A rewrite system to build planar subdivisions

David Cazier and Jean-François Dufourd
Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection
LSIIT, URA CNRS 1871
Université Louis Pasteur, Dép. Informatique
7, rue René Descartes 67084 Strasbourg Cedex
{cazier, dufourd}@dpt-info.u-strasbg.fr

Abstract

Algebraic specifications allied to rewriting are more and more used in design and logical prototyping of programs. We show how these techniques can be applied to a basic problem in computational geometry, namely the construction of planar subdivisions. We build up a simple, complete and convergent system of rules to cope with this problem and suggest a formal method to study and classify the specified algorithms.

1 Introduction

For about twenty years, an interesting approach in program design has been relying on the use of formal methods, especially algebraic specifications [1] [2] allied to rewriting [3]. Firstly, this approach allows designers to focus on conceptual and logical aspects of the problems to solve. Formal objects are described with abstract data type generators, and the behaviour of operations on these objects is modeled by equations. Secondly, specifications can be made operational with a correct orientation of equations, and techniques of logical prototyping are used to point out possible design errors.

The use of these technics has been fruitfull in various areas like langage for computer graphics [4] [5], mecanical proof in geometry [6], and geometrical modeling [7] [8]. We show how they can be applied to a basic problem in computational geometry, namely the boolean operations which are fundamental in geometrical modeling and so deserve a faultless hence formal definition. To be well understood, we limit the present paper to boolean operations on planar subdivisions. Such a subdivision is a partition of the plane into vertices, edges and faces. Boolean operations amount to the refinement of superposed subdivisions, as in the Bentley-Ottmann algorithms [9]. In fact, these problems may be generalized to the self-refinement of embedded combinatorial maps.

We place ourselves in a formal and precise algebraic framework, especially for the definition and manipulation of combinatorial maps. We give a simple but complete definition of self-refinement and show how this transformation can be described with a terminating and confluent rewrite system. Finally, we propose a formal approach to describe and study, in terms of rewriting, concrete and efficient refinement algorithms that leads us to a classification of such algorithms.

2 Selfrefinement of subdivisions

To compute the union, intersection or difference between two planar subdivisions, a convenient way is to construct a new subdivision that contains all the elements of the two, taking into account the intersections and overlappings existing between them. Precisely, superposed edges or vertices are merged, intersecting edges are cut at their intersection points and edges that overlap some vertices are cut at these incidence points. The result of the boolean operations can then be obtained from the corefinement of the subdivisions by selecting the required parts.

We generalize this idea through the notion of the self-refinement of a set of vertices, edges and faces. It consists in transforming it into another one that represents a subdivision of the plane. In figure 1, two subdivisions (a) are superposed to form such a set (b), which is self-refined (c).

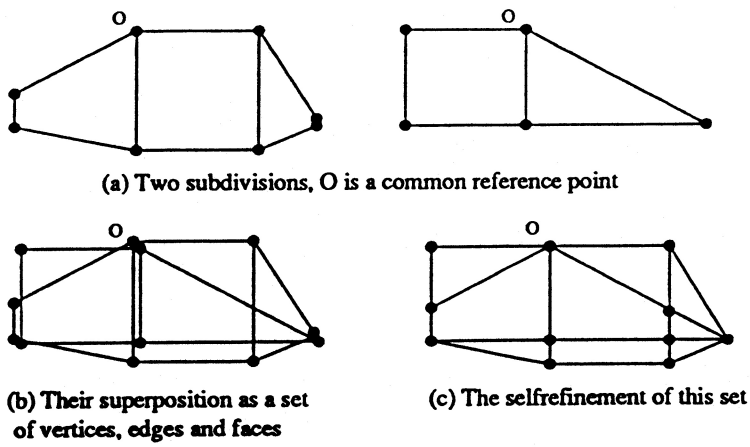


Figure 1: *Example of self-refinement*

It is usual now, for a geometrical object, to distinguish its topology from its embedding. The topology concerns the parts of the object, that is to say its vertices, edges and faces, as well as their adjacency relationships. The embedding concerns the position and shape of these parts. These two aspects coexist in the notion of combinatorial embedded maps which supplies an easy, precise and concise description of subdivisions [10]. Interest of topology was shown in [11] that presents a good alternative to describe subdivisions and topological operators.

3 Combinatorial maps and self-refinement

Let us recall here some basic notions on maps. A *map* is a triplet (B, α_0, α_1) where B is a finite set of *darts*, α_0 is an involution on B, that is to say a permutation such that $\alpha_0(\alpha_0(x)) = x$, for all x, and α_1 is a permutation on B.

Two darts are said to be *linked* with respect to α_0 (resp. α_1), or 0-linked (resp. 1-linked), if they belong to the same orbit with respect to α_0 (resp. α_1). Darts are generally interpreted as half-edges. So, two 0-linked darts form a *topological edge*, and an orbit with respect to α_1 defines a *topological vertex* of the map. See figure 2 where half-segments are associated with darts.

Example 3.1 Figure 2 presents drawing conventions and a map with $B = \{1,\ldots,7,-1,\ldots,-7\}$ and in cyclic notation α_0 = (-1,1) (-2,2) (-3,3) (-4,4) (-5,5) (-6,6) (-7,7) and α_1 = (1, 2) (-2, 3) (-3, -4, 7) (4, -6) (-7, 6, 5, -1) (-5). Thus $\alpha_0(1) = -1$, $\alpha_0(-1) = 1$, and the orbit $<\alpha_0>(1) = \{1, -1\}$ defines an edge. Similarly, $\alpha_1(6) = 5$, $\alpha_1(5) = -1$, $\alpha_1(-1) = -7$, $\alpha_1(-7) = 6$, $<\alpha_1>(6) = \{5,-1,-7,6\}$, and the vertex dart 6 belongs to contains the darts -7,6,5 and -1. □

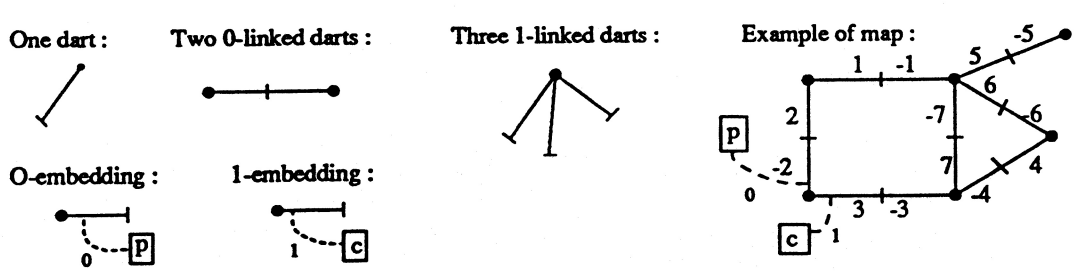


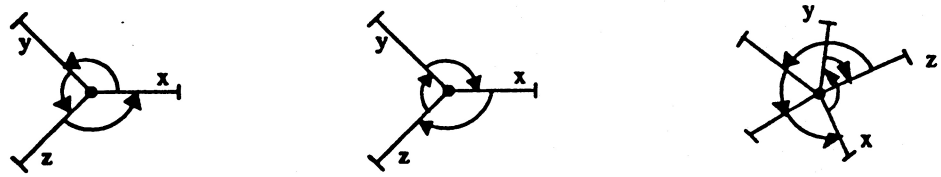
Figure 2: *Conventions and graphical representations*

The *geometry* of the subdivisions is formally described by the embedding of combinatorial maps. This embedding consists in associating each topological part of the map with a geometrical object having the same dimension. We associate *points* with vertices (0-embedding) and *Jordan arcs* with edges (1-embedding).

Example 3.2 In figure 2, dart -2 is 0-embedded on point p, dart 3 is 1-embedded on curve c and then implicitly that is also the case of dart -3. □

Although that notion can be formally defined, we simply say that a map is *planar* if it can be embedded without any self intersection or overlapping. That way, self-refinement of any embedded map transforms it into a planar map which is planarly embedded, or in fact that correctly models a planar subdivision. Then, the result must satisfy the following five conditions : (*i*) all the darts of a vertex have to be embedded on the same point ; (*ii*) two distinct vertices have to be embedded on distinct points ; (*iii*) curves on which edges are embedded must not overlap any 0-embeddings of the vertices ; (*iv*) the 1-embeddings of two distinct edges must not intersect themselves ; (*v*) the vertices have to be *arranged.*

Let us explain the condition (*v*). As edges of a given vertex are examined following 1-links order, the associated curves have to turn counter-clockwise around the vertex (see figure 3). Finally, to those necessary conditions, we add an additional one that leads to a better representation : (*vi*) the map must not contain any null edge, that is to say an edge that is embedded on a null curve.



(a) Arranged, $\alpha_1 = (x, y, z)$ (b) Not arranged, $\alpha_1 = (x, z, y)$ (c) Not arranged, $\alpha_1 = (\ldots, x, y, z, \ldots)$

Figure 3: *Arranged and not arranged vertices : circle arcs represent 1-links*

The self-refinement we have just defined corresponds to a generic description of all refinement problems. Particular algorithms may be described by restricting the kinds of sets that are used. The selrefinement operation could be used on maps coming from superposed maps, as described before, in order to carry boolean operations. If the starting set is a set of segments, self-refinement corresponds to intersections finding algorithms as discussed in [12] [13]. If we limit to two the number of edges belonging to the same vertex, we have got a polygon clipping algorithm.

4 Formal specification of subdivisions

The first phase to formalize the refinement is to give an algebraic specification of combinatorial maps and operations handling them. Maps are defined from four basic functional *generators* : $v, l0, l1, em0$ [7] [8]. Generator v creates the empty map, $l0(M, x, y)$ and $l1(M, x, y)$ link in map M dart x to y with respect to α_0 or α_1. Generator $em0(M, x, p)$ embeds dart x on point p. So, a map is described by first order equivalent terms. In the following, only 0-embeddings are considered. Thus, an edge is implicitely 1-embedded on the line segment between the 0-embeddings of its extremities.

A set of functional *selectors*, *destructors* and *constructors* on maps can then be defined through a first order equational theory. For instance, selectors $eqv(M, x, y)$, $eqev(M, x, y)$ and $eqee(M, x, y)$ respectively test the equality of the two vertices, the 0-embeddings of the two vertices, and the 1-embeddings of the two edges darts x and y belong to. Among the constructors, $cutee(M, x, p)$ cuts at point p the edge x belongs to and $merge(M, x, y)$ merges the two distinct vertices x and y belong to, if they are embedded on the same point. Finally, the destructor $dv(M, x)$ deletes dart x from the vertex it belongs to. Other easily understandable selectors appear in the rules of section 5.

$$R_1 : \frac{M}{dee(M,x)} \text{ if } \begin{cases} M = l0(M',x,y) \\ nulledge(M,x) \end{cases} \qquad R_2 : \frac{M}{dee(M,z)} \text{ if } \begin{cases} M = l0(l0(M',x,y),z,t) \\ eqee(M,x,z) \end{cases}$$

$$R_3 : \frac{M}{cutee(M,x,p)} \text{ if } \begin{cases} M = em0(l0(M',x,y),z,p) \\ \neg nulledge(M,z) \\ s = getlembed(M,x) \\ incident(p,s) \end{cases}$$

$$R_4 : \frac{M}{cutee(cutee(M,x,i),z,i)} \text{ if } \begin{cases} M = l0(l0(M',x,y),z,t) \\ s = getlembed(M,x) \\ s' = getlembed(M,x) \\ secant(s,s') \end{cases}$$

with $i = intersection(s,s')$

$$R_5 : \frac{M}{merge(M,x,y)} \text{ if } \begin{cases} M = em0(em0(M',x,p),y,q) \\ eqp(p,q) \\ \neg eqv(M,x,y) \\ arrangeable(M,x,y) \end{cases} \qquad R_6 : \frac{M}{dv(M,x)} \text{ if } \begin{cases} M = em0(M',x,p) \\ \neg arranged\text{-}dart(M,x) \end{cases}$$

Table 1: Rewrite system for embedded map self-refinement

5 Rewrite system for the self-refinement of maps

We define the self-refinement of maps throughout a set of elementary and independent operations that are nicely described as rules of a conditional modulo rewrite system [14]. In the rules of table 1, numerators represent the starting map and denominators represent it after one rewrite step. Rules can be applied only when conditions described after the *if* are satisfied. Rules are graphically depicted in figure 4.

Rule R1 deletes a null edge, represented as a loop in figure 4, when it occcurs. When two edges are superposed, rule R2 deletes the second one. Rule R3 does the incidence cutting, that is to say it cuts in two parts an edge incident to a vertex. Rule R4 realizes the intersection cutting, that is to say it cuts two edges at their intersection point, if any. The two last rules handle vertices. Rule R5 merges two distinct vertices embedded on equal points, if possible. Finally, rule R6 deletes a not aranged dart from the vertex it belongs to.

The first expected property for a rewrite system, as for any computation, is termination. A rewrite system is said to be *terminating* when there do not exist infinite sequences of rule applications, whatever starting data were. To prove the termination of the rewrite system, we make use of a *semantic ordering* [15] based on a measure of maps. The measure of each numerator is proven to be strictly greater than the measure of the corresponding denominator. To do this, equations of the specification are used, and each representative term of maps is considered. Hence the rewrite system we have defined is terminating.

The second property we could expect for our rewrite system is *confluence*. A rewrite system is confluent if the result of any sequence of rule applications does not depend upon the order the rules are applied. For a terminating system, confluence is equivalent to *local confluence*. Here, technical difficulties are due to the fact that rewriting is made modulo the equational theory E that is defined by the specifications of the maps[3].

To prove the rewrite system is confluent, we show that if there exist two ways to rewrite a map M_0 in map M_1 or M_1', then there exist two sequences of rewrite steps such that M_1 rewrites in M_n, M_1' rewrites in M_m' and the maps M_n and M_m' are equal in some sense. The last equality is the

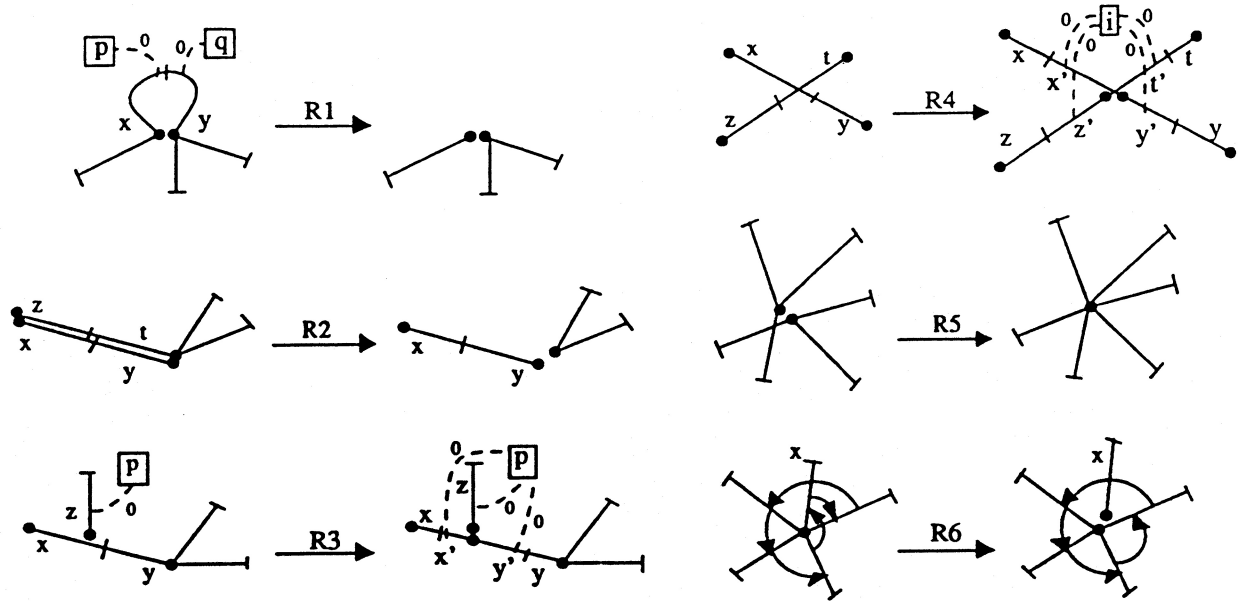


Figure 4: *Graphical illustration of the rewrite rules*

equality modulo the equations of E and modulo dart renaming, because darts numbers depend on the order they are created.

Our terminating and confluent rewrite system is is then *convergent* [3]. This implies that, for each map, there exists one unique normal form modulo map equalities. So, the rewrite system can be seen as a function of map normalization which projects any map into its self-refinement.

6 Concrete algorithms and complexity

A naïve use of the rewrite system described above is to test for each dart and each couple of darts if a rule can be executed. Such an abstract algorithm is not deterministic, because darts are randomly chosen. To describe a concrete, that is to say real and efficient, algorithm, we have to hold a strategy to choose darts. We achieve this goal adding to the rewrite system a control structure that yields the dart or couple of darts that is going to be examined. A rewrite rule then describes the transformations of the map *and* those of the control structure.

For instance, rules R1 and R3 become :

$$R_1 : \frac{S,M}{d(d(S,x),y),dee(M,x)} \text{ if } \begin{cases} x=f(S) \\ nulledge(M,x) \\ y=\alpha_0(M,x) \end{cases} \qquad R_3 : \frac{S,M}{i(i(S,x'),y'),cutee(M,x,p)} \text{ if } \begin{cases} (x,s)=f(S) \\ \neg nulledge(M,x) \\ p=gem0(M,x) \\ s=getlembed(M,x) \\ incident(p,s) \end{cases}$$

where S is a classical data structure such as a list or a tree, f is a function that searches in S the darts that have to be examined, $i(S,x)$ and $d(S,x)$ are functions that insert and delete a dart in S. Darts x' and y' are the new darts created by the edge cutting. Some other rules are added to the rewrite system. They describe the changes on the control structure when refinement rules cannot be used.

We have proposed the generic mechanism to describe a concrete self-refinement algorithm. Different control structures lead to different algorithms. If lists are used, the system corresponds to the algorithm that steps through the darts with two imbricated loops. If more complex structures such as trees, heaps or dictionaries are used, we can describe, through of variant of our rewrite system, plan sweep algorithm like those proposed in [12] [13].

A classification of refinement algorithms can thus be done. It is based upon the kind of structures and the kind of research functions that are used. The interest of this kind of classification is the clear separation between data structures used to handle maps and data structures used to improve control and thus complexity of the algorithms.

7 Conclusions

We have defined topological and geometrical operations for the construction and the handling of planar subdivisions. To achieve this, we base our approach on the combinatorial map mathematical model. The use of algebraic specifications makes definitions more precise and makes us capable to clearly define integrity constraints on objects and functions. The joint use of rewriting techniques leads us to express a complex problem as a set of elementary and independent transformations. So we describe completely in a formal way map self-refinement. Moreover, the use of techniques proper to rewriting, allows us to prove the termination and confluence of this normalization.

Operations on maps and rewrite rules have been implemented in Prolog. Using this logical prototyping, we were able to quickly verify, in a practical way, the validity of our specifications. So, the specification formalism allied to the rewriting expressiveness conduct to a safe and rigorous design of algorithms, even in computational geometry. Finally, those techniques can also be used to study algorithms complexity. In our case, the prototype we made in Prolog allowed us a practical study of the different strategies and control structures for the application of rules.

References

[1] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification 1. equations and initial semantics*, volume 6 of *EATCS Monograph on Theoretical Computer Science*. Springer, 1985.

[2] M. Wirsing. Algebraic specifications. In *Formal models and semantics*, Handbook of Theoretical Computer Science, chapter 13, pages 675–788. Elsevier, 1990.

[3] N. Dershowitz and J.P. Jouannaud. Rewrite systems. In *Formal models and semantics*, Handbook of Theoretical Computer Science, chapter 6, pages 243–320. Elsevier, 1990.

[4] W.R. Mallgren. *Formal specification of interactive graphic programming languages*. ACM Dist. Dissertation. MIT Press, USA, 1982.

[5] D.A. Duce, E.V.C. Fielmding, and L.S. Marshall. Formal specification of a small example based on GKS. *ACM Trans. on Graphics*, 7(3):180–197, 1988.

[6] B. Brüdelin. Using geometric rewrite rules for solving geometric problems symbolically. *Theoretical Computer Science*, 116:291–303, 1993.

[7] J.F. Dufourd. Algebraic map-based topological kernel for polyhedron modellers : algebraic specification and logic prototyping. In *Proc. Eurographics*, pages 649–662, 1989.

[8] Y. Bertrand and J.F. Dufourd. Algebraic specification of a 3D-modeler based on hypermaps. *CVGIP : Graphical models and image processing*, 56(1):29–60, 1994.

[9] J.L. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, 28:643–647, 1979.

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

[11] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoï diagrams. *ACM Trans. on Graphics*, 4(2):74–123, April 1985.

[12] J. Nievergelt and F.P. Preparata. Plane sweep algorithms for intersecting geometric figures. *Com. of ACM*, 25(10):739–747, 1982.

[13] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segment in the plane. *Journal of ACM*, 39(1):1–54, 1992.

[14] N. Dershowitz and M. Okada. A rational for conditional equational programming. *Theoretical Computer Science*, 75:111–138, 1990.

[15] E. Bevers and J. Lewi. Prooving terminaison of (conditional) rewrite systems. A semantic approach. *Acta Informatica*, 30:537–568, 1993.