In a way, the ideas of Lienhardt about boundary-representation models are extended, interpreted and rewritten. He has shown that the topology of graphical objects can easily be described by mathematical concepts of map and generalized map. The map specification problems are emphasized, to take into account topology and, to some extent, geometry. Indications are also given about implementation. A technique of algebraic specification is used to define a universe of graphical objects and operations based on the maps and their extensions. First, a unifying concept of hypermap is specified, for which different levels of operations are given. In the next step, this notion is specialized to retrieve the maps and the generalized maps with their own operations. The fundamental properties of the model of boundary representation are defined, and then obtained from the formal specification. It is shown that the notion of hypermap helps to precisely define the morphology of the objects, i.e. geometrical embedding and photometry, which can be attached to elements of hypermaps. The case of surfaces is specially considered. The question of implementation is also considered. From an algebraic specification, numerous concrete representations can be developed. Two different representations are demonstrated: a rapid prototyping in OBJ3 and an efficient pointer realization in C.

Lienhardt\textsuperscript{1} has shown that the mathematical concepts of n-dimensional map, or n-map, and generalized map, or n-g-map, allow the unification, abstraction and generalization of most of the boundary-representation models proposed in the literature. When \( n = 2 \), for instance, they allow the description of all kinds of surface subdivision topology, and of their components and characteristics, with maximum language precision and economy.

The key issue of the present paper is the formal specification of topological subdivisions, by means of maps, g-maps and operations, for which most definitions and properties are Lienhardt's. Another important, but secondary, preoccupation is morphology: these propositions allow the inclusion in the same framework of geometric and photometric attributes of graphical objects.

In the area of computer graphics, sophisticated mathematical models are built to describe complex objects and operations to be implemented. In fact, there is an actual gap between the mathematical abstractions and the concrete objects of the programming languages and, from high-level rigorous concepts, unsafe programs are sometimes written. This situation is related to the software development techniques. Modern methods advocate a good formal specification to fill up this gap\textsuperscript{3-5}. They have been initiated in computer graphics by Mallgren\textsuperscript{6,7} and Duce and Fielding\textsuperscript{8}, they are now used for all kinds of problems\textsuperscript{9,10} and explored for boundary representation\textsuperscript{11}, especially for those based on maps and generalizations\textsuperscript{12,13}.

Thus, some known descriptions of boundary representation models have been made at the implementation level, dealing, for example, with lists and pointers\textsuperscript{14}. Others have described basic operators with schemata, texts in natural language or programs\textsuperscript{15,16}. With such techniques, it is difficult to precisely and completely define the properties of the models, particularly integrity constraints, that must be maintained during the manipulation of the graphical objects. On the other hand, recent models\textsuperscript{17-21} use classical algebra, where integrity constraints can be readily expressed, but not so easily related data structural operations, because of the lack of a well adapted language. They are also far from being ready for programming.

The formal specification techniques are between these two extreme limits: they offer a precise formalism to describe objects, constraints and operations, but they are rather close to programming languages\textsuperscript{22}. Although the emphasis here is on maps and their extensions, formal specification techniques work for any algebraically valid data structure, and all the structures quoted by Lienhardt\textsuperscript{1}, i.e. winged-edge, quad-edge, facet-edge, cell-tuple, etc., could be reformulated in this way.

As a formal specification technique, we use only an algebraic functional specification approach\textsuperscript{23-27}. Thanks to an equational axiomatization, it gives us a very neat and elegant description of the sorts (symbolic notations

---

Jean-François Dufourd

Departement d'Informatique, Université Louis-Pasteur, 7, Rue René Descartes, F-67084 Strasbourg Cedex, France

Paper received: 4 May 1990.
for types or classes) of objects, functional operations, and properties. In order to simplify the specification and to reduce the programming effort, maps and g-maps are unified by the notion of hypermap, or h-map, of which they are subsorts.

Our specification involves sorts and operations dealing with geometry, i.e., embedding in a Euclidean space, and photometry of objects, as well as topology. All these elements are considered at n dimensions, but topology and morphology are treated in quite a different way. The case of surfaces, of which concrete examples have been developed, will be stressed.

It is shown how sorts and operations can be incrementally built, in a progressive and modular way, in order to obtain descriptions which can be supported by a computer-aided software engineering (CASE) system like LARCH\textsuperscript{19} or OBJ\textsuperscript{26}. Thus, some ideas of the formal expression of our specifications have been taken from the SRI language OBJ\textsuperscript{26}. However, to avoid long explanations, the paper is rather far from the constraints of such a specification system.

Finally, from this point of view, implementation is a subsequent phase that leans on specification. It is well known that several implementations which confirm the specification can be proposed. Two very different implementations of hypermaps will shortly be shown. In order to demonstrate the efficiency of the algebraic specification method, one of them is rapid prototyping, realized through a straightforward translation into the language OBJ\textsuperscript{26}. The other one corresponds to an efficient pointer implementation presented with C.

They are two facets of a project of a 3D modeller whose development is in progress at Strasbourg University.

Definitions of the n-hypermaps, an algebraic specification of topological operations and important properties are given. The notions of n-maps and n-g-maps are put back into this framework, and the 2D case is insisted upon. Morphological concepts are introduced, and the 2D case with morphological operations is completed. Two implementations of hypermaps are proposed with OBJ\textsuperscript{3} and C, respectively. Possible extensions of this approach are given.

**SPECIFICATION OF HYPERMAPS**

**Basic mathematical definitions**

For reasons of design and development, map\textsuperscript{n-w} and generalized map\textsuperscript{w} are unified by the notion of an n-dimensional hypermap (see Cori\textsuperscript{31} for n = 1).

**Definition 1:** Let \( n \) be a non-negative integer. An n-dimensional hypermap, or n-h-map, is an \( n + 2 \)-tuple \( H = (D, \alpha_0, \alpha_1, \ldots, \alpha_n) \), where \( D \) is a finite set of darts, and \( \alpha_0, \alpha_1, \ldots, \alpha_n \) are permutations on \( D \).

In an n-h-map, a dart \( x \) is k-free if it is a fixed point of \( \alpha_k \) (i.e., such that \( \alpha_k(x) = x \)), otherwise it is k-tied. A dart which is k-free for all \( k \leq n \) is free. If \( \alpha_0, \alpha_1, \ldots, \alpha_n \) are without fixed points, the n-h-map is closed, otherwise it is opened. Furthermore, every n-h-map with \( n \geq 0 \) is called a hypermap, or h-map (see Figure 1).

**Definition 2:** In an n-h-map \( H \), any permutation \( \tau \) of \( D \) classifies the darts in cyclic disjointed sequences, called orbits of \( H \) with respect to (w.r.t.) \( \tau \). The orbit containing dart \( x \) is the sequence:

\( \langle \tau \rangle(x) = (x_0, x_1, x_2, \ldots, x_{i-1}) \)

where \( i \) is the smallest strictly positive integer such that \( \tau^i(x) = x \).

**Example:** In Figure 1, \( \langle \alpha_0 \rangle(1) = (1, -1), \langle \alpha_0 \rangle(4) = (4), \langle \alpha_0 \rangle(3) = (3, 5, 6), \langle \alpha_0 \rangle(2) = (2) \).

We call \( \langle \alpha \rangle(x) \) the k-orbit of \( x \).

**Algebraic specification of a basic kernel**

Hypermaps can be specified algebraically using a specification system like LARCH\textsuperscript{24} or OBJ\textsuperscript{26}. However, in order to simplify the presentation, we use a straightforward mathematical expression, without the syntactical and semantical constraints of a real specification language.

In Table 1, a basic kernel H-MAP 1 of 'atomic' operations for the sort h-map is defined in a purely functional way. We suppose that the integer, Boolean and dart sorts are known through a basic specification ROOT, with the classical operations, including equality for darts, denoted =.

Three basic generators of h-maps, \( v, i \) and \( l \), allow the creation of h-maps:

- \( v(n) \) is the empty \( n \)-h-map. It is without darts and therefore without links;
- \( i(h, x) \) is the insertion of dart \( x \) in h-map \( h \), giving a new h-map. In \( i(h, x) \), \( x \) is a fixed point of \( x \) for any dimension;
- \( l(h, k, x, y) \) is the linking (or tying) of dart \( x \) after dart \( y \) in its k-orbit of h-map \( h \). Darts \( x \) and \( y \) must exist in h-map \( h \), be different and \( x \) must be k-free.

**Example:** H-map \( h \) in Figure 1 is built from:

\( h1 = i(i(i(i(i(i(v(2), 1), -1), 0, 1, -1), 2), -2), 0, 2, -2), 1, 2, 1, -1, -2) \) by

\( h = i(i(i(i(i(i(h1, 3), 4), 5), 6), 1, 3, 5), 1, 3, 5), 1, -1, -2, 1, 6, 5) \).
Table 1. Algebraic specification of the basic kernel H-MAP 1

\textbf{spec} \ H-MAP 1 \textbf{using} \ ROOT

\textbf{sort} \ h-map

\textbf{operations}

\begin{align*}
\text{\texttt{v}} & : \text{integer} \rightarrow h-map \quad /* \text{vacuous, or empty, n-h-map} */ \\
\text{\texttt{i}} & : h-map \times \text{dart} \rightarrow h-map \quad /* \text{insertion of a new dart} */ \\
\text{\texttt{l}} & : h-map \times \text{integer} \times \text{dart} \times \text{dart} \rightarrow h-map \quad /* \text{linking of a dart in a k-orbit} */ \\
\text{\texttt{d}} & : h-map \times \text{dart} \rightarrow h-map \quad /* \text{deletion of a dart} */ \\
\text{\texttt{r}} & : h-map \times \text{integer} \times \text{dart} \rightarrow h-map \quad /* \text{rupture of a linking} */
\end{align*}

\textbf{preconditions}

\begin{align*}
\text{\texttt{prec}} (v(n)) & = 0 < n \\
\text{\texttt{prec}} (i(h, x)) & = \neg e(h, x) \\
\text{\texttt{prec}} (l(h, k, x, y)) & = 0 \leq k \leq \delta(h) \wedge e(h, x) \wedge e(h, y) \wedge x \neq y \\
& \wedge x(h, k, x) = x \\
\text{\texttt{prec}} (a(h, x)) & = \text{\texttt{prec}} (a^{-1}(h, k, x)) = 0 \leq k \leq \delta(h) \wedge e(h, x) \\
\text{\texttt{prec}} (\phi(h, x)) & = e(h, x) \\
\text{\texttt{prec}} (d(h, x)) & = 0 \leq k \leq \delta(h) \wedge e(h, x) \wedge x(h, k, x) \neq x
\end{align*}

\textbf{axioms}

\begin{align*}
1. & \quad \delta(v(n)) = n \\
2. & \quad \delta(i(h, x)) = \delta(h) \\
3. & \quad \delta(l(h, k, x, y)) = \delta(h) \\
4. & \quad e(v(n), x) = \text{false} \\
5. & \quad e(i(h, x), z) = (z = x) \vee e(h, z) \\
6. & \quad e(l(h, k, x, y), z) = e(h, z) \\
7. & \quad \phi(i(h, x), z) = (z = x) \vee \phi(h, z) \\
8. & \quad \phi(l(h, k, x, y), z) = z \neq x \wedge z \neq y \wedge \phi(h, z) \\
9. & \quad (a(h, x), k, z) = (z = x \text{ then } x \text{ else } a(h, k, z)) \\
10. & \quad (l(h, k, x, y), j, z) = (j = k \text{ then if } z = y \text{ then } x \\
& \quad \text{ else if } z = x \text{ then } a(h, k, y) \text{ else } a(h, k, z) \\
& \quad \text{ else if } z = x \text{ then } a(h, k, y) \text{ else } a^{-1}(h, k, z) \\
11. & \quad \delta^{-1}(i(h, x), k, z) = (z = x \text{ then } h \text{ else } i(d(h, z), x)) \\
12. & \quad \delta^{-1}(l(h, k, x, y), j, z) = (j = k \text{ then if } z = y \text{ then } h \\
& \quad \text{ else if } z = y \text{ then if } a(h, k, y) = y \text{ then } h \\
& \quad \text{ else if } z = y \text{ then if } a(h, k, y) \neq y \text{ then } h \\
& \quad \text{ else if } z = y \text{ then if } a(h, k, y) \neq y \text{ then } h) \\
13. & \quad d(i(h, x), z) = (z = x \text{ then } h \text{ else } i(d(h, z), x)) \\
14. & \quad d(l(h, k, x, y), z) = j(d(h, z), k, x, y) \\
15. & \quad r(i(h, x), k, z) = i(r(h, k, z), x) \\
16. & \quad r(l(h, k, x, y), j, z) = (j = k \text{ then if } z = x \text{ then } h \\
& \quad \text{ else if } z = x \text{ then if } a(h, k, y) = y \text{ then } h \\
& \quad \text{ else if } z = y \text{ then if } a(h, k, y) \neq y \text{ then } h \\
& \quad \text{ else if } z = y \text{ then if } a(h, k, y) \neq y \text{ then } h) \\
\end{align*}

Five \textbf{basic selectors} on \texttt{h-maps}, \(\delta, e, \phi, a\) \text{ and } \(a^{-1}\), define the \texttt{h-map} states:

• \(\delta(h)\) gives the dimension of \texttt{h-map} \(h\);
• \(e(h, x)\) tests existence of dart \(x\) in \texttt{h-map} \(h\);
• \(\phi(h, x)\) tests freeness of dart \(x\) in \texttt{h-map} \(h\);
• \(a(h, k, x)\) gives the successor of dart \(x\) in its \(k\)-orbit of \(h\); it corresponds to \(a(x)\);
• \(a^{-1}(h, k, x)\) gives the predecessor of \(x\) in its \(k\)-orbit of \(h\); it corresponds to \(a^{-1}(x)\).
Dart $x$ must exist in $h$ for application of the last three operations.

**Example**: in h-map $h$ of Figure 1, $\delta(h) = 2$, $\epsilon(h, 1) = \text{true}$ and $\iota(h, 10) = \text{false}$. $\phi(h, 4) = \text{true}$. $\chi(h, 2, 3) = 1$, $\chi^{-1}(h, 1, 3) = 6$

Two generators $d$ and $r$, also called destructors, are the inverses of $i$ and $l$. They allow h-map destruction:

1. $d(h, x)$ is the deletion of dart $x$ from h-map $h$, giving a new h-map. For application of this operation, dart $x$ must exist in $h$ and be free;
2. $r(h, k, x)$ is the rupture of the $k$-link of dart $x$ in h-map $h$. It removes $x$ from its $k$-orbit in $h$. After that, $x$ is $k$-free in the new h-map. For application of this operation, $x$ must exist in $h$ and be $k$-tied.

**Example**: the complete deletion of the darts 1 and $-1$, with all their bindings, from h-map $h$ of Figure 1, is defined by the following expression:

$$d(d(r(r(r(r(r(h, 2, 1), -1), 0, 1), 1), 1), 1), 1), 1), -1)$$

**Preconditions** allow an operations domain definition. Equational conditional axioms specify the mutual behaviour of these operations, i.e. the effect of the basic generators on the basic selectors – from 1 to 12 – and on the destructors – from 13 to 16.

This systematic way for writing the axioms does not assure their correctness (see below). However, since it respects the rules of algebraic presentation, it always makes certain of their minimality. That is also true for all following specifications.

Note that instead of building efficient (at some point of view) data structures for the h-maps, the H-MAP-1 specification only keeps a raw history of their construction. In this sense, such a specification is neutral with regard to premature design decisions, which are devoted to the implementation level.

**Semantics and properties**

Classically, the above specification of hypermap is a theory in equational first order logic. Let $T_{h}$ be the set of the closed terms, i.e. without variables, whose sort is h-map.

**Example**: term $i(i(i(i(v(2), 1), -1), 0, 1, -1), 2), 2, 0, 2, -2), 1, 1, 2), 1, -1, -2)$ is in $T_{h}$.

In order to give some properties, the semantics of equality $=$ of hypermaps must be precise. It can be considered from two points of view, with two equivalence relations in $T_{h}$, denoted $=$, and $\equiv$, respectively.

**Definition 3**: if $h1$ and $h2$ are terms in $T_{h}$,

1. $h1 = h2$ iff $h1$ and $h2$ can be proven equal by using the axioms;
2. $h1 \equiv h2$ iff $h1$ and $h2$ cannot be distinguished through any selector, i.e. through any observation.

Relations $=,$ and $\equiv$, are called equalities in initial and terminal algebra, respectively. It is clear that $h1 = h2$ implies $h1 \equiv h2$, but the converse is not true.

**Example**: h-maps $h1 = i(i(i(i(v(2), 1), 1), 2), 1), 1, 1, 2), 0, 2, 1$ and $h2 = i(i(i(i(v(2), 1), 1), 2), 1), 1, 1, 2), 0, 3, 2)$ are such that $h1 \neq h2$, because their equality cannot be proven by using axiomatics. However, $h1 = h2$ because, in the same conditions, applications of selectors $d$, $i$, $r$, $x$ or $x^{-1}$ on $h1$ and $h2$ always lead to the same results.

In initial algebra, where $=$ has the semantics of $=_{\equiv}$, previous specification allows us to obtain the following properties:

1. For any dimension $k$, $x$ is a permutation with inverse $x^{-1}$.
2. $\phi(h, x) = \text{true}$ iff $x(h, k, x) = x$, for each dimension $k$;
3. Every term of sort $n$-h-map can be rewritten using axiomatics without the operations $d$ and $r$, i.e. with only $v$ (once), $i$ and $l$. Such a term is called a normal form of an h-map in initial algebra.

**Example**: $h3 = r(h2, 0, 2)$ can be rewritten $i(i(i(v(2), 2), 1), 1, 2, 1, 3)$, which is a normal form in initial algebra.

In terminal algebra, where $=$ has the semantics of $=_{\equiv}$, other properties add to the previous ones:

1. When the terms in play make sense (i.e. the preconditions of the used operations are satisfied), applications of operations $i$ or $i$ and $l$ can be inverted.
2. Every normal form of $n$-h-map in initial algebra can be rewritten in a term which is built from $v$ by composition of $i$, followed by compositions of $l$. Such a term is called normal form of an h-map in terminal algebra.

**Example**: the initial normal form $i(i(i(i(v(2), 2), 1), 2), 1, 2, 1, 3)$ can be rewritten $i(i(i(v(2), 2), 1), 1, 2, 1, 3)$, which is a normal form in terminal algebra.

Some of these properties can be considered as integrity constraints for the objects. If the preconditions are satisfied for the use of the operations, these constraints are automatically respected. In the following, the semantics are those of initial algebra. Note that it is possible to confuse $=_{\equiv}$ and $=_{\equiv}$, by adding appropriate axioms which give the properties of $=_{\equiv}$, to $=_{\equiv}$, for example:

17. $i(i(h, x), y) = i(i(h, y), x)$
18. $i(i(h, k, x), y) = i(i(h, x), z), k, x, y)$

Finally, with the definitions of consistency and sufficient completeness of Guttag and Horning, the following property makes sure that the specification can be safely used.

Roughly speaking, a specification $S_{1}$ is consistent, relative to a specification $S_{0}$, if it is impossible to prove from $S_{0}$ a new theorem of $S_{1}$. Here, the basic specification $S_{0}$ is that of Booleans and integers, and it should not be possible to prove from the h-map specification $S_{1}$ a result like $\text{true} = \text{false}$, or $0 = 1$.

A specification $S_{1}$ is sufficiently complete, relative to a specification $S_{0}$, if every term of $S_{1}$ without a variable...
of form $f(a_1, \ldots, a_n)$, where $f: s_1 \times \ldots \times s_n \rightarrow s$ is a function symbol whose target-sort $s$ is defined in $S_0$. Here, for instance, $\epsilon(l(l(i(i(v, 1), 2), 2, 1, 2), 0, 2, 1), 3)$ is rewritten as $false$.

Property: the specification of $h$-maps is consistent and sufficiently complete.

In what follows, although this is not explicitly mentioned, all specifications have this property.

The proofs of all the above results can be easily obtained by induction in the language of terms of the theory. Therefore, they are omitted.

Secondary selectors

Many secondary selectors can be defined from the previous kernel. In Table 2, for instance:

- $\text{dim}(h)$ is the effective dimension of $h$, i.e. the greater order of its linkings;
- $\text{nd}(h)$ is the number of darts inserted in $h$;
- $\text{nI}(h, k)$ is the number of $k$-linkings in $h$;
- $\text{eql}(h, k, x, y)$ is the test of $k$-equivalence, i.e. the belonging to the same $k$-orbit of $h$ for $x$ and $y$.

The use of the last operation is constrained by the fact that $x$ and $y$ must exist in $h$.

Auxiliary functions such as $\text{max}$ and $\text{expath}$ have the following meaning:

- $\text{max}(m, n)$ gives the greater between integers $m$ and $n$; it is assumed to be defined in ROOT;

- $\text{expath}(h, k, x, z, y)$ tests the existence of a path, which does not pass through dart $x$, from dart $z$ through dart $y$ in orbit $\langle \alpha_k \rangle(x)$ of $h$-map $h$.

Here, a purely functional specification is given, but not an implementation. Operations such as $\text{dim}$ and $\text{eql}$ can be implemented efficiently by selecting an appropriate data structure for a hypermap (see below), which respects the specification.

Union, connected components and elements

New operations which are especially relevant in the field of CAD are defined to deal with components of complex graphical objects.

Definition 4: let $H^1 = (D^1, \alpha_1', \alpha_1', \ldots, \alpha_{\chi_1}')$ and $H^2 = (D^2, \alpha_2', \alpha_2', \ldots, \alpha_{\chi_2}')$ be two $h$-maps such that $D^1 \cap D^2 = \emptyset$. The union of $H^1$ and $H^2$ is the $h$-map $H = (D, \alpha_1, \alpha_2', \ldots, \alpha_{\chi_2})$ such that $D = D^1 \cup D^2$, $n = \max(n_1, n_2)$, and for $k = 1, \ldots, n$, $\alpha_k(x) = \alpha_k'(x)$ if $x \in D^1$ and $k \leq n_1$, $\alpha_k(x)$ if $x \in D^2$ and $k \leq n_2$, $x$ otherwise.

Other mathematical definitions resulting from the generalized maps\textsuperscript{1} can immediately be translated into the framework of the hypermaps.

Definition 5: an $n$-$h$-map $H$ can be considered as an oriented multigraph $M(H)$ on $D$, with $n + 1$ binary relations, which corresponds to functions $s_0, \alpha_1, \ldots, \alpha_n$. The $n$-$h$-map $H$ is said to be connected if multigraph $M(H)$ is connected, otherwise it has the same connected components.

Table 2. Algebraic specification $H$-MAP 2

<table>
<thead>
<tr>
<th>spec</th>
<th>H-MAP 2 using H-MAP 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations</td>
<td>dim: $h$-map $\rightarrow$ integer</td>
</tr>
<tr>
<td></td>
<td>nd: $h$-map $\rightarrow$ integer</td>
</tr>
<tr>
<td></td>
<td>nl: $h$-map $\times$ integer $\rightarrow$ integer</td>
</tr>
<tr>
<td></td>
<td>eql: $h$-map $\times$ integer $\times$ dart $\times$ dart $\rightarrow$ Boolean</td>
</tr>
<tr>
<td>preconditions</td>
<td>prec (eql($h$, $k$, $x$, $y$)) $\equiv$ $0 \leq k \leq \delta(h) \land \varepsilon(h, x) \land \varepsilon(h, y)$</td>
</tr>
<tr>
<td>axioms</td>
<td>dim($\nu(n)$) $= -1$</td>
</tr>
<tr>
<td></td>
<td>dim($i(h, x)$) $= dim(h)$</td>
</tr>
<tr>
<td></td>
<td>dim($l(h, k, x, y)$) $= \max(k, dim(h))$</td>
</tr>
<tr>
<td></td>
<td>nd($\nu(n)$) $= 0$</td>
</tr>
<tr>
<td></td>
<td>nd($i(h, x)$) $= nd(h)+1$</td>
</tr>
<tr>
<td></td>
<td>nd($l(h, k, x, y)$) $= nd(h)$</td>
</tr>
<tr>
<td></td>
<td>nl($\nu(n)$, $k$) $= 0$</td>
</tr>
<tr>
<td></td>
<td>nl($i(h, x)$, $k$) $= nl(h, k)$</td>
</tr>
<tr>
<td></td>
<td>nl($l(h, k, x, y)$, $k$) $= if ; j = k ; then ; nl(h, k) + 1 ; else ; nl(h, k)$</td>
</tr>
<tr>
<td></td>
<td>eql($h$, $k$, $x$, $y$) $= (x = y \lor \text{expath}(h, k, x, \alpha(h, k, x), y))$</td>
</tr>
<tr>
<td></td>
<td>where \text{expath}($h$, $k$, $x$, $z$, $y$) $= (z = y \lor z \neq x \land \text{expath}(h, k, x, \alpha(h, k, z), y))$</td>
</tr>
</tbody>
</table>

volume 23 number 2 march 1991
Table 3. Algebraic specification H-MAP 3

**spec H-MAP 3 using H-MAP 2**

**operations**

- union: h-map x h-map → h-map
- eq: h-map x dart x dart → Boolean
- nc: h-map → integer
- cc: h-map x dart → h-map
- pr: h-map x integer → h-map

/* union of disjoint h-maps */
/* equivalence of two darts */
/* number of connected components */
/* connected component of a dart */
/* k-dimensional projection */

**preconditions**

- prec (union(h1, h2)) = δ(h1) = δ(h2) ∧ (δ(h1, x) ⊆ δ(h2, x))
- prec (eq(h, x, y)) = δ(h, x) ∧ δ(h, y)
- prec (cc(h, z)) = δ(h, z)
- prec (pr(h, k)) = 0 ≤ k ≤ δ(h)

**axioms**

- union(v(n1), h2) = h2
- union(i(h, x, h2)) = (union(h, h2), x)
- union(l(h, k, x, y, h2)) = (union(h, h2), k, x, y)

- eq(δ(h, x), t) = (z = t v z ≠ x ∧ t ≠ x ∧ eq(h, z, t))
- eq(δ(h, z, x, t)) = eq(h, z, x) v eq(h, z, t) ∧ eq(h, t, x) v eq(h, t, y))

**Definition 6:** let H = (B, z0, z1, ..., zn) be an n-h-map (with n ≥ 1). For k such as 0 ≤ k ≤ n, the (n-1)-h-map H_k = (B, z0, z1, ..., z_k-1, z_k, z_k+1, ..., z_n) is the h-map of the k-elements of H. The connected components of H_k are the k-elements of H.

With n = 2, for instance, 0-elements are named vertices, 1-elements edges and 2-elements faces.

Table 3 gives a specification H-MAP 3 for five new functions, union, eq, nc, cc and pr:

- union(h1, h2) is the union of disjoint h-maps h1 and h2, with the same dimension;
- eq(h, x, y) is a test of equivalence, i.e. the adherence to the same connected component of h-map h for darts x and y;
- nc(h) is the number of connected components of h-map h;
- cc(h, z) is the connected component of h which contains dart z;
- pr(h, k) is the h-map of the k-dimensional elements of h-map h.

The definition of function cc is a recursive compact form for h-maps of Kruskal’s algorithm for finding connected components in a graph. Here, these components are also represented by h-maps, which must sometimes be united by function union.

Note that pr(h, k) is in fact considered as an n-h-map and not an (n - 1)-h-map. A more complex specification can be written which eliminates this drawback.12

**MAPS AND GENERALIZED MAPS**

**Mathematical definitions**

The definitions of map and g-map1 can be rewritten using the concept of hypermap, with the assumption that the set of darts D may be empty.

**Definition 7:** let n ≥ 1. An n-dimensional map M, or n-map, is an (n - 1)-h-map M = (D, x_0, x_1, ..., x_n), where:

- x_0, x_1, ..., x_n are involutions;
- x_0, x_1, more simply written x_i x_j, are involutions, for 0 ≤ i < j + 2 ≤ n - 1.
Involutions and fixed points in the specification

To maintain an involution \( \alpha \) for dimension \( k \) in h-map \( h \), the use of the linking operation \( l \) at dimension \( k \) must be constrained by a stronger precondition:

\[
\text{prec} \ (l(h, k, x, y)) \equiv \exists \ 0 \leq k \leq \delta(h) \land \varepsilon(h, x) \land \varepsilon(h, y) \\
\land x \neq y \land \alpha(h, k, x) = x \land \alpha(h, k, y) = y
\]

Thus, in the case of maps or g-maps, some operations of H-MAP 3 can be more constrained and also, notably, simplified. In what follows, it is supposed that new operations which override the old ones with the same names are defined in specifications respectively called MAP 1 (for maps) and G-MAP 1 (for g-maps) using H-MAP 3. That is the case for \( v, i, l, \alpha, d, r, \text{union}, cc, pr \). For instance, if \( \alpha \) is an involution for dimension \( k \), the definition of \( \alpha \) for g-maps is more concise and \( \alpha^{-1} \) is not necessary. The new operations allow us to manage maps and g-maps 'in construction', i.e. before they verify all constraints in Definitions 7 and 8.

Definition 8: An n-dimensional generalized map \( G \), or n-g-map, is an n-h-map \( G = (D, \alpha_0, \alpha_1, ..., \alpha_{n-1}) \), where:

- \( \alpha_0, \alpha_1, ..., \alpha_{n-1} \) are involutions without fixed points;
- \( \alpha_k \) is an involution;
- \( \alpha_k \) are involutions, for \( 0 \leq i < i + 2 \leq j \leq n \).

All other definitions of Lienhardt are available under the same conditions.

Table 4. MAP 1 and G-MAP 1 specifications

<table>
<thead>
<tr>
<th>spec MAP 1 using H-MAP 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts mapc &lt; h-map</td>
</tr>
<tr>
<td>map &lt; mapc [ 0 \leq i &lt; i + 2 \leq j \leq \delta(m) - 1 \supset \varepsilon(m, x) \supset (\alpha(m, j, \alpha(m, i, x)) = x)]</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>spec G-MAP 1 using H-MAP 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts g-mapc &lt; h-map</td>
</tr>
</tbody>
</table>
| g-map < g-mapc \[ 0 \leq k \leq \delta(g) - 1 \supset nd(g) = 2^{n}l(g, k) \\
| \land (0 \leq i < i + 2 \leq j \leq \delta(g) \supset \varepsilon(g, x) \supset (\alpha(g, j, \alpha(g, i, x)) = x)]  |
| ...                        |
deal with objects of sort mapc or g-mapc are very similar to the ones which are in H-MAP 3, and therefore are not written.

However, the conditions after iff in Table 4 are very expensive to check a posteriori. It is easier to provide consistency checks during the construction of elements of maps or g-maps. This enforces a hierarchical dimension after dimension construction that conforms to Lienhardt.

To illustrate these questions with surfaces, a number of operations based on the previous ones to build 2-g-maps by operations on simple edges (i.e. couples \((x, z_i(x))\), is presented below.

**Operations for building 2-g-maps**

We deal here with the following ‘realistic’ constraints, which are verified by a 2-g-map in construction, of sort 2-g-mapc:

- (c1) \(z_0\) is an involution without a fixed point;
- (c2) \(z_1\) and \(z_2\) are involutions, possibly with fixed points;
- (c3) \(x, z_i\) is an involution, possibly with fixed points.

A true 2-g-map (Definition 8) is a special case of such a 2-g-mapc, with the additional condition that \(z_0\) is an involution without a fixed point.

In Table 5, 2-G-MAP 1 specifies four operations, ie, de, se and ue, designed to build or destroy 2-g-maps:

- \(ie(g, x_1, y_1, x_2, y_2)\) is the insertion of a new simple edge \((x_1, x_2)\) in 2-g-mapc \(g\), with \(x_1\) [resp. \(x_2\)] 1-linked to \(y_1\) [resp. \(y_2\)] and without a sewing. This unique operation unifies all cases of simple edge insertion, as illustrated by Figure 4;
- \(de(g, x_1)\) is the deletion of simple edge \((x_1, x_2)\) from 2-g-mapc \(g\), i.e. the inverse of the previous operation \(ie\) and \(de\) are generalizations of the well known Euler operators \((\gamma, \beta)\);
- \(se(g, x_1, x_2)\) is the sewing of two simple edges given by two distinct darts \(x_1\) and \(x_2\) in 2-g-mapc \(g\). All cases of edge sewings are represented in Figure 5. The first one is a bending of a simple edge, and the third one is a flattening of two simple edges;
- \(ue(g, x_1)\) is the unsewing of a simple edge \((x_1, x_2)\), i.e. the inverse of se.

Note that the basic operations of the 2-g-maps \(v, i, l\), etc., must be redefined, in the same way as for the

**Table 5. Specification of 2-g-maps** (auxiliary elements \(g_1, x_2, y_1\) and \(y_2\) are defined by ‘macro-definitions’ after where)

<table>
<thead>
<tr>
<th>spec 2-G-MAP 1 using G-MAP 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts 2-g-mapc &lt; g-mapc iff (\delta(g) \leq 2) 2-g-map &lt; 2-g-mapc iff (nd(g) = 2 \cdot nd(1, 1))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i(e(g, x_1))) is the insertion of edge ((x_1, x_2)) in 2-g-mapc (g)</td>
</tr>
<tr>
<td>(d(e(g, x_1))) is the deletion of edge ((x_1, x_2)) in 2-g-mapc (g)</td>
</tr>
<tr>
<td>(s(e(g, x_1, x_2))) is the sewing of two simple edges ((x_1, x_2)) in 2-g-mapc (g)</td>
</tr>
<tr>
<td>(u(e(g, x_1))) is the unsewing of edge ((x_1, x_2)) in 2-g-mapc (g)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\prec i(e(g, x_1, y_1, x_2, y_2))) (\equiv \neg e(g, x_1) \land e(x_2) \land x_1 \neq x_2 \land y_1 \neq y_2) (\land (e(g, y_1) \land e(x_1) = y_1 \lor x_1 = y_1 = x_2)) (\land (e(g, y_2) \land e(x_2) = y_2 \lor x_2 = y_2 = x_2))</td>
</tr>
<tr>
<td>(\prec d(e(g, x_1))) (\equiv e(g, x_1) \land e(x_1))</td>
</tr>
<tr>
<td>(\prec s(e(g, x_1, x_2))) (\equiv e(g, x_1) \land e(x_2) \land x_1 \neq x_2 \land e(g, x_1) = x_1 \land e(g, x_2) = x_2)</td>
</tr>
<tr>
<td>(\prec u(e(g, x_1))) (\equiv e(g, x_1) \land e(x_1) \land x_1 \neq x_2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>axioms</th>
</tr>
</thead>
</table>
| \(i(e(g, x_1, y_1, x_2, y_2)) =\)
| \(\text{if } x_1 = y_2 \text{ then } h(g, 1, 1, x_1)\)
| \(\text{else if } x_1 = y_1 \text{ then } h(g, 1, 2, x_2)\)
| \(\text{else} \) \(\text{if } x_2 = y_2 \text{ then } g\) \(\text{else} \) \(\text{if } x_2 = y_1 \text{ then } h(g, 1, 1, x_1)\)
| \(\text{else} \) \(h(g, 1, 2, x_1, x_2)\) |
| \(\text{where } g = h(r(g, x_1, x_2), 0, x_1, x_2)\) |

| \(d(e(g, x_1))\) = \(d(d(r(g, x_1, x_2), x_1, x_2)\) |
| \(\text{where } g = h(g, 1, 1, x_1)\) |
| \(\text{else if } x_1 = y_1 \text{ then } h(g, 1, 2, x_2)\) |
| \(\text{else if } x_2 = y_2 \text{ then } g\) |
| \(\text{else} \) \(\text{if } x_2 = y_1 \text{ then } h(g, 1, 1, x_1)\) |
| \(\text{else} \) \(h(g, 1, 2, x_1, x_2)\) |
| \(r(g, x_1, x_2) = x(g, 0, x_1) = y(g, 0, x_2)\) |
| \(\text{else} \) \(h(g, 1, 2, x_1, x_2)\) |
| \(\text{where } g = h(r(g, x_1, x_2), 0, x_1, x_2)\) |

| \(s(e(g, x_1, x_2))\) = \(h(g, 1, 2, x_1, x_2)\) |
| \(\text{else if } x_1 = y_1 \text{ then } h(g, 1, 2, x_2)\) |
| \(\text{else if } x_2 = y_2 \text{ then } g\) |
| \(\text{else} \) \(\text{if } x_2 = y_1 \text{ then } h(g, 1, 1, x_1)\) |
| \(\text{else} \) \(h(g, 1, 2, x_1, x_2)\) |
| \(\text{where } g = h(r(g, x_1, x_2), 0, x_1, x_2)\) |

| \(u(e(g, x_1))\) = \(h(g, 1, 2, x_1, x_2)\) |
| \(\text{else if } x_1 = y_2 \text{ then } h(g, 1, 1, x_1)\) |
| \(\text{else if } x_2 = y_1 \text{ then } h(g, 1, 2, x_2)\) |
| \(\text{else} \) \(\text{if } x_2 = y_1 \text{ then } h(g, 1, 2, x_1, x_2)\) |
| \(\text{where } g = h(r(g, x_1, x_2), 0, x_1, x_2)\) |

106 computer-aided design
g-maps with respect to the h-maps. They override in 2-G-MAP 1 the basic operations of G-MAP 1.
All of these operations are defined by compositions of the v, i, l, d and r operations, defined for g-map, but also usable for 2-g-map by inheritance. From the axiomatics, it is easy to prove the following properties by induction:

Properties: the exclusive use of v(2), ie, de, se, et and ue operations, in respect to their preconditions, leads to a 2-g-map which respects the constraints (c1), (c2) and (c3). A true 2-g-map g is such a 2-g-map with the additional constraint: ndl(g) = 2*ndl(g-1).

- Under the same conditions, the operations se and ue are commutative and permutative in the terminal algebra.
- Any 2-g-map g is equal, in the terminal algebra, to a 2-g-map which is built by composition of insertions ie, followed by sewings se, from the empty 2-g-map v(2).

The reader is referred to Dufourd for more details about these specifications. Thus, for 2-g-maps, we can forget the basic dart-layer and consider exclusively the simple edge-layer. Following this method, high-level operations can be incrementally built.

Frontier and boundaries

G-MAP 1 specification may be completed with new operations. The first ones deal with the frontier, the definition of which in Lienhardt can be rewritten:

Definition 10: let n ≥ 1 and G = (D, αo, α1, ..., αn) be an n-g-map. The frontier of G is the (n-1)-g-map δG = (D', α', α', ..., α') where:
- D' is the set of the n-free darts in G;
- for i ≤ n-2, α' is the restriction of αi to D';
- α' is defined, for all x ∈ D', in a unique way by α' = (D' \ {x}) ∩ α\"(x).

Let us define in the G-MAP 2 specification (see Table 6) the operations σ and δ:

- σ(g, x) is the dart which follows dart x in the frontier of n-g-map g;
- δ(g, x) is the dart which follows dart x in the frontier of n-g-map g;
- δ(g) is the frontier of g-map g. Following a standard recursion technique, δ is defined from operations δ0 and δ1, which maintain a whole copy of g as first parameter;
- δ0(g, n, g) is the g-map g with all n-free darts and without linkings;
- δ1(g, n, g) restores in the previous one the good linkings of the frontier.

Note that during the transformations described for δ1, the constructed object is not a g-map but a g-mapc, because it has too many fixed points. It is a true g-map, in the sense of Definition 8, only at the end.

Map of hypervolumes of a g-map

The notion of hypervolume of an n-g-map is useful to characterize the orientability, as pointed out by Lienhardt and shown below.

Definition 11: let n ≥ 1 and an n-g-map G = (D, αo, α1, ..., αn). The n-map (D, αo, α1, ..., αn) is called n-map of hypervolumes of G.

Operation hv is defined in the G-MAP 3 specification (see Table 7):

- hv(g) is the map of hypervolumes of g. This operation is defined from auxiliary operations hv0 and hv1, which keeps a whole copy of g as its first parameter;
- hv0(g) is g without links and with a by 1 reduced dimension;
- hv1(g, n, g) restores the good links of the hypervolumes in the previous map, by means of function ho. ho(m, k, x, y) links the k-orbits of two darts x and y in mapc m.

Note that the constructing object is always a map and the ‘destroyed’ one (third parameter of hv1) a g-mapc.
Table 6. Specification of the frontier of a g-map

```plaintext
spec G-MAP 2 using G-MAP 1

operations
σ: g-map × dart → dart /* following dart in the frontier */
\hat{\sigma}: g-map → g-map /* g-map of frontier of a g-map */
\check{\sigma}: g-map × integer × g-mapc × → g-mapc /* auxiliary function */
\check{\sigma}': g-map × integer × g-mapc × → g-mapc /* auxiliary function */

preconditions
prec (σ(g, x)) ≡ e(g, x) ∧ x(g, δ(g), x) = x
prec (\check{\sigma}(g)) ≡ 1 ≤ δ(g)

axioms
σ(g, x) = if x(g, n, y) = y then y else σ(g, y) where n = δ(g), y = x(g, n, x(g, n - 1, x))
\check{\sigma}(g) = \check{\sigma}')(g, δ(g), g)
```

Table 7. Specification of the map of hypervolumes

```plaintext
spec G-MAP 3 using G-MAP 2, MAP 1

operations
hv: g-map → map /* map of hypervolumes of a g-map */
\check{\sigma}: g-map → mapc /* auxiliary function */
\check{\sigma}': g-map × integer × g-mapc → mapc /* auxiliary function */
l0: mapc × integer × dart × dart → mapc /* auxiliary function */

preconditions
prec (hv(g)) ≡ 1 ≤ δ(g)

axioms
hv(g) = hv1(g, δ(g), g)
hv1(g, n, v(n)) = hv0(g)
hv1(g, n, i(g1, x)) = hv1(g, n, g1)
hv1(g, n, l(g1, k, x, y)) =
if e(g2, x) ∧ e(g2, y) then l(g2, k, x, y)
else if k = n - 1
then if e(g2, x) ∧ x(g2, n - 1, x) = σ(g, x) then l(g2, k, x, σ(g, x))
else if e(g2, y) ∧ x(g2, n - 1, y) = σ(g, y)
then l(g2, k, y, σ(g, y)) else g2
else g2
where g2 = \check{\sigma}')(g, n, n)
\check{\sigma}0(g, n, v(n)) = v(n - 1)
\check{\sigma}0(g, n, l(g1, x)) = if x(g, n, x) = x then i(\check{\sigma}0(g, n, g1), x) else \check{\sigma}0(g, n, g1)
\check{\sigma}0(g, n, l(g1, k, x, y)) = \check{\sigma}0(g, n, g1)
```

108 computer-aided design
Characteristics of 2-g-maps

Let us now return to the case of the 2-g-maps. The previous functions allow the calculation of their characteristics, as defined by Lienhardt. In the 2-G-MAP 2 specification of Table 8, the following operations are defined:

- \( b(g) \) is the number of boundaries of a 2-g-map \( g \);
- \( c(g) \) is the Euler characteristic of \( g \);
- \( q(g) \) is the orientability factor of \( g \);
- \( gs(g) \) is the genus of \( g \);
- \( ori(g) \) is true iff \( g \) is orientable.

These characteristics are directly expressed from the basic elements of a 2-g-map by means of functions. In a real implementation, they could be stored and modified at each updating of the 2-g-map. For more developments about these characteristics, in particular the orientability, the reader is referred to References 1, 19 and 20.

MORPHOLOGICAL ASPECTS

Morphological sorts and basic operations for n-h-maps

In order to deal with geometry, an n-g-map must be embedded in an n-dimensional space. Form attributes are associated to elements of every dimension: points are associated to vertices (0-elements), curves to edges (1-elements), surfaces to faces (2-elements), etc. To take into account photometry, we also decorate these elements in any dimension, with other attributes such as colour, thickness of draughts, texture, etc.

In fact, for specification and programming, it is easier and more general to deal with h-maps than with g-maps. Hence, new morphological sorts and operations are introduced. The idea is to associate morphological attributes to the darts, one association per element at dimension \( k \), i.e. \( k \)-element, to a unique representative dart. For example, a point and a colour are associated to a vertex by inserting the coordinates of the point and the colour number in a unique dart in the vertex, i.e. a 0-element. Automatically, the other darts of the vertex also take on these attributes. For simplification, geometry and photometry are considered together, despite the fact that separate and different treatments must be provided. The symbol \( \text{attribute}(k) \) denotes the sort of the attributes of \( k \)-elements.

In H-MAP 4 specification (see Table 9):

- \( ia(h, k, x, a) \) is the insertion of a \( k \)-attribute \( a \) for the dart \( x \) in h-map \( h \);
- \( da(h, k, x) \) is the deletion of the \( k \)-attribute of all darts which are in the same \( k \)-element as dart \( x \);
- \( za(h, k, x) \) is the existence of a \( k \)-attribute for dart \( x \) in h-map \( h \);
- \( d\text{fa}(h, x) \) is the attribute freeness at any dimension for dart \( x \) in h-map \( h \);
- \( z\text{a}(h, k, x) \) is the \( k \)-attribute for dart \( x \) in h-map \( h \);
- \( eqk(h, k, x, y) \) is true iff darts \( x \) and \( y \) belong to the same \( k \)-element of h-map \( h \).

<table>
<thead>
<tr>
<th>Table 8. Characteristics of 2-g-maps</th>
</tr>
</thead>
<tbody>
<tr>
<td>spec 2-G-MAP 2 using 2-G-MAP 1, G-MAP 3</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>ori: 2-g-map → Boolean</td>
</tr>
<tr>
<td>/* orientability of a 2-g-map */</td>
</tr>
<tr>
<td>axioms</td>
</tr>
<tr>
<td>c(g) = (nc(pr(g, 2)) + nc(pr(g, 0)) + nc(pr(g, 1)) - nd(g))/2</td>
</tr>
<tr>
<td>ori(g) = (2 * nc(g) = nc(hv(g)))</td>
</tr>
<tr>
<td>q(g) = if ori(g) then 0 else if odd(b(g) + c(g)) then 1 else 2</td>
</tr>
<tr>
<td>gs(g) = 1 - (b(g) + c(g) + q(g))/2</td>
</tr>
</tbody>
</table>

volume 23 number 2 march 1991 109
Table 9. Specification of morphological sorts and operations

spec H-MAP 4 using H-MAP 3

sort attribute(k: integer) /* morphological (geometrical and photometrical) attributes at dimension k: k-attributes */

operations

ia: h-map x integer x dart x attribute(k) -* h-map /* insertion of a k-attribute */
da: h-map x integer x dart --* h-map /* deletion of a k-attribute */
z(k): h-map x integer x dart -> Boolean /* existence of a k-attribute */
phi: h-map x dart -> Boolean /* morphological freeness */
aeqk: h-map x integer x dart x dart -> Boolean /* access to a k-attribute */
sa: h-map x integer x dart -> Boolean /* belonging to a k-element */

preconditions

prec (ia(h, k, x, a)) = 0 ~ k < ~ e(h) A ~ a(h, k, x)
prec (da(h, k, x)) = 0 ~ k < ~ e(h) A ~ a(h, k, x)
pprec (sa(h, k, x)) = 0 ~< k ~< ~ e(h)
prec (~a(h, x)) = ~ a(h, x)
pprec (~a(h, k, x)) = 0 ~< k ~< ~ e(h) A e(h, x) A ~ a(h, k, x)
pprec (eqk(h, k, x, y)) = 0 ~< k ~< ~ e(h) A ~ a(h, k, x) A ~ a(h, k, y)

axioms

~a(i(h, x), j, z) = ~ a(h, j, z)
~a(l(h, k, x, y), j, z) = if j = k then ~ a(h, j, y) else if ~ a(h, j, x) A ~ a(h, j, y) then ~ a(h, j, z) else ~ a(h, j, z)
~a(ia(h, k, x, a), j, z) = if j = k and eqk(h, k, x, z) then a else ~ a(h, j, z)
d(a(i(h, x), j, z), k, x) = if j = k then l(a(i(h, x), k, x), y) else if eqk(h, j, x, y) A ~ a(h, j, x) then l(a(i(h, y), k, x), x, y) else l(a(i(h, x), k, x), y) ~ a(h, j, y) A eqk(h, j, x, y) then l(a(i(h, y), k, x), x, y) else l(a(i(h, x), k, x), y)
deqk(h, k, x, y) = eqk(pr(h, k), x, y)

de(ia(h, k, x, a), j, z) = if j = k and eqk(h, k, x, z) then a else ~ a(h, j, z)

de(ia(h, k, x, a), j, z) = if j = k and eqk(h, k, x, z) then a else ~ a(h, j, z)

de(ia(h, k, x, a), j, z) = if j = k and eqk(h, k, x, z) then a else ~ a(h, j, z)

de(ia(h, k, x, a), j, z) = if j = k and eqk(h, k, x, z) then a else ~ a(h, j, z)

de(ia(h, k, x, a), j, z) = if j = k and eqk(h, k, x, z) then a else ~ a(h, j, z)

Note that $\phi(h, z)$ is the conjunction of $\zeta(h, k, z)$, for $k$ varying from 1 to $\delta(h)$, built by the auxiliary recursive function $\phi_k$.

Repercussions on the topological layer

In the previous section, new generators of h-maps have been introduced at the most basic level, and some preconditions and axioms of the primitive topological operations must be modified. For instance, suppose that two darts are topologically linked at dimension $k$ by means of operation $l$. Then, at any dimension $j \neq k$, either one or both of the two darts are free of $j$-attributes, or else the two attributes are identical, because the darts are in the same $j$-element. Therefore, the topological layer of specification H-MAP 4 must be revisited, as shown in Table 10.

Preconditions and axioms for more complex topological operations of the previous specifications H-MAP 2, 3, MAP 1, G-MAP 1, etc., must be revised in the same way. Note that they are mixed operators, in the sense of Lienhardt, i.e. topological and morphological operators. But the two points of view could be strictly separated, at least as far as the basic operators are concerned.
**Table 10. Complement for the H-MAP 4 specification**

<table>
<thead>
<tr>
<th>spec H-MAP 4 (following)</th>
</tr>
</thead>
<tbody>
<tr>
<td>preconditions</td>
</tr>
<tr>
<td>prec ( {(h, k, x, y) \mid 0 \leq k \leq \delta(h) \land e(h, x) \land e(h, y) \land x \neq y \land x(h, k, x) = x \land x(h, k, y) = y \land x(h, k) = x \lor z(h, j, x) \lor z(h, j, y) \land eq(pr(h, j), x, y)) } |</td>
</tr>
<tr>
<td>prec ( {(d(h, x)) \mid \neg e(h, x) \land e(h, x) \lor z(h, k, x) = x } |</td>
</tr>
<tr>
<td>axioms</td>
</tr>
<tr>
<td>( e(a(h, k, x), z) = e(h, z) )</td>
</tr>
<tr>
<td>( \phi(a(h, k, x), a, j, z) = \phi(h, j, z) )</td>
</tr>
<tr>
<td>( x(a(h, k, x), a, j, z) = x(h, j, z) )</td>
</tr>
<tr>
<td>( x^{-1}(a(h, k, x), a, j, z) = x^{-1}(h, j, z) )</td>
</tr>
<tr>
<td>( d(a(h, k, x), a, j, z) = d(h, j, z, k, x, a) )</td>
</tr>
<tr>
<td>( r(a(h, k, x), a, j, z) = r(h, j, z, k, x, a) )</td>
</tr>
</tbody>
</table>

**Morphological operations on 2-g-maps**

As above for the topology, let us now focus on the 2-dimensional case, with user realistic constraints and operations. We also suppose that new morphological operations for g-maps override those ones of h-maps.

**Table 11. Morphological operations for 2-g-maps in construction**

<table>
<thead>
<tr>
<th>spec 2-G-MAP 3 using G-MAP 4, 2-G-MAP 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations</td>
</tr>
<tr>
<td>( iea1: 2-g\text{-}map \times point \rightarrow 2-g\text{-}map ) /* insertion of a simple edge with two points */</td>
</tr>
<tr>
<td>( iea2: 2-g\text{-}map \times dart \times point \rightarrow 2-g\text{-}map ) /* insertion of a simple edge with one point */</td>
</tr>
<tr>
<td>( dea: 2-g\text{-}map \times dart \rightarrow 2-g\text{-}map ) /* deletion of a simple edge */</td>
</tr>
<tr>
<td>( sea: 2-g\text{-}map \times dart \times dart \rightarrow 2-g\text{-}map ) /* sewing of two simple edges with motion */</td>
</tr>
<tr>
<td>( uea: 2-g\text{-}map \times dart \rightarrow 2-g\text{-}map ) /* unsewing two simple edges */</td>
</tr>
<tr>
<td>preconditions</td>
</tr>
<tr>
<td>prec ( (iea1(g, p1, p2)) = true )</td>
</tr>
<tr>
<td>prec ( (iea2(g, y1, p2)) = e(g, y1) \land \neg x(g, 1, y1) = y1 )</td>
</tr>
<tr>
<td>prec ( (dea(g, x1)) = prec (dea(g, x1)) )</td>
</tr>
<tr>
<td>prec ( (sea(g, x1, z1)) = prec (sea(g, x1, z1)) \land z1 \neq a(g, 0, x1) \land z1 \neq x(g, 1, x1) )</td>
</tr>
<tr>
<td>prec ( (uea(g, x1)) = prec (uea(g, x1)) )</td>
</tr>
<tr>
<td>axioms</td>
</tr>
<tr>
<td>( iea1(g, p1, p2) = ia(ia(ie(g, x1, x2, x2), 0, x1, p1), 0, x2, p2) ) where ( x1 = newd(g), x2 = newd(i(g, x1)) )</td>
</tr>
<tr>
<td>( iea2(g, y1, p2) = ia(ie(g, x1, x2, x2), 0, x1, p1), 0, x2, p2) ) where ( x1 = newd(g), x2 = newd(i(g, x1)) )</td>
</tr>
<tr>
<td>( dea(g, x1) = if x1 = y1 then if x2 = y2 then g1 else ia(ia(g1, 1, x2), 0, y2, p2) ) else if ( x2 = y2 ) then ia(ia(g1, 1, x1), 0, y1, p1)</td>
</tr>
<tr>
<td>( sea(g, x1, z1) = se(da(da(ia(g, 0, x1), 0, z1), 0, z1)), x1, z1) )</td>
</tr>
<tr>
<td>( uea(g, x1) = ia(ia(ia(ia(g1, 0, x1), 0, z1), 0, x2, p2), 0, x1, p1), 0, z1, p1) = prec (uea(g, x1)) )</td>
</tr>
</tbody>
</table>

In specification 2-G-MAP 3 (see Table 11) morphological 2-g-maps in construction are defined, i.e. of sort 2-g-map, whose elements are only decorated with geometrical attributes at dimension 0; a point of the 3D Euclidean space is always associated to every vertex. For simplicity, only straight line edges join two vertices and delimit plane or non-plane faces, hence topological loops and geometrical bendings are prohibited. In addition, the geometry of edges and faces is not minutely checked: degeneracies and self-intersections are permitted. But, constraints of all kinds could be expressed in this framework.

The new operations are the following:

- \( iea1(g, p1, p2) \) creates a new simple edge joining points \( p1 \) and \( p2 \);
- \( iea2(g, y1, p2) \) creates a new simple edge joining the vertex containing dart \( y1 \) and \( p2 \), with new free darts. Loops are automatically prohibited. Note that, because \( g \) is a g-map, there is no need to reorder darts around the vertex of \( y1 \), as would be the case with h-map;
- \( dea(g, x1) \) deletes the simple edge containing dart \( x1 \), with its attributes if necessary;
The new sort point is a renaming of attribute(0), with ordinary operations concerning 3D points. There is only a single auxiliary operation on h-maps, also usable for 2-g-maps in construction:

- newd(h) gives a new dart, which is different from all darts of h-map h; this operation is not specified here, because it is close to implementation.

Note that, to delete or unsew a simple edge, attributes are systematically deleted and restored where it is necessary. At implementation level, this can be done in another way.

To use operations like iod1, it is necessary to previously select a dart, for instance by means of the coordinates of the vertex that contains it. This is an important problem that we could also formally specify in this framework.

PROTOTYPING AND IMPLEMENTATION

Several proposals have been made to implement boundary representations. The realization through lists with pointers is probably the most well known; for example, with the winged-edge data structure\(^1\), the quad-edge structure\(^7\) and the face adjacency hypergraph representation\(^6\). Relational systems have also been suggested by others.

For topological maps, several attempts have been made at different applications; for example, lists with pointers in a software of plants–images synthesis\(^8\), a CAD system for architecture\(^4\), a graphical editor\(^5\), and a relational database in a cartographic system\(^7\). In the following, two very different examples of implementations of h-maps are demonstrated – rapid prototyping, and efficient procedural pointer realization.

Rapid prototyping using OBJ3

In the rapid prototyping approach, the logic theory resulting from an algebraic specification is transformed into a rewriting system\(^7\). Functional languages with matching, such as ML\(^9\) or MIRANDA\(^6\), or logical ones with unification such as PROLOG\(^6\), can be used for this kind of manipulation. But the most straightforward way is to rewrite the formal descriptions into an algebraic specification language, such as LARCH\(^24\) or OBJ3\(^8\).

Table 12 is a brief illustration of our current preoccupation about formal modelling, including genericity for darts and h-map normalization, with dart ordering. An operational translation of the H-MAP 1 specification into OBJ3 – version 1988 for Sun 3 – is presented, with two objects (in fact, algebras), ROOT and H-MAP 1, and a theory, DART. Thanks to the clause is protecting, object ROOT contains all the features of built-in object INT, i.e. the sorts Bool, Nat, Int and NZINT, respectively, for Booleans, naturals, integers and non-null integers, with their operations\(^9\). DART is the theory of totally ordered sets with elements of sort Dart, and mix ordering \(<\). H-MAP 1 is a generic object, with formal parameter D for an algebra which satisfies the DART theory. The operation names are kept unchanged, except for the Greek letters changed to Roman ones, \(x\) rewritten \(a\), and \(d\) renamed \(dn\).

After sorts, operation profiles and variables are declared, axioms of two types are written: equational ones, after eq, and conditional, after ceq. Note that the form if \(\ldots\) then \(\ldots\) else \(\ldots\) fi is allowed in equational axioms.

The semantics of OBJ3 is that of initial algebra with equality \(=\), denoted \(\equiv\). This semantics coincides with an operational mechanism of rewriting, where the axioms are oriented from left to right in order to give (possibly conditional) rewriting rules. Thus, \(r1 = r2\) iff the normal forms of \(r1\) and \(r2\) are syntactically equal.

In order to test the specification, Table 13 declares an object H-MAP1-INT which instantiates generic object H-MAP 1 with D replaced by INT, i.e. Dart by Int, and \(<\) by the ordinary ordering of integers. H-maps \(h1\) and \(h\) are constants which correspond to the example of Figure 1.

After compiling the previous objects and theory, OBJ3 allows use of operations, with questions (after red) and answers (with the number of rewrites), for instance:

<table>
<thead>
<tr>
<th>Questions</th>
<th>OBJ3 Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>red e(h, 2) rewrites: 44</td>
<td>Bool: true</td>
</tr>
<tr>
<td>red e(h, 10) rewrites: 67</td>
<td>Bool: false</td>
</tr>
<tr>
<td>red e(h, 0, 1) rewrites: 55</td>
<td>NZINT: (-1)</td>
</tr>
<tr>
<td>red e(h, 1, 0, 1) rewrites: 38</td>
<td>Hmap: h1(h1(h1(h1(0, 1), 2), 1), 1), (-1)</td>
</tr>
</tbody>
</table>

In Table 12, preconditions are not mentioned. They could be taken in account by introducing new error features, as in Goguen\(^7\). All the specifications of the previous sections have been translated (without preconditions) into OBJ3 and checked. Some of them are very inefficient, due to the slowness of the sequential rewriting engine of OBJ3, but above all to the inherent exponential complexity of certain recursive definitions. Recall that this rapid prototyping is only useful to verify logical specifications, and not to develop applications. A following paper will relate in detail this experiment with OBJ3.

Pointer implementation

As another short example, the previous H-MAP 4 specification can be written in the C language, as shown in Table 14. For simplicity, it is assumed that all \(k\)-elements and connected components are represented by an object of the single type element carrying general data. This object is linked to the darts by means of a unique representative dart.

In order to make manipulation of the objects easy, the following types of pointers are defined: dart,
for clarity, it is assumed that all morphological attributes are relevant from a unique type attribute.

In this implementation, the basic access operations (selectors) are very easily written as functions. For
Table 13. Test of H-MAP 1 specification

\[
\text{obj } \text{H-MAP1-INT is protecting H-MAP1[INT] .}
\]

\[
\begin{align*}
\text{op } h: & \rightarrow \text{Hmap} . \\
\text{op } h1: & \rightarrow \text{Hmap} . \\
\end{align*}
\]

\[
\begin{align*}
h1 &= h1(h1(h1(h1(2), 1), -1), 0, 1, -1, 2), -2), 0, 2, -2(1, 1, 2), 1, -1, -2 ) . \\
h &= h1(h1(h1(h1(3), 4), 5), 6, 1, 3, 5), 2, 1, 3), 2, -1, -2(1, 1, 6, 5) .
\end{align*}
\]

Table 14. Pointer implementation of h-maps in C language

```
typedef struct stdart
{
    hmap sh;
    dart sa, sal[m];
    element sel[m];
    element sc;
    dart snd, spd;
    Bool mark;
} stdart, *dart;
```

```
typedef struct stelement
{
    hmap sh;
    dart srd;
    attribute sa;
    dart snd, spd;
} selement, *element;
```

```
typedef struct sthmap
{
    int sdn;
    dart std;
    element sle[m];
    element sic;
    int snb, sc;
    Bool sor;
} sthmap, *hmap;
```

example, operation a (for z) is:

```
dart a(h, k, x) hmap h; int k; dart x;
{
    return (x->sa[k]);
}
```

Remember that x must belong to h by the precondition of operation a. Another way of programming is by testing this precondition in the function body and, in case of a negative answer, sending an error message or a signal. Since each dart points on its connected component, dart equivalence test eq can merely be programmed by using the equality between pointers:

```
Bool eq(h, x, y) hmap h; dart x, y;
{
    return(x->sc == y->sc);
}
```

Except for the g-maps, the update operations (generators) are more difficult to program, especially the deletions of objects. Of course, operations like i or l do not create a new hmap, but modify the old one in the main memory. But the updating of pointers is expensive during modifications of h-maps. They often need a traversal of hypermaps with dart marking. This explains the presence of a permanent mark field in the dart structure.

Finally, one has to make sure that these functions and procedures are correct, i.e., they conform to the algebraic specification. This can be done systematically by means of proof techniques similar to those of Hoare. Their application is not always really difficult, but long and tedious.

A following paper will explain in more detail transformations from the algebraic specification to an efficient implementation in C language for a Silicon Graphics 4D workstation, in the framework of our 3D modeller project.
CONCLUSIONS

Sorts of objects and operations for handling combinatorial h-maps, maps, g-maps and morphological elements have been described in modules of algebraic specifications in a hierarchical way. The approach is expected to be a sound foundation to build complex graphics systems. Only a first kernel has been presented, but it can be extended by new sorts and high-level operations and adapted for real applications. For instance, our own experience with a cartographic system and models of plant growing has shown that these propositions are realistic.

At the moment, we are continuing with a project on the B-rep geometric 3D modeller, where algebraic specification, rapid prototyping in OBJ3, PROLOG and programming in C are brought into play. In this framework, Euler operators and generalizations have been specified and implemented, as well as operators to build, subdivide and sew together elementary opened or closed solids such as cube, sphere, cylinder, torus, etc. Other high-level operators, like Boolean ones, are studied but not wholly defined yet.

In general, the algebraic specification technique has demonstrated a good security factor and a high possibility of reutilization. Indeed, it offers a clear, elegant and abstract functional description. The equational axiomatization provides a mathematical framework for expressing and proving properties – in fact, integrity constraints – of the described objects, as well as proving the consistency and the completeness of the specification. It would be very instructive to compare this approach to other methods, like pre-post axiomatic specifications, and derivatives such as VDM, or Z language, also used for specifying boundary representation. In the area of computer graphics, these formal practices seem more and more necessary.

In boundary representation, the algebraic techniques allow the incremental building of sorts and operations, the separation of topological aspects and morphological ones (geometric embedding and morphology), and their mixage if necessary. In this way, preconditions for hierarchized operations have been defined, which lead to sound graphical objects satisfying integrity constraints. Topological constraints have been emphasized, but the way is open for complex geometric or morphological ones. Expressing constraints of usual objects in a CAD domain, mechanics for instance, could be very useful. Our framework would allow such a study; it would become very difficult in another approach too close to implementation.

Rapid prototyping of specifications into an executable program is an important verification of the feasibility of software projects. Although other languages can be applied, such as functional languages or logical ones after some transformations, OBJ3 seems very convenient. Indeed, whenever large graphics software is to be developed, modularity, genericity and inheritance are needed. All that is also given by OBJ3. But it would be interesting to compare it with other computer-aided specification systems.

Formal specification is unfortunately not the panacea: it is a very precise tool to logical problem setting, but some very important problems remain outside its framework. That is particularly the case for the question of man–machine communication, a major subject in interactive computer graphics, where formal methods seem ineffective.

Finally, a great part of the difficulties of a sale and efficient implementation from specifications has no good solution. As pointed out by Bergstra et al., this transition has been very little studied. Attempts with low-level efficient languages like C are useful, but difficult, because they need a lot of non-trivial transformations during the development. In the future, high-level programming languages such as object-oriented ones, would certainly be most convenient.

ACKNOWLEDGEMENTS

This research is much influenced by the works of the ‘Strasbourg 3D modeling group’. I would like to thank Y Bertrand and P Lienhardt for their pertinent comments on the successive versions of this paper. I am very grateful to the CAD reviewers for their careful reading and helpful remarks. They gave me much work, but they greatly contributed to improve the meaning of the text. I also thank J Korczak and P Frath for a hard task: the correction of the English translation of the paper.

This research was partly supported by the PRC Mathématiques et Informatique and the GRECO de Programmation, CNRS, France.

REFERENCES

5. Spivey, J M The Z Notation Prentice-Hall, UK (1989)


19 Lienhardt, P ‘Extension of the notion of map and subdivision of three dimensional space’ Proc. STACS Bordeaux, France (1988) pp 301–311


29 Tutte, W T ‘Graph theory’ Encyclopedia of Mathematics and its Applications Addison-Wesley, USA (1984)


31 Cori, R ‘Un code pour les graphes planaires et ses applications’ Astérisque Vol 27 SMF Paris, France (1975)

32 Bertrand, Y ‘Spécification algébrique et prototypage logique d’un noyau de modeleur 3-D basé sur les cartes’ Rapport techn. CRI Univ. L Pasteur, Strasbourg, France (1990)


