcal{P}(P \odot m)$. Only the first condition can be violated, meaning that $\iota^{\langle o \rangle}(P \odot m, G\langle o \rangle(a))$ is not a subgraph of G, once (a, h) is mapped onto a. By induction hypothesis, $\mathcal{P}(P)$ holds. Therefore, one of the added link does not belong to G. Let i and j be the dimensions such that the superfluous link is of dimension j mapped from a i-link. Since the algorithm did not stop on the 'relabeling failure' case, the relabeling means that all links $b \overset{i}{\bullet\!-\!\bullet} c$ in $G\langle o \rangle(a)$ results in $(b, m) \overset{j}{\bullet\!-\!\bullet} (c, m)$ in G, contradicting that the link does not belong to G. Therefore, $\mathcal{P}(P \odot m)$ holds.

Assume that the algorithm fails to run Step 2.2, when constructing the label $\langle o^m \rangle$. The 'relabeling failure' was triggered by a dimension i from $\langle o \rangle$. Let b_i be the other extremity of the i-link incident to a. Because the algorithm failed a relabeling was decided for i. Let it be j, i.e., $R_m^{\langle o \rangle}(i) = j$. The failure state also means there are b and c in $G\langle o \rangle(a)$ such that $b \overset{i}{\bullet\!-\!\bullet} c$ belongs to $G\langle o \rangle(a)$ but $(b, m) \overset{j}{\bullet\!-\!\bullet} (c, m)$ does not belong to G. Therefore $\iota^{\langle o \rangle}(P \odot m, G\langle o \rangle(a))$ would not be a subgraph of G, once (a, h) is mapped onto a. $\square$

Termination. Since G is finite, the algorithm stops eventually. By induction, when it stops, all nodes of $\mathcal{S}$ belong to $V^{\circledast}$, i.e., have an orbit type as label and extended incident arcs. Since $\mathcal{P}(\mathcal{S})$ holds, $\iota^{\langle o \rangle}(\mathcal{S}, G\langle o \rangle(a))$ is the unique subgraph of G where (a, h) is mapped onto a. Besides, for all nodes v of $\mathcal{S}$, there exists no arc of the form $(b, v) \overset{i}{\bullet\!-\!\bullet} (c, v')$ in G that does not belong to $\iota^{\langle o \rangle}(\mathcal{S}, G\langle o \rangle(a))$, where b and c are darts of $G\langle o \rangle(a)$, v' is a node of $\mathcal{S}$, and i a dimension of $[\![0, n]\!]$. Since G is connected, $\iota^{\langle o \rangle}(\mathcal{S}, G\langle o \rangle(a))$ is the complete graph G, once (a, h) is mapped onto a. $\square$

Appendix B. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.gvc.2022.200049.

References

[1] Gould D. Complete maya programming: An extensive guide to MEL and C++ API. Elsevier; 2003.

[2] Squillacote AH, Ahrens J, Law C, Geveci B, Moreland K, King B. The paraview guide, Vol. 366. Kitware Clifton Park, NY; 2007.

[3] Conlan C. The blender python API: Precision 3D modeling and add-on development. A Press; 2017.

[4] Damiand G, Lienhardt P. Combinatorial maps: Efficient data structures for computer graphics and image processing. CRC Press; 2014.

[5] Lienhardt P. Subdivisions of N-dimensional spaces and N-dimensional generalized maps. In: Proceedings of the fifth annual symposium on computational geometry. SCG '89, New York, NY, USA: Association for Computing Machinery; 1989, p. 228–36. http://dx.doi.org/10.1145/73833.73859.

[6] Lienhardt P. Topological models for boundary representation: a comparison with n-dimensional generalized maps. Comput Aided Des 1991;23(11):59–82. http://dx.doi.org/10.1016/0010-4485(91)90100-B.

[7] Edmonds JR. A combinatorial representation for oriented polyhedral surfaces. (Ph.D. thesis), University of Maryland; 1960.

[8] Akleman E, Chen J, Gross JL. Extended graph rotation systems as a model for cyclic weaving on orientable surfaces. Discrete Appl Math 2015;193:61–79. http://dx.doi.org/10.1016/j.dam.2015.04.015.

[9] Poudret M, Comet JP, Le Gall P, Arnould A, Meseure P. Topology-based geometric modelling for biological cellular processes. In: International conference on language and automata theory and applications. 2007, p. 497–508.

[10] Bellet T, Poudret M, Arnould A, Fuchs L, Le Gall P. Designing a topological modeler kernel: A rule-based approach. In: Shape modeling international conference. IEEE; 2010, p. 100–12. http://dx.doi.org/10.1109/SMI.2010.31.

[11] Rozenberg G, editor. Handbook of graph grammars and computing by graph transformation: Vol. I. Foundations. USA: World Scientific Publishing Co., Inc.; 1997.

[12] Ehrig H, Ehrig K, Prange U, Taentzer G. Fundamentals of algebraic graph transformation. Monographs in theoretical computer science. An EATCS series, Berlin Heidelberg: Springer-Verlag; 2006, http://dx.doi.org/10.1007/3-540-31188-2.

[13] Heckel R, Taentzer G. Graph transformation for software engineers: with applications to model-based development and domain-specific language engineering. Cham: Springer International Publishing; 2020, http://dx.doi.org/10.1007/978-3-030-43916-3, ISBN 978-3-030-43915-6 978-3-030-43916-3.

[14] Prusinkiewicz P, Lindenmayer A, Hanan J. Development models of herbaceous plants for computer imagery purposes. In: Proceedings of the 15th annual conference on computer graphics and interactive techniques. SIGGRAPH '88, vol. 22, New York, NY, USA: Association for Computing Machinery; 1988, p. 141–50. http://dx.doi.org/10.1145/54852.378503.

[15] Prusinkiewicz P, Samavati F, Smith C, Karwowski R. L-system description of subdivision curves. Int J Shape Model 2003;09(01):41–59. http://dx.doi.org/10.1142/S0218654303000048.
[16] Smith C, Prusinkiewicz P, Samavati F. Local specification of surface subdivision algorithms. In: Pfaltz JL, Nagl M, Böhlen B, editors. Applications of graph transformations with industrial relevance. Lecture notes in computer science, Berlin, Heidelberg: Springer; 2004, p. 313–27. http://dx.doi.org/10.1007/978-3-540-25959-6_23.
[17] Poudret M, Arnould A, Comet J-P, Le Gall P. Graph transformation for topology modelling. In: Ehrig H, Heckel R, Rozenberg G, Taentzer G, editors. Graph transformations. Lecture notes in computer science, vol. 5214, Berlin, Heidelberg: Springer; 2008, p. 147–61. http://dx.doi.org/10.1007/978-3-540-87405-8_11.
[18] Pascual R, Le Gall P, Arnould A, Belhaouari H. Topological consistency preservation with graph transformation schemes. Sci Comput Program 2022;214:102728. http://dx.doi.org/10.1016/j.scico.2021.102728.
[19] Bellet T, Arnould A, Belhaouari H, Le Gall P. Geometric modeling: Consistency preservation using two-layered variable substitutions. In: de Lara J, Plump D, editors. Graph transformation. Lecture notes in computer science, Cham: Springer; 2017, p. 36–53. http://dx.doi.org/10.1007/978-3-319-61470-0_3.
[20] Smelik RM, De Kraker KJ, Tutenel T, Bidarra R, Groenewegen SA. A survey of procedural methods for terrain modelling. In: Proceedings of the CASA workshop on 3D advanced media in gaming and simulation. 2009, p. 25–34.
[21] Müller P, Wonka P, Haegler S, Ulmer A, Van Gool L. Procedural modeling of buildings. In: ACM SIGGRAPH 2006 papers. SIGGRAPH '06, New York, NY, USA: Association for Computing Machinery; 2006, p. 614–23. http://dx.doi.org/10.1145/1179352.1141931.
[22] Parish YIH, Müller P. Procedural modeling of cities. In: Proceedings of the 28th annual conference on computer graphics and interactive techniques. SIGGRAPH '01, New York, NY, USA: Association for Computing Machinery; 2001, p. 301–8. http://dx.doi.org/10.1145/383259.383292.
[23] Stava O, Pirk S, Kratt J, Chen B, Měch R, Deussen O, Benes B. Inverse procedural modelling of trees. Comput Graph Forum 2014;33(6):118–31. http://dx.doi.org/10.1111/cgf.12282.
[24] Wu F, Yan DM, Dong W, Zhang X, Wonka P. Inverse procedural modeling of facade layouts. ACM Trans Graph 2014;33(4). http://dx.doi.org/10.1145/2601097.2601162.
[25] Garcia-Dorado I, Aliaga DG, Bhalachandran S, Schmid P, Niyogi D. Fast weather simulation for inverse procedural design of 3D urban models. ACM Trans Graph 2017;36(2). http://dx.doi.org/10.1145/2999534.
[26] Emilien A, Vimont U, Cani MP, Poulin P, Benes B. WorldBrush: interactive example-based synthesis of procedural virtual worlds. ACM Trans Graph 2015;34(4). http://dx.doi.org/10.1145/2766975.
[27] Hu Y, Dorsey J, Rushmeier H. A novel framework for inverse procedural texture modeling. ACM Trans Graph 2019;38(6). http://dx.doi.org/10.1145/3355089.3356516.
[28] Št'ava O, Beneš B, Měch R, Aliaga DG, Krištof P. Inverse procedural modeling by automatic generation of L-systems. Comput Graph Forum 2010;29(2):665–74. http://dx.doi.org/10.1111/j.1467-8659.2009.01636.x.
[29] Prusinkiewicz P, Lindenmayer A. The algorithmic beauty of plants. New York, NY: Springer New York; 1990.
[30] Rozenberg G, Salomaa A. The mathematical theory of L systems. Academic Press; 1980.
[31] Guo J, Jiang H, Benes B, Deussen O, Zhang X, Lischinski D, Huang H. Inverse procedural modeling of branching structures by inferring L-systems. ACM Trans Graph 2020;39(5):155:1–155:13. http://dx.doi.org/10.1145/3394105.
[32] Santos E, Coelho RC. Obtaining L-systems rules from strings. In: 2009 3rd Southern conference on computational modeling. 2009, p. 143–9. http://dx.doi.org/10.1109/MCSUL.2009.21.
[33] Kripac J. A mechanism for persistently naming topological entities in history-based parametric solid models. In: Proceedings of the third ACM symposium on solid modeling and applications. SMA '95, New York, NY, USA: Association for Computing Machinery; 1995, p. 21–30. http://dx.doi.org/10.1145/218013.218024.
[34] Farjana SH, Han S. Mechanisms of persistent identification of topological entities in CAD systems: A review. Alex Eng J 2018;57(4):2837–49. http://dx.doi.org/10.1016/j.aej.2018.01.007.
[35] Cardot A, Marcheix D, Skapin X, Arnould A, Belhaouari H. Persistent naming based on graph transformation rules to reevaluate parametric specification. Comput-Aided Des Appl 2019;16(5):985–1002. http://dx.doi.org/10.14733/cadaps.2019.985-1002.
[36] Liu HTD, Kim VG, Chaudhuri S, Aigerman N, Jacobson A. Neural subdivision. ACM Trans Graph 2020;39(4). http://dx.doi.org/10.1145/3386569.3392418.
[37] Loop C. Smooth subdivision surfaces based on triangles (Master's thesis), The University of Utah; 1987.
[38] López-Fernández JJ, Garmendia A, Guerra E, de Lara J. An example is worth a thousand words: Creating graphical modelling environments by example. Softw Syst Model 2019;18(2):961–93. http://dx.doi.org/10.1007/s10270-017-0632-7.
[39] Dinella E, Dai H, Li Z, Naik M, Song L, Wang K. Hoppity: Learning graph transformations to detect and fix bugs in programs. In: International conference on learning representations. 2020, p. 17, URL https://openreview.net/forum?id=SJeqs6EFvB.
[40] Igarashi T, Hughes JF. A suggestive interface for 3D drawing. In: Proceedings of the 14th annual ACM symposium on user interface software and technology. UIST '01, New York, NY, USA: Association for Computing Machinery; 2001, p. 173–81. http://dx.doi.org/10.1145/502348.502379.
[41] Xu X, Peng W, Cheng CY, Willis KD, Ritchie D. Inferring CAD modeling sequences using zone graphs. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2021, p. 6062–70, URL http://arxiv.org/abs/2104.03900.
[42] Sharma G, Goyal R, Liu D, Kalogerakis E, Maji S. CSGNet: Neural shape parser for constructive solid geometry. In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2018, p. 5515–23.
[43] Du T, Inala JP, Pu Y, Spielberg A, Schulz A, Rus D, Solar-Lezama A, Matusik W. InverseCSG: Automatic conversion of 3D models to CSG trees. ACM Trans Graph 2018;37(6). http://dx.doi.org/10.1145/3272127.3275006.
[44] Kania K, Zięba M, Kajdanowicz T. UCSG-net – unsupervised discovering of constructive solid geometry tree. 2020, arXiv:2006.09102.
[45] Weiler K. Edge-based data structures for solid modeling in curved-surface environments. IEEE Comput Graph Appl 1985;5(1):21–40. http://dx.doi.org/10.1109/MCG.1985.276271.
[46] König B, Nolte D, Padberg J, Rensink A. A tutorial on graph transformation. In: Heckel R, Taentzer G, editors. Graph transformation, specifications, and nets: In memory of hartmut ehrig. Lecture notes in computer science, Cham: Springer International Publishing; 2018, p. 83–104. http://dx.doi.org/10.1007/978-3-319-75396-6_5.
[47] Bommes D, Lévy B, Pietroni N, Puppo E, Silva C, Tarini M, Zorin D. Quad-mesh generation and processing: A survey. Comput Graph Forum 2013;32(6):51–76. http://dx.doi.org/10.1111/cgf.12014.
[48] Belhaouari H, Arnould A, Le Gall P, Bellet T. Jerboa: A graph transformation library for topology-based geometric modeling. In: Giese H, König B, editors. Graph transformation. Lecture notes in computer science, vol. 8571, Cham: Springer International Publishing; 2014, p. 269–84. http://dx.doi.org/10.1007/978-3-319-09108-2_18.
[49] Catmull E, Clark J. Recursively generated B-spline surfaces on arbitrary topological meshes. Comput Aided Des 1978;10(6):350–5.
[50] Warren J. Subdivision methods for geometric design. 1995, p. 111.
[51] Dyn N, Levine D, Gregory JA. A butterfly subdivision scheme for surface interpolation with tension control. ACM Trans Graph 1990;9(2):160–9. http://dx.doi.org/10.1145/78956.78958.
[52] Powell MJD, Sabin MA. Piecewise quadratic approximations on triangles. ACM Trans Math Software 1977;3(4):316–25. http://dx.doi.org/10.1145/355759.355761.
[53] Guzmán J, Lischke A, Neilan M. Exact sequences on Powell–Sabin splits. Calcolo 2020;57(2):13. http://dx.doi.org/10.1007/s10092-020-00361-x.
[54] Doo D, Sabin M. Behaviour of recursive division surfaces near extraordinary points. Comput Aided Des 1978;10(6):356–60. http://dx.doi.org/10.1016/0010-4485(78)90111-2.
[55] Richaume L, Andres E, Largeteau-Skapin G, Zrour R. Unfolding level 1 menger polycubes of arbitrary size with help of outer faces. In: Couprie M, Cousty J, Kenmochi Y, Mustafa N, editors. Discrete geometry for computer imagery. Lecture notes in computer science, Cham: Springer International Publishing; 2019, p. 457–68. http://dx.doi.org/10.1007/978-3-030-14085-4_36.
[56] McKay BD, Piperno A. Practical graph isomorphism, II. J Symbolic Comput 2014;60:94–112. http://dx.doi.org/10.1016/j.jsc.2013.09.003.
[57] Hsieh SM, Hsu CC, Hsu LF. Efficient method to perform isomorphism testing of labeled graphs. In: Gavrilova ML, Gervasi O, Kumar V, Tan CJK, Taniar D, Laganá A, Mun Y, Choo H, editors. Computational science and its applications. Lecture notes in computer science, Berlin, Heidelberg: Springer; 2006, p. 422–31. http://dx.doi.org/10.1007/11751649_46.