Master Sciences et Technologies
mention STIC (Sciences et Technologie de l'Information et de la Communication)

Méta-Modeleur, un générateur de noyaux de modeleurs

Mathieu Poudret

Mémoire de Stage recherche M2, spécialité Fondements de l'Ingénierie de l'Informatique et de l'Image

effectué sous la direction de :
Agnès Arnould
Yves Bertrand

au Laboratoire Signal, Image, Communications, FRE n°2731
Université de Poitiers - UFR Sciences Fondamentales et Appliquées

Poitiers, juillet 2005

Remerciements

Avant tout, je tiens à remercier Agnès Arnould et Yves Bertrand pour leur disponibilité tout au long du stage. Je les remercie également pour l'aisance avec laquelle ils m'ont fait partager leurs connaissances scientifiques.

Cette année de Master 2 aura été l'année la plus décisive de mon cursus universitaire. C'est aussi pour cette raison qu'elle a été pour moi, l'une des plus stressante. Aussi, je tiens à remercier Agnès et Yves pour leur bienveillance ainsi que leurs discours apaisants, dans ces moments où ma nature angoissée a rendu mon travail plus difficile.

Je remercie aussi Pascal Lienhardt qui a su à plusieurs reprises trouver le temps de venir discuter de mon stage d'une manière générale, et de modèles topologiques en particulier.

Je remercie également Eric Andres sans qui je n'aurais peut-être jamais réalisé qu'il est parfois inutile d'être stressé.

Enfin, je remercie ma famille et mes amis d'avoir accepté que je réduise parfois ma vie sociale à néant pour cause de Méta-Modélisation, ainsi que Pierre-François et Sébastien pour m'avoir permis de passer une année universitaire dont je n'aurais pas voulu rater une seule minute...

Table des matières

Résumé

La majorité des modeleurs géométriques à base topologique sont basés sur un modèle fixe, adapté à un domaine particulier (géologie, architecture, etc.). Devant la multiplicité de ces domaines, les chercheurs ont besoin de toujours plus de modeleurs, dont le temps de développement peut être très élevé.

L'objectif du stage est de créer un Méta-Modeleur. Cet outil doit générer le code d'un noyau de modeleur à partir de la description succincte d'un modèle topologique et de ses opérations. Les mois de développement actuels seront ainsi transformés en jours de choix ou mise au point du modèle.

La première partie du rapport est consacrée à la création d'une structure de données générique dont seule l'instanciation la spécialise en fonction d'un modèle particulier. La seconde partie explicite une méthode grâce à laquelle les contraintes de cohérence du modèle peuvent être utilisées lors de la génération du code des opérations.

1 Introduction

Les modeleurs géométriques sont des logiciels de création et de manipulation d'objets géométriques. Ils utilisent une représentation fixe des objets, choisie en fonction du domaine d'application visé. Ce sont des logiciels importants dont le coût de développement peut représenter de nombreuses années/hommes.

Dans un environnement de recherche en modélisation géométrique, les modèles utilisés sont nombreux car les natures et dimensions des objets représentés sont très variables en fonction des applications (villes, bâtiments, animations, couches géologiques, etc.). De ce fait, les chercheurs consacrent beaucoup de temps à développer de nouveaux modeleurs nécessaires à leur recherche.

Pourtant, des points communs existent entre les modèles topologiques. À partir d'une étude de cette intersection, nous souhaitons mettre en oeuvre un Méta-Modeleur, c'est-à-dire un générateur de noyaux de modeleurs. La simple description du modèle topologique et de ses opérations sera transformée automatiquement en noyau de modeleur. Les mois de développement actuels seront ainsi transformés en jour de choix ou mise au point du modèle.

1.1 La modélisation géométrique à base topologique

Ce document traite dans une large mesure de modélisation géométrique à base topologique. Aussi, il est utile de rappeler les bases de ce concept souvent confondu avec la modélisation géométrique « classique ». Dans celle-ci, seule la forme et la position des objets est représentée. La modélisation géométrique à base topologique s'intéresse en plus à la structure des objets. Par exemple, dans un espace à trois dimensions, nous parlons de sommets, arêtes, faces, volumes et de relations d'incidences ou d'adjacences entre ces éléments. C'est à la topologie combinatoire qu'est due cette approche permettant de réduire considérablement les temps de calculs de nombreux algorithmes basés sur le parcours de scènes complexes.

1.1.1 Les premiers modèles topologiques

Depuis les années 70, les modèles topologiques ont suivi de nombreux courants, dont le modèle fil de fer a été l'initiateur. Il fut rapidement abandonné en raison d'une trop grande ambiguïté dans la compréhension des objets qu'il permet de manipuler. Par exemple, la géométrie d'un cube représenté en fil de fer (figure 1A) peut être interprétée de plusieurs manières (figure 1B).

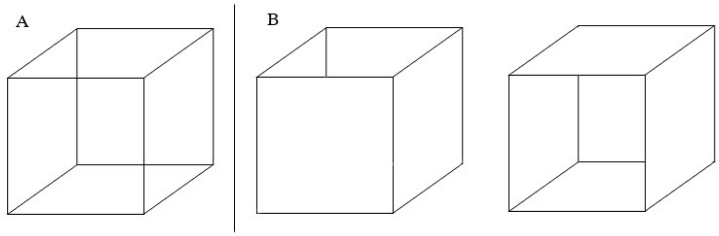

Fig. 1 – Ambiguïté du modèle fil de fer.

Le modèle fil de fer a laissé place à deux grandes familles de modèles topologiques : la géométrie constructive solide (CSG, Constructive Solid Geometry) et la représentation par les bords (B-Rep, Boundaries-Representation).

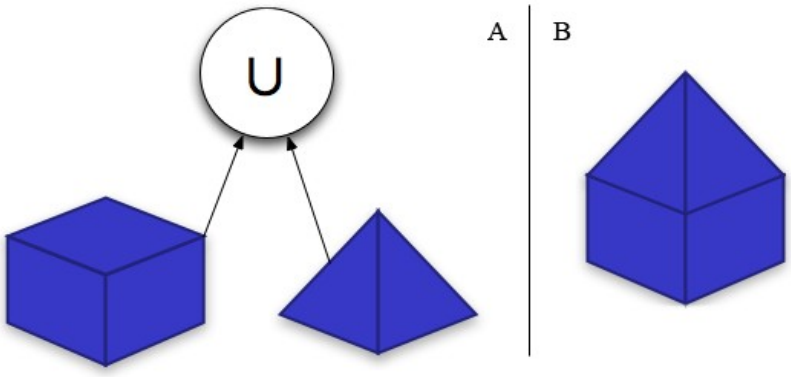

Fig. 2 – Un arbre CSG et son interprétation.

En CSG, les objets sont donnés par une suite d'opérations booléennes représentée la plupart du temps par un arbre binaire (figure 2A). Les feuilles de l'arbre sont occupées par une primitive (cube, cylindre, pavé, etc.), les noeuds correspondent à une opération booléenne (union, intersection, etc.).

L'objet final existe seulement d'une manière implicite. Il est donné par un parcours de l'arbre depuis les feuilles jusqu'à la racine. Sur la figure 2, l'objet final présenté en 2B est l'interprétation de l'arbre présenté en 2A. Cette représentation implicite des objets pose des difficultés lorsqu'il s'agit d'attacher des propriétés à des objets ou parties d'objets.

Cette limitation est levée avec le modèle B-Rep. Dans celui-ci les objets sont représentés explicitement par un découpage de leur surface. Ainsi, chaque morceau peut être complètement caractérisé. En revanche, avec un tel modèle il est impossible d'établir des liens entre les volumes puisque topologiquement l'objet est en dimension 2.

1.1.2 Les structures cellulaires

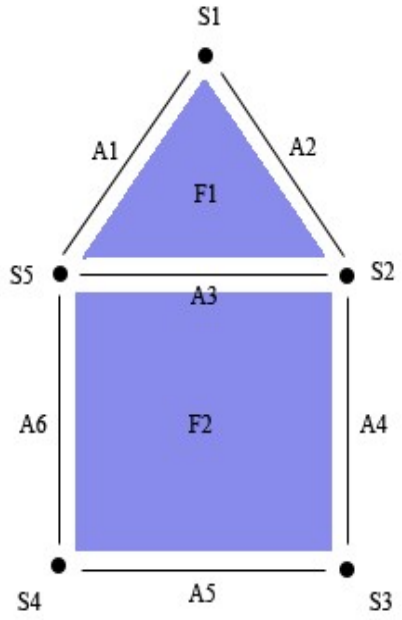

FIG. 3 – Une subdivision cellulaire.

Plus récemment, pour répondre aux limitations sous-jacentes aux familles présentées plus haut, de nouveaux modèles topologiques ont fait leur apparition. Ces modèles appartiennent à la classe des subdivisions d'espace (ou structures cellulaires). Dans celle-ci, un espace de dimension n est partitionné en $n+1$ familles de cellules de dimension 0 à n munies de relations d'incidences et/ou d'adjacences. Un exemple de subdivision cellulaire est donné en figure 3, où $S_{i=1,\ldots,5}$ sont des 0-cellules[1] (sommets), $A_{j=1,\ldots,6}$ sont des 1-cellules (arêtes) et $F_{k=1,2}$ sont des 2-cellules (faces). Sur cette figure, ni les relations d'incidences, ni les relations d'adjacences ne sont représentées.

[1]On note i-cellule, une cellule de dimension i.

Cc sont aux subdivisions d'espace que nous allons nous intéresser dans le cadre de notre Méta-Modeleur. Cette classe peut être divisée en plusieurs sous-classes. Par exemple, on peut distinguer les modèles topologiques dans lesquels les cellules sont représentées explicitement des modèles topologiques dans lesquels elles sont données implicitement ou bien explicitement et implicitement à la fois. En section 2, nous étudions un premier modèle dans lequel les cellules sont données de manière implicite, puis un second dans lequel elles sont données explicitement.

1.1.3 Où réapparaît la géométrie

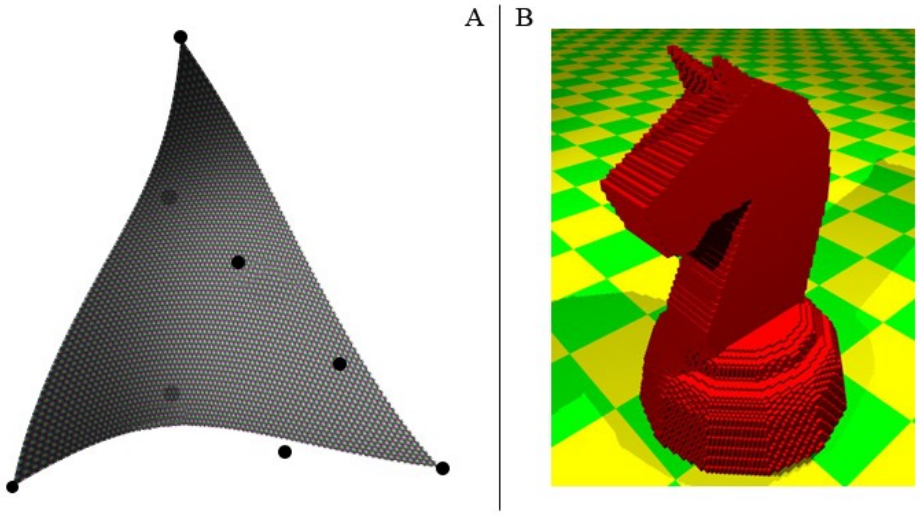

FIG. 4 – Plongement de Bézier et plongement discret.

Les subdivisions d'espaces se limitent à une représentation topologique des objets, et en tant que telle ne fournissent aucune information géométrique. Si l'on souhaite visualiser les objets, il est nécessaire d'associer un modèle de plongement aux modèles topologiques. Dans la pratique, un modèle de plongement attache une information géométrique aux différentes cellules topologiques.

L'un des modèles de plongement les plus couramment utilisés consiste à associer un point de l'espace à chaque sommet. Ceci permet de tracer des arêtes entre les sommets, selon qu'ils sont incidents ou non à une même arête dans la représentation topologique de l'objet. Ce plongement simple donne une représentation linéaire des objets.

Il existe des plongements plus complexes dont deux représentant fréquemment utilisés sont les plongements de Bézier (figure 4A) et les plongements discrets (figure 4B).

Il convient de distinguer la topologie du plongement non seulement au niveau de la description de l'objet, mais aussi au niveau des opérations de construction et/ou modification des objets. Par exemple, l'opération de triangulation modifie la topologie d'un objet sans toucher à sa géométrie, tandis que les opérations de déformation (comme tirer sur les sommets d'un cube) modifient la géométrie, mais n'ont aucun effet sur la topologie.

1.2 La génération de code

Qu'il s'agisse de simuler des phénomènes réels (géologie, biologie, etc.) ou de créer des divertissements (cinéma, jeux vidéos, etc.), la modélisation géométrique est utilisée dans de nombreux domaines. Devant la multiplicité de ces champs d'utilisation, les scientifiques doivent régulièrement créer ou adapter des modèles géométriques ainsi que les modeleurs associés. Or, la création d'un modeleur est très coûteuse en temps de développement.

Pourtant, de part leur nature, il apparaît que les modèles topologiques ne sont pas complètement décorrélés. Ainsi, à l'aide d'une étude précise de ce lien existant entre les différents modèles, nous souhaitons créer un outil capable de générer automatiquement le code d'un noyau de modeleur. Cet outil, que nous appelons Méta-Modeleur, aura pour seule entrée la description succincte d'un modèle topologique et de ses opérations. Grâce à lui, l'utilisateur n'aura plus besoin de programmer manuellement l'ensemble du modeleur correspondant au modèle topologique utilisé, le temps de développement en sera alors considérablement réduit.

Ainsi, le Méta-Modeleur est un générateur de code puisqu'il doit traduire la description d'un modèle géométrique et de ses opérations, en code de noyau de modeleur basé sur ce modèle. Comme tout compilateur, il comprend une partie classique d'analyse de la description de départ, mais en raison de sa grande spécialisation, la partie génération de code est beaucoup plus spécifique. Ainsi, une part importante de ce rapport sera consacrée aux aspects théoriques de cette seconde partie.

1.3 Contenu du rapport

De nombreux documents sont consacrés à la modélisation géométrique à base topologique, mais ceux-ci se limitent dans la plupart des cas à l'étude

d'un modèle fixe ainsi qu'à son utilisation dans un domaine particulier. Ainsi, nous ne nous attardons pas sur les résultats de la recherche bibliographique effectuée en début de stage. Néanmoins, des références sont données lorsque nous abordons l'étude de modèles exemples.

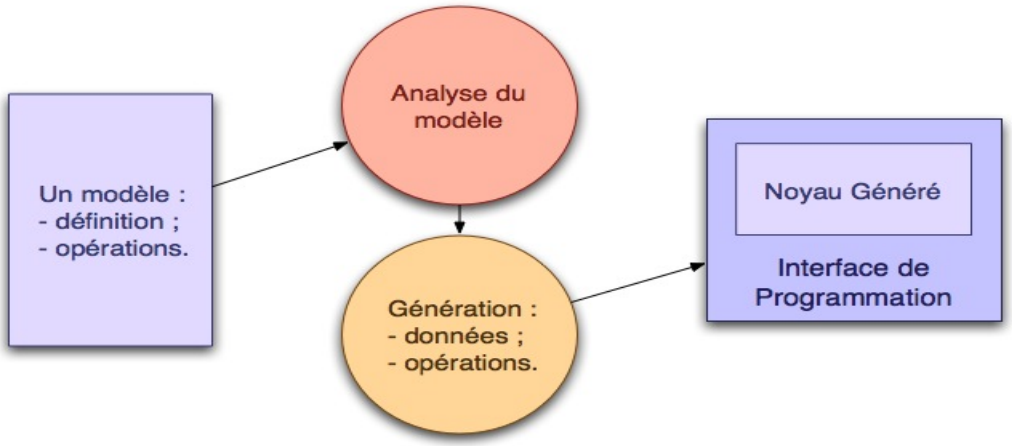

Fig. 5 – L'architecture du Méta-Modeleur.

Afin de faciliter l'introduction du plan de ce document, nous nous appuyons sur l'architecture que nous nous sommes fixée pour notre Méta-Modeleur (figure 5). Sur ce schéma, le Méta-Modeleur (au centre) est composé de deux grandes parties. Le premier module est consacré à l'analyse lexicale puis syntaxique de la description du modèle d'entrée (à gauche sur le schéma), il aboutit à la construction d'un arbre de syntaxe abstraite contenant les données nécessaires à la génération du code. Cette première partie est à la fois technique et classique en ce sens qu'elle est commune à la plupart des projets de compilation, par conséquent nous travaillons directement sur l'arbre de syntaxe abstraite. Dans ce rapport, nous allons détailler l'étude suite à laquelle nous avons fixé son contenu ainsi que l'utilisation que nous en faisons, en vue de générer du code.

Sur la figure, la réécriture en code fonctionnel de la description d'entrée est donnée en deux points. Ainsi, après avoir donné deux exemples de modèles topologiques, nous consacrons une seconde partie à la génération d'une structure de données permettant de stocker et de manipuler les objets créés suivant le modèle d'entrée. Enfin, la dernière partie du rapport est consacrée à la solution que nous souhaitons mettre en oeuvre afin de générer le code des

opérations.

Notons que pour rendre exploitable le code généré par le Méta-Modeleur, nous devons lui associer une interface de programmation (à droite sur le schéma). Cette partie n'a pas été traitée lors de ce stage, elle n'est donc pas abordée dans le rapport. En revanche, dans la section 3.3.2, nous traitons brièvement l'association d'une interface graphique au noyau généré.

2 Deux modèles géométriques à base topologique

Si nous souhaitons montrer que notre Méta-Modeleur fonctionne correctement, nous devons montrer qu'il sait gérer l'ensemble des modèles topologiques existants. S'il génère un noyau correct pour ceux-ci, il est raisonnable de penser qu'il fonctionnera sur de nouveaux modèles topologiques, ou sur des extensions des modèles connus. Ainsi, la conception d'un générateur de noyaux de modeleurs nécessite de faire une étude préalable des différents modèles que l'on peut rencontrer. Cependant, et ce à cause de leur trop grand nombre, il n'est pas raisonnable de tous les analyser en détail.

Bien que nous ayons étudié un grand nombre de modèles dans le cadre de la conception de ce générateur de noyaux, nous nous limitons ici à une description détaillée de deux d'entre eux : les cartes généralisées [Lie89] et les ensembles semi-simpliciaux [May67]. Dans les cartes généralisées, les cellules topologiques (sommets, arêtes, faces, etc.) sont représentées implicitement (elles sont données par un parcours des éléments de base du modèle : les brins), tandis que dans les ensembles semi-simpliciaux, elles apparaissent explicitement dans la structure du modèle. En raison de cette différence notable dans la manière de représenter les cellules topologiques, ces deux modèles sont bien représentatifs des modèles topologiques existants. Ils permettent ainsi de définir précisément les besoins du Méta-Modeleur.

2.1 Les cartes généralisées

Définition du modèle

Les cartes généralisées appartiennent à la famille des subdivisions d'espace, elles sont une extension des cartes combinatoires [Tut84] [BS85] et permettent de représenter la topologie des quasi-variétés[2] de dimension n, orientables ou non, avec ou sans bord.

Dans la figure 6, l'objet A est une quasi-variété : il est constitué de deux 3-cellules collées le long d'une 2-cellule. En revanche, ni le B ni le C n'entrent dans la classe des quasi-variétés : dans un cas une 0-cellule est incidente à plus de deux 1-cellules, et dans l'autre deux 3-cellules sont collées le long d'une 1-cellule.

[2]Une quasi-variété de dimension n est un objet de dimension n obtenu par assemblage de n-cellules le long de $(n-1)$-cellule. Une $(n-1)$-cellule ne peut pas appartenir au bord de plus de deux n-cellules.

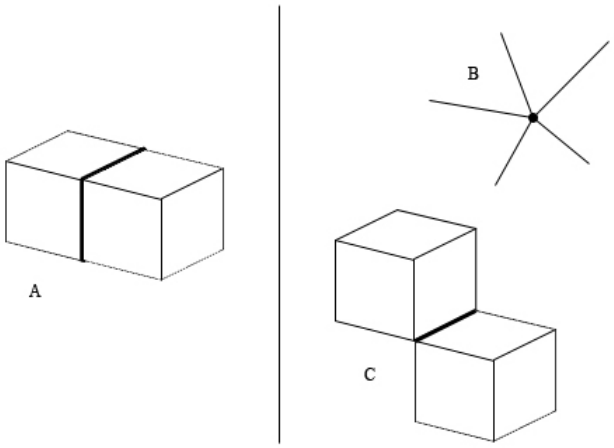

FIG. 6 – Quasi-variétés, exemple et contre-exemples.

Avant de définir la notion de cartes généralisées, nous devons introduire une notation. Si β et γ sont des applications de $E \to E$, nous notons $\beta\gamma$ la composition $\beta \circ \gamma$, et $b\beta\gamma$ l'application de cette composition à un élément b de E.

Intuitivement, une carte généralisée de dimension n (ou n-G-carte) est la donnée d'un ensemble d'éléments abstraits appelés brins, liés entre eux par des involutions[3] (chaque brin étant lié à $n+1$ brins distincts ou non). La cohérence des objets ainsi construits est assurée par la donnée d'une contrainte. La définition 1 décrit mathématiquement une n-G-carte.

Définition 1 (Cartes généralisées de dimension n)**.** *Une carte généralisée de dimension $n \geq 0$ (ou n-G-carte) est une algèbre $G = (B, \alpha_0, ..., \alpha_n)$, où :*

- *B est un ensemble de brins ;*
- *$\alpha_0, ..., \alpha_n$ sont des involutions sur B ;*
- *$\alpha_i\alpha_j$ est une involution pour tout i, j vérifiant $0 \leq i < i+2 \leq j \leq n$.*

Un exemple d'opération

La définition seule d'un modèle ne suffit pas. Les constructions et modifications des objets nécessitent de définir des opérations. Ainsi, définissons

[3]Une application f est une involution si et seulement si $f = f^{-1}$.

l'opération de couture, qui, associée à l'opération d'ajout d'un brins, permet de construire tout objet appartenant à la classe des quasi-variétés.

Coudre deux brins d'une même carte consiste à les lier par une involution. Si nous assimilons les brins à des demi-arêtes, lier deux brins par α_0 permet d'obtenir une arête complète, α_1 permet de coudre des arêtes entre elles, α_2 de coudre des faces, α_3 des volumes. Plus généralement, deux i-cellules sont liées par α_i. Cependant, dans la plupart des cas l'ajout de ce seul lien entre les brins à coudre ne suffit pas. En effet, pour respecter la contrainte de cohérence des G-cartes, il est nécessaire d'effectuer un parcours d'orbites[4] à partir des deux brins, puis de lier entre eux chaque brin de ces orbites.

Après avoir défini la notion d'isomorphisme dans une carte généralisée (définition 2), nous donnons une définition rigoureuse de l'opération de couture (définition 3), puis nous l'illustrons dans un exemple simple de construction topologique (figure 7).

Définition 2 (Isomorphisme dans une carte généralisée de dimension n)**.** *Soient $G = (B, \alpha_0, ..., \alpha_n)$ une n-G-carte, b_1 et $b'_1 \in B$ et un sous-ensemble $I = \{\alpha'_1, ..., \alpha'_k\} \subset \{\alpha_0, ..., \alpha_n\}$. Une application $\varphi : B \to B$ réalise un isomorphisme de $< I > (b_1)$ sur $< I > (b'_1)$ si et seulement si :*

- *φ réalise une bijection de $< I > (b)$ sur $< I > (b')$;*
- *pour tout $\alpha_i \in I$, pour tout $b \in < I > (b)$, on a $b\alpha_i\varphi = b\varphi\alpha_i$.*

Définition 3 (Couture de brins dans une carte généralisée de dimension n)**.** *Soient $G = (B, \alpha_0, ..., \alpha_n)$ une n-G-carte, b et b' deux brins de B et un entier i vérifiant $0 \leq i \leq n$. Soit φ un isomorphisme de $< \alpha_0, ..., \alpha_{i-2}, \alpha_{i+2}, ..., \alpha_n > (b)$ dans $< \alpha_0, ..., \alpha_{i-2}, \alpha_{i+2}, ..., \alpha_n > (b')$ avec $\varphi(b) = b'$. Nous définissons $G' = (B', \alpha'_0, ..., \alpha'_n)$, la n-G-carte résultant de la i-couture de b et b' dans G par :*

- *$B = B'$;*
- *pour tout j tel que $0 \leq j \leq n$ et $j \neq i$, $\alpha'_j = \alpha_j$;*
- $\alpha'_i = \begin{cases} \varphi(e) \text{ si } e \in < \alpha_0, ..., \alpha_{i-2}, \alpha_{i+2}, ..., \alpha_n > (b) ; \\ \varphi^{-1}(e) \text{ si } e \in < \alpha_0, ..., \alpha_{i-2}, \alpha_{i+2}, ..., \alpha_n > (b') ; \\ \alpha_i(e) \text{ sinon.} \end{cases}$

[4]Soient $G = (B, \alpha_0, ..., \alpha_n)$ une n-G-carte, $b \in B$ et un sous-ensemble $\{\alpha'_1, ..., \alpha'_k\} \subset \{\alpha_0, ..., \alpha_n\}$. L'orbite $< \alpha_1, ..., \alpha_k > (b)$ est l'ensemble des brins b' tel qu'il existe une composition c quelconque de $\alpha'_1, ..., \alpha'_k$ telle que $bc = b'$. Dit plus simplement, une orbite $< \alpha_1, ..., \alpha_k > (b)$ est l'ensemble des brins atteignables à partir de b, par composition d'involutions de $\{\alpha_1, ..., \alpha_k\}$.

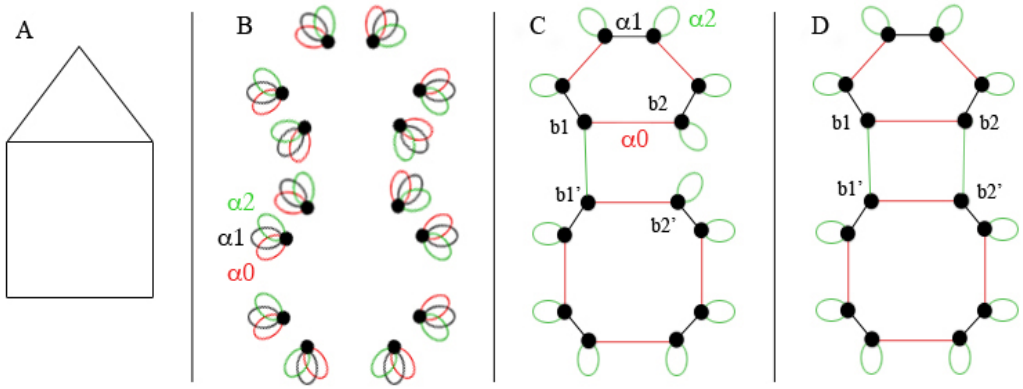

FIG. 7 – Construction d'un objet 2D.

Afin de mieux comprendre la manière dont est construit un objet, représentons une maison (figure 7A) sous la forme d'une 2-G-carte.

La première étape consiste à se donner une carte constituée d'un ensemble de brins isolés[5] (figure 7B). La topologie de la maison est obtenue par coutures successives. Les arêtes du toit et de la façade sont obtenues par couture en α_0, les faces correspondant au toit et à la façade sont ensuite obtenues par couture des arêtes par α_1.

La partie la plus délicate consiste à coudre la façade et le toit, c'est-à-dire coudre deux faces entre elles. Il s'agit pour ceci de coudre par α_2, b_1 avec b'_1 (figure 7C) et b_2 avec b'_2 (figure 7D). Remarquons que si b_1 et b'_1 sont liés par α_2 mais que b_2 et b'_2 sont indépendants (figure 7C), l'objet final n'est pas une 2-G-carte en ce sens qu'il ne respecte pas la contrainte de cohérence du modèle : $(\alpha_0\alpha_2 \circ \alpha_0\alpha_2)(b1) = b'_1 \neq b_1$, ce qui signifie que $\alpha_2\alpha_0$ n'est pas une involution. C'est pourquoi si nous suivons la définition de la couture (définition 3), le fait de coudre b_1 et b'_1 par α_2 conduit à coudre b_2 avec b'_2 (figure 7D), ce qui permet de maintenir la cohérence de l'objet. Ainsi le résultat de la couture est une 2-G-carte au sens de la définition 1.

2.2 Les ensembles semi-simpliciaux

Définition du modèle

Les ensembles semi-simpliciaux appartiennent eux aussi à la famille des subdivisions d'espace. Leur espace de représentation est plus large que celui

[5]On appelle brin isolé, tout brin b tel que pour tout i vérifiant $0 \leq i \leq n$, $b\alpha_i = b$.

des G-cartes puisqu'il ne se limite pas aux quasi-variétés mais permet de représenter la topologie de tout objet géométrique subdivisé. Un ensemble semi-simplicial est une famille d'objets abstraits appelés simplexes, liés entre eux par des opérateurs de bords pour lesquels nous donnons une contrainte de cohérence. La définition 4 décrit un ensemble semi-simplicial de dimension n.

Définition 4 (Ensemble semi-simplicial de dimension n)**.** *Un ensemble semi-simplicial de dimension $n \geq 0$, est une algèbre $S = (K, (d_j)_{j=0,...,n})$, où :*

- *$K = \bigcup_{i=0}^{n} K^i$, avec K^i un ensemble fini de simplexes de dimension i ;*
- *pour tout $i \geq 1$, d_j est une application $K^i \rightarrow K^{i-1}$ appelée opérateur de bord, avec $0 \leq j \leq i$;*
- *pour tout i tel que $2 \leq i \leq n$, pour tout k et j tels que $0 \leq k < j \leq i$, pour tout $\sigma \in K^i$, $\sigma d_j d_k = \sigma d_k d_{j-1}$.*

Le dernier point de cette définition peut être reformulé de la manière suivante : pour tout simplexe σ de dimension n (on parle de n-simplexe), deux faces[6] quelconques de σ de dimension $n-1$, alors ces deux faces ont au moins une face de dimension $n-2$ commune.

Un exemple d'opération

L'opération de base des ensembles semi-simpliciaux est l'identification. Tout les objets appartenant à la classe de représentation du modèle peuvent être obtenus par composition de cette opération et de l'opération d'ajout d'un simplexe.

Intuitivement, identifier deux simplexes σ_1 et σ_2 de même dimension appartenant à un même ensemble semi-simplicial S revient à identifier leurs bords[7] puis à :

- retirer σ_1 et σ_2 de S ;
- ajouter à S un nouveau simplexe σ_3 de même dimension que σ_1 et σ_2 ;
- modifier les opérateurs de bords de manière à respecter les contraintes de cohérence du modèle.

[6] σ' est une face de σ s'il existe un opérateur de bord liant σ à σ'.

[7] Le bord d'un i-simplexe σ est l'ensemble semi-simplicial contenant l'ensemble des simplexes σ' tels qu'il existe $l_{i-1}, ..., l_j$ vérifiant $\sigma d_{l_{i-1}} ... d_{l_j} = \sigma'$.

Définition 5 (Identification de simplexes dans un ensemble semi-simplicial de dimension n)**.** *Soient* $S = (K, (d_j)_{j=0,\dots,n})$, σ_1 *et* σ_2 *deux* k*-simplexes.* $S' = (K', (d'_j)_{j=0,\dots,n})$, *l'ensemble semi-simplicial résultant de l'identification de* σ_1 *et* σ_2 *dans* S *est défini par :*

- $K' = K - \{\sigma_1, \sigma_2\} \cup \{\sigma_3\}$ *où* σ_3 *est un nouveau simplexe issu de l'identification de* σ_1 *et* σ_2 *;*
- *si* $k \geq 1$, $\sigma_3 d'_i = \sigma_i$, *où* $\sigma_i = \sigma_1 d_i = \sigma_2 d_i$ *pour tout* i *tel que* $0 \leq i \leq k$ *;*
- *pour tout* $\sigma \in K$, $\forall i \leq k+1$ *si* $(\sigma d_i = \sigma_1$ *ou* $\sigma d_i = \sigma_2)$ *alors* $\sigma d'_i = \sigma_3$, *sinon* $\sigma d'_i = \sigma d_i$.

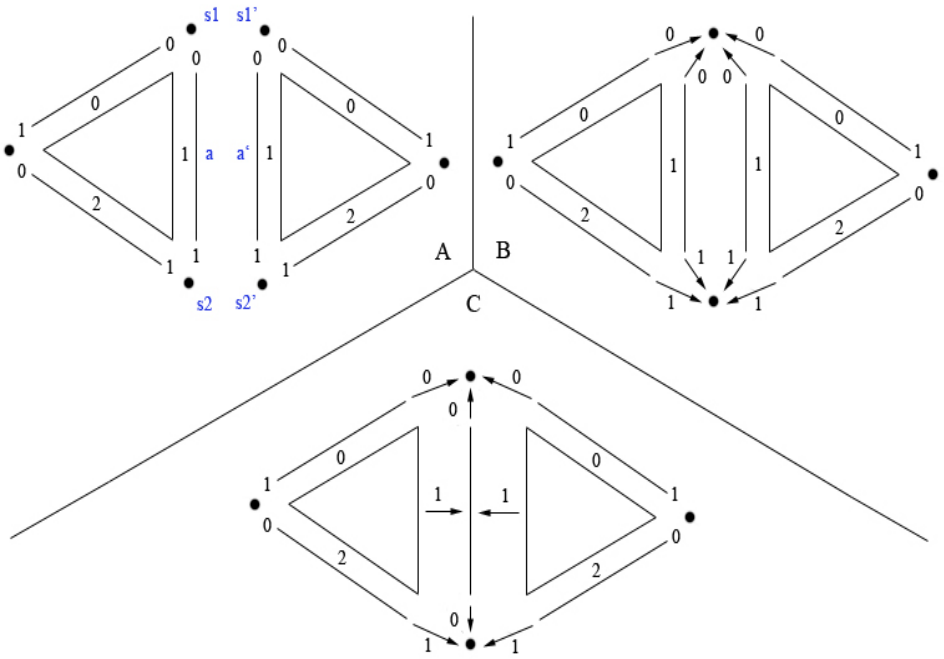

FIG. 8 – Identification de deux simplexes.

Dans la figure 8, nous souhaitons identifier les deux 1-simplexes (deux arêtes) a et a' (figure 8A). La première étape consiste à identifier le bord de a au bord de a'. Pour cela, on identifie s_1 à s'_1 et s_2 à s'_2 (figure 8B). Il suffit ensuite d'identifier a à a' selon les étapes données dans la définition 5.

3 Générer une structure de données

Le module de génération comprend deux grandes parties dont la première est consacrée à la génération de la structure de données. Cette structure est utilisée dans le noyau de modeleur afin de stocker les objets construits suivant le modèle topologique donné en entrée du Méta-Modeleur. Le travail effectué sur la génération de cette structure a joué un rôle clé dans le développement du générateur de noyaux de modeleurs. En effet, dans cette section, nous proposons une structure de données générique unique dont seule l'instanciation dépend du modèle d'entrée. La section 4 sera consacrée aux opérations de construction et/ou modification des objets.

3.1 Le choix d'une structure

3.1.1 Vers une structure générique

Le problème posé est le suivant : à partir d'un modèle d'entrée possédant *a priori* une architecture propre, nous devons générer une structure de donnée adaptée. Cependant, à la lecture d'une définition mathématique de modèle telles que celles données en exemple dans la section 2, le choix de la structure la mieux adaptée n'est pas immédiat, qui plus est si ce choix doit être effectué de manière automatique par un générateur de code. En effet, choisir la meilleure structure de données est dans la plupart des cas un problème indécidable, puisque cela suppose par exemple de savoir comparer des structures en termes de taille de données et d'efficacité. Or, ces deux critères dépendent essentiellement de l'utilisation faite de la structure, c'est-à-dire des spécificités éventuelles de objets manipulés, mais surtout de la nature et de la fréquence des opérations réalisées.

Ainsi, plutôt que de générer une structure de données potentiellement différente pour chaque modèle topologique, et afin de faciliter la génération de cette structure de données, nous proposons une structure utilisable par tous les modèles topologiques, mais qui sera néanmoins spécialisée pour chacun au moment de son instanciation.

Dans [Lie91], l'auteur compare les modèles topologiques connus et en conclut que les modèles topologiques ordonnés[8] (les cartes généralisées en sont un exemple) sont équivalents en ce sens qu'on peut facilement passer d'un modèle à l'autre lorsqu'ils permettent de représenter la même classe

[8]On appelle modèle ordonné tout modèle utilisant un seul type d'élément de base sur lequel agit des fonctions élémentaires. Les cellules, bords et composantes connexes y sont définis implicitement.

d'objets. Dans le cadre du Méta-Modeleur, lorsque nous parlons de structure générique, nous n'entendons pas utiliser les conclusions de [Lie91] afin de ramener le modèle d'entrée à un modèle connu. Nous souhaitons instancier notre structure générique directement à partir du modèle topologique donné en entrée du Méta-Modeleur, indépendamment d'un éventuel modèle de référence.

Si l'on considère l'ensemble des modèles topologiques connus, on note des similitudes dans leur architecture. En effet, quel que soit le modèle étudié, nous avons toujours des éléments de base (éventuellement étiquetés par une dimension) ainsi que des applications permettant de passer d'un élément à un autre (nous parlons de liens entre ces éléments). De plus, des propriétés sur ces liens expriment les contraintes de cohérence du modèle topologique considéré.

Néanmoins, il convient de préciser que la nature des éléments de base (brins, simplexes, etc.), la nature des liens (involutions, opérateurs de bord, etc.), ainsi que le nombre de liens attachés à un élément de base varie sensiblement d'un modèle topologique à un autre. Les contraintes de cohérence, pour leur part, n'ont pas d'impact sur la structure de donnée. Elles interviennent au niveau des opérations et sont donc traitées dans la section suivante.

À partir des observations que nous venons d'effectuer, nous pouvons définir notre structure de données topologique générique. Elle est constituée d'un ensemble d'éléments de base où chaque élément de base est la donnée d'une dimension et d'un ensemble de liens indexés vers ses voisins (ce nombre de liens peut varier en fonction de la dimension de l'élément, mais est constant pour un élément de base donné). La définition 6 décrit mathématiquement cette structure générique abstraite.

Définition 6 (Structure topologique générique)**.** *La structure topologique générique de dimension n est un couple $T = (E, L)$, où :*

- $E = \bigcup_{i=0}^{n} E^i$*, avec E^i un ensemble fini d'éléments de base de dimension i ;*
- $L = \bigcup_{i=0}^{n} L^i$*, où L^i est l'ensemble des liens des éléments de E^i, c'est-à-dire tel que tout lien l de L^i est une application de E^i vers E^j ($0 \leq j \leq n$).*

3.1.2 Validation de la structure générique

Avant de proposer une implantation de notre structure générique, nous devons vérifier qu'elle fonctionne pour les modèles topologiques donnés en exemple dans la partie 2.

Dans un premier temps, nous souhaitons utiliser notre structure générique avec les cartes généralisées (voir partie 2.1). Dans le cas d'une carte généralisée de dimension n, nous instancions notre structure topologique générique de la manière suivante :

- $E = E^0$, est l'ensemble des brins de la carte, notons que dans une carte généralisée les brins ont tous la même dimension (0 par défaut) ;
- $L = L^0$, est l'ensemble de liens correspondants aux involutions.

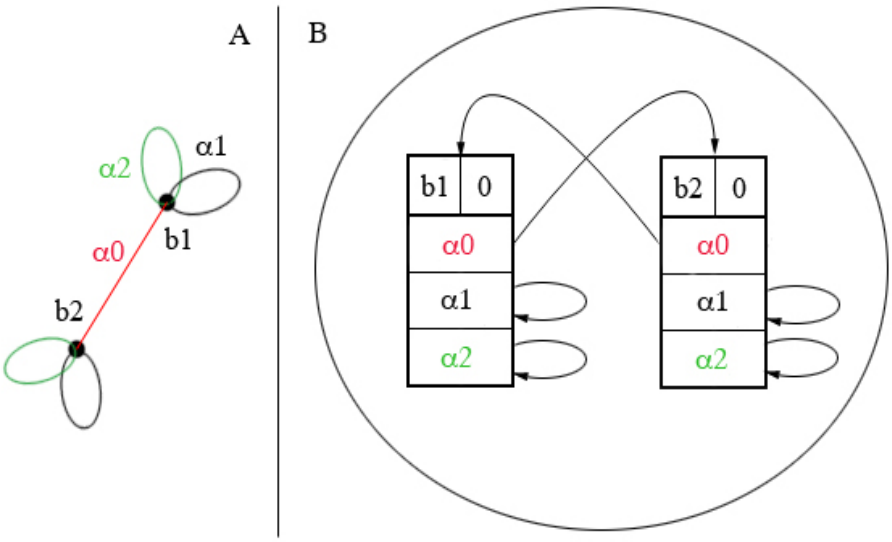

FIG. 9 – Application de la structure générique aux 2-G-cartes.

La figure 9A donne un exemple de 2-G-carte dans lequel deux brins b_1 et b_2 sont liés par α_0 pour former une arête. Sur la figure 9B, nous avons deux brins b_1 et b_2 auxquels on attache l'étiquette (la dimension) par défaut : « 0 », et l'indexage des liens par α_0, α_1 et α_0. Les liens eux-mêmes sont représentés par les flèches sortant de chacun des index. Pour faciliter la compréhension du schéma, le nom des éléments de base apparaît bien qu'il ne figure pas réellement dans la structure abstraite. Dans une carte généralisée, si deux

brins b_1 et b_2 sont liés par une involution α_i alors $b_1\alpha_i = b_2$ et $b_2\alpha_i = b_1$. Ainsi, si $b_1 \neq b_2$, une involution est représentée par deux liens dans la structure de donnée. Lorsque $b_1 = b_2$ (dans l'exemple, c'est le cas pour α_1 et α_2), une involution est représentée par une boucle. Il est donc facile de stocker une carte généralisée à l'aide de notre structure générique.

Si nous traitons les ensembles semi-simpliciaux de dimension n, nous instancions notre structure topologique générique de la manière suivante :

- $E = \bigcup_{i=0}^{n} E^i$, avec E^i l'ensemble des simplexes de dimension n ;
- $L = \bigcup_{i=0}^{n} L^i$, où L^i est l'ensemble des opérateurs de bords liants les i-simplexes de E^i au $(i-1)$-simplexes de E^{i-1}.

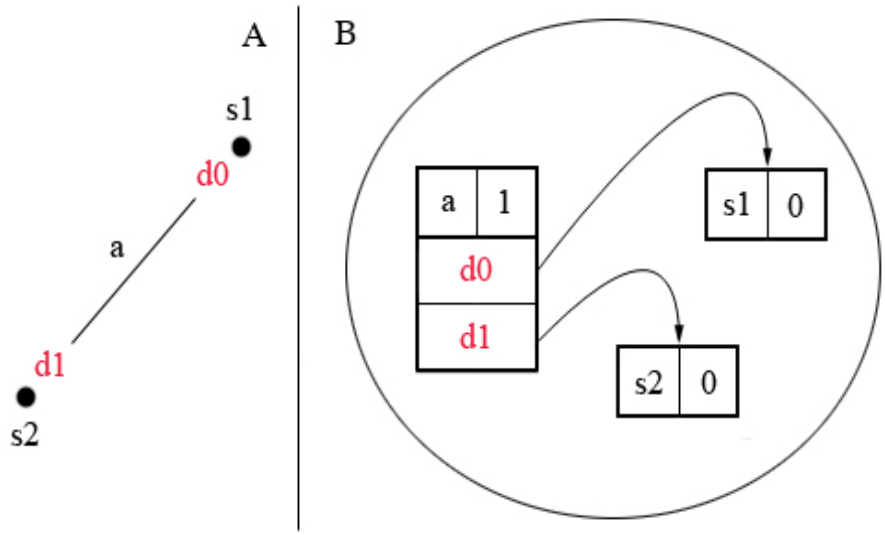

FIG. 10 – Application de la structure générique aux ensembles semi-simpliciaux de dimension 2.

Sur la figure 10, nous appliquons la structure générique aux ensembles semi-simpliciaux (voir partie 2.2). Cette figure utilise les mêmes conventions graphiques que précédemment. La figure 10A contient deux 0-simplexes s_1 et s_2 ainsi qu'un 1-simplexe a. L'arête a est liée aux sommets s_1 et s_2 par deux opérateurs de bords d_0 et d_1. Contrairement aux G-cartes, les éléments de

base des ensembles semi-simplicaux peuvent être de plusieures dimensions (0 ou 1 dans notre cas). Les 0-simplexes n'ont pas d'opérateurs de bord, c'est pourquoi dans la figure 10B, les représentations s_1 et s_2 ne contiennent aucun lien.

Enfin, à l'inverse des involutions qui par leur nature nécessitent d'être représentées par deux arcs, nous associons à chaque opérateur de bord un et un seul lien dans la structure de donnée. Ainsi sur la figure 10A $ad_0 = s_1$, ce qui se traduit en 10B par une unique flèche liant l'index d_0 de a à s_1.

Nous venons de montrer que dans le cas des cartes généralisées et des ensembles semi-simpliciaux, la structure de donnée générique répond bien aux besoins. Or, ces deux modèles sont bien représentatifs des modèles traités.

3.1.3 Implantation de la structure générique

Notre structure générique est en fait un graphe orienté particulier dans lequel les arcs sont indexés. En effet, si nous implantons les éléments de base par les sommets du graphe et les liens entre les éléments de base par les arcs, pour chaque sommet du graphe et chaque étiquette d'arc (où une étiquette est un nom de lien, par exemple « α_0 », « d_0 », etc.) il existe un unique arc dans le graphe. Cette condition permet une implantation plus optimisée des graphes dans laquelle l'accès aux voisin des sommets par un lien donné s'effectue en temps constant. Tandis que dans une implantation de graphes généraux, par exemple par listes d'adjacences, le temps d'accès aux voisins est linéaire en le nombre de brins.

Pour faciliter la réalisation d'une première version du Méta-Modeleur au cours du stage, nous avons choisi d'utiliser une bibliothèque de graphes généraux existante : OCamlGraph [CFS05], que nous présentons dans la partie suivante où nous introduisons par ailleurs OCaml [Inr85], qui est le langage de programmation utilisé à la fois dans le code du Méta-Modeleur et dans le code généré. Comme nous le voyons dans la partie 3.3, cette bibliothèque affiche déjà de très bonnes performances et peut de plus être spécialisée conformément à la remarque précédente.

3.2 OCaml et OCamlGraph : brève introduction

3.2.1 Un langage fonctionnel

OCaml est un langage de programmation fonctionnelle. À ce titre, il s'agit d'un langage dans lequel les fonctions sont des données comme les autres. En particulier, il est facile de passer des fonctions en argument ou même

en sortie d'autres fonctions. Cette grande simplicité dans la manipulation des fonctions, alliée au polymorphisme, permet d'écrire facilement des programmes généraux, à partir d'un code générique. Le langage OCaml peut être interprété ou compilé. L'interpréteur est un outil intéressant puisqu'il permet à l'utilisateur d'avoir une trace de l'exécution de son programme sans se soucier de la gestion des entrées/sorties.

```
1 : let f1 = fun x g -> (g x) + 1 ;;
2 : val f1 : 'a -> ('a -> int) -> int = <fun>
3 :
4 : let f2 = fun () -> (fun x -> x) ;;
5 : val f2 : unit -> 'a -> 'a = <fun>
6 :
7 : let f3 = fun g  -> (fun x -> g(x) + 1) ;;
8 : val f3 : ('a -> int) -> 'a -> int = <fun>
```

Code 1 – Exemples de fonctions OCaml.

Dans le code source 1 les lignes 1, 4 et 7 sont des déclarations de fonctions, elles sont introduites avec le mot clé *fun*. Les lignes 2, 5 et 8, retournées par l'interpréteur OCaml, donnent le type des fonctions le plus général en utilisant la variable de type *'a*.

La fonction *f1* (ligne 1) prend deux valeurs en paramètres : une fonction g et une donnée x d'un type quelconque. Elle renvoie un entier. Cette ligne de code nous permet d'aborder un autre point important d'OCaml : la synthèse de type. Aucune déclaration n'indique explicitement que *f1* retourne un entier. Le type de retour est synthétisé par l'interpréteur, lors de l'analyse de l'expression *g(x) + 1*. C'est du type de l'opérateur *+* (*int -¿ int -¿ int*), qu'est déduit le type de retour de la fonction *f1*. De même, le type de g (fonction d'une donnée quelconque vers les entiers) est déduit du *+*. De plus, remarquons que l'instruction *(g x)* est l'application de la fonction g à la valeur x. Cette notation est typique des langages fonctionnels.

En informatique, en raison des effets de bords, les fonctions peuvent ne pas avoir d'argument, c'est le cas de la fonction *f2* (ligne 4), ou ne pas avoir de valeur de retour (comme les fonctions d'affichage). On utilise alors le type *unit*, dont l'unique valeur est *()*. Le type de retour de *f2* est une fonction, en l'occurence : la fonction identité.

Enfin, *f3* (ligne 7) est un exemple de fonction qui à la fois renvoie une fonction et prend une fonction en paramètre.

3.2.2 Les modules OCaml

En plus de son aspect fonctionnel, OCaml propose une surcouche efficace (en fait un langage au-dessus du langage) qui facilite l'écriture de modules. La puissance de ce sur-langage est due à l'utilisation de foncteurs permettant d'écrire des modules paramétrés par d'autres modules. Un foncteur est une application de l'ensemble des modules dans l'ensemble des modules.

```
1 : module SortList.Make :
2 :     functor (ord : OrderedType) ->
3 :         sig
4 :             type elt = Ord.t
5 :             type t
6 :             val empty : t
7 :             val add : elt -> t -> t
8 :             ...
9 :         end ;;
```

Code 2 – Implantation d'un module de listes ordonnées.

Prenons l'exemple de l'implantation d'un module de listes ordonnées. Dans le code source 2, nous donnons la signature d'un foncteur *SortList.make*, de création d'un module de listes. Dans le cadre de cette brève introduction à OCaml, nous nous contentons de la signature du foncteur, sans détailler le corps de son implantation. *SortList.make* prend en paramètre le module des éléments de la liste. Ce module *ord* (ligne 2) de signature *OrderedType* (disponible dans la librairie standard d'OCaml), doit proposer un type pour les éléments de base ainsi qu'une fonction de comparaison entre ces éléments. Le foncteur *SortList.Make* retourne un module dont la signature est celle d'une liste, *t* étant le type de la liste (ligne 5). Ce nouveau module propose entre autres une constante liste vide (*empty*, ligne 6), une fonction d'ajout d'un élément (*add*, liste 7), etc. À la ligne 4, nous renommons le type des éléments *Ord.t* de la liste en *elt* afin de rendre le code plus lisible.

La fonction *add* donne un exemple d'utilisation d'une structure de donnée persistante. En programmation impérative, lorsque l'on modifie un objet (une liste, un tableau, etc.) la modification est effectuée en place, c'est-à-dire directement sur l'objet d'origine. Typiquement, une fonction d'ajout d'un élément à une liste ne retourne rien et la liste d'origine est modifiée par effet de bord. En utilisant une structure de donnée persistante (on parle de programmation fonctionnelle pure), l'ajout d'un élément à une liste nécessite de créer et ren-

voyer une nouvelle liste, contenant les mêmes éléments que la liste d'origine, plus l'élément ajouté. C'est pour cette raison que la valeur de retour de la fonction *add* est du type *t*. Cette reconstruction des données n'est pas handicapante puisqu'elle est gérée efficacement par le compilateur OCaml. De plus elle permet dans bien des cas d'éviter à l'utilisateur de gérer lui-même l'espace mémoire alloué à ses structures.

```
 1 : module MySortList = SortList.Make
 2 :     (struct
 3 :         type t = int * int
 4 :         let compare = Pervasives.compare
 5 :     end) ;;
 6 :
 7 : let liste_vide =
 8 :     MySortList.empty ;;
 9 : val liste_vide : MySortList.t = <abstr>
10 :
11 : let liste_non_vide =
12 :     MySortList.add (0,0) liste_vide ;;
13 : val liste_non_vide : MySortList.t = <abstr>
```

Code 3 – Implantation d'un module de listes ordonnées.

La variable *MySet* du code source 3 reçoit le résultat d'un appel au foncteur *SortList.Make* avec pour paramètre un module d'éléments ordonnés (lignes 2 à 5). Ce module introduit un type *t*, correspondant à un couple d'entiers (ligne 3), puis une fonction *compare* (ligne 4) de comparaison entre ces couples. On remarque que l'appel à un foncteur s'effectue de la même manière qu'un appel de fonction classique. Les lignes 7 à 11 donnent un exemple d'utilisation du module *MySortList*. On y crée une liste vide (lignes 7 et 8) puis ajoute un couple d'entiers à cette liste (lignes 11 et 12).

3.2.3 OCamlGraph

OCamlGraph [CFS05] est une bibliothèque générique de graphes développée en OCaml. Sa conception a été considérablement facilitée par l'utilisation des modules, et notamment des foncteurs. OCamlGraph propose un grand nombre de structures de données différentes pour représenter les graphes : orientés ou non, structures persistantes ou modifiées en place, sommets et arcs avec ou sans étiquettes, marques sur les sommets, etc. De plus, elle implante

de nombreux algorithmes classiques de graphes, écrits indépendamment de la structure de données utilisée.

OCamlgraph étant une bibliothèque en cours de développement, un dialogue avec ses développeurs nous a permis de l'enrichir, puis de valider son utilisation dans le cadre de notre projet de Méta-Modeleur. Ajoutons enfin que si à terme nous souhaitons ajouter une structure de graphes spécialisée dans laquelle les arcs sont indexés (offrant ainsi la possibilité d'accéder en temps constant aux voisins d'un sommet par un lien donné), nous pouvons sans problème conserver l'interface de programmation fournie avec OCamlGraph tout en utilisant notre propre structure de données.

3.3 OCamlGraph et la modélisation géométrique

En tant que bibliothèque générique de graphes, OCamlGraph n'est pas consacrée à un domaine d'utilisation particulier. Or, c'est dans le contexte de la modélisation géométrique à base topologique que nous souhaitons l'utiliser. Nous avons donc besoin d'une implantation efficace en particulier pour ce qui concerne le parcours du graphe et de ses différentes parties. Afin de choisir l'implantation la plus adaptée à nos besoins et de vérifier son efficacité, nous nous sommes donnés deux tests critiques.

Dans un premier temps, nous avons comparé OCamlGraph à un modeleur géométrique. Cette comparaison porte sur la vitesse de parcours des objets. Ensuite, nous avons utilisé OCamlGraph dans le cadre de la programmation d'un noyau de modeleur basé sur des 2-G-cartes, afin de nous assurer qu'elle est bien adaptée à la modélisation géométrique à base topologique.

3.3.1 Temps de parcours des structures

Ce premier test a pour but de vérifier qu'OCamlGraph reste compétitive, lorsqu'il s'agit de traiter des objets (ici des graphes) volumineux. Par graphes volumineux, nous entendons des graphes de plusieurs dizaines de milliers de noeuds. Un tel test est indispensable puisqu'il est fréquent, en modélisation géométrique à base topologique, de traiter de tels objets lorsqu'il s'agit de manipuler des scènes complexes.

Moka [VD03], un modeleur à base de 3-G-cartes écrit en C++, permet de créer puis de travailler sur de telles scènes, tout en proposant des temps de calculs performants. Ainsi, dans ce premier test, nous allons comparer OCamlGraph à Moka.

Le mode opératoire est le suivant : nous créons deux programmes ; dans le premier nous utilisons le noyau de Moka afin de créer une sphère topologique

dont nous connaissons la taille en termes de nombre de brins, puis avec d'OCamlGraph nous générons un graphe dans lequel le nombre de sommets est égal à ce nombre de brins. Enfin, nous parcourons ces deux structures à l'aide d'un parcours de volumes dans le premier cas, et d'un parcours en profondeur dans le second.

Par parcours, nous entendons le fait de passer sur tout les éléments de base (brins ou noeuds) de l'objet considéré, ceci en utilisant les liens proposés dans les deux structures (involutions ou arcs), et en n'effectuant aucun traitement sur les éléments de base. Dans les cartes généralisées de dimension 3, le parcours d'un volume s'effectue en utilisant les liens α_0, α_1 puis α_2. Ainsi, si dans le cadre de notre comparatif nous parcourrons une 3-G-Carte de taille n dans Moka, nous devons parcourir un graphe de n sommets et $3 \times n$ arcs. Afin d'obtenir un résultat significatif, nous effectuons ce test pour $n = 1000$, 10000, 40000, 70000 puis 130000.

Il convient de préciser que dans un parcours en profondeur, le problème de la non-indexation des arcs n'intervient pas puisqu'à aucun moment on ne doit descendre dans un voisin particulier. Les voisins des sommets sont tous traités, et dans un ordre quelconque. Ceci n'amoindrit pas la valeur de notre test puisque cette comparaison porte uniquement sur le comportement d'OCamlGraph face à des données volumineuses.

Dans ce premier test, nous ne comparons pas le parcours d'une même 3-G-Carte, dans deux modeleurs différents. Le graphe créé avec OCamlGraph étant généré aléatoirement, nous n'avons aucun contrôle sur sa forme. Néanmoins cela nous suffit puisque nous ne nous intéressons pas à la forme exacte de la structure, mais à sa taille en mémoire.

Enfin, bien que les deux programmes à comparer soient écrits dans deux langages différents, nous les avons compilés dans les mêmes conditions, en C++ d'un côté et en OCaml de l'autre. À cette fin, nous avons utilisé des options de compilations similaires dans les deux langages et avons effectué les tests sur la même machine avec à chaque fois, les mêmes ressources disponibles.

Le résultat de ce comparatif est donné en figure 11. L'écart entre la courbe d'OCamlGraph et celle de Moka est presque nul. De plus, les deux courbes suivent la même progression linéaire, en fonction du nombre n d'éléments de base. Remarquons qu'il existe tout de même un écart significatif autour de $n = 70000$, pour le moment nous ne savons pas expliquer cet écart.

Malgré cette exception, les résultats de ce test montrent qu'OCamlGraph se comporte bien lorsqu'elle est confrontée à de gros objets.

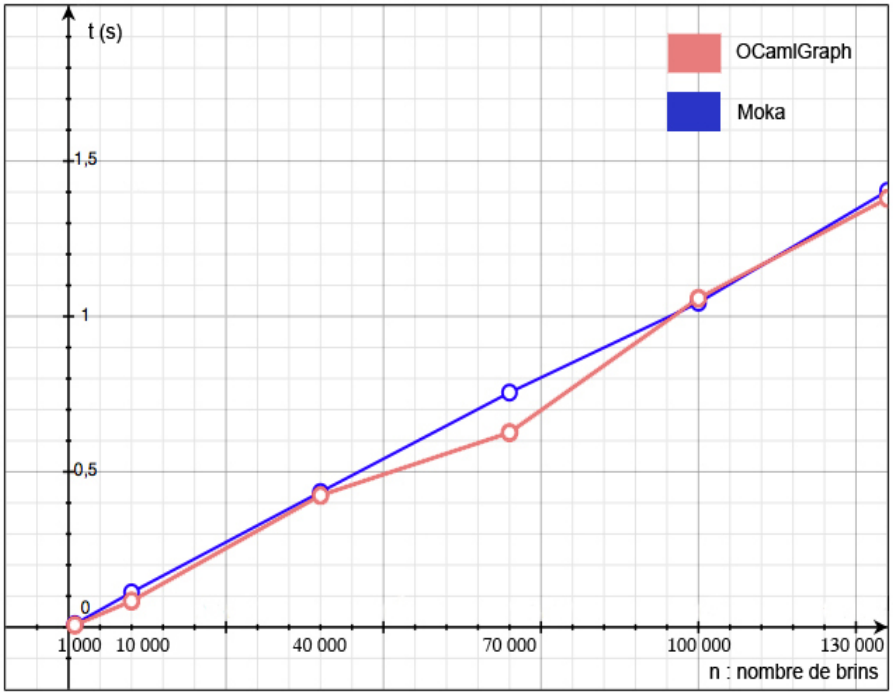

Fig. 11 – Comparaison OCamlGraph/Moka.

3.3.2 Programmation d'un noyau de modeleur

Nous venons de montrer qu'OCamlGraph permet de traiter de gros objets, nous devons maintenant vérifier qu'elle est bien adaptée au cas particulier de la modélisation géométrique à base topologique. Ainsi, en guise de second test, nous allons programmer un noyau de modeleur à l'aide d'OCamlGraph. Nous choisissons un noyau à base de 2-G-cartes. De ce fait il est facile de créer une petite interface graphique 2D sur laquelle l'utilisateur peut interagir. L'écriture de ce noyau de modeleur constitue de plus une base idéale à la génération de code puisqu'il nous permet d'identifier les informations dont nous avons besoin au moment de la génération. Enfin, le fait de créer une interface graphique introduit la manière dont nous allons traiter le plongement des objets dans l'espace, et nous permet de mieux appréhender la manière dont le noyau généré pourra à terme être intégré à une interface existante.

Dans le noyau de modeleur que nous proposons, quatre opérations de base sont implantées :

- l'ajout d'un brin isolé ;
- la suppression d'un brin isolé ;
- la couture par α_0, α_1 ou α_2 ;
- la décousure par α_0, α_1 ou α_2.

Structure de donnée

Dans ce paragraphe, nous implantons les cartes généralisées de dimension 2 par l'une des structures de données proposées dans la bibliothèque OCamlGraph. En section 2, nous avons vu qu'aux sommets du graphe nous faisons correspondre les éléments de base du modèle (ici les brins), puis aux arcs les liens. L'indexation des arcs, inexistante dans les graphes généraux, est donnée par les étiquettes portées par ces arcs. Cette manière de représenter un modèle topologique par un graphe nous guide vers une structure particulière parmi celles implantées dans OCamlGraph : la structure de graphes orientés étiquetés. Selon les développeurs d'OCamlGraph, la version impérative abstraite de cette structure est la plus performante, c'est donc celle que nous utilisons. Ainsi, à la ligne 16 du code source 4, nous appelons le foncteur *Graph.Imperative.Digraph.AbstractLabeled* correspondant exactement à notre choix d'implantation.

Deux modules sont déclarés en en-tête du code source 4. Le premier est un module très simple de points 2D (lignes 1 à 3). Il est passé en paramètre du

```
 1 : module Point2D = struct
 2 :     type t = int * int
 3 : end ;;
 4 : open Point2D ;;
 5 :
 6 : module Label = struct
 7 :     type t = ALPHA_0 | ALPHA_1 | ALPHA_2
 8 :     let compare = compare
 9 :     let hash = Hashtbl.hash
10 :     let equal = (=)
11 :     let default = ALPHA_0
12 : end ;;
13 : open Label ;;
14 :
15 : module Gmap2D =
16 :     (Graph.Imperative.Digraph.AbstractLabeled
17 :         (Point2D))
18 :             (Label) ;;
19 : open Gmap2D ;;
```

Code 4 – Création de la structure de graphe.

foncteur *Graph.Imperative.Digraph.AbstractLabeled* et est utilisé pour l'affichage, nous y revenons dans un paragraphe consacré à l'interface graphique de notre modeleur.

Label (lignes 6 à 12) est le module des étiquettes attachées aux arcs. Ces étiquettes, utilisées comme système d'indexation des liens, sont de type t (ligne 7, notons qu'à l'extérieur du module *Label*, le type t est appelé *Label.t*). Dans le cas des 2-G-cartes, il comprend trois constructeurs constants *ALPHA_0 ALPHA_1* et *ALPHA_2*, qui correspondent chacun à un index de lien. En tant que paramètre d'un foncteur, *Label* doit avoir une certaine signature, choisie par les développeurs d'OCamlGraph. Pour cette raison, nous associons à *Label* quatre fonctions : *compare* (ligne 8) permet de définir un ordre sur les étiquettes ; *hash* est une fonction de hachage ; *equal* est la fonction d'égalité entre deux étiquettes ; *default* nous donne la valeur par défaut d'une étiquette. En OCaml, les foncteurs peuvent renvoyer un module (voir section 3.2.2) ou bien renvoyer un nouveau foncteur, c'est le cas de *Graph.Imperative.Digraph.AbstractLabeled*. *Label* est passé en paramètre de ce nouveau foncteur dont le résultat est le module *Gmap2D* (ligne 15) correspondant à la structure de graphe que nous avons choisi.

Opérations

Nous venons de créer la structure de données utilisée pour stocker une 2-G-carte. Intéressons nous alors brièvement au code des opérations de construction et modification des objets. Dans la section 3, nous analysons ces opérations en dehors du contexte de la programmation de notre modeleur-test. Nous allons dans un premier temps détailler la fonction d'ajout d'un brin isolé, puis nous détaillons la fonction de couture de deux brins.

```
 1 : let add_dart m label =
 2 :     let dart = V.create label in
 3 :     let a_0 = E.create dart ALPHA_0 dart in
 4 :     let a_1 = E.create dart ALPHA_1 dart in
 5 :     let a_2 = E.create dart ALPHA_2 dart in
 6 :     add_vertex m dart ;
 7 :     add_edge_e m a_0 ;
 8 :     add_edge_e m a_1 ;
 9 :     add_edge_e m a_2
10 : ;;
```

Code 5 – Ajout d'un brin isolé.

En toute généralité, il n'existe aucune contrainte sur le nombre d'arcs que nous attachons aux sommets d'un graphe. Or dans une 2-G-carte, nous associons exactement trois involutions à chaque brin. C'est dans les opérations que nous devons nous assurer que les nombres et types des liens sont corrects.

Ainsi lors de l'ajout d'un brin isolé (code source 5) nous ajoutons au graphe un sommet *dart* (ligne 6) auquel nous attachons 3 arêtes *a_0*, *a_1* et *a_2* (ligne 7 à 9). Les déclarations de ces quatre nouveaux objets sont effectuées dans les lignes 2 à 5. Afin de faciliter la lecture du code, nous passons par des variables plutôt que d'utiliser directement la valeur de retour des fonctions *V.create* et *E.create* utilisées respectivement pour la création d'un nouveau sommet et d'un nouvel arc. Dans une 2-G-carte, lors de l'ajout d'un brin isolé b, il est lié à lui-même par involution : $b\alpha_0 = b$, $b\alpha_1 = b$ et $b\alpha_2 = b$. Ainsi, lorsque nous créons les trois arcs implantants les involutions, nous les étiquetons avec les indexes *ALPHA_0*, *ALPHA_1* et *ALPHA_1*, puis nous les faisons boucler sur le nouveau sommet *dart*.

En procédant ainsi, nous sommes assurés que les brins isolés ajoutés à la carte sont bien construits. Par nature le Méta-Modeleur doit générer le code des opérations, nous avons donc un contrôle total sur ce code. Il devient alors facile de s'assurer, comme c'est le cas ici, que les éléments de base construits sont conformes à leur définition.

Le code source 6 détail la fonction *sew_rec*, utilisée pour coudre deux brins dans une 2-G-carte. On suppose qu'au moment de l'appel les deux orbites à coudre sont isomorphes, ce qui nous assure que la couture est bien possible (notons qu'une manière de vérifier que la couture est possible est d'essayer de coudre). Les arguments de *sew_rec* sont les suivants :

- *map*, la carte dans laquelle on veut coudre ;
- *alpha*, l'involution par laquelle on veux coudre ;
- *invol_list*, les involutions utilisées lors des parcours d'orbites ;
- *d_1* et *d_2*, les deux brins à coudre.

Nous ne commentons pas en détail la fonction *sew_rec*, qui ne comporte que peu d'intérêts autres que techniques. Néanmoins, il est important de préciser que l'on distingue deux parties. Une partie locale (lignes 4 à 11) dans laquelle nous lions les deux brins courants par l'involution *alpha*, et une partie propagation (ligne 13 à 27). Dans la section 2, nous avons donné un exemple de couture par α_2 dans laquelle l'ajout seul d'un lien α_2 entre les deux brins à coudre ne suffit pas. Pour respecter les contraintes de cohérence du modèle, cette couture doit être propagée le long de deux orbites. Ainsi, le rôle de cette propagation est de rétablir (de propager) les contraintes.

```
1 : let rec sew_rec map alpha invol_list d_1 d_2 =
2 :    Mark.set d_1 1 ; Mark.set d_2 1 ;
3 :
4 :    let suiv_1 = succ_e map d_1 in
5 :    let suiv_2 = succ_e map d_2 in
6 :    let new_edge_1 = E.create d_1 alpha d_2 in
7 :    let new_edge_2 = E.create d_2 alpha d_1 in
8 :    remove_edge_e map (get_invol suiv_1 alpha) ;
9 :    remove_edge_e map (get_invol suiv_2 alpha) ;
10 :   add_edge_e map new_edge_1 ;
11 :   add_edge_e map new_edge_2 ;
12 :
13 :   let rec sub_sew_rec l = match l with
14 :        [] -> ()
15 :      | invol::tail ->
16 :         let d_1_s = E.dst
17 :            (get_invol suiv_1 invol) in
18 :         if ((Mark.get d_1_s)=0) then (
19 :            let d_2_s = E.dst
20 :               (get_invol suiv_2 invol) in
21 :            sew_rec
22 :               map alpha invol_list d_1_s d_2_s
23 :         ) else
24 :            sub_sew_rec tail
25 :   in
26 :
27 :   sub_sew_rec invol_list
28 : ;;
```

Code 6 – Couture de 2 brins.

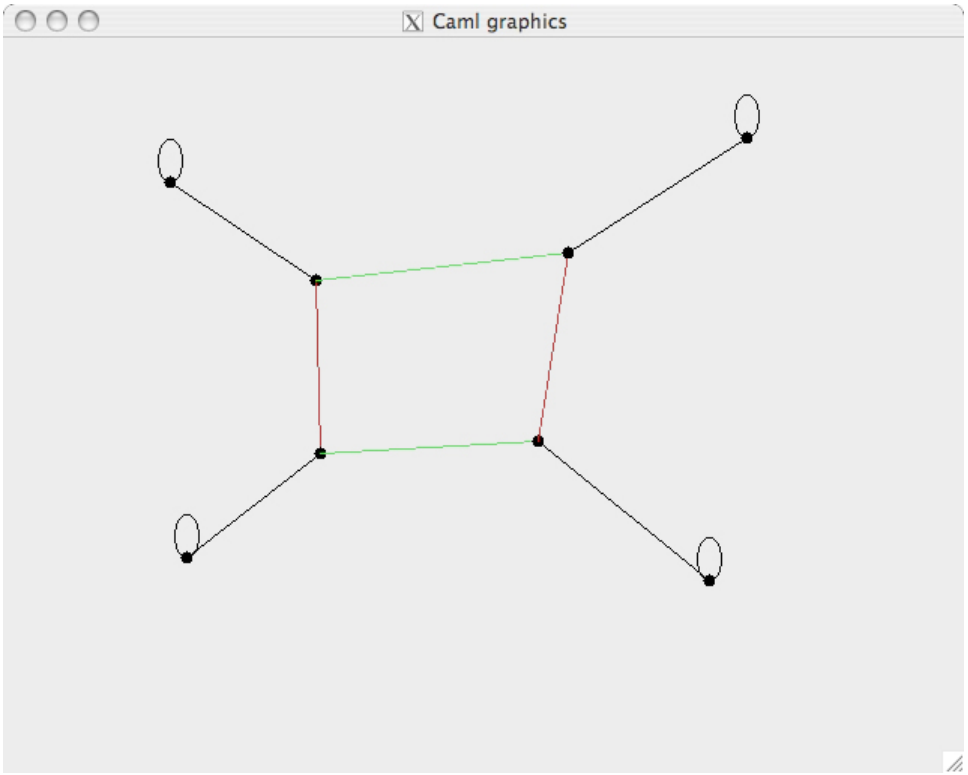

FIG. 12 – Interface du modeleur test.

Interface

Afin de rendre notre modeleur de test plus attractif, nous avons conçu une petite interface graphique sur laquelle l'utilisateur peut interagir. Par exemple pour coudre deux brins l'utilisateur appuie sur la touche « s », clique sur les deux brins à coudre, puis entre l'indice de l'involution par laquelle il veut coudre. Une copie d'écran de notre interface est montrée en figure 12.

Le développement de cette interface nous a permis de réfléchir à la manière dont le Méta-Modeleur peux générer le plongement des objets. Sur la figure 12, les brins sont représentés par des points et les involutions par des droites reliant deux points. Une boucle est attachée à un point lorsque le brin correspondant est à lié à lui même par l'une des trois involutions α_0, α_1 ou α_2. L'unique donnée utilisée lors de cet affichage est un point dans l'espace que nous associons à chaque brin. Dans OCamlGraph, il est possible d'étiqueter les sommets du graphe par une donnée d'un type choisi par l'utilisateur. Ainsi nous associons notre donnée de plongement à chaque sommet, et donc

à chaque brin. Le type couple d'entiers est décrit dans le module *Point2D* du code source 4. Le module des étiquettes de sommet doit être passé en paramètre du foncteur de création du graphe, ainsi à la ligne 17 *Point2D* est passé à *Graph.Imperative.Digraph.AbstractLabeled*.

Dans notre structure générique, les éléments de base du modèle sont étiquetés par leur dimension. Cependant nous ne la faisons pas figurer dans le cadre de notre modeleur-test puisque les brins d'une carte généralisée ont tous la même dimension. Lors de la génération de code, la dimension des éléments de base doit être donnée en plus du plongement, dans l'étiquette des sommets.

Ce second test, dans lequel nous avons utilisé l'implantation de notre structure générique dans un cas concret de modélisation géométrique à base topologique, est tout à fait positif. En effet, cette implantation à l'aide d'OCamlGraph est bien adaptée à notre projet puisqu'elle nous a permis de programmer en un temps très court (une quinzaine d'heures) un véritable noyau de modeleur auquel nous avons associé une petite interface utilisateur.

3.4 La génération du code de la structure de données

À l'issue de cette partie du rapport, la génération d'une structure de données adaptée au modèle d'entrée devient immédiate. En effet, il s'agit désormais de spécialiser notre structure générique. En termes d'implantation, ceci revient à ajouter des contraintes (au sens de contraintes structurelles) à l'une des structures de graphe proposées par OCamlGraph. Suite au travail effectué sur notre modeleur-test, nous avons identifié précisément les informations nécessaires à la création de ce graphe. Ainsi, la grammaire de description du modèle d'entrée doit contenir les informations suivantes :

- le nom des éléments de base ainsi que leur dimension ;
- le nom et la nature des liens ;
- une information de plongement associée à chaque élément de base.

Remarquons que le nom des éléments de base est utilisé uniquement pour rendre plus explicite le code généré par le Méta-Modeleur. Le nom des liens sert en revanche de nom pour les constructeurs constants du type utilisé lors l'indexation des arcs (ligne 7 du code source 4). En OCaml, le nom des constructeurs doit commencer par une majuscule, lors de l'écriture du code, on préfixe par une majuscule les noms donnés par l'utilisateur.

Ainsi, dans le cas des cartes généralisées de dimension 2, l'arbre de syntaxe abstraite de la description du modèle (l'arbre généré par l'analyse syntaxique) contient :

- « brin » 0 « point2d »,
- « alpha_0 » INVOLUTION « brin »
- « alpha_1 » INVOLUTION « brin »
- « alpha_2 » INVOLUTION « brin »

Ceci signifie que l'élément de base des 2-G-cartes est le « brin », qu'il est de dimension 0, et que pour l'afficher nous avons besoin d'un « point2d ». De plus, à chaque « brin » sont associées trois involutions « alpha_0 », « alpha_1 » et « alpha_2 ».

Si l'on souhaite que le langage d'entrée permette une description succincte des modèles, il doit comporter des briques de base. Les involutions font partie de ces mots-clés que le langage se doit de proposer. Ceci est justifié par le fait que l'on retrouve souvent les involutions dans la définition d'un modèle topologique. En procédant ainsi, l'utilisateur n'a pas besoin de définir ce qu'est une involution. En particulier le Méta-Modeleur sait que si un brin b est lié à un brin b' par involution, il existe deux liens dans la structure de données, un lien de b vers b' puis un lien de b' vers b.

Si l'entrée du Méta-Modeleur décrit les ensembles semi-simpliciaux de dimension n, l'arbre de syntaxe abstraite contient :

- « simplex_0 » 0 « point2d »
- « simplex_1 » 1 « couple_de_point2d »
- « simplex_2 » 2 « triplet_de_point2d »
- « d_0_1 » APPLICATION « simplex_1 » « simplex_0 »
- « d_1_1 » APPLICATION « simplex_1 » « simplex_0 »
- « d_0_2 » APPLICATION « simplex_2 » « simplex_1 »
- « d_1_2 » APPLICATION « simplex_2 » « simplex_1 »
- « d_2_2 » APPLICATION « simplex_2 » « simplex_1 »

Les ensembles semi-simpliciaux de dimension 2 possèdent trois éléments de base nommés « simplex_0 », « simplex_1 » et « simplex_2 », de dimensions respectives 0, 1 et 2. Pour plonger un « simplex_0 » (un sommet) nous avons besoin d'un « point2d », pour plonger un « simplex_1 » d'un

« couple_de_point2d » et enfin pour plonger un « simplex_2 » d'un « triplet_de_point2d ». À chaque « simplexe_1 » sont associées deux applications « d_0_1 » et « d_1_1 » de l'ensemble des « simplex_1 » vers l'ensemble des « simplex_0 ». Et enfin, à chaque « simplexe_2 » sont associées trois applications « d_0_2 », « d_1_2 » et « d_2_2 » de l'ensemble des « simplex_2 » vers l'ensemble des « simplex_1 ». Ici encore, les applications doivent faire partie des briques de base du Méta-Modeleur.

Notons que dans cet exemple, les opérateurs de bords sont doublement indicés (le second indice étant la dimension du simplexe auquel ils sont rattachés). Ceci facilite la lecture du code, mais reste facultatif. En particulier dans la définition des ensembles semi-simpliciaux (définition 4), les opérateurs de bords n'ont qu'un seul indice.

Dans ce paragraphe, nous n'étudions pas les contraintes de cohérence du modèle. Ces contraintes interviennent au niveau des opérations de construction ; c'est donc dans la partie suivante du rapport, consacrée à la génération du code des opérations, que nous les traitons.

Nous avons testé la génération de la structure de données à partir des arbres syntaxiques dont nous venons de détailler le contenu. Par exemple, dans les cas des 2-G-cartes on génère le même code que celui présenté dans le code source 4 (page 29). Néanmoins, le type *Point2D* n'es pas généré puisque dans un premier temps nous ne souhaitons pas traiter l'affichage dans le cadre de notre Méta-Modeleur. Le générateur se contente de passer *Point2D* en paramètre du foncteur *Graph.Imperative.Digraph.AbstractLabeled*, mais l'utilisateur doit implanter lui-même ce module lié au plongement.

4 Générer les opérations

Dans la section 3, nous avons répondu au problème de la génération par le Méta-Modeleur d'une structure de données pour les objets topologiques. Dans cette section, nous nous intéressons à la génération du code des opérations de construction et de modification des objets. Nous venons de voir que les contraintes de cohérence des modèles interviennent au niveau des opérations. Par exemple, lors de la couture par α_2 de deux brins d'une 2-G-carte, il est nécessaire d'effectuer deux parcours d'orbites en ajoutant un lien α_2 à chaque étape de ces parcours. Ceci permet de respecter la contrainte des 2-G-cartes, qui précise que $\alpha_0\alpha_2$ est une involution (voir partie 2.1).

Nous n'avons pas de garantie qu'à partir de ces contraintes telles qu'elles sont écrites dans les définitions de la partie 2, il est possible de déduire des algorithmes complets. Cette traduction de formules mathématiques en code opérationnel est un problème difficile auquel nous ne pouvons apporter de solution dans le cadre d'un stage. Néanmoins, nous allons proposer une manière de générer le code des opérations à partir d'une formulation plus constructive de ces contraintes.

Afin d'introduire notre solution, nous analysons quatre opérations de base : l'ajout et la suppression d'un élément de base ou d'un lien.

4.1 L'ajout et la suppression d'un élément isolé

Dans les modèles topologiques, nous distinguons deux types de liens : les opérateurs d'adjacence et les opérateurs d'incidence. On appelle opérateur d'adjacence un opérateur tel qu'il permet de lier deux éléments de base de même dimension : c'est le cas des involutions dans les cartes généralisées. Un opérateur d'incidence lie deux éléments de base de dimensions distinctes, les opérateurs de bords utilisés dans les ensembles semi-simpliciaux appartiennent à cette seconde catégorie. Nous traitons différemment l'ajout d'un élément isolé selon qu'il est lié à ces voisins par un opérateur d'adjacence, ou par un opérateur d'incidence.

4.1.1 Premier cas : opérateurs d'adjacence

Dans les cartes généralisées nous appelons brins isolé un brin qui n'est lié à aucun autre brin, lui mis à part. Ainsi, ajouter un brin isolé à une carte généralisée de dimension n revient à ajouter un brin à l'ensemble des éléments de base, puis à lier ce brin à lui-même par les involutions. La définition de cette opération est donnée en définition 7. Un exemple est donné en figure 13.

Définition 7 (Ajout d'un brin isolé dans une carte généralisée de dimension n)**.** *Soient $G = (B, \alpha_0, ..., \alpha_n)$ une n-G-carte, la carte $G' = (B', \alpha'_0, ..., \alpha'_n)$ résultant de l'ajout d'un brin isolé b' à G est défini par :*

- $B' = B \cup \{b'\}$;
- *pour tout brin $b \in B$ tel que $b \neq b'$, on a $b\alpha'_i = b\alpha_i$ pour $i = 0, ..., n$;*
- $b'\alpha'_i = b'$ *pour* $i = 0, ..., n$.

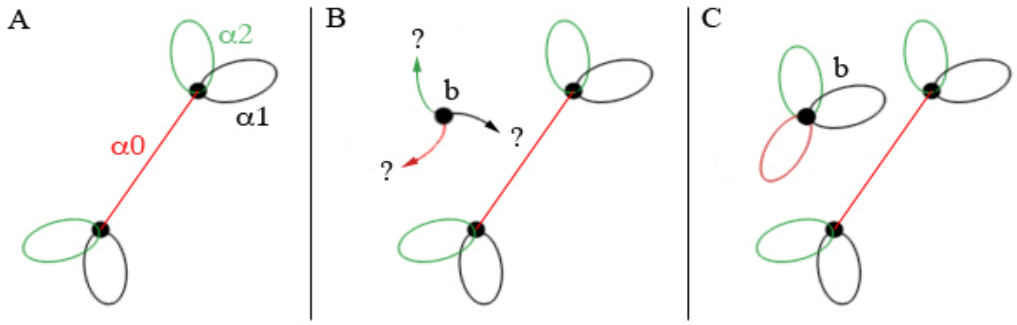

FIG. 13 – Ajout d'un brin isolé.

Le résultat de l'ajout d'un brin isolé est bien une G-carte car il vérifie la contrainte de cohérence des G-cartes.

Sur la figure 13, nous souhaitons ajouter un brin isolé b à la 2-G-carte de la figure 13A. L'ajout de b sans préciser les liens (figure 13B) n'est pas possible. En effet, un lien est une application de l'ensemble des brins vers l'ensemble des brins et doit par conséquent être défini pour tout brin de la 2-G-carte. Ainsi, conformément à la définition 7, on lie b à lui-même par involution (figure 13C). Ces nouveaux liens sont bien des involutions puisque $b\alpha_0\alpha_0 = b$, $b\alpha_1\alpha_1 = b$ et $b\alpha_2\alpha_2 = b$. De plus, la contrainte « $\alpha_0\alpha_2$ est une involution » est bien respectée car $b\alpha_0\alpha_2\alpha_0\alpha_2 = b$.

Au niveau de l'implantation, lors de l'ajout d'un brin isolé à une carte généralisée de dimension n, il suffit d'ajouter au graphe un sommet sur lequel nous faisons boucler $n + 1$ arcs indexés par l'involution correspondante. On remarque que dans ce cas, le fait d'implanter les opérateurs d'adjacence par une boucle lors de l'ajout d'un élément de base isolé suffit à vérifier les contraintes de cohérence du modèle.

4.1.2 Second cas : opérateurs d'incidence

A priori, dans le cas des ensembles semi-simpliciaux, on ne peut pas parler de l'ajout d'un simplexe isolé en ce sens que s'il est isolé (sous-entendu sans bord), l'objet n'est plus un ensemble semi-simplicial. En effet, les opérateurs de bords sont des applications. Ainsi pour tout k-simplexe σ ($k > 0$), σd_i avec $0 \leq i \leq k$ existe. Ceci étant, on ne peut ajouter un k-simplexe à un ensemble semi-simplicial sans lui associer de bord. Étant donné cette contrainte, il existe plusieurs manières d'ajouter un simplexe à un ensemble semi-simplicial.

Une première idée est de n'autoriser l'ajout d'un simplexe, que si les simplexes qui constitueront son bord existent déjà dans l'ensemble semi-simplicial. Il suffit dans ce cas d'ajouter le nouveau simplexe, puis de le lier à son bord. Une variante étant d'ajouter du même coup le simplexe et son bord.

Enfin, soit $S = (K, (d_j)_{j=0,\dots,n})$ un ensemble semi-simplicial de dimension n. On suppose qu'il existe dans K, un sous-ensemble L de $n+1$ simplexes (un par dimension) tel que pour chaque simplexe isolé ajouté, on lui associe un bord constitué de simplexes de L. Les simplexes de L constituent en quelque sorte les bords par défaut des autres simplexes de l'ensemble semi-simplicial. C'est à ce troisième cas que nous allons nous intéresser. Ainsi, nous devons redéfinir l'ensemble semi-simplicial vide comme étant l'ensemble semi-simplicial réduit à L. Dès à présent nous appellons élément de base isolé, un élément de base dont tous les simplexes du bord appartiennent à L. Nous pouvons alors parler de l'ajout d'un simplexe isolé. La définition 8 décrit mathématiquement cette opération. L'exemple de la figure 14 illustre la définition.

Définition 8 (Ajout d'un simplexe isolé dans un ensemble semi-simplicial de dimension n)**.** *Soit $S = (K, (d_j)_{j=0,\dots,n})$, un ensemble semi-simplicial. Soit $L = \bigcup_{i=0}^{n} \{\epsilon_i\}$, un sous-ensemble de K où ϵ_i (avec $i > 0$) est un i-simplexe de tel que $\epsilon_i d_j = \epsilon_{i-1}$, pour $0 \leq j \leq i$. $S' = (K', (d'_j)_{j=0,\dots,n})$, l'ensemble semi-simplicial résultant de l'ajout dans S d'un i-simplexe β est défini par :*

- *$S' = S \cup \{\sigma\}$;*
- *si $i > 0$ alors, $\beta d'_j = \epsilon_{i-1}$, pour $0 \leq j \leq i$;*
- *pour tout i-simplexe $\sigma \neq \beta$ de K', $\sigma d'_j = \sigma d_j$, pour $0 \leq j \leq i$.*

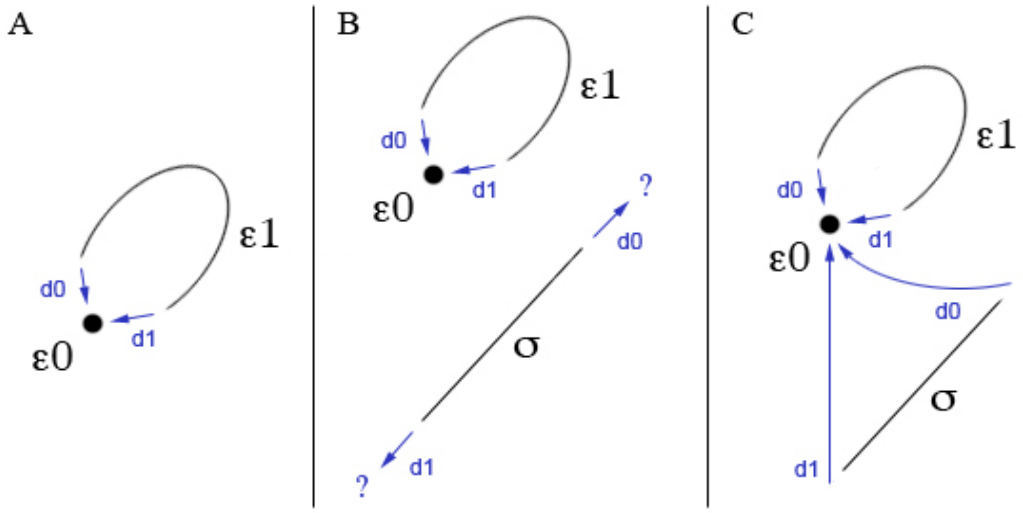

FIG. 14 – Ajout d'un simplexe.

À chaque nouveau simplexe, nous associons un bord, ainsi le résultat de l'opération est bien un ensemble semi-simplicial. En particulier la contrainte de cohérence des ensembles semi-simpliciaux est respectée si la contrainte est vérifiée par les simplexes de L, ce qui est le cas par hypothèse.

La figure 14 donne un exemple d'ajout d'un 1-simplexe σ à l'ensemble semi-simplicial vide de dimension 1 (figure 14A). Comme pour les G-cartes, l'ajout seul de σ sans mise à jour des opérateurs de bords ne suffit pas (figure 14B). Ainsi, sur la figure 14C, nous associons un bord par défaut (constitué du 0-simplexe ϵ_0) à σ. Ainsi, nous assurons le respect des contraintes.

Du point de vue de l'implantation, lors de l'ajout d'un k-simplexe σ, il suffit d'ajouter un nouveau sommet d'étiquette k, puis si $k > 0$, d'ajouter $k+1$ arcs indexés par le nom de l'opérateur de bord correspondant, vers le $k-1$-simplexe ϵ_{k-1}. Dans ce cas, le fait d'attribuer une valeur par défaut aux opérateur d'incidence, suffit à assurer la cohérence du modèle. Ceci rejoint les observations effectuées lors de l'étude des opérateurs d'adjacence.

4.1.3 La suppression d'un élément de base isolé

Indépendamment du modèle utilisé, supprimer un élément de base isolé e revient, dans le graphe correspondant au modèle, à supprimer les arcs sortant de e, puis à supprimer le sommet e.

4.2 L'ajout et le retrait d'un lien par reformulation des contraintes

Dans les opérations que nous venons d'étudier, le traitement des opérateurs d'adjacence et d'incidence suffit à respecter les contraintes de cohérence du modèle. Lors de l'ajout ou du retrait d'un lien entre deux éléments de bases, nous devons considérer ces contraintes. Ici, nous nous intéressons à la manière dont elles interviennent lors de ces deux opérations et proposons, à partir d'une reformulation des contraintes de cohérence, une solution pour la génération du code des opérations.

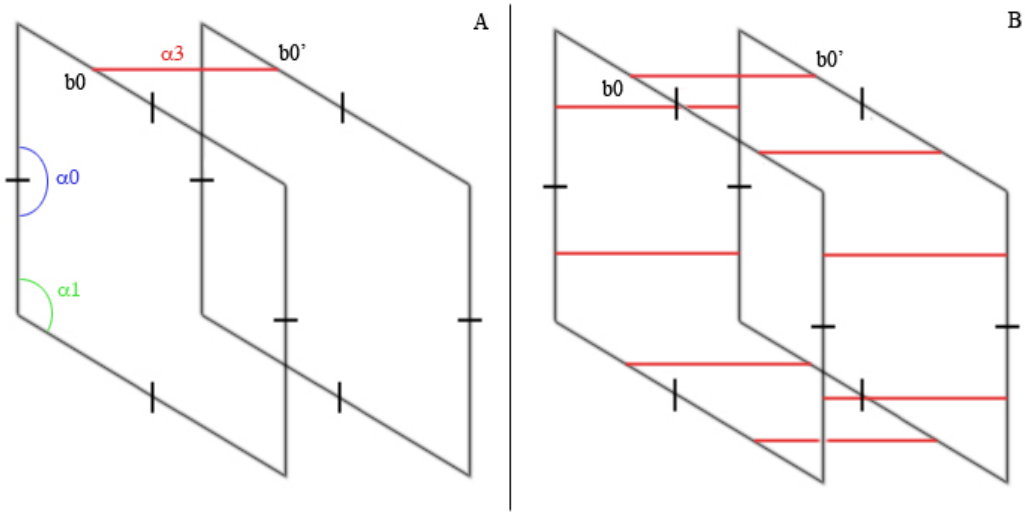

FIG. 15 – Couture par α_3.

Dans les cartes généralisées, les opérations de couture et décousure correspondent respectivement à l'ajout et à la suppression d'un lien entre deux éléments de base. Dans la section 2.1.2, où nous définissons la couture (définition 3), nous donnons un exemple de couture par α_2 dans une 2-G-carte. Afin de mettre en évidence la manière dont intervient la contrainte de cohérence des G-cartes dans l'opération lors de la couture, étudions un exemple plus complexe : la couture par α_3 dans une 3-G-carte (figure 15).

Dans la partie 3.3.2, où nous présentons notre modeleur-test, nous avons mis en évidence une décomposition de la couture, en deux parties distinctes : une partie locale et une partie associée à la propagation de la contrainte de cohérence. Cette décomposition apparaît dans l'exemple de la figure 15. Sur cette figure, les brins sont représentés par des demi-arêtes, les liens α_0 par des segments séparant deux demi-arêtes, les liens α_1 par des sommets entre

deux arêtes et les liens α_3 par un connecteur entre deux demi-arêtes. Sur cette figure, les boucles (par exemple en α_2) ne sont pas représentées.

Dans la partie A du schéma, nous ajoutons un lien α_3 entre deux brins b_0 et b'_0. Ceci correspond à la partie locale de l'opération. La génération du code correspondant à ce simple ajout d'un lien est triviale puisqu'il suffit d'ajouter deux arcs au graphe représentant la 3-G-carte. Dans la partie B, nous ajoutons les liens nécessaires au respect des contraintes de cohérence. Nous avons appelé ce processus la propagation, en ce sens qu'elle propage les contraintes de cohérence sur tout l'objet, de manière à le rendre cohérent. Lors de la génération du code des opérations, la principale difficulté est de générer cette propagation.

Dans le modèle des 3-G-cartes, la contrainte de cohérence est la suivante : $\alpha_i\alpha_j$ est une involution pour tout i, j vérifiant $0 \leq i < i+2 \leq j \leq 3$. Dans la figure 16, nous essayons de donner l'intuitions de ce à quoi correspond cette contrainte.

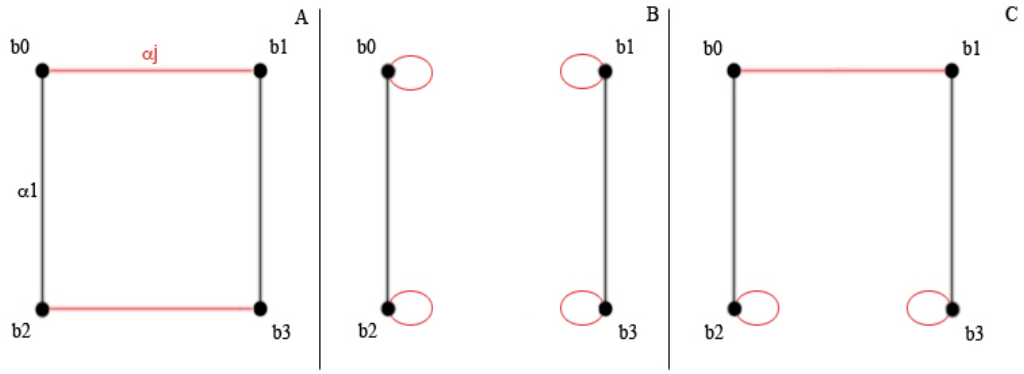

FIG. 16 – La contrainte des G-cartes.

Sur cette figure, où les brins sont représentés par un point, on suppose que $0 \leq i < i+2 \leq j$. Les objets des parties A et B respectent la contrainte de cohérence. En effet, pour tout $i \in \{0,1,2,3\}$, $b_i\alpha_i\alpha_j$ est une involution. Cependant le fait de vérifier la contrainte ne permet pas de conclure sur la structure de l'objet, puisque par exemple en A, b_0 et b_1 sont liés par α_2, tandis qu'en B ce lien n'existe pas. En C, l'objet est incohérent. En particulier, $b_0\alpha_i\alpha_j\alpha_i\alpha_j = b_1 \neq b_0$. Cet état indéterminé peut se produire dans au moins deux cas.

Premier cas : un lien α_j vient d'être ajouté entre b_0 et b_1. Alors, par

propagation, il faut ajouter un lien α_j entre b_2 et b_3. On se ramène ainsi au cas A. Il est possible de reformuler ceci en : si on ajoute un lien α_j entre b_0 et b_1 alors il faut ajouter un lien α_j supplémentaire entre $b_0\alpha_i = b_2$ et $b_1\alpha_i = b_3$. Bien sûr, une autre manière de rendre cohérent l'objet représenté en C est de retirer le lien α_j entre b_0 et b_1, mais nous supposons que ce lien vient d'être ajouté et que nous ne souhaitons plus y toucher.

Second cas : le lien α_j vient d'être supprimé entre b_2 et b_3. Alors, par propagation, il faut supprimer le lien α_j entre b_0 et b_1. On se ramène ainsi au cas B. Il est possible de reformuler ceci en : si on supprime le lien α_j entre b_2 et b_3 alors il faut supprimer le lien α_j entre b_0 et b_1.

De ces deux cas, nous pouvons déduire une reformulation des contraintes de cohérence des 3-G-cartes :

$$\begin{aligned} b_1\alpha_3 = b_2 &\Leftrightarrow b_1\alpha_0\alpha_3 = b_2\alpha_0 \\ b_1\alpha_3 = b_2 &\Leftrightarrow b_1\alpha_1\alpha_3 = b_2\alpha_1 \\ b_1\alpha_2 = b_2 &\Leftrightarrow b_1\alpha_0\alpha_2 = b_2\alpha_0 \end{aligned}$$

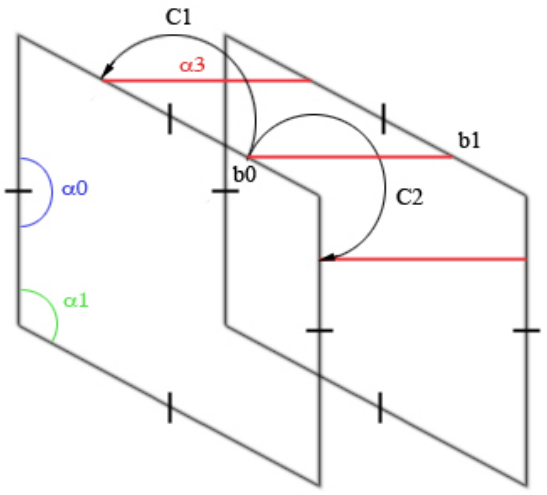

FIG. 17 – La reformulation des contraintes.

Sur la figure 17, on souhaite coudre b_0 et b_1 par α_3. On commence par ajouter ce nouveau lien (partie locale de l'opération). La première contrainte correspond à la flèche $C1$: b_0 et b_1 sont liés par α_3 alors $b_0\alpha_0$ et $b_1\alpha_0$ doivent à leur tour être liés par α_3. La seconde contrainte est représentée par la flèche $C2$. Nous venons de vérifier les contraintes à partir de b_0 et b_1, il suffit

maintenant de la vérifier pour les deux couples de brins qui viennent d'être liés. De proche en proche, on coud toute l'orbite, ainsi l'objet résultant est cohérent.

Lorsqu'on itère sur les contraintes, on peut retomber sur des brins déjà traités (pour lesquels la contrainte a été vérifiée). Nous devons donc utiliser un système de marquage des brins afin de nous assurer qu'on ne traite pas deux fois les mêmes brins.

La décousure est obtenue de manière analogue, avec les mêmes contraintes de cohérences. Il suffit de retirer les liens (faire apparaître des boucles), au lieu de les ajouter.

4.3 La génération du code des opérations

À partir d'une reformulation plus constructive des contraintes, il est facile de générer du code. Par exemple dans le cas étudié dans le paragraphe 4.2, il suffit de remplacer l'implication mathématique par un couple de « si alors sinon ». Nous obtenons alors l'algorithme suivant :

```
DEBUT verifier_contraintes(b1,b2)
   marquer(b1) ; marquer(b2) ;

   SI ((b1.a3 = b2) ET (non_marqué(b1.a0))) ALORS
      lier(b1.a0, b2.a0, a3) ;
      verifier_contraintes(b1.a0, b2.a0) ;
   FIN_SI
   SI ((b1.a3 = b1) ET (non_marqué(b1.a0))) ALORS
      délier(b1.a0, b2.a0, a3) ;
      verifier_contraintes(b1.a0, b2.a0) ;
   FIN_SI

   SI ((b1.a3 = b2) ET (non_marqué(b1.a1))) ALORS
       lier(b1.a1, b2.a1, a3) ;
       verifier_contraintes(b1.a1, b2.a1) ;
   FIN_SI
   SI ((b1.a3 = b1) ET (non_marqué(b1.a1))) ALORS
       délier(b1.a1, b2.a1, a3) ;
       verifier_contraintes(b1.a1, b2.a1) ;
   FIN_SI

   SI ((b1.a2 = b2) ET (non_marqué(b1.a0))) ALORS
```

```
      lier(b1.a0, b2.a0, a2) ;
      verifier_contraintes(b1.a0, b2.a0) ;
   FIN_SI
   SI ((b1.a2 = b1) ET (non_marqué(b1.a0))) ALORS
      délier(b1.a0, b2.a0, a2) ;
      verifier_contraintes(b1.a0, b2.a0) ;
   FIN_SI
FIN
```

Dans cet algorithme :

- *lier(b1,b2,ai)* signifie que l'on lie les deux brins b_1 et b_2 par α_i ;
- *délier(b1,b2,ai)* signifie que l'on retire le lien α_i entre b_1 et b_2 ;
- *b.ai* équivaut à la notation $b\alpha_i$.

Générer l'algorithme complet de couture de deux brins b_1 et b_2 par α_2 devient alors très simple puisqu'il se réduit à :

```
// Opération locale
lier(b1,b2,a2) ;

// Propagation
vérifier_contraintes(b1,b2) ;
```

De la même manière, la décousure est donnée par :

```
// Opération locale
délier(b1,b2,a2) ;

// Propagation
vérifier_contraintes(b1,b2) ;
```

Dans ces deux algorithmes, seule la partie locale doit être donnée par l'utilisateur.

Nous n'avons pas eu le temps de tester cette génération du code des opérations. En revanche, nous avons vérifié que notre méthode de reformulation des contraintes de cohérence ainsi que la décomposition des opérations en une partie locale et une partie propagation était possible sur d'autres modèles que les G-Cartes (par exemple sur les ensembles semi-simpliciaux).

5 Conclusion

Dans un premier temps, nous avons mené une étude approfondie des modèles topologiques existants. Au cours de ce rapport, nous nous appuyons sur seulement deux d'entre eux : les cartes généralisées et les ensembles semi-simpliciaux. Cependant, notre étude ne se limite pas à ces deux seuls modèles. Les arêtes ailées [Bau75] [Wei75], les cartes [Lie89], chaînes de cartes, etc. sont autant d'autres modèles auxquels nous nous sommes intéressés afin de concevoir le Méta-Modeleur.

Ce travail d'analyse nous a permis d'identifier des similitudes entre les modèles topologiques. Cette mise à jour des intersections des modèles connus nous a permis de définir une structure de donnée topologique générique de stockage des objets, dont seule l'instanciation la spécialise en fonction du modèle donné en entrée du Méta-Modeleur. Après avoir montré que cette structure fonctionne sur les modèles étudiés, nous avons pu proposer puis tester une implantation sous forme de graphes de cette structure générique.

Enfin, suite à ce premier travail de génération de la structure de données, nous nous sommes attelés à apporter une réponse au problème de la génération du code des opérations. Pour cela, nous avons montré qu'il était possible de construire des algorithmes complets à partir d'une formulation plus constructive des contraintes de cohérence des modèles. Cette solution a été testée sur plusieurs opérations issues des cartes généralisées puis des ensembles semi-simpliciaux.

Une première perspective à notre travail serait de spécialiser la structure de graphe proposé par OCamlGraph. Dans l'implantation de notre structure générique, les étiquettes des arcs sont utilisées comme indexage des liens. Or, l'implantation de graphes d'OCamlGraph ne permet pas d'accéder en temps constant à un lien dont nous connaissons l'étiquette. Néanmoins, l'aspect modulaire d'OCamlGraph permet à l'utilisateur d'implanter ses propres graphes tout en conservant l'interface de programmation.

En début de stage nous avons commencé par réfléchir à un langage de description des modèles topologiques. Après quelques essais, nous avons défini un premier langage très proche de la définition mathématique d'un modèle. Autour de ce premier langage, nous avons programmé un analyseur lexicale puis syntaxique nous permettant de construire un générateur d'arbre de syntaxe abstraite, stockant et ordonnant les données contenues dans la description du modèle d'entrée. C'est suite à ce premier travail que nous nous

sommes intéressés à la génération du code. Très tôt, nous avons identifié la génération de la structure de stockage des objets comme étant l'une des deux principales difficultés du stage, la seconde étant la génération d'algorithmes constructifs à partir des contraintes de cohérence du modèle. Lorsque nous avons proposé notre structure, il devenait clair que notre premier langage de description des modèles allait être modifié. Il devait faire figurer explicitement les éléments de bases du modèle ainsi que les liens entre ces éléments. Dès lors, il était plus prudent de travailler directement sur l'arbre de syntaxe abstraite du langage d'entrée. Fixer la grammaire de ce langage d'entrée était difficile avant notamment d'avoir proposé une solution à la génération du code des opérations. Maintenant que nous avons une solution à ce second problème, il est possible de fixer le langage d'entrée. Cette partie technique est l'une des étapes à effectuer, dans la continuité du stage. Lorsque le langage d'entrée sera parfaitement défini, il deviendra possible de lui associer un modèle mathématique afin de montrer que le code produit par le Méta-Modeleur correspond bien à la description donnée en entrée.

Parmi les modèles sur lesquels nous avons travaillé, on distingue principalement deux classes. Dans la première, les cellules topologiques (sommet, arêtes, faces, etc.) sont données implicitement. Ceci signifie qu'aucun élément de base ne leur correspond, ils sont donnés implicitement par un parcours des élément de bases. C'est le cas dans les cartes généralisées où par exemple les faces sont données par une orbite. La seconde classe correspond aux modèles dans lesquels les cellules sont données explicitement. Ainsi, à chaque cellule est associé un élément de base du modèle. Les ensembles semi-simpliciaux sont un exemple de tel modèle. Une étude intéressante consisterait à vérifier que nos solutions fonctionnent dans le cas des modèles ou les cellules sont données à la fois implicitement et explicitement. Le modèle des arêtes ailées contient cette redondance d'information. Par exemple les faces sont données par une orbite, mais elles existent aussi en tant qu'éléments de base du modèle. *A priori*, cette redondance d'information ne doit pas poser de problème si les contraintes de cohérence du modèle sont correctement construites ; nous devons le vérifier.

Lors de la conception de notre structure de donnée générique, et particulièrement lorsque nous avons étudié le fonctionnement d'OCamlGraph, nous nous sommes questionné sur ce que pourrait donner une bibliothèque modulaire de modèles topologiques. Par exemple, de nombreux modeleurs géométriques à base topologique se limitent à un unique modèle de plongement. En effet, il est parfois difficile de séparer la topologie de la géométrie (et donc du plongement), en particulier dans certaines opérations comme le

chanfreinage. Si, dès l'écriture du noyau, le plongement était un paramètre du modèle, la frontière entre la géométrie et le plongement serait plus marquée. Ceci faciliterait l'écriture de modeleurs dans lesquels le modèle de plongement ne serait pas imposé. Ainsi, l'idée d'un modèle paramétrable est une option qui s'offre à nous dans la continuité de ce stage puisque c'est ce vers quoi le Méta-Modeleur pourrait évoluer. Dans ce cas, il conviendra de ne plus parler de Méta-Modeleur au sens de générateur de noyaux de modeleurs, mais plutôt de modèle générique.

La création d'un générateur de noyaux de modeleurs géométrique à base topologique est un projet ambitieux en ce sens qu'il demande une bonne expérience de la modélisation géométrique. Bien que ce projet se soit déroulé dans le cadre d'un stage pour lequel par définition une telle expérience n'existe pas, nous avons réussi à répondre aux deux principaux problèmes posé par le Méta-Modeleur. Nous avons proposé une structure de donnée générique pouvant se spécialiser en un modèle topologique quelconque. De plus, nous sommes capables de générer des algorithmes complets à partir des contraintes de cohérence du modèle. Ainsi, nous savons maintenant générer le code d'un noyau de modeleur à partir d'une description succincte d'un modèle, répondant ainsi aux spécifications principales du Méta-Modeleur.

Références

[Bau75] B. Baumgart. A polyhedron representation for computer vision. In *AFIPS Nat. Conf. Proc.*, volume 44, pages 589–596, 1975.

[BS85] R. Bryant and D. Singerman. Foundations of the theory of maps on surfaces with boundaries. *Quart. J. Math. Oxford Ser. (2)*, 36(141) :17–41, 1985.

[CFS05] F. Conchon, J.-C. Filliâtre, and J. Signoles. Le foncteur sonne toujours deux fois. In *JFLA'05*, pages 79–93. INRIA, March 2005.

[Inr85] Inria. Objective caml. caml.inria.fr/, 1985.

[Lie89] P. Lienhardt. Subdivision of n-dimensional spaces and n-dimensional generalized maps. In *Annual Symposium on Computational Geometry SCG'89*, pages 228–236, Saarbruchen, Germany, June 1989. ACM Press.

[Lie91] P. Lienhardt. Topological models for boundary representation : a comparison with n-dimensional generalized maps. *Computer-Aided Design*, 23(1) :59–82, January 1991.

[May67] J.-P. May. *Simplicial Objects in Algebric Topology*, volume 11 of *Van Nostrand Mathematical Studies*. Van Nostrand, première edition, 1967.

[Tut84] W. Tutte. *Graph Theory*, volume 21 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, 1984.

[VD03] F. Vidil and G. Damiand. Moka. www.sic.sp2mi.univ-poitiers.fr/moka/, 2003.

[Wei75] K. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *Comput. Graph. Appl.*, 5(1) :21–40, 1975.