N^o d'ordre : **3904**

THÈSE

présentée à

l'Université Louis Pasteur de Strasbourg - Département d'informatique
Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection
UMR 7005 CNRS/ULP

par

Mme Caroline ESSERT-VILLARD

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ LOUIS PASTEUR
MENTION SCIENCES
SPÉCIALITÉ INFORMATIQUE

SÉLECTION DANS L'ESPACE DES SOLUTIONS ENGENDRÉES PAR UN PLAN DE CONSTRUCTION GÉOMÉTRIQUE

---

soutenue publiquement le 16 novembre 2001
devant la commission d'examen composée de :

M. Jean-Daniel BOISSONNAT, Rapporteur Externe,
Directeur de recherche à l'INRIA de Sophia-Antipolis

M. Dominique MICHELUCCI, Rapporteur Externe,
Professeur à l'Université de Bourgogne

M. René CAUBET, Examinateur,
Professeur à l'Université Paul Sabatier de Toulouse

M. Jean-Michel DISCHLER, Rapporteur Interne,
Professeur à l'Université Louis Pasteur de Strasbourg

M. Jean-François DUFOURD, Directeur de Thèse,
Professeur à l'Université Louis Pasteur de Strasbourg

M. Pascal SCHRECK, Invité,
Maître de conférences à l'Université Louis Pasteur de Strasbourg

Remerciements

Tout d'abord, je tiens à remercier le professeur Jean-François Dufourd de m'avoir proposé ce sujet, et d'avoir dirigé ce travail. Son expérience, ses conseils avisés, et ses corrections judicieuses m'ont été très précieux.

Un grand merci de tout coeur à Pascal Schreck pour tout ce qu'il m'a apporté, tant au niveau professionnel que personnel, et dont la liste serait trop longue à faire ici. Merci pour tout le temps qu'il m'a consacré. Pour avoir partagé avec moi sa vision du monde et ses blagues Carambar, sa musique et parfois ses angoisses, mais surtout son amitié.

Je remercie le professeur Dominique Michelucci pour sa grande disponibilité et sa sympathie, les professeurs Jean-Daniel Boissonnat, Jean-Michel Dischler, et René Caubet pour m'avoir fait l'honneur de s'intéresser à ce travail et avoir accepté de faire partie de mon jury de thèse.

Je remercie le professeur Dominique Bechmann, qui a été la première à me donner le goût de la recherche, en me proposant un sujet et en encadrant mon DEA.

Merci également à tous ceux qui m'ont entourée dans mon travail, encouragée, et soutenue pendant toutes ces années, mes indispensables collègues et amis : Pascal, Dominique, Vince et Zizou, sans oublier Fab, Lolo, Niko, Sylvain, Mooby, Pierre, Arash, Marie-Claire, Lili, ainsi que les petits nouveaux qui constituent une relève énergique, et tous ceux que j'oublie... Merci a tous mes amis d'ici et d'ailleurs.

Et enfin, je tiens à remercier de tout mon coeur mes proches, ceux qui me soutiennent depuis la nuit des temps, mes parents, mon frère, mes grands-parents, ma belle-famille, et par dessus tout ceux qui m'ont supportée, soutenue et aimée chaque jour de cette thèse et à qui je dédie ce mémoire : Antoine et Arthur.

Table des matières

Table des figures

Introduction

Présentation du sujet

Le travail présenté dans ce mémoire s'insère dans le cadre de la modélisation géométrique en CAO (Conception Assistée par Ordinateur), en vue principalement de la conception automatisée d'objets solides. De nombreuses spécialités sont intéressées par ce sujet, comme celles qui concernent la conception de pièces mécaniques industrielles (automobile, aéronautique, jouets, etc.), ou encore l'architecture, et même des domaines plus artistiques comme les arts plastiques. Il trouve également des extensions dans les domaines de l'animation d'objets et de la simulation, par exemple d'articulations ou de coulissements.

Cette étude se situe plus précisément dans le cadre des constructions géométriques sous contraintes. En CAO, les contraintes géométriques sont souvent utilisées sous forme de systèmes de cotes pour décrire des figures de façon précise, ainsi que les propriétés géométriques qu'elles doivent respecter. Ces contraintes sont de type distance, angle, tangence, égalité entre entités, etc. Le plus souvent, une esquisse approximative ressemblant à l'objet final que l'on souhaite obtenir est tracée sommairement à l'aide d'une souris. Les contraintes imposées s'ajoutent à des contraintes implicites d'incidence et d'adjacence induites par le tracé de l'esquisse. Finalement, si l'on souhaite pouvoir utiliser par la suite l'objet ainsi modélisé, en vue par exemple d'un usinage ou d'une simulation, il est absolument nécessaire que la figure calculée respecte les contraintes. Un système de CAO doit donc pouvoir les résoudre et donner les solutions possibles au problème posé.

Afin de résoudre de tels problèmes de constructions géométriques sous contraintes, de nombreux types de solveurs ont été réalisés en CAO, utilisant diverses techniques de résolution. Il existe principalement deux grandes classes de solveurs : ceux qui utilisent une approche numérique, et ceux qui suivent une approche symbolique. Les premières, qui sont le plus souvent choisies, consistent à résoudre le système d'équations numériques provenant de la cotation. Les secondes consistent à conserver un historique de la construction en produisant à partir des contraintes un *plan de construction* dans lequel les valeurs des cotes sont abstraites symboliquement. De plus, certains solveurs emploient une combinaison de ces approches avec une technique de décomposition d'un problème en sous-problèmes plus petits et plus facilement solubles, puis un assemblage des sous-figures ainsi résolues.

Lors de précédents travaux, une approche symbolique basée sur le raisonnement géo-

métrique a été étudiée au sein de notre équipe. Cette approche efficace présente un certain nombre d'avantages, notamment celui de permettre des manipulations de figures telles que l'animation en temps réel, difficilement réalisable avec une approche numérique. Elle est associée à une méthode originale de décomposition ascendante et d'assemblage de sous-figures. Elle a été concrétisée dans le domaine de la CAO par un prototype nommé *YAMS* (Yet Another Meta-Solver), qui associe un solveur géométrique avec le modeleur 3D à base topologique *Topofil*. Les outils de résolution proposés sont basés sur la géométrie classique, plus particulièrement les constructions géométriques, ainsi que sur le raisonnement symbolique, en particulier les systèmes à base de règles. Ils sont suffisamment spécialisés pour résoudre la plupart des problèmes de CAO.

Cependant, quelle que soit l'approche choisie, numérique ou symbolique, algébrique ou géométrique, un système de contraintes ne définit généralement pas une figure unique. Lorsque l'ensemble des solutions est fini et non vide, le système est dit bien-contraint. Dans le cas d'un système bien-contraint, l'exploration de l'espace des solutions produites n'est pas aussi aisé qu'on pourrait le supposer. En effet, l'existence d'équations polynomiales dont le degré est supérieur ou égal à 2, d'un point de vue algébrique, ou bien d'intersections multiples, d'un point de vue géométrique, conduit rapidement à une explosion combinatoire du nombre de solutions. Par rapport à notre approche, cette augmentation du nombre de solutions est le résultat de la présence de multifonctions dans le plan de construction élaboré par *YAMS* à partir d'un système bien-contraint.

Dans la plupart des cas, les utilisateurs de CAO désirent une solution unique à leur problème lorsqu'ils dessinent un objet. C'est pourquoi il est important pour un solveur de pouvoir identifier la solution qui est la plus pertinente par rapport aux attentes de l'utilisateur, qu'il a matérialisées par les contraintes qu'il a posées, et par l'esquisse qu'il a tracée. En effet, examiner toutes les solutions les unes après les autres serait un travail trop fastidieux lorsque leur nombre devient très grand. La réponse la plus couramment proposée est l'utilisation d'heuristiques pour filtrer les résultats déjà calculés. Ceci est généralement caractérisé par des temps de calcul très longs, et il reste encore souvent plus d'une solution après le filtrage.

L'approche symbolique développée dans *YAMS* permet de structurer l'espace des solutions en utilisant le plan de construction, qui a été établi précédemment lors de la phase de résolution. En effet, celui-ci peut être représenté sous la forme d'un arbre, qui présente un embranchement à chaque fois qu'un choix est imposé par la présence d'une multifonction. Cette structure arborescente peut être exploitée de plusieurs manières. Nous en avons expérimenté trois, une ayant pour objet l'optimisation de l'organisation du plan de construction, et les deux autres la sélection d'une solution parmi un espace de solutions.

Premièrement, il est possible de réorganiser le plan de construction statiquement, ou dynamiquement par détection d'échec pendant le parcours de l'arbre, grâce à l'utilisation d'une méthode dérivée du problème SAT.

Deuxièmement, nous proposons une méthode efficace qui consiste à comparer les entités géométriques de la solution avec celles de l'esquisse au fur et à mesure de leur calcul. Cela

permet ainsi d'éliminer au plus tôt les solutions indésirables, représentées par des sous-arbres, en élaguant les branches correspondantes dans l'arbre dès qu'une incompatibilité est détectée, avant que leur calcul ne soit complètement terminé. Cette méthode permet, dans le cas de contraintes métriques, d'obtenir dans tous les cas une branche unique, donc une solution unique. Cependant, cette approche présente quelques limites dues à la présence éventuelles de contraintes non métriques, ou booléennes, telles que la tangence ou l'égalité d'entités.

Une troisième méthode nous permet donc de remédier à ce problème résiduel, et permet également à l'utilisateur de pouvoir accéder tout de même aux autres solutions dans le cas où il ne serait pas totalement satisfait de celle qui lui a été proposée. Il s'agit d'outils interactifs qui font intervenir l'utilisateur. Ils sont notamment inspirés des outils de débogage présents dans la plupart des environnements de développement informatique, et tirent donc avantage du plan (également appelé programme) de construction. Le *designer* a ainsi la possibilité de reprendre l'interprétation numérique, en l'effectuant cette fois-ci pas à pas, afin de choisir lui-même le résultat à sélectionner à certains embranchements. Il a également l'opportunité de manipuler à la souris une solution déjà calculée, afin de déplacer les éléments dont le placement ne lui conviennent pas. Tous ces outils font bien entendu appel à des opérations sous-jacentes sur l'arbre et le plan de construction, et sont totalement transparentes pour l'utilisateur.

Toutes ces méthodes, aussi bien automatiques qu'interactives, traitant le problème de la multiplicité des solutions, ont donné lieu au développement d'un prototype nommé *SAMY* (Sélection Automatique : Modules pour Yams). Celui-ci a été ajouté au noyau de *YAMS* en tant que groupe de modules, d'abord de sélection automatique de solutions, puis de navigation. Manipulant des plans de construction, il intervient donc après la phase de résolution de *YAMS*, lorsque le plan de construction a déjà été produit.

Dans un souci de clarté, et afin de démarrer cette recherche sur des bases solides, nous avons opté pour une phase préliminaire à ce travail, qui est la description précise et rigoureuse des modèles que nous utilisons. C'est pourquoi une première partie de notre étude consiste à formaliser les notions, entités, et opérations sur lesquelles nous travaillons, ainsi que les problèmes posés. C'est le langage *OBJ3*, qui permet de réaliser des spécifications algébriques ainsi que de tester ensuite ces spécifications, qui a été choisi pour cette formalisation. Il a permis d'apporter plus de rigueur au cœur de nos travaux.

Plan du mémoire

Le Chapitre 1 présente sommairement les principales méthodes (numériques et symboliques, par décomposition, etc.) couramment utilisées pour résoudre des problèmes de constructions géométriques sous contraintes. Il aborde ensuite les réponses que certains auteurs ont commencé à apporter depuis quelques années pour résoudre la question des solutions multiples engendrées par la résolution des systèmes de contraintes.

Dans un deuxième temps, le Chapitre 2 expose notre approche de la résolution de

contraintes géométriques ainsi que toutes les notions qui lui sont relatives, puis détaille le prototype *YAMS* auquel elle a donné lieu. Un exemple mettant en jeu plusieurs centaines de contraintes permet de mieux appréhender la façon d'utiliser ce système.

Le Chapitre 3 détaille ensuite la spécification algébrique qui permet de décrire formellement et avec précision toutes les notions nécessaires pour constituer une base solide pour ce travail. Ce chapitre offre une vue d'ensemble claire et rigoureuse sur tous les concepts que nous utilisons, comme les définitions, le plan de construction, l'arbre des solutions, le graphe de dépendance, etc. Il ébauche également un début de réponse au problème posé par la multiplicité des solutions en proposant différentes sortes de tris topologiques sur le plan de construction afin de le réorganiser.

Dans le Chapitre 4, nous proposons une première approche de sélection pour traiter le problème des solutions multiples. Il s'agit d'une méthode automatique efficace, basée sur la notion de déformation continue du système de contraintes, qui tire parti de la structure en forme d'arbre de l'espace des solutions pour l'élaguer au fur et à mesure du calcul des solutions. Divers exemples illustrent cette méthode et permettent de mieux la comprendre. Mais quelques limites restreignent cependant son utilisation.

Le Chapitre 5 montre comment les limites évoquées au chapitre précédent peuvent être surmontées grâce à l'emploi des divers outils interactifs, faisant appel à une intervention de l'utilisateur. Ces outils permettent d'effectuer une interprétation numérique pas à pas, ou de manipuler directement une solution à l'aide de la souris. La conception, l'utilisation, et les fonctionnalités de ces outils sont détaillées dans ce chapitre, illustrés par quelques exemples significatifs.

Le Chapitre 6, plus technique, présente l'aspect général et détaille les fonctionnalités du module *SAMY*, puis en montre son fonctionnement à travers quelques exemples d'utilisation.

Une conclusion clôt la présentation de ce travail avec un bilan et l'évocation de certaines perspectives. Elle est suivie d'une bibliographie offrant les références des ouvrages auquels se rapportent les citations faites tout au long de ce mémoire.

Finalement, quelques informations complémentaires sont données en annexe, afin de permettre une meilleure compréhension sans pour autant trop alourdir le mémoire en lui-même. Elle comprend la signature de l'univers géométrique, ainsi que quelques parties de code source en C++.

Chapitre 1

Résolution de contraintes géométriques et sélection de solutions

Dans le domaine de la Conception Assistée par Ordinateur (ou CAO), la résolution automatique de figures représentées par des esquisses cotées est un sujet qui a déjà été étudié par beaucoup d'auteurs. C'est pourquoi, après avoir introduit un peu plus précisément la problématique de cette étude, ce chapitre de synthèse aborde les principales approches de résolution de contraintes et de sélection de solutions fréquemment rencontrées.

Notons que le sujet de notre recherche est l'étude de la structuration d'un espace de solutions produit par un solveur géométrique, et son exploration. Bien que la résolution de systèmes de contraintes en elle-même ne fasse pas partie de notre sujet, celui-ci en dépend naturellement. En particulier, dans notre cas, c'est le solveur *YAMS* qui produit l'espace des solutions. Ce choix de solveur a été fait pour des raisons historiques, mais aussi parce que nous pensons que l'approche formelle suivie dans ce solveur se prête bien à notre étude. Afin de justifier plus précisément le choix du solveur, nous recensons, à la suite de la problématique, les méthodes de résolution de contraintes géométriques qui nous semblent les plus caractéristiques. Pour ne pas égarer le lecteur, nous en ferons une présentation synthétique et relativement succincte. Ainsi, toujours dans cette optique, nous citerons principalement les articles fondateurs, en renvoyant le lecteur à [SABC98, Dur98, LK98, JAMSR01, Wan01, CH01a, CH01b] pour des références plus récentes sur le sujet.

1.1 Problématique

Comme nous l'avons évoqué brièvement dans l'introduction de ce mémoire, l'utilisateur auquel nous nous intéressons souhaite, pour diverses raisons liées à sa spécialité, concevoir automatiquement un objet solide. Peut-être même souhaite-t-il animer cet objet, ou l'articuler. Il a à l'esprit une ébauche de figure, disons l'allure générale de l'objet qu'il souhaite créer. Il possède également un cahier des charges, lui imposant toutes sortes de contraintes

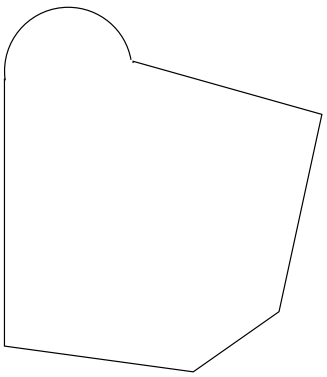


Fig. 1.1: Une esquisse tracée grâce à un modeleur

sur cet objet, en plus des contraintes géométriques classiques ou de bon sens qui lui sont propres. Ces contraintes apportent une description précise des propriétés qui doivent être respectées par l'objet.

Parmi les outils classiques proposés en CAO, un modeleur lui permet de tracer une *esquisse* de son objet. L'utilisateur peut dessiner à la souris des points, segments, arcs de cercles, cercles qui composeront sa figure. La Fig.1.1 montre un exemple du genre d'esquisse qu'il est possible de tracer grâce à un tel modeleur, et représente une pièce quelconque en 2D.

Cependant, l'esquisse que l'utilisateur a tracée ne prend pas encore en compte toutes les contraintes qui doivent s'appliquer à son objet. Elle vérifie déjà les contraintes d'incidence et d'adjacence, car celles-ci sont implicites, et sont liées à la nature même du dessin de l'esquisse. Mais elle ne respecte encore aucune autre contrainte.

Pour cela, l'utilisateur réalise graphiquement une spécification déclarative de la pièce qu'il a à l'esprit. Il ajoute ainsi les contraintes supplémentaires qu'il souhaite appliquer à sa figure. Celles-ci peuvent être par exemple des contraintes de longueur d'un segment, de distance entre deux entités géométriques, d'angle, de tangence, etc. Il les place sur l'esquisse, de manière à obtenir ainsi une *esquisse cotée*. La Fig.1.2 reprend l'exemple de la Fig.1.1 en y ajoutant une cotation. Elle représente la pièce cotée, sur laquelle nous avons nommé des points caractéristiques afin de mieux préciser les contraintes définissant l'objet représenté, l'hexagone *cdefga*. Afin de repérer plus facilement les différentes cotes qu'il a placées, celles-ci sont également nommées.

En dessin technique, la signification de la double-flèche matérialisant une cote n'est pas la même suivant les positions des lignes de cotes. C'est ainsi que dans l'exemple proposé, $k1$ traduit une contrainte de distance entre les points a et b, et $k5$ une contrainte de distance entre le point e et la droite fg. Notons qu'ici, les droites sont considérées comme orientées, ainsi que les angles. Ajoutons enfin que cette esquisse fait intervenir un point extérieur, nommé b.

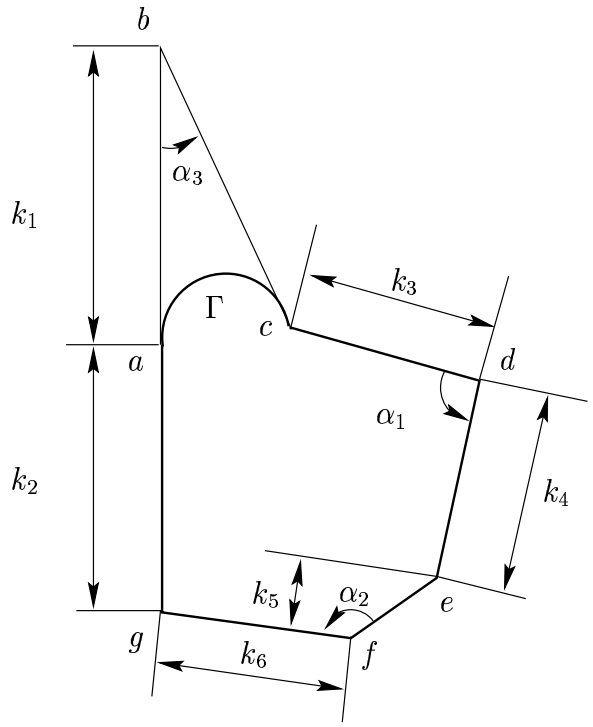


Fig. 1.2: Une esquisse cotée

L'esquisse cotée constitue ainsi la représentation graphique d'un système de contraintes, qui doit ensuite être résolu afin de générer une figure vérifiant toutes les contraintes, les propriétés géométriques qui ont été spécifiées soit implicitement par le tracé de l'esquisse soit déclarativement. C'est l'objet du *solveur*, deuxième composant de l'outil de résolution automatique de figures contraintes.

Suivant la méthode de résolution implantée par le solveur, une, quelques, ou toutes les solutions du système de contraintes peuvent être produites. Par exemple, quatre solutions engendrées par le système de contraintes donné dans l'exemple précédent sont présentées à la Fig.1.3.

Il existe un grand nombre de méthodes de résolution différentes. Elles ont fait l'objet de recherches de la part de beaucoup d'auteurs. Dans les sections suivantes, nous rappelons les principales d'entre elles.

La production de solution(s) est une des facettes de la problématique des constructions géométriques en CAO. Une fois les contraintes résolues, se posent généralement plusieurs autres questions. Comme nous l'avons vu plus haut, les solveurs, suivant la méthode de résolution utilisée et les valeurs particulières des données, peuvent fournir une seule ou bien plusieurs solutions à l'énoncé. Dans le cas d'une solution unique, la solution trouvée est-elle celle que l'utilisateur attendait ? Lorsque les solutions sont nombreuses, commment

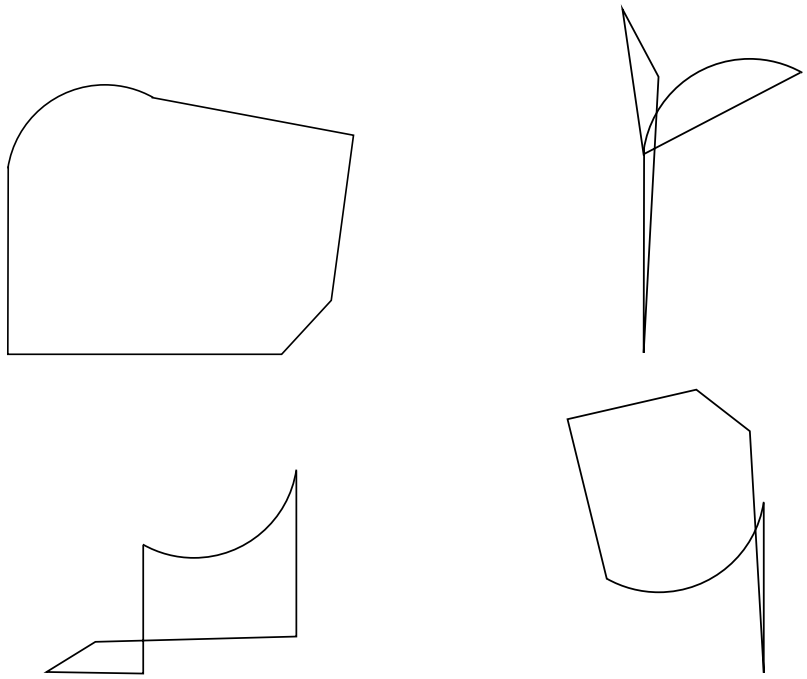


Fig. 1.3: Quatre solutions engendrées par la résolution de l'esquisse cotée

sélectionner la solution ? C'est pourquoi nous évoquons ensuite également les réponses apportées jusqu'à présent par les différents auteurs à ces questions.

1.2 Résolution de contraintes géométriques

En créant Sketchpad, I.E. Sutherland [Sut63] a été l'un des pionniers de la modélisation géométrique à l'aide de contraintes, et fut à l'origine d'un grand nombre de travaux. Depuis, diverses façons de résoudre des contraintes ont été explorées. Deux classes principales peuvent être distinguées : les approches numériques et les approches symboliques. Nous exposons dans ce chapitre quelques-unes de ces techniques de résolution (voir aussi [AM95]), en essayant de voir comment est gérée la question de la pertinence de la solution proposée à l'utilisateur.

1.2.1 Approches numériques

Ce premier type d'approche est très souvent choisi pour résoudre des systèmes de contraintes. Il consiste à résoudre numériquement le système d'équations correspondant à la cotation. La résolution s'appuie essentiellement sur des méthodes itératives provenant de l'analyse numérique, mais aussi sur des méthodes de décomposition.

Les techniques de l'analyse numérique peuvent être mises en œuvre pour obtenir des solutions approchées. Elles reposent sur des idées simples (dichotomie, méthode de la sécante, de la tangente, etc.), et procèdent toutes par approximations successives. Le principal avantage de ces méthodes est qu'elles permettent aisément d'intégrer de nouveaux types de contraintes. En revanche, elles possèdent l'inconvénient de ne proposer souvent qu'une seule solution, bien que les systèmes non linéaires en possèdent généralement un nombre qui croît exponentiellement avec le nombre d'équations et leur degré. Nous présentons ci-dessous quelques méthodes classiques.

Relaxation de contraintes Cette méthode [Bor81] fut l'une des premières à être utilisée pour résoudre ce type de problème. Elle a été introduite en modélisation géométrique par I.E. Sutherland [Sut63], et a donné lieu à son prototype nommé Sketchpad. Elle a également été utilisée plus récemment par L. Solano Albajes et P. Brunet Crosa dans [SABC98]. Elle est basée sur un système de définition d'objets par contraintes, et s'effectue en deux étapes. La première phase est une phase formelle de propagation des degrés de liberté dans un graphe. Si elle échoue, la phase algébrique numérique de relaxation entre en jeu. Elle agit par perturbation des valeurs affectées à des variables, provoquant par répercussion une modification des autres valeurs par propagation dans le graphe, jusqu'à ce que l'erreur totale soit minimisée. Le principal inconvénient de cette méthode est que la convergence est assez lente. De plus, elle ne propose qu'une seule solution.

Méthode de Newton-Raphson Cette approche itérative [PFTV88] est celle qui est la plus couramment utilisée en CAO, du fait de son implantation facile et de sa rapidité de convergence. Elle peut être utilisée lorsque les contraintes sont exprimées comme des équations algébriques de forme implicite ($f(x) = 0$).

Dans le cas simple à 1 inconnue, la méthode de Newton-Raphson génère une séquence d'approximations successives d'une solution de $f(x) = 0$ en utilisant la suite

$$x_{r+1} = x_r - \frac{f(x_r)}{f'(x_r)} \tag{1.1}$$

D'un point de vue géométrique, le problème revient à chercher l'intersection de la courbe représentative de f avec l'axe Ox. Le principe est de considérer la tangente en un point comme une bonne approximation de la courbe. C'est pour cette raison qu'elle est parfois aussi appelée méthode de la tangente.

En 2 dimensions, nous avons 2 équations à résoudre simultanément :

$$\begin{cases} F(x, y) = 0 \\ G(x, y) = 0 \end{cases}$$

On utilise alors le Jacobien J défini par

$$J = \begin{bmatrix} \dfrac{\partial F}{\partial x} & \dfrac{\partial F}{\partial y} \\ \dfrac{\partial G}{\partial x} & \dfrac{\partial G}{\partial y} \end{bmatrix}$$

La suite 1.1 définie avec 1 inconnue devient alors au point X_r

$$J(X_r)\delta_r = -\mathrm{F}(X_r)$$

où $X_r = \begin{bmatrix} x_r \\ y_r \end{bmatrix}$, $\delta_r = \begin{bmatrix} h_r \\ k_r \end{bmatrix}$, $\mathrm{F}(X_r) = \begin{bmatrix} F(x_r) \\ G(x_r) \end{bmatrix}$, $h_r = x_{r+1} - x_r$ et $k_r = y_{r+1} - y_r$. En dimension n, nous avons x_r, δ_r, et $\mathrm{F}(X_r)$ qui deviennent des vecteurs $n \times 1$, tandis que $J(X_r)$ est une matrice $n \times n$.

Cette méthode présente cependant deux inconvénients majeurs. Le premier est d'être dépendante de la valeur initiale, et d'avoir un comportement chaotique suivant cette valeur initiale. H. Lamure et D. Michelucci ont ainsi montré que les bassins d'attraction des systèmes d'équations algébriques sont des objets fractals. Un exemple graphique pour $z^3 - 1 = 0$ est donné dans [LM95]. Le deuxième est de ne proposer qu'une seule solution. C'est un inconvénient sérieux, puisqu'aucune autre possibilité n'est proposée dans le cas où la solution est incompatible avec l'application voulue.

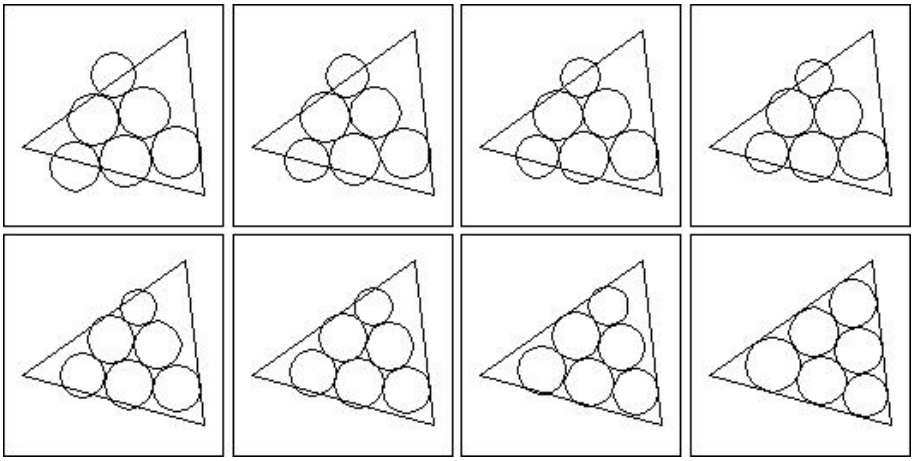


Fig. 1.4: Progression de la résolution par homotopie en 7 pas de t

Méthode par homotopie L'homotopie est une notion bien connue en topologie [Mau80]. Elle a été utilisée notamment par D. Michelucci et H. Lamure [LM95], qui ont proposé une méthode de résolution de contraintes géométriques par homotopie (en anglais,

continuation method). Considérons un système $G(X) = 0$ de n équations indépendantes à n inconnues représentant l'ensemble des contraintes. On définit le système $F(X) = G(X) - G(X_0)$ ayant pour solution l'approximation initiale X_0 donnée par un utilisateur. Une interpolation linéaire H est établie entre F et G de la façon suivante :

$$H(t, X) = tG(X) + (1 - t)F(X)$$

où H est une homotopie telle que $H(0, X) = F(X)$ et $H(1, X) = G(X)$. En faisant varier le paramètre t pour une valeur de X donnée, H suit une courbe (ou chemin) homotopique. L'idée est de suivre la courbe partant de $(t = 0, X = X_0)$. Si le chemin homotopique passe par un point $(t = 1, X)$, alors une solution de $H(1, X) = 0$ et donc de $G(X) = 0$ est trouvée. Différentes manières itératives de suivre les chemins homotopiques sont présentées en détail dans [LM95]. Un exemple de progression de la résolution par homotopie est présenté Fig.1.4.

Cette méthode offre une réponse aux problèmes de non-convergence de la méthode de Newton-Raphson. Cependant, elle est plus lente que cette dernière. Ceci est dû au fait que le passage d'un point de la courbe homotope à un autre est réalisé par approximations en résolvant des contraintes par une méthode de type Newton-Raphson.

Notons qu'il est possible avec cette méthode d'essayer de suivre, lorqu'on le souhaite, plusieurs chemins homotopiques et d'obtenir ainsi toutes les solutions réelles pour un système de contraintes (voir plus de détails dans [Dur98]).

D'autres méthodes plus récentes comme le calcul d'intervalles ont été étudiées, mais pour celle-ci comme pour les méthodes précédentes, il est difficile, coûteux, voire impossible d'avoir toutes les solutions. De plus, lorsqu'on obtient une solution, rien ne permet d'affirmer que celle-ci est bien la solution recherchée. Même la méthode par homotopie, qui paraît être la mieux à même de produire une solution ressemblant à l'esquisse tracée par l'utilisateur, peut parfois hésiter entre deux chemins symétriques.

Lorsque la solution n'est pas celle que l'utilisateur attend, celui-ci n'a d'autre choix que de relancer la résolution en modifiant les valeurs initiales, ou l'approximation initiale, afin de faire converger sur une autre solution, ou bien dans le cas de la dernière méthode, de reparcourir un chemin homotopique en suivant une autre bifurcation lorsqu'il en existe une. Dans le cas défavorable où il faudrait parcourir toutes les solutions, le processus serait alors extrêmement fastidieux.

1.2.2 Méthodes symboliques

Une résolution formelle (ou symbolique) du système de contraintes présente l'avantage de fournir une solution générale au problème posé. La résolution s'effectue en deux phases distinctes : la phase symbolique qui, après abstraction des valeurs numériques, produit un procédé de construction général, puis une phase d'interprétation numérique lors de laquelle on réintroduit les valeurs des cotes pour produire la ou les solutions.

L'aspect numérique n'intervient donc qu'en tant que dernière étape de la résolution, ce qui permet de ne pas avoir à recommencer entièrement le processus de résolution lorsqu'on souhaite voir une autre solution que celle qui est proposée. Il suffit de reprendre seulement le dernier stade, l'interprétation numérique, pour passer à une autre solution.

Ce type d'approche possède des propriétés intéressantes, comme celle de pouvoir manipuler de façon efficace la figure définie en faisant varier les valeurs des paramètres, ce qui peut donner lieu par exemple à une animation de la figure.

Nous présentons ici les principales méthodes symboliques de résolution de systèmes de contraintes géométriques.

Méthode de Ritt-Wu et bases de Gröbner L'approche algébrique exacte consiste à transformer formellement le système d'équations polynomiales S représentant les contraintes du problème. La résolution exacte de systèmes d'équations polynomiales fait appel à la théorie de l'élimination dans les anneaux de polynômes [Kal91, Wan01]. Dans ces techniques, on met sous forme triangulaire un système polynomial en cherchant une base particulière de l'*idéal engendré* par les polynômes. Rappelons cette notion. Si $\{f_1, \ldots, f_r\}$ est un ensemble de polynômes de l'anneau $A = K[x_1, \ldots, x_n]$, où K est un corps et $x_1, \ldots, x_n$ des variables indépendantes, l'ensemble $I = \{f = q_1 f_1 + \cdots + q_r f_r \mid q_i \in A \text{ avec } 1 \leq i \leq r\}$ est appelé l'idéal engendré par $\{f_1, \ldots, f_r\}$. On remarque que les zéros d'un ensemble de polynômes $P = \{f_1, \ldots, f_r\}$ sont également les zéros des polynômes de l'idéal I engendré par P. De plus, si l'on connaît une base P' de I, alors les polynômes de P' admettent les mêmes zéros que P. Les méthodes algébriques formelles consistent à calculer une telle base.

La méthode de Ritt-Wu [Cho88] repose sur un algorithme permettant de trouver à partir d'un ensemble de polynômes S des systèmes triangulaires ayant les mêmes racines que S. L'algorithme cherche une décomposition de S en ensembles caractéristiques irréductibles de polynômes, chaque ensemble contenant les polynômes générateurs d'un idéal premier dont les zéros sont des solutions de S [Kal91, Che92]. Cette méthode a été mise en œuvre en liaison avec la méthode de Lebesgue [Leb50] pour résoudre algébriquement des problèmes de constructions à la règle et au compas, lors d'un stage de DEA effectué au sein de notre équipe par G. Chen [Che92].

Une autre manière de mettre sous forme triangulaire un système d'équations repose sur l'utilisation des *bases de Gröbner*. L'idéal I engendré par l'ensemble S possède des bases, appelées bases de Gröbner, possédant d'intéressantes propriétés, comme, en choisissant un ordre lexicographique sur les variables, d'être sous forme triangulaire. On peut alors en dériver les solutions numériques du problème posé. A partir d'une famille génératrice quelconque d'un idéal I, l'algorithme de Buchberger [Buc85] permet de produire une base de Gröbner. Cet algorithme possède malheureusement une complexité exponentielle.

Cette technique de résolution de systèmes d'équations a été mise en œuvre par L.W. Ericson et C.K. Yap [EY88] pour un éditeur géométrique, LINETOOL. Dans ce système, les auteurs utilisent des structures de données particulières permettant un calcul plus efficace des bases de Gröbner. Toutefois, il ne s'agit là que d'heuristiques et la complexité de

l'algorithme reste exponentielle. Cette méthode est coûteuse mais l'accent est mis sur la précision et non sur l'interactivité de l'outil.

Approches à base de règles de construction géométriques. Les méthodes géométriques tirent parti de règles de construction et des propriétés de la géométrie euclidienne pour établir, par raisonnement, une construction des figures répondant aux contraintes. Cette démarche se concrétise assez naturellement sous la forme d'un système à base de connaissances. L'énoncé et les déductions ultérieures sont placés dans une base de faits initiale. Les règles de construction et certaines propriétés géométriques constituent une base de règles.

B. Brüderlin [Brü88] est l'un des premiers à mettre en œuvre ce type d'approche dans une optique CAO pour résoudre les problèmes de contraintes pour le dessin de pièces mécaniques. Les contraintes 2D acceptées sont de type :

$\mathrm{D}(P, Q, k)$	la distance entre les points P et Q est de k
$\mathrm{s}(P, Q, \alpha)$	α est l'angle entre le segment PQ et l'axe Ox du repère du plan
$\mathrm{d}(P, Q) = \mathrm{d}(R, S)$	les segments PQ et RS ont des longueurs égales
$\mathrm{c}(P, [x, y])$	le point P a pour coordonnées (x, y)

Brüderlin exprime un ensemble de règles de construction sous la forme d'un système de réécriture [Brü93]. Chaque règle de construction est traduite en une règle de réécriture.

Dans sa méthode, Brüderlin aborde également des problèmes en aval de la résolution. Il propose, pour choisir automatiquement parmi plusieurs solutions, d'introduire des contraintes du type "figure convexe" ou "segments qui ne se coupent pas", ce qui constitue une première amorce dans la recherche de solutions pertinentes.

B. Aldefeld, dans [Ald88] puis [AMRV92], se penche sur la même problématique et propose une architecture détaillée d'un système de résolution géométrique. Les contraintes forment la base de faits. Des informations sémantiques comme le degré de restriction sont attachées aux types de contraintes. Une base de règles contient les règles de construction. A partir de ces données, un moteur d'inférences avec une stratégie de chaînage avant est utilisé pour engendrer un plan de construction formel destiné à être interprété numériquement. Pour faciliter la tâche du concepteur, des contraintes métriques (angles droits, droites parallèles) sont extraites de l'esquisse et sont pondérées par un coefficient représentant une estimation de l'intention du dessinateur. Par l'ajout de ces contraintes, le problème est bien souvent sur-contraint. Le moteur d'inférences met alors en œuvre une heuristique basée sur la pondération des contraintes pour choisir celles qui doivent être résolues. Cependant, ces méthodes ne prennent pas en compte la possibilité de décomposer un problème.

1.2.3 Méthodes par décomposition

D'autres méthodes de résolution sont également souvent citées et utilisées : il s'agit d'approches par décomposition [Owe91, Sun86, VSR92, AA94, AAJM93, BFH+95]. Elles sont basées sur l'idée de découpage du problème en petites parties plus simples à résoudre, puis d'assemblage de ces parties une fois résolues localement. Pour ces résolutions locales, les méthodes par décomposition peuvent utiliser aussi bien des méthodes numériques que symboliques.

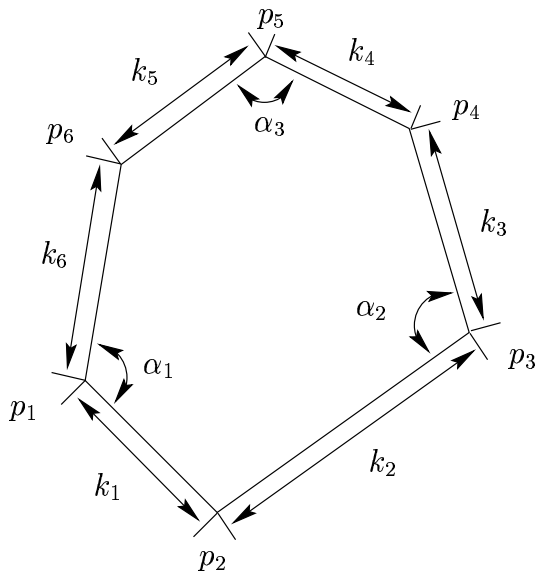


Fig. 1.5: Problème bien traité par une méthode de décomposition

Méthode d'Owen J.C. Owen fut l'un des précurseurs en matière de résolution par décomposition. La méthode qu'il propose [Owe91] tente de résoudre des problèmes de constructions à base de contraintes de distances et d'angles. D'autres contraintes géométriques, comme les contraintes de coïncidence, de tangence, de parallélisme, peuvent s'exprimer avec des valeurs particulières de distances et d'angles. Les entités considérées sont les points, les droites et les cercles. L'ensemble des contraintes est traduit en un graphe, appelé *graphe des contraintes*, dans lequel les sommets sont les entités et les arêtes les contraintes et relations d'incidence.

La méthode d'Owen consiste à décomposer un graphe de contraintes suivant toutes les paires d'articulations, afin de séparer le graphe en plusieurs composantes triconnexes, jusqu'à obtenir des sous-graphes représentant soit des cas de figures de triangles, soit des composantes triconnexes. Une seconde phase a pour but la résolution numérique des triangles contraints. Enfin, les triangles sont réassemblés suivant l'ordre inverse de la décomposition.

La Fig.1.6(a) montre le graphe de contraintes pour le problème de la Fig.1.5, qui est constituée d'un hexagone $p_1p_2p_3p_4p_5p_6$ contraint par 6 longueurs : $k_1 = p_1p_2, \ldots, k_6 = p_6p_1$, ainsi que 3 angles : $\alpha_1 = \widehat{p_6p_1p_2}, \alpha_2 = \widehat{p_2p_3p_4}, \alpha_3 = \widehat{p_4p_5p_6}$. Notons que cette méthode

considère des angles non orientés. On y voit apparaître à la fois les contraintes métriques, représentées par des arcs valués, et les contraintes d'incidence liant les points (p_i) aux droites (l_i). Ce graphe possède trois paires d'articulations : (p_6p_2), (p_6p_4) et (p_4p_2). La décomposition de ce graphe suivant les paires d'articulation aboutit aux quatre sous-graphes représentés à la Fig.1.6(b).

Cette méthode présente l'inconvénient d'être dépendante de la structure de données sous forme de graphe. Elle se limite au cas des contraintes binaires : la technique utilisée (recherche des paires d'articulation) limite la nature des sommets, qui ne peuvent être que des entités géométriques dont le degré de liberté vaut 2. Les graphes triconnectés non prévus ne sont pas résolus (dans la version présentée en 1991), comme par exemple un quadrilatère dont la résolution utilise la règle du parallélogramme.

Les problèmes liés à l'espace des solutions n'ont, à notre connaissance, pas été traités par les différents auteurs qui se sont intéressés à cette méthode.

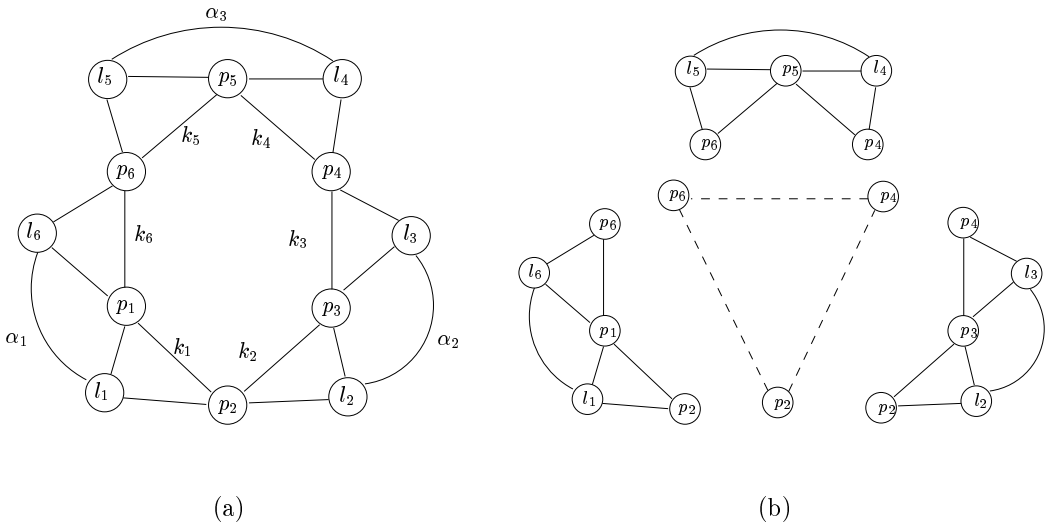


Fig. 1.6: Résolution selon la méthode d'Owen

Méthode de Sunde La méthode d'Owen relève d'une approche descendante où le problème est considéré dans sa totalité pour être fragmenté. La méthode initialement proposée par G. Sunde [Sun86] et étudiée plus avant et mise en œuvre par A. Verroust, F. Schonek et D. Roller [VSR92], procède au contraire de manière ascendante, et permet une résolution incrémentale. Les problèmes contiennent uniquement des contraintes de distance et d'angles, et les entités sont des points.

Cette méthode agit par application de règles de constructions géométriques. Une structure de données contient toutes les données de départ, sur lesquelles sont appliquées les règles au fur et à mesure de la résolution, afin de construire la figure petit à petit.

Elle est plus puissante que la méthode d'Owen, car elle ne se limite pas au cas biconnecté. Cependant, ses structures de données traduisent mal les contraintes de type tangence. De plus, à notre connaissance, la méthode de Sunde n'a pas été implantée de ma-

nière efficace alors que celle d'Owen a fourni un module de construction très rapide et largement utilisé dans les logiciels de CAO, notamment *D-Cubed* (http ://www.d-cubed.co.uk/) [Dcu93, Dcu95]. Ce produit étant industriel, les détails sont tenus secrets, et nous n'avons pas d'information quant à la sélection de solution ou l'exploration de l'espace de solutions.

Méthode de Ait-Aoudia L'étude de S. Ait-Aoudia [AA94] s'appuie une démarche analogue à celle d'Owen. Dans cette méthode, l'ensemble des contraintes est traduit en un graphe de contraintes. Pour la résolution, quatre traitements du graphe sont utilisés conjointement. Le premier traitement réalise une propagation des degrés de liberté comme dans Sketchpad [Sut63]. Le deuxième traitement est une décomposition suivant les paires d'articulation comme la pratique Owen. Le troisième s'occupe de résoudre un cas particulier. Quant au quatrième et dernier traitement, il intervient lorsque les autres traitements ont échoué et consiste à appliquer la méthode de Newton-Raphson.

La décomposition est ici d'approche descendante. Cette méthode est intéressante car elle illustre le fait que la décomposition permet l'utilisation conjointe de différentes méthodes.

Une autre méthode étudiée par S. Ait-Aoudia ([AAJM93]), H. Lamure et D. Michelucci ([LM97]), et plus généralement par Hoffmann *et al.* dans [CH01a, CH01b], est basée sur la recherche de composantes irréductibles d'un système d'équations associé au système de contraintes. Cette méthode s'appuie sur le théorème de König-Hall et sur la décomposition de Dulmage-Mendelsohn.

Méthode de Bouma, Fudos et Hoffmann La méthode présentée par W. Bouma, I. Fudos et C. Hoffman [BFH+95] procède par décomposition ascendante. Un système de résolution géométrique permet la construction de *clusters*. Cette notion correspond à celle de sous-figure résolue. Chaque cluster contient des points, des droites, des cercles, tous définis dans un repère.

Après la formation des clusters, deux cas peuvent se produire. Soit la résolution aboutit à la production d'un seul cluster contenant tous les éléments de l'énoncé et la construction est alors terminée avec succès, soit plusieurs clusters sont produits. Dans ce dernier cas, trois règles de construction sur les clusters sont appliquées pour les assembler. L'assemblage de tous les clusters pour n'en former qu'un seul n'est pas toujours possible mais ces trois règles permettent de traiter tous les cas résolus par le système de Verroust *et al.* [VSR92], excepté pour certaines règles, comme par exemple la règle du parallélogramme.

Cette méthode, qui rappelle en bien des points celle de Sunde, avec cependant quelques différences (elle est moins liée aux structures de données, les règles d'assemblage sont plus générales), est très intéressante car elle montre d'une part une résolution locale et d'autre part un mécanisme d'assemblage de clusters. Ce travail est d'autant plus notable ici que ses auteurs sont parmi les premiers à s'être intéressés de près au problème de la sélection de solutions, comme nous le verrons à la Section 1.3.

Plus récemment, des travaux ont été menés par J. Y. Lee et K. Kim ([LK98]) dans la continuité de cette méthode.

Un des intérêts de ces méthodes par décomposition est qu'elles permettent de résoudre un plus grand nombre de problèmes. C'est pourquoi notre équipe a choisi de suivre un type d'approche similaire, utilisant également un processus de décomposition/assemblage, pour créer le prototype *YAMS*, tout en privilégiant cependant autant que possible des méthodes de résolution locales symboliques et en évitant les méthodes itératives.

Ajoutons que, d'une façon générale, choisir la solution la plus pertinente par rapport à l'esquisse n'est pas une chose aisée lorsqu'on utilise une méthode numérique. Cela est dû au fait que la résolution doit le plus souvent être recommencée sur d'autres données initiales, lorsque la solution produite n'est pas jugée satisfaisante, et que l'on souhaite en visualiser une autre. Les approches numériques ne semblent donc pas les plus adéquates pour répondre à ce genre de problème.

Notre approche de l'exploration et de la sélection dans un espace de solutions s'appuie sur le résultat symbolique produit par *YAMS*. Un des buts de cette thèse est ainsi de montrer que l'approche symbolique géométrique utilisée dans *YAMS* se prête bien à l'exploration de l'espace des solutions. Mais ce n'est pas le seul avantage de cette approche. En effet, pour l'animation de figures par exemple, il est également préférable d'avoir une résolution produisant une solution générale, qui n'est pas à recalculer à chaque fois. Si l'on n'effectue que l'interprétation numérique finale lorsqu'on modifie les valeurs des paramètres, on obtient un processus capable de calculer en temps réel, et donc d'animer des figures de façon satisfaisante.

1.3 Sélection de solutions et exploration de l'espace des solutions

Le problème de la sélection de la solution la plus pertinente parmi toutes les possibilités proposées, et plus généralement celui de la ressemblance entre deux figures, sont des sujets qui n'ont à notre connaissance jamais fait l'objet jusqu'à présent d'une recherche spécifique approfondie. Il en va de même pour l'exploration de l'espace des solutions produites par la résolution. Certains auteurs mentionnent leur existence, sans pour autant, semble-t-il, en chercher une solution. D'autres ont toutefois tenté d'ébaucher une réponse.

Tout d'abord, remarquons avec R. Anderl et R. Mendgen [AM95] que l'importance de l'esquisse ainsi que des outils de saisie doit être soulignée. En effet, l'esquisse et l'outil qui permet de la saisir sont la base sur laquelle s'appuie la résolution. Celle-ci, et donc la ou les figures qui vont en résulter, sont alors dépendantes de leur précision. De même, la précision des éventuelles contraintes implicites qui peuvent être détectées à partir de l'esquisse (comme par exemple des perpendicularités) sont dépendantes de la sensibilité de l'outil, qui peut jouer un rôle prépondérant lors de la résolution, plus particulièrement lorsque des méthodes ne fournissant qu'une solution à la fois sont utilisées.

B. Brüderlin, quant à lui, évoque brièvement dans [SB91] l'existence de solutions ambiguës, puis propose l'introduction de contraintes supplémentaires pour les dissiper. Ceci

suppose d'une part l'intervention de l'utilisateur, et d'autre part un fort risque d'obtenir un système sur-contraint.

De son côté, J. Owen expose une première idée, déjà ébauchée par B. Aldefeld ([Ald88]), pour résoudre ces ambiguïtés dans [Owe91]. Il s'agit de comparer les placements relatifs des éléments de la figure solution avec ceux de l'esquisse donnée. Par exemple, un point se situe-t-il à droite ou à gauche d'une droite sur la figure ? P. Serré propose d'autres contraintes de nature topologique (appartenance à un secteur angulaire, à des portions du plan,...), et la prise en compte de contraintes de chiralité qui semblent plus discriminatoires [CRS98]. Cependant, pour ce type de méthodes, d'une part le nombre de critères à poser est très important, et d'autre part ces critères de comparaison sont à notre avis très insuffisants pour pouvoir choisir la solution souhaitée, comme nous le montrerons à la Section 4.1.1.

C. Hoffmann et ses collaborateurs sont parmi les premiers à s'intéresser de plus près au problème de solutions multiples que rencontrent les solveurs. Dans [BFH+95], ils évoquent trois possibilités afin de trouver parmi les solutions proposées quelle est celle qui était attendue par l'utilisateur. La première est de faire bouger des éléments jugés mal positionnés sur la figure solution présentée. Mais cette idée, leur paraissant mettre en jeu des opérations très complexes, n'a pas été exploitée. La deuxième est de sur-contraindre le système. Cependant, un système sur-contraint est effectivement plus restrictif, mais en contrepartie il peut également se révéler ambigu. C'est pourquoi cette voie n'a pas non plus été poursuivie.

Les méthodes automatiques ne leur paraissant pas prometteuses, ils proposent donc finalement comme troisième possibilité d'utiliser une combinaison entre le dialogue homme-machine et certaines heuristiques pour sélectionner la solution désirée. Tout d'abord, une première alternative est basée sur l'idée de comparaison des placements relatifs des éléments de la figure. Cependant, comme nous l'avons dit plus haut, ce critère est insuffisant.

Une deuxième alternative est de proposer à l'utilisateur un mode "incrémental" [Fud93], qui permet de choisir un "niveau" de la construction dans lequel on pourra sélectionner une des solutions possibles, afin de modifier la figure proposée si elle ne convient pas. Dans l'exemple (tiré de [Fud93]) de la Fig.1.7 (les figures présentées ici ont été importées via Internet et contiennent des textes un peu difficiles à lire), on a une esquisse cotée constituée de 4 segments et d'un arc dont le but évident est d'arrondir la jonction entre $Sg9$ et $Sg7$. Après avoir changé les contraintes d'angles de 70° à 30°, la solution proposée par leur solveur, qui est présentée Fig.1.8 (pour des raisons de place, l'arc a été tronqué), présente un "croisement". L'utilisateur choisit alors de faire une modification au niveau 7, où $Ar10$, $Sg7$ et $Pt6$ sont mis en évidence. L'utilisateur sélectionne parmi les 4 solutions possibles, la solution numéro 4 au lieu de la solution 2, et obtient la Fig.1.9. Puis, il effectue une seconde modification au niveau 4, où $Ar10$, $Sg9$ et $Pt8$ sont mis en surbrillance. L'utilisateur sélectionne ici la solution 1 au lieu de la 3 sur 4 possibles, pour obtenir la Fig.1.10. La bonne solution est enfin trouvée en prenant le complémentaire de l'arc $Ar10$, Fig.1.11.

Cette méthode ne semble cependant pas toujours très intuitive. En effet, la Fig.1.9 paraît visuellement très éloignée des Fig.1.8 et 1.11, dont elle représente pourtant une

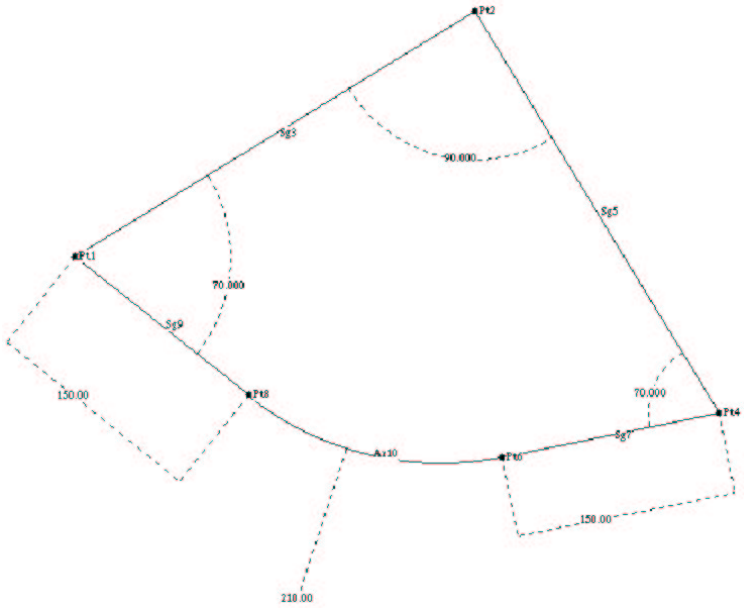


Fig. 1.7: Esquisse cotée : l'arc a pour but d'arrondir la jonction entre $Sg9$ et $Sg7$

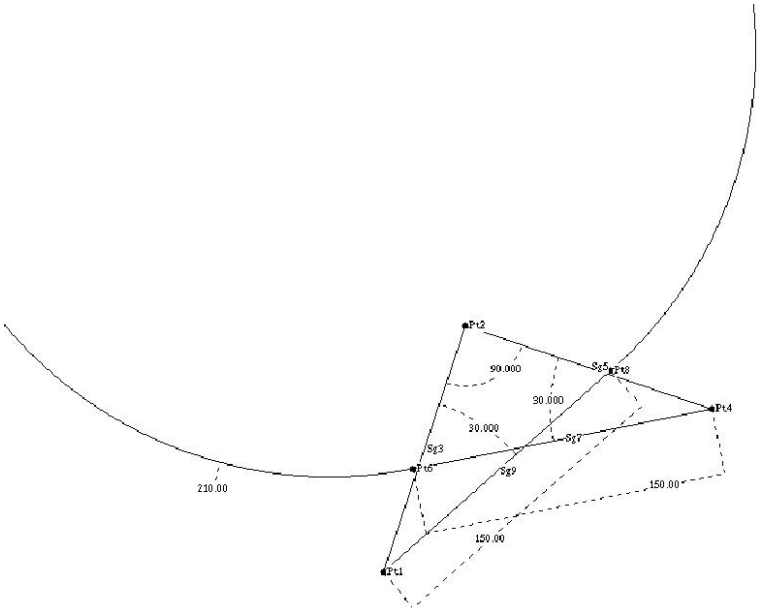


Fig. 1.8: Solution proposée avec angles contraints à 30°

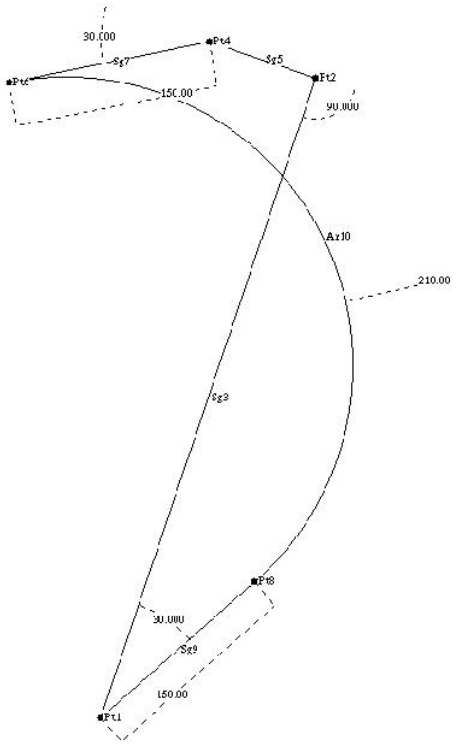


Fig. 1.9: Modification au niveau 7

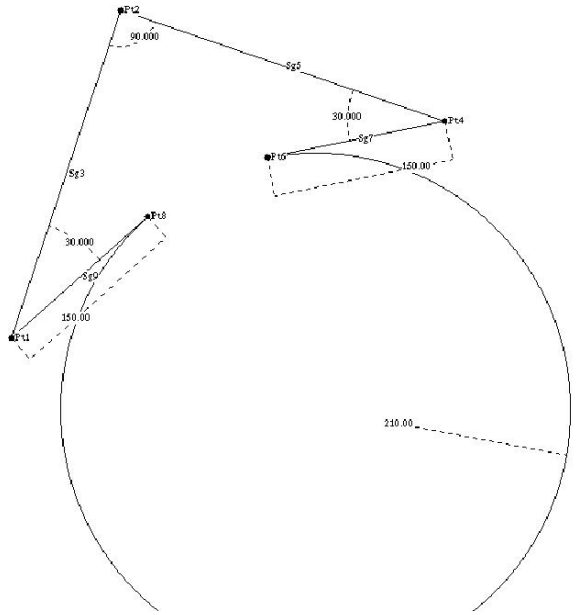


Fig. 1.10: Modification au niveau 4

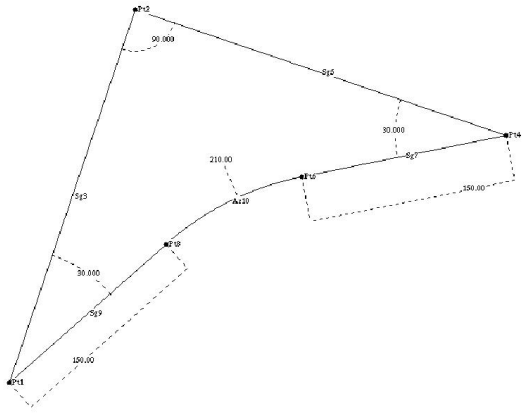


Fig. 1.11: Le complémentaire de l'arc achève la modification

étape intermédiaire. Un autre inconvénient de cette dernière méthode est qu'il semblerait qu'après chaque changement dans un niveau, la figure doive être "régénérée" avant de pouvoir effectuer un autre changement dans un autre niveau, ce qui peut rendre la tâche exagérément longue si la figure est compliquée. De ce point de vue, l'utilisation d'une méthode symbolique de résolution nous paraît plus prometteuse.

Une autre approche développée récemment dans le cadre des micro-mondes en géométrie étudie la continuité des construction dynamiques, afin d'éviter les "sauts" lors des déformations ([Kor99]). Bien que notre étude ait été menée indépendamment, nous verrons plus loin que la méthode de gel d'une branche est apparentée à cette notion.

Conclusion

Finalement, ce problème de la sélection de solution parmi plusieurs proposées, ou plus généralement de la recherche de la solution réellement attendue par l'utilisateur et non simplement d'une des solutions répondant à l'énoncé, n'a pas été traité jusqu'à présent de façon satisfaisante. C'est pourquoi nous nous proposons d'étudier comment résoudre ce problème, dans un premier temps de façon automatique, et dans un deuxième temps en faisant intervenir l'utilisateur, en l'aidant à explorer l'espace des solutions soit parce que la solution proposée par la méthode automatique ne le satisferait pas, soit parce que la méthode automatique n'aurait pas été capable de ne fournir qu'une seule solution, soit tout simplement parce que l'utilisateur serait curieux de voir les autres solutions possibles. Pour cela, nous tirons parti des propriétés de la méthode de résolution qui a été suivie dans *YAMS*, et qui sera expliquée dans le chapitre suivant.

Chapitre 2

Présentation de *YAMS*

Avant d'aborder notre sujet proprement dit, il nous semble indispensable de présenter le solveur qui se situe en amont. L'approche de résolution de systèmes de contraintes qui a été adoptée par le groupe de constructions géométriques automatisées du LSIIT est à la fois formelle et géométrique : les outils de résolution qui sont proposés sont basés sur la géométrie classique, plus particulièrement les constructions géométriques, et sur le raisonnement symbolique, précisément les systèmes à base de règles. Ils sont suffisamment spécialisés pour être efficaces dans la résolution de la plupart des problèmes de CAO. Ils ont dans un premier temps donné lieu à *Progé*, prototype de résolution de problèmes de constructions géométriques à la règle et au compas, qui a été proposé par P. Schreck au cours de son travail de thèse [Sch93], et inspiré des travaux de M. Buthion [But75]. Ces derniers sont toutefois plutôt orientés vers l'Enseignement Assisté par Ordinateur (EAO) et l'Intelligence Artificielle.

Cette approche a ensuite été concrétisée par un autre prototype nommé *YAMS*, plus orienté vers la CAO, qui associe un solveur géométrique au modeleur 3D à base topologique *Topofil*. *YAMS* travaille essentiellement en 2D, mais il possède également des fonctions lui permettant de construire des objets 3D par extrusion. Une description précise de *YAMS* ainsi que de son association avec *Topofil*peut être trouvée dans [DMS95, DMS97, DMS98, Mat97]. Nous ne présenterons donc dans ce chapitre que la partie solveur de *YAMS*, et plus particulièrement les processus de résolution et d'interprétation, de décomposition de problèmes et d'assemblage de sous-systèmes, ainsi que son architecture multi-agents avec tableau noir. Nous en ferons une présentation informelle, puisqu'il ne s'agit pas d'un travail effectué dans le cadre de cette thèse, mais de l'approche qui est le point de départ de notre propre étude.

Le solveur de *YAMS* prend en entrée une esquisse cotée, saisie par l'utilisateur grâce au modeleur *Topofil*. A notre sens, une solution doit avoir la forme d'un *plan de construction* général, c'est-à-dire une forme simplifiée de programme de construction ne contenant explicitement ni itération ni structure conditionnelle, à la différence de ceux produits par *Progé* [Sch93], et dans lequel aucune valeur numérique n'apparaît. Celui-ci peut ensuite être

interprété sur des données numériques particulières pour donner une ou des solutions numériques et graphiques. C'est ce mécanisme que nous allons expliquer dans les paragraphes qui suivent, ainsi que le fonctionnement général de *YAMS*. Nous conclurons ce chapitre par le traitement d'un exemple caractéristique avec *YAMS*, afin d'illustrer et de suivre au fur et à mesure le déroulement complet de la résolution d'un système de contraintes.

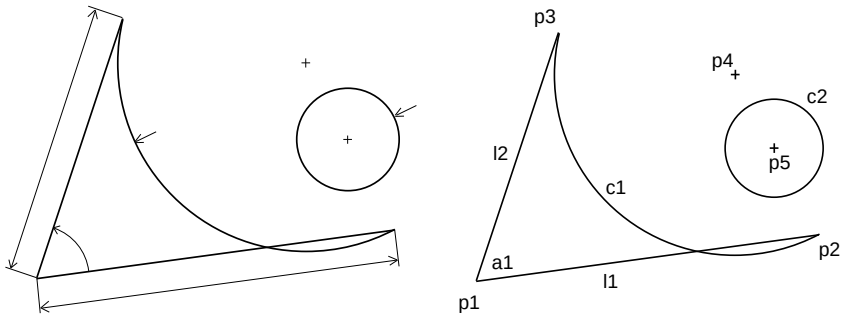
2.1 Résolution symbolique

Dans un premier temps, à partir de l'esquisse cotée, le solveur associe des identificateurs aux objets géométriques. Ces objets peuvent être de différentes *sortes géométriques* : *longueur*, *angle*, *point*, *droite*, *cercle*. Puis, il transforme les contraintes en paramètres formels pour former le système de contraintes géométriques, tel qu'il est défini ci-dessous.

Définition 1 (système de contraintes) *Un* système de contraintes géométriques *est un triplet $S = (U, X, C)$, où U est un ensemble de paramètres, X est un ensemble d'inconnues, et C est un ensemble de contraintes de la forme $C = \left\{p_1(U,X), \ldots, p_r(U,X)\right\}$, où chaque $p_i(U,X)$ est un terme prédicatif (ou prédicat), c'est-à-dire une contrainte, dont les variables sont dans X ou dans U. On impose que $X \cap U = \emptyset$.*

Lorsque l'utilisateur donne une esquisse cotée, il fournit en fait à *YAMS* des valeurs numériques, par exemple $distpp(p1, p2, 5)$, qui sont abstraites à la volée pour produire des contraintes, telles que $distpp(p1, p2, k2)$ et $k2 = initl(5)$. Plus précisément, l'utilisateur fournit un système instancié $S_u = (\emptyset, X, C_u)$, où u est le n-uplet des valeurs des cotes, et *YAMS* le transforme en un système $S = (U, X, C)$, où chaque symbole de U est l'abstraction d'une valeur de cote, et C est l'ensemble de contraintes donné par l'utilisateur où chaque valeur de cote est remplacé par un symbole de U. Il est possible de revenir à S_u depuis S en instanciant les symboles de U avec des valeurs de u. Un avantage de l'approche formelle est que l'on peut changer les valeurs de u sans pour autant affecter le processus de résolution.

Exemple 1 *Un exemple d'esquisse cotée est donné à la Fig.2.1(a). Le premier objet, constitué de deux segments et d'un arc de cercle, est soumis à des contraintes topologiques (incidence et adjacence) déduites de l'esquisse, ainsi qu'à des contraintes métriques données par l'utilisateur. Comme on peut le voir sur le schéma, ce sont une contrainte de longueur sur chaque segment, une contrainte d'angle entre les deux segments, et une contrainte sur le rayon de l'arc de cercle. Notons au passage que l'angle est orienté ; nous reviendrons sur cet aspect plus loin dans cette section, à la Remarque 2. Le deuxième objet de la figure est un cercle, dont on impose que le centre soit aussi le centre de l'arc de cercle du premier objet. Le rayon de ce cercle est aussi contraint, il doit être égal à la moitié du rayon de l'arc de cercle. L'esquisse telle qu'elle a été dessinée ne respecte pas ces contraintes métriques, mais respecte les contraintes d'incidence et d'adjacence. La Fig.2.1(b) montre comment YAMS associe des identificateurs aux objets géométriques. Par convention, p_i,*



(a) esquisse cotée (b) association d'identificateurs

Fig. 2.1: Exemple d'esquisse

c_i, l_i, k_i, *et* a_i *sont les noms choisis pour les points, cercles, droites, longueurs et angles, respectivement. Le système de contraintes correspondant à cet exemple est donné à la Table 2.1. On peut y trouver par exemple la contrainte* centre(c2,p5) *qui signifie que* p5 *est le centre de* c2, onl(p3,l2) *qui veut dire que le point* p3 *se situe sur la droite* l2, *ou encore* fixorgpl(p1,l1,p2) *qui fixe le point* p1 *à l'origine et la droite* l1 *sur l'axe* Ox, p2 *étant sur* l1. *Tous les types de contraintes peuvent être consultés à l'Annexe A.*

Tab. 2.1: Contraintes correspondant à l'exemple de la Fig.2.1

egal_p(p5, p4)	angle(p1, p2, p1, p3, a1)	onc(p2, c1)
centre(c2, p5)	distpp(p1, p2, k2)	onl(p3,l2)
centre(c1, p4)	distpp(p1, p3, k1)	onl(p2, l1)
radius(c2, k4)	fixorgpl(p1, l1, p2)	onl(p1, l2)
radius(c1, k3)	onc(p3, c1)	onl(p1, l1)

Remarque 1 *Nous distinguons deux types de contraintes : les contraintes dites* métriques *et les contraintes dites* booléennes. *Une contrainte est dite métrique si elle contient des paramètres dont on peut mesurer une instance sur l'esquisse, et telle qu'avec ces valeurs la contrainte est vérifiée sur l'esquisse. Ce sont par exemple des contraintes de longueur, d'angle, etc. Au contraire, une contrainte dite booléenne ne possède pas de paramètre et n'est généralement pas vérifiée sur l'esquisse. On peut citer par exemple les contraintes de tangence, d'égalité entre entités géométriques.*

Au prix d'une modification de l'énoncé, on peut parfois remplacer une contrainte booléenne par des contraintes métriques en ajoutant des paramètres. Mais l'énoncé ainsi obtenu n'est alors pas équivalent à l'énoncé initial, et ne correspond pas forcément à l'intention de l'utilisateur.

Avec les mêmes notations que précédemment, et dans le cas où toutes les contraintes

sont métriques, si u est le n-uplet des valeurs lues sur l'esquisse, alors l'esquisse est une solution de S_u.

Ensuite vient la phase de résolution formelle, qui a pour but de produire un procédé de construction des figures solutions. *YAMS* produit ainsi des définitions de la forme $y = g(u_1, \ldots, u_s, x_1, \ldots, x_k)$, qui assurent la correspondance entre des termes fonctionnels et des identificateurs. Les définitions sont regroupées pour former le plan de construction, résultat de la phase formelle. Ce plan indique comment et dans quel ordre doivent être construits les objets géométriques, afin de produire la figure. Plus formellement, un plan T est le résultat d'une résolution symbolique du système de contraintes S si T est un système triangulaire, composé de définitions de la forme $x_i = g(u_1, \ldots, u_s, x_1, \ldots, x_{i-1})$ où $1 \leq i \leq r$, r étant le nombre de variables de T, et $S \equiv T$, c'est-à-dire S et T ont les mêmes solutions réelles distinctes quelles que soient les valeurs des paramètres de u. Nous disons que T est un système *triangulaire résolu*. En instanciant les symboles des paramètres de U avec des valeurs d'un n-uplet u, on obtient $S_u \equiv T_u$.

Exemple 2 *Un plan de construction correspondant à l'esquisse cotée de l'Exemple 1, où les valeurs des paramètres ont été données par l'utilisateur, est présenté à la Table 2.2, à lire colonne après colonne à partir de la gauche. Afin de rendre cet exemple plus clair, quelques symboles fonctionnels et leurs profils ont été rassemblés à la table 2.3. L'Annexe A peut également être consultée.*

Tab. 2.2: Plan de construction correspondant à l'esquisse cotée de la Fig.2.1

k4 = initl(200)	p1 = initp(0,0)	c4 = mkcir(p1,k1)
k3 = initl(400)	l1 = initd(p1,0)	p3 = interlc(l1,c3)
a1 = inita(1.570796)	l2 = lpla(p1,l1,a1)	c1 = medradcir(p2,p3,k3)
k2 = initl(300)	c3 = mkcir(p1,k2)	p5 = centre_of(c1)
k1 = initl(200)	p2 = interlc(l1,c3)	c2 = mkcir(p5,k4)

Remarque 2 *Notons que YAMS traite des contraintes d'angles orientés. Dans les définitions du plan de construction, cette particularité des angles ne transparaît pas, et la construction en elle-même n'en tient pas compte. Elle reste cependant représentée par les valeurs des instanciations des paramètres, dont le signe est fonction de l'orientation de l'angle. Par exemple, si l'on utilise la définition* $l2 = lpla(p1, l1, a1)$ *pour construire une droite* $l2$ *passant par le point* $p1$*, et formant un angle* $a1$ *avec la droite* $l1$*, alors le résultat sera différent selon que l'on a instancié* $a1$ *auparavant par* $a1 = inita(1.2)$ *ou bien par* $a1 = inita(-1.2)$*. De plus, l'ensemble des contraintes, qui rendent compte de l'orientation, est à tout moment accessible en vue d'une vérification de la satisfaction de celles-ci. Ainsi, la contrainte* $angle(p1, p2, p1, p3, a1)$ *est différente de la contrainte* $angle(p1, p3, p1, p2, a1)$*, puisqu'elle représente un angle opposé.*

Les distances quant à elles ne sont pas signées. Par exemple, $distpp(p1, p2, k)$ *est équivalent à* $distpp(p1, p2, -k)$*.*

TAB. 2.3: Quelques symboles fonctionnels et leurs profils

symbole	profil	commentaire
interlc	$line \times circle \rightarrow point$	intersection cercle-droite
mkcir	$point \times long \rightarrow circle$	cercle de centre et rayon connus
lpla	$point \times line \times angle \rightarrow line$	droite passant par un point et faisant un certain angle avec une droite connue
medradcir	$point \times point \times long \rightarrow circle$	cercle passant par deux points et de rayon connu
intercc	$circle \times circle \rightarrow point$	intersection cercle-cercle

2.2 Décomposition en sous-figures

Le solveur de *YAMS* est capable de décomposer le système de contraintes initial en systèmes plus petits appelés *sous-systèmes*. Dans *YAMS*, cette décomposition est un processus ascendant : les sous-systèmes sont découverts au cours de la résolution. La philosophie est de résoudre les sous-figures indépendamment en des sous-plans, en utilisant un repère pour chacune, puis de les coller ensemble grâce à un mécanisme appelé *assemblage*, en les déplaçant dans un repère commun. La décomposition du système de contraintes géométriques est en effet basée sur l'invariance des contraintes par le groupe des déplacements.

YAMS utilise une collaboration entre différentes méthodes locales, privilégiant les méthodes symboliques comme celles évoquées au Chapitre 1 (pour l'instant, essentiellement des systèmes à base de connaissances, ...). Ces méthodes sont coordonnées entre elles par une architecture multi-agents avec tableau noir.

L'emploi de méthodes géométriques ne permet pas toujours de résoudre complètement un système de contraintes. Dans ce cas, on est amené à manipuler des sous-systèmes de contraintes, résolus par différentes méthodes, et à les réunir dans le but de former une seule forme triangulaire résolue contenant la totalité d'un système. Il existe souvent plusieurs façons de décomposer un système de contraintes. L'obtention d'une décomposition particulière en sous-systèmes est généralement due au choix du repère de positionnement.

Lorsqu'un sous-système S' de S est résolu, on le remplace par son *bord*. Le but est de produire de nouvelles contraintes métriques, par construction, conformes à l'énoncé, et qui portent sur les inconnues communes à S' et à $S - S'$. Le bord est donc un sous-système B de S, dont l'ensemble des inconnues est constitué des inconnues communes à S' et à $S - S'$, les paramètres sont ses inconnues localisées (pour une définition plus détaillée, voir [Mat97] et [DMS98]). Par exemple, sur la Fig.2.2, les bords ont été représentés en pointillés. Étant donné que les sous-systèmes sont invariants par déplacement, on peut donc effectuer des déplacements pour les assembler, en utilisant leurs bords. On ajoute donc pour cela une nouvelle sorte à l'univers géométrique : la sorte *déplacement*.

Exemple 3 *Sur la Fig.2.2(a) est représentée l'esquisse d'une figure impossible à résoudre sans décomposer le système. En 2.2(b), le système a été décomposé en trois sous-systèmes. Le bord de chaque sous-figure est tracé en pointillés. Après transformations appliquées sur deux des sous-figures, on obtient une figure assemblée (2.2(c)). Le résultat final est présenté en 2.2(d).*

La décomposition du système de contraintes en sous-systèmes implique l'utilisation d'une numérotation particulière pour différencier les objets selon le repère dans lequel on les considère. A chaque repère différent, dans lequel est construite une sous-figure, est associé un entier. Par exemple, le point $p1$ dans le repère associé à l'entier 1 sera identifié par $p1.1$, tandis que le même point dans le repère commun par exemple associé à l'indice 2 sera noté $p1.2$. Cette notation sera plus détaillée dans la Section 3.1.2. Cependant, dans le reste de ce document, afin de ne pas alourdir la notation, les indices seront omis lorsqu'aucune décomposition n'a été nécessaire à la résolution d'un système de contraintes, et que par conséquent tous les éléments sont construits dans le même repère.

Cette notation nous permettra plus tard de repérer plus facilement les parties du plan de construction correspondant aux différentes sous-figures, en vue de tris, ou d'opérations à effectuer ou non sur certaines sous-figures.

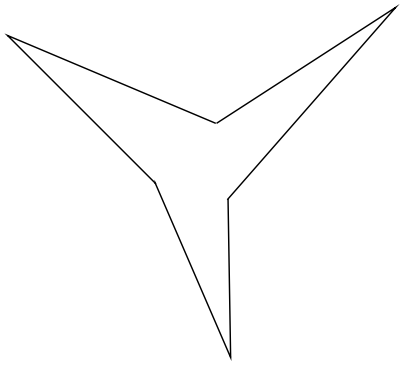
2.3 Interprétation numérique de base

Dans une seconde phase, les cotes données par l'utilisateur sont utilisées comme paramètres effectifs pour l'interprétation numérique du plan de construction produit par la phase de résolution symbolique. Nous reviendrons en détail sur cette phase un peu plus loin dans ce mémoire. On distingue plusieurs niveaux dans l'interprétation numérique : l'interprétation des sortes géométriques, éléments de base, l'interprétation des termes fonctionnels, et l'interprétation des prédicats.

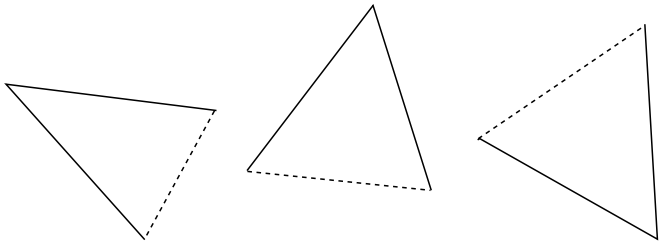
2.3.1 Interprétation des sortes géométriques

Les sortes sont interprétées par un n-uplet de réels. Plus précisément, l'interprétation d'une sorte α est un ensemble E_α en bijection avec une partie de $\mathbb{R}^n$. Voici la forme des n-uplets pour chaque sorte :

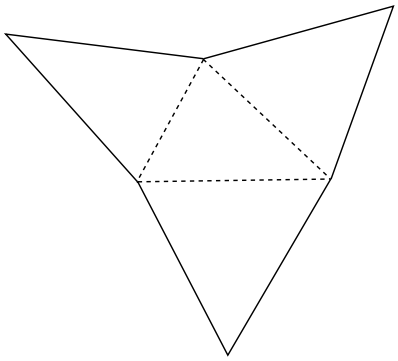
Longueur :	(l)	longueur réelle
Angle :	(a)	angle en radians compris dans l'intervalle $]-\pi, \pi]$
Point :	(x, y)	coordonnées dans le plan muni du repère $(O, \vec{\imath}, \vec{\jmath})$
Droite :	(a, d)	angle entre la droite et l'axe Ox du repère du plan et distance de la droite algébrique à l'origine O
Cercle :	(x, y, r)	coordonnées du centre et rayon
Déplacement :	(x_1, y_1, a, x_2, y_2)	



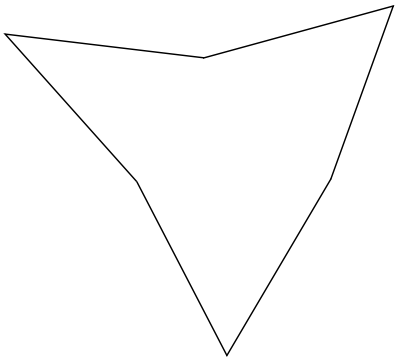
(a) esquisse



(b) 3 sous-figures résolues localement et leurs bords



(c) 3 sous-figures assemblées grâce à leurs bords



(d) figure finale

Fig. 2.2: Processus de décomposition/assemblage

(x_1, y_1) coordonnées de l'origine du repère source, (x_2, y_2) coordonnées de l'origine du repère destination, a angle orienté dans le sens source-destination entre les directions des deux repères

2.3.2 Interprétation des termes fonctionnels

Les termes fonctionnels utilisés produisent dans la plupart des cas une solution ou aucune, selon les instances des paramètres, mais peuvent éventuellement aussi produire plusieurs solutions. C'est par exemple le cas de l'intersection entre une droite et un cercle, symbolisée par *interlc*, qui peut dans certaines circonstances produire deux points. De même, *medradcir*, qui construit un cercle de rayon connu passant par deux points connus produit généralement deux cercles distincts. Chaque symbole fonctionnel est donc associé à ce qu'on appelle une *multifonction* numérique, qui est une fonction renvoyant zéro, un ou plusieurs résultats.

L'existence de multifonctions dans un plan de construction introduit des choix dans le processus d'interprétation. Une fois que les valeurs sont assignées aux paramètres, on peut considérer l'interprétation comme la construction d'un arbre étiqueté avec les valeurs numériques. L'interprétation d'une définition de la forme $y = g(u_1, \ldots, u_s, x_1, \ldots, x_r)$ produit un embranchement de degré k si la multifonction g a un *maximum* de k résultats. A la fin de l'interprétation, l'arbre représente l'espace des solutions, et une solution correspond aux étiquettes d'une branche.

Durant l'évaluation, il peut arriver qu'une multifonction ne fournisse aucun résultat. Par exemple, si les valeurs sont telles qu'on cherche l'intersection entre un cercle et une droite non secants, alors la multifonction *interlc* ne produira aucun résultat. Dans ce cas, l'interprétation s'arrête pour cette branche. Il se peut aussi que le nombre de solutions produites n'atteigne pas le maximum. Nous faisons donc une distinction entre l'*arbre des solutions potentielles*, que nous appellerons également dans la suite de ce mémoire *arbre des possibilités*, et l'*arbre des solutions réelles*, que nous abrègerons par *arbre des solutions*. L'arbre des solutions est un extrait de l'arbre des possibilités. Notons que même si l'arbre des solutions est plus petit que l'arbre des possibilités, il peut cependant croître très vite et devenir très large, comme nous l'évoquons dans l'Exemple 5. Bien entendu, cet arbre n'est pas réellement construit dans la pratique, mais exploré par backtracking.

Exemple 4 *Un arbre des solutions produit pour notre exemple, après assignation des paramètres, est présenté Fig.2.3, à droite du plan de construction. Chaque nœud de l'arbre correspond au résultat pour un identificateur, qui est écrit avec un chiffre entre parenthèses qui permet une distinction entre les solutions des multifonctions. Une définition précise de numérotation des solutions d'une multifonction sera donnée au Chapitre 4, à la Section 4.2.2. Les huit solutions (ou branches) sont ensuite tracées à la Fig.2.4.*

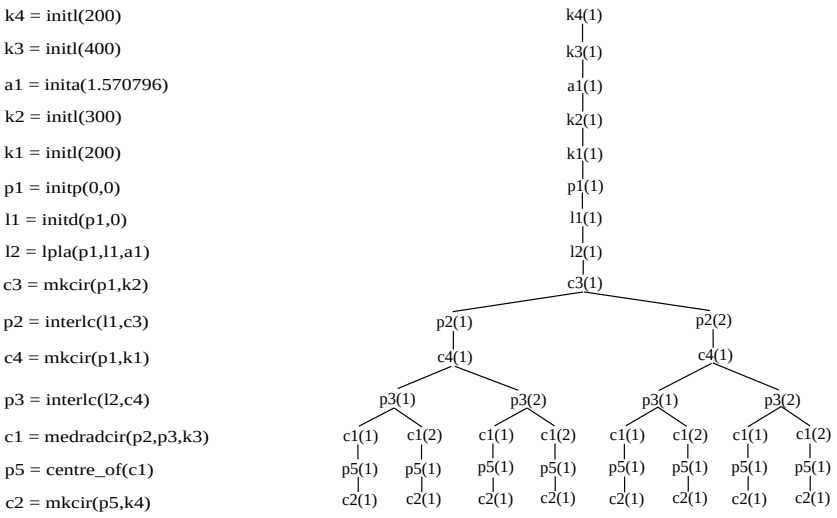


Fig. 2.3: Plan de construction correspondant à la Fig.2.1, et son arbre des solutions

2.3.3 Interprétation des prédicats

L'interprétation des prédicats n'est pas utilisée dans la phase d'interprétation à proprement parler. Elle est utilisée dans la vérification numérique des contraintes qui intervient juste en conclusion de cette phase.

En effet, le plan de construction qui a été généré permet de construire non seulement toutes les solutions, mais aussi d'autres figures qui sont des "fausses solutions", provenant par exemple de l'absence de prise en compte de l'orientation durant la phase formelle. Les fausses solutions peuvent rapidement être éliminées grâce à ce simple test qu'est la vérification numérique des contraintes (à ϵ près), puisqu'elles ne satisfont pas les contraintes données. Par exemple, parmi les solutions de l'Exemple 4 données à la Fig.2.4, quatre peuvent être éliminées car l'angle $a1$ est l'opposé de ce qui est demandé dans les contraintes. Il s'agit des deux solutions en haut à droite et des deux solutions en bas à gauche. Notons que dans cette phase de filtrage, les solutions redondantes car identiques à un déplacement près sont également éliminées. Cependant, comme le montre l'Exemple 5, cela peut s'avérer encore insuffisant. D'autres heuristiques sont alors nécessaires pour élaguer l'arbre des solutions de façon drastique, en éliminant les figures qui ne "ressemblent" pas à l'esquisse. Ces heuristiques, ainsi que notre notion de "ressemblance" entre figures, sont détaillées au Chapitre 4.

Ajoutons que l'interprétation des prédicats a déjà été utilisée dans la méthode analytique de résolution pour poser le système d'équations.

Exemple 5 *La Fig.2.5(a) montre une esquisse composée de 15 triangles adjacents dont on demande qu'il soient équilatéraux. Les contraintes imposent que les longueurs de tous leurs côtés soient égales à une constante donnée. Ce type de configuration a notamment été*

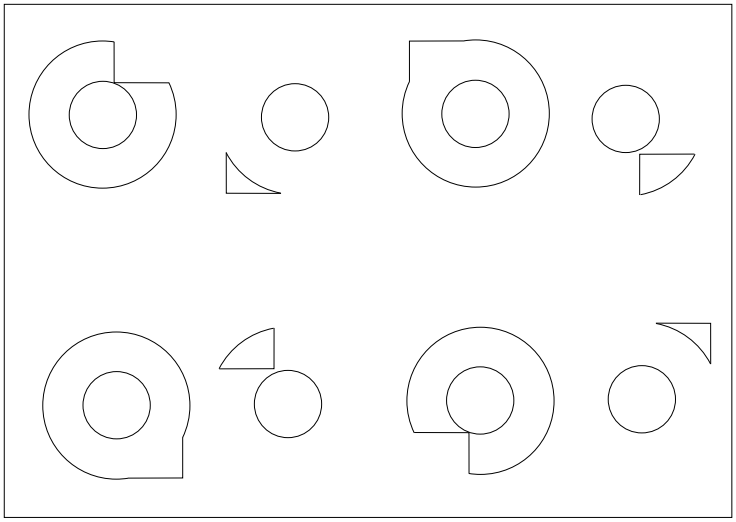


Fig. 2.4: Les solutions engendrées

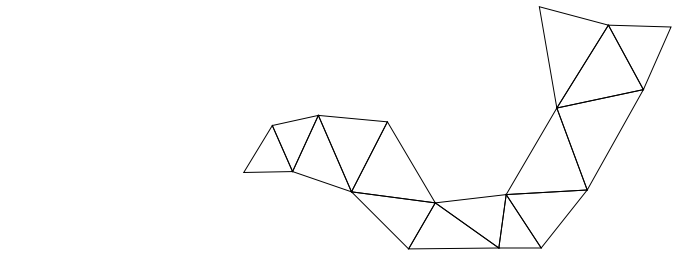
proposé par Owen [Owe91], et est connu pour avoir 2^{p-2} *solutions distinctes, où p est le nombre de points. Dans notre cas, avec* 17 *points, nous obtenons* 32768 *solutions. La plupart de ces solutions contiennent des triangles superposés, puisque leurs côtés sont égaux. Nous en présentons quelques-unes à la Fig.2.5(b). L'espace des solutions ne peut être réduit car toutes les figures vérifient les contraintes.*

Le grand avantage de l'approche utilisée par *YAMS* est que lorsque l'on souhaite essayer d'autres valeurs pour les paramètres, la phase de résolution symbolique n'est pas remise en cause. Seule la phase d'interprétation numérique est relancée en changeant l'instanciation des paramètres, ce qui permet un gain de temps considérable. Cela permet également d'animer des figures contraintes, car l'interprétation est un processus instantané. En faisant varier les valeurs des paramètres au cours du temps, et en réinterprétant à chaque changement, on obtient un mouvement en temps réel. Une articulation sera obtenue par exemple en faisant varier la valeur d'un angle, un coulissement en faisant varier une distance.

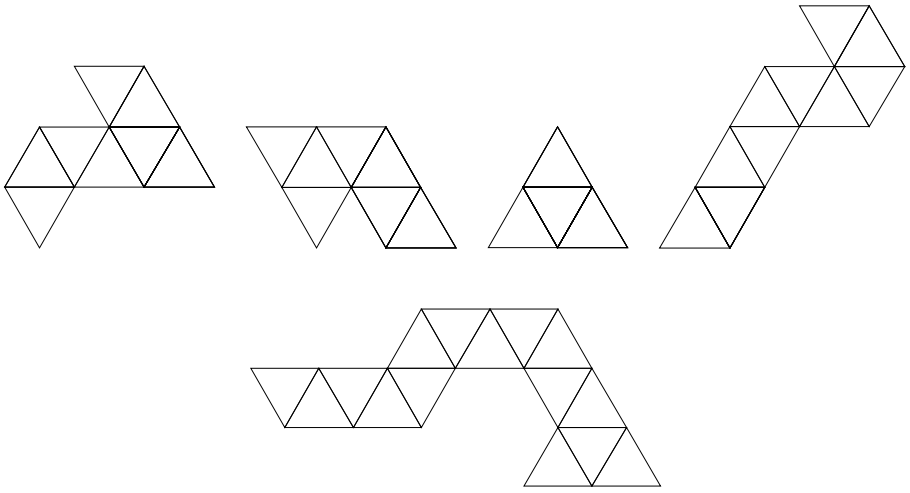
Nous avons donc vu comment *YAMS* traite la résolution d'un système de contraintes, et quelles méthodes il utilise tout au long du processus pour produire les solutions. Nous allons maintenant donner quelques indications sur l'architecture de *YAMS*.

2.4 Architecture multi-agents

Comme nous l'avons déjà évoqué, l'originalité de cette approche réside dans sa capacité à fédérer des méthodes de résolution locales très différentes. Actuellement, deux d'entre elles sont des systèmes à base de connaissances, la troisième est une méthode analytique. C'est grâce à une architecture multi-agents du système que ces méthodes sont coordonnées entre



(a) esquisse



(b) cinq solutions parmi 32768

Fig. 2.5: Exemple des 15 triangles équilatéraux

elles.

Cette architecture est présentée à la Fig.2.6. On y trouve trois grandes composantes : le tableau noir, les agents, le superviseur. Le tableau noir contient les informations communes et correspond ainsi à une mémoire partagée. Ensuite, les agents ont en charge le traitement des informations du tableau noir. Pour finir, le superviseur contrôle les informations du tableau noir et active les agents.

Dans ce système, le tableau noir tient une place importante. Il correspond à la mise en œuvre d'un univers géométrique. Il contient les systèmes de contraintes, de l'énoncé jusqu'aux formes triangulaires résolues. D'un point de vue logique, le tableau noir correspond à l'univers, augmenté tout au long de la construction de termes clos provenant des agents et du superviseur. Les termes et formules contenus dans le tableau noir sont d'une part les axiomes généraux de l'univers géométrique, et d'autre part les termes clos qui sont, initialement, des axiomes correspondant à l'énoncé auxquels s'ajoutent ensuite des théo-

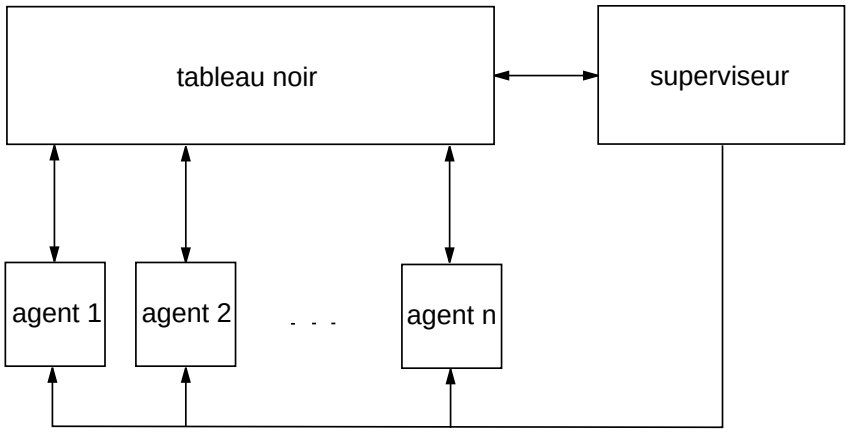


Fig. 2.6: Système multi-agents

rèmes correspondant aux bords des sous-figures et aux connaissances sur les constructions courantes. Les théorèmes entrant dans le tableau noir induisent eux-mêmes de nouveaux termes par l'application du modus ponens avec les axiomes généraux, permettant ainsi de prendre en compte la permutabilité pour la contrainte de distance ou encore la relation de Chasles.

A tout moment, le tableau noir peut être consulté et complété par chaque méthode de résolution locale. Au cours de la résolution, le tableau noir contient des informations sur les parties déjà résolues, notamment le bord. Après chaque définition constructive d'un nouvel élément, le superviseur détecte les assemblages possibles avec les autres sous-plans déjà obtenus. S'il en existe, il interrompt la méthode en cours pour les réaliser et met à jour le tableau noir. Ainsi, l'assemblage est fait *au plus tôt*. La résolution formelle s'arrête lorsque toutes les inconnues de l'énoncé sont dans un même plan de construction, ou sinon lorsque plus aucune méthode ne parvient à ajouter de nouvelles définitions. Dans le premier cas la résolution formelle *réussit*, dans le deuxième elle *échoue*.

2.5 Traitement d'un exemple par *YAMS*

Afin d'illustrer le fonctionnement du propotype *YAMS*, nous allons présenter un exemple de contruction géométrique et sa résolution par *YAMS*. Nous pourrons ainsi suivre au fur et à mesure l'exécution du processus dans sa totalité. De plus, la compréhension intuitive des mécanismes mis en œuvre dans cette résolution permettra d'appréhender plus facilement les idées développées ensuite dans le reste de ce mémoire. Pour cela, nous allons reprendre l'exemple déjà évoqué à la Section 1.1. Cet exemple est développé plus en détail dans [Mat97] et [DMS98].

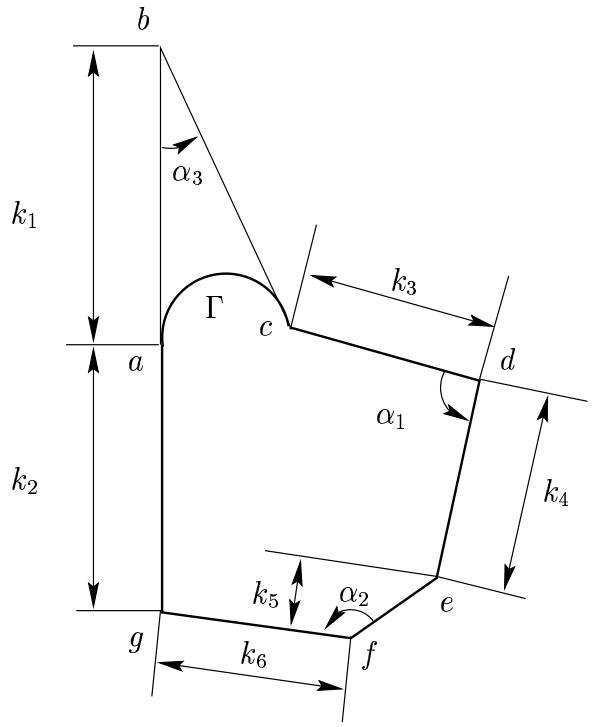


Fig. 2.7: Un système de contraintes

2.5.1 Enoncé

Tout d'abord, nous rappelons à la Fig.2.7 l'esquisse cotée déjà présentée à la Section 1.1. Cette figure précise toutes les contraintes imposées à notre esquisse, l'hexagone *cdefga*, en faisant intervenir le point extérieur b. Certaines contraintes sont indiquées sur la figure par la cotation, et d'autres, comme les contraintes de tangence, sont implicites. Pour notre exemple, l'énoncé se compose de contraintes données littéralement plus bas, complété par des contraintes topologiques issues du dessin lui-même, comme l'incidence points-droites et points-cercles. Notons la présence d'un cercle de rayon indéterminé.

distance du point a au point $b = k_1$
distance du point a au point $g = k_2$
distance du point c au point $d = k_3$
distance du point d au point $e = k_4$
distance du point e à la droite $fg = k_5$
distance du point f au point $g = k_6$
angle orienté entre les droites dc et $de = \alpha_1$
angle orienté entre les droites fe et $fg = \alpha_2$
angle orienté entre les droites ba et $bc = \alpha_3$
les points a, b, g sont alignés

la courbe Γ est un arc de cercle
la droite ab est tangente à Γ en a
la droite bc est tangente à Γ en c

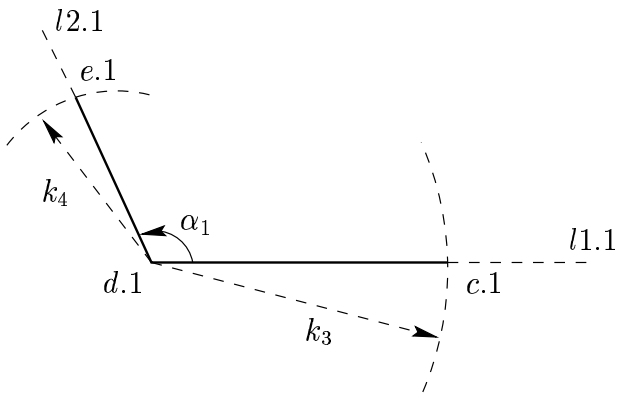


Fig. 2.8: Figure auxiliaire dans un premier repère

2.5.2 Construction par décomposition/assemblage

Le système de contraintes posé ne peut être résolu en une fois. Il est nécessaire de recourir à une décomposition en plusieurs sous-systèmes qui sont résolus localement puis assemblés. Une solution particulière pour chacun de ces sous-systèmes est schématisée respectivement Fig.2.8, 2.9 et 2.10. Leurs repères ont été numérotés par 1, 2 et 3, donc tous les éléments de ces constructions ont été respectivement suffixés par .1, .2, et .3.

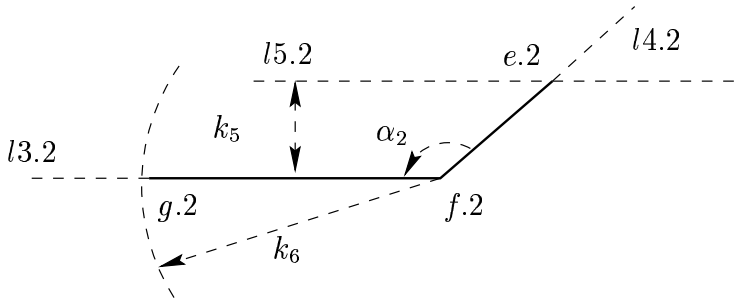


Fig. 2.9: Figure auxiliaire dans un deuxième repère

Il est ici possible de continuer la construction en utilisant les propriétés métriques des sous-figures déjà construites. Ainsi, par la sous-figure auxiliaire $(c.1, d.1, e.1)$, on sait que la distance $c.3e.3$ est égale à $c.1e.1$. De même, par la sous-figure auxiliaire $(e.2, f.2, g.2)$, on a $g.3e.3$ qui est égal à $g.2e.2$. Le point $e.3$ est ainsi déterminé comme étant l'intersection des cercles de centres respectifs $c.3$ et $g.3$ et de rayons respectifs $c.1e.1$ et $g.2e.2$. Il est ainsi possible de compléter la sous-figure auxiliaire 3, comme montré Fig.2.11.

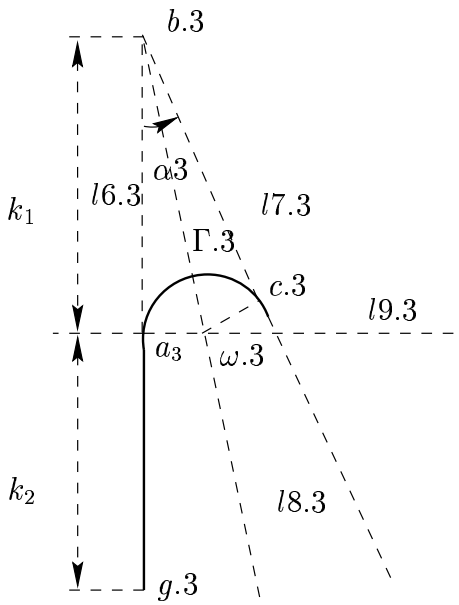


Fig. 2.10: Figure auxiliaire dans un troisième repère

La construction de la figure se termine alors en assemblant les sous-figures 1 et 2 à la figure 3, grâce au calcul des déplacements uniques $\varphi_{1\to 3}$ et $\varphi_{2\to 3}$ tels que $\varphi_{1\to 3}(c.1, d.1, e.1) = (c.3, d.3, e.3)$ et $\varphi_{2\to 3}(e.2, f.2, g.2) = (e.3, f.3, g.3)$. Cet assemblage est schématisé à la Fig.2.12.

La façon de construire chaque sous-figure est détaillée dans un plan de construction partiel, produit par la résolution locale, dans lequel les éléments sont suffixés par le numéro du repère correspondant. La concaténation des plans de constructions partiels de chacune des trois sous-figures 1, 2 et 3 forme un plan de construction général ramené à la sous-figure 3. Celle-ci est une solution particulière, dans le dernier repère fixé, du problème initial. Toutes les figures obtenues par déplacement sont également des solutions du problème. Remarquons enfin qu'au lieu du repère de la troisième figure, le repère de l'une quelconque des deux autres aurait aussi bien pu être privilégié, produisant ainsi deux autres solutions particulières mais identiques à un déplacement près.

2.5.3 *YAMS* en pratique

La Fig.2.13 montre l'esquisse cotée telle qu'elle est saisie par l'utilisateur au moyen du modeleur intégré au prototype. Les contraintes d'angles et de distances entre points sont indiquées à l'écran par des lignes de cote et lignes d'attache mais sans qu'apparaissent la valeur des cotes. Une première série de contraintes d'ordre topologique peut être directement déduite de l'esquisse, donnant lieu à l'ensemble de prédicats suivant :

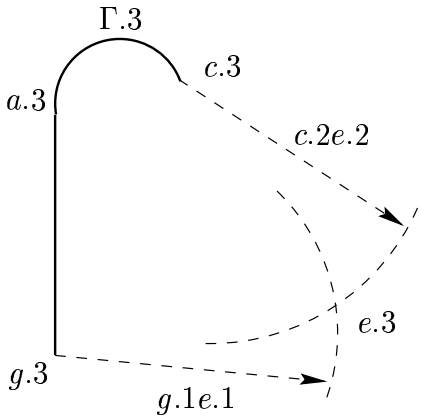


Fig. 2.11: Figure auxiliaire dans un troisième repère

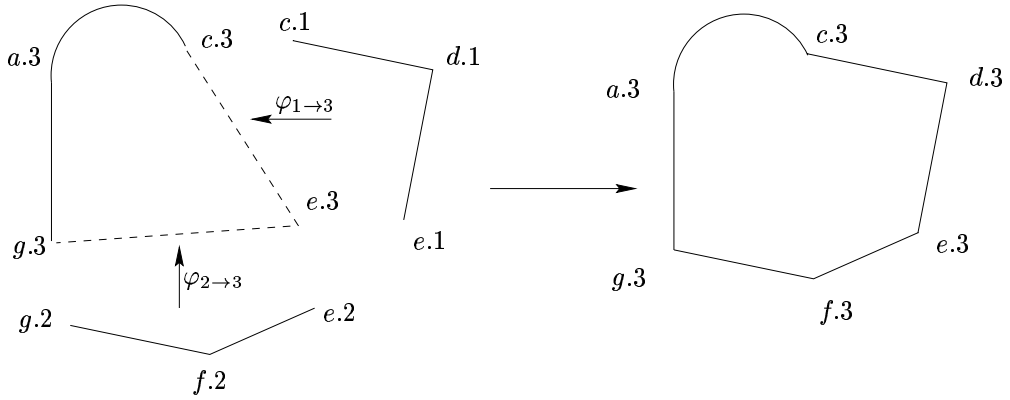


Fig. 2.12: Assemblage des sous-figures

```
onl(p1, l1)          *** contrainte d'incidence point-droite : p1 est sur l1
onl(p1, l2)
onl(p2, l2)
onl(p2, l3)
onl(p3, l1)
onl(p3, l4)
onl(p4, l4)
onl(p4, l5)
onl(p5, l3)
onl(p6, l5)
onl(p5, l6)
onl(p6, l6)
onl(p7, l6)
onc(p4, c1)          *** contrainte d'incidence point-cercle : p4 est sur c1
onc(p7, c1)
centre(c1, p8)       *** contrainte issue du plongement : p8 est le centre de c1
```

Le nommage des éléments est montré à la Fig.2.14. Les noms sont attribués de façon automatique. Rappelons que, par convention, les noms des points commencent toujours

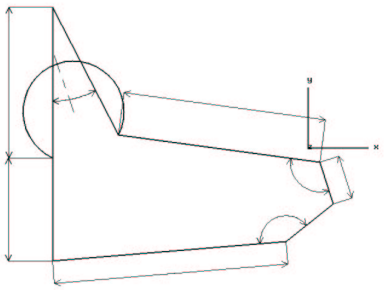


FIG. 2.13: Esquisse cotée

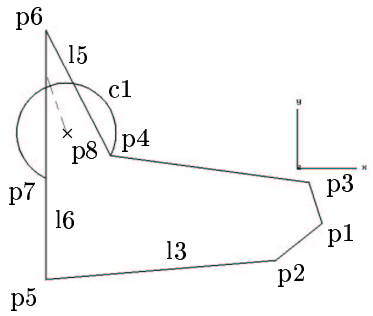


FIG. 2.14: Attribution automatique des noms symboliques

par la lettre p, les droites par l et les cercles par c.

Les contraintes placées par l'utilisateur fournissent un deuxième ensemble de prédicats, présenté ci-dessous, qui peut être ajouté au premier. Les longueurs sont associées automatiquement à des noms commençant par la lettre k, et les angles à des noms commençant par la lettre a.

```
distpp(p1, p3, k1)          *** les points p1 et p3 sont distants de la longueur k1
distpp(p3, p4, k2)
distpp(p7, p6, k3)
distpp(p5, p7, k4)
distpp(p5, p2, k5)
distpl(p1, l3, k6)          *** le point p1 et la droite l3 sont distants de la longueur k6
angle(p3, p1, p3, p4, a1)   *** les vecteurs p3p1 et p3p4 forment un angle de mesure a1
angle(p2, p5, p2, p1, a2)
angle(p6, p7, p6, p4, a3)
tgcl(c1, l5)                *** le cercle c1 et la droite l5 sont tangents
tgcl(c1, l6)
```

Enfin, les valeurs numériques saisies par l'utilisateur sont ajoutées sous forme de prédicats d'initialisation de longueurs et d'angles à des valeurs réelles :

```
k5 = initl(298.0)
```

k4 = initl(200.0)
k3 = initl(300.0)
k2 = initl(206.0)
k1 = initl(170.0)
k6 = initl(70.0)
a3 = inita(0.73)
a2 = inv_angle(inita(2.40))
a1 = inv_angle(inita(1.57))

A partir de cet ensemble de prédicats formant le système de contraintes, *YAMS* résout le problème grâce à son système multi-agents qui, après décomposition puis assemblage comme expliqué à la Section 2.5.2, produit le plan de construction suivant, dont les éléments sont suffixés en fonction de leur appartenance à une sous-figure :

valeurs de cotes
$$\begin{cases} k5 = \text{initl}(298.0) \\ k4 = \text{initl}(200.0) \\ k3 = \text{initl}(300.0) \\ k2 = \text{initl}(206.0) \\ k1 = \text{initl}(170.0) \\ k6 = \text{initl}(70.0) \\ a3 = \text{inita}(0.733038) \\ a2 = \text{inv\_angle}(\text{inita}(2.40)) \\ a1 = \text{inv\_angle}(\text{inita}(1.57)) \end{cases}$$

construction 1
$$\begin{cases} p5.1 = \text{initp}(-420.732239, -185.07) \\ l6.1 = \text{initd}(p5.1, 1.57) \\ a13 = \text{inv\_angle}(\text{supp\_angle}(a3)) \\ c3.1 = \text{mkcir}(p5.1, k4) \\ p7.1 = \text{interlc}(l6.1, c3.1) \\ c4.1 = \text{mkcir}(p7.1, k3) \\ p6.1 = \text{interlc}(l6.1, c4.1) \\ a10.1 = \text{inita}(1.570796) \\ l7.1 = \text{lpla}(p7.1, l6.1, a10.1) \\ a12.1 = \text{bissect}(a3) \\ l8.1 = \text{lpla}(p6.1, l6.1, a12.1) \\ p8.1 = \text{interll}(l8.1, l7.1) \\ l5.1 = \text{lpla}(p6.1, l6.1, a13) \\ c1.1 = \text{mkcir2}(p8.1, p7.1) \\ p4.1 = \text{interlc}(l5.1, c1.1) \end{cases}$$

construction 2
$$\begin{cases} p1.2 = \text{initp}(42.42, -91.29) \\ l1.2 = \text{initd}(p1.2, -1.26) \\ c6.2 = \text{mkcir}(p1.2, k1) \\ p3.2 = \text{interlc}(l1.2, c6.2) \\ l4.2 = \text{lpla}(p3.2, l1.2, a1) \\ c7.2 = \text{mkcir}(p3.2, k2) \\ p4.2 = \text{interlc}(l4.2, c7.2) \end{cases}$$

construction 3
$$\begin{cases} p5.4 = \text{initp}(-420.73, -185.07) \\ l3.4 = \text{initd}(p5.4, 0.08) \\ l9.4 = \text{ldl}(l3.4, k6) \\ c11.4 = \text{mkcir}(p5.4, k5) \\ p2.4 = \text{interlc}(l3.4, c11.4) \\ l2.4 = \text{lpla}(p2.4, l3.4, a2) \\ p1.4 = \text{interll}(l2.4, l9.4) \end{cases}$$

distances déduites
$$\begin{cases} k9 = \text{fdist}(p4.2, p1.2) \\ k10 = \text{fdist}(p5.4, p1.4) \end{cases}$$

construction 1
$$\begin{cases} c15.1 = \text{mkcir}(p4.1, k9) \\ c16.1 = \text{mkcir}(p5.1, k10) \\ p1.1 = \text{intercc}(c16.1, c15.1) \end{cases}$$

assemblage
$$\begin{cases} dep1.1 = \text{make\_dep\_pp}(p1.1, p5.1, p1.4, p5.4) \\ p2.1 = \text{transfp}(p2.4, dep1.1) \\ dep2.1 = \text{make\_dep\_pp}(p1.1, p4.1, p1.2, p4.2) \\ p3.1 = \text{transfp}(p3.2, dep2.1) \end{cases}$$

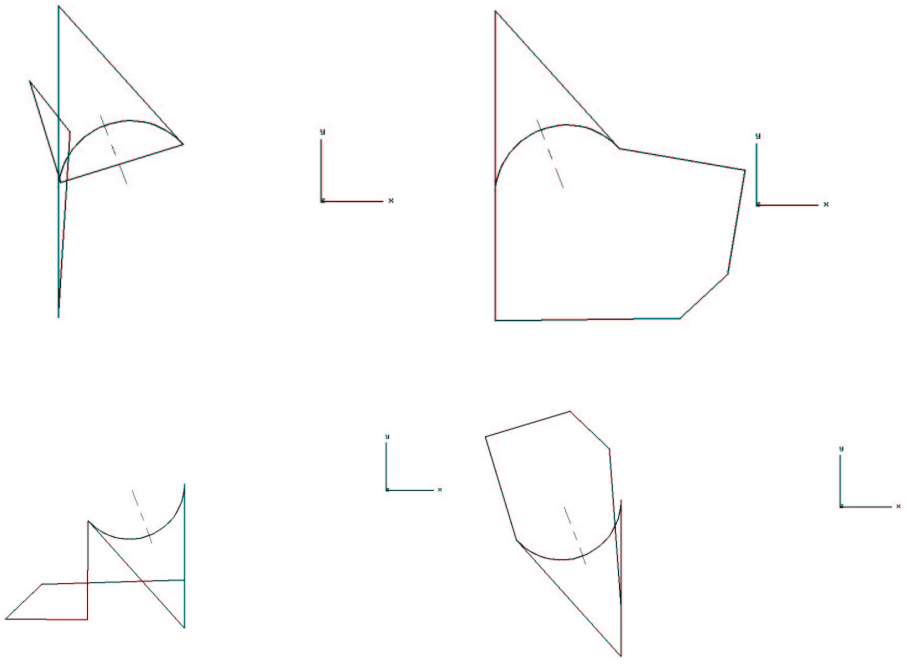


Fig. 2.15: Quatre solutions produites par YAMS

La Fig.2.15 montre des copies d'écran des quatre solutions proposées par *YAMS* après l'interprétation numérique de ce plan de construction. Nous avons ensuite ajouté la contrainte $fixorgpl(p5, l3)$ qui impose au point $p5$ de se trouver à l'origine O du repère du plan euclidien et à la droite $l3$, incidente à $p5$, d'être confondue avec l'axe Ox. Un nouveau plan de construction est calculé, similaire au précédent. La phase de résolution numérique conduit de nouveau à quatre solutions (voir copies d'écran de la Fig.2.16) qui sont équivalentes modulo un déplacement à celles produites précédemment.

On s'aperçoit que trois d'entre elles sont éloignées de l'esquisse et sûrement aussi des attentes de l'utilisateur. C'est le genre de problème que nous souhaitons pouvoir traiter : comment choisir de façon automatique la solution attendue par l'utilisateur parmi toutes les solutions disponibles ? Nous développerons donc ce sujet à partir du Chapitre 4.

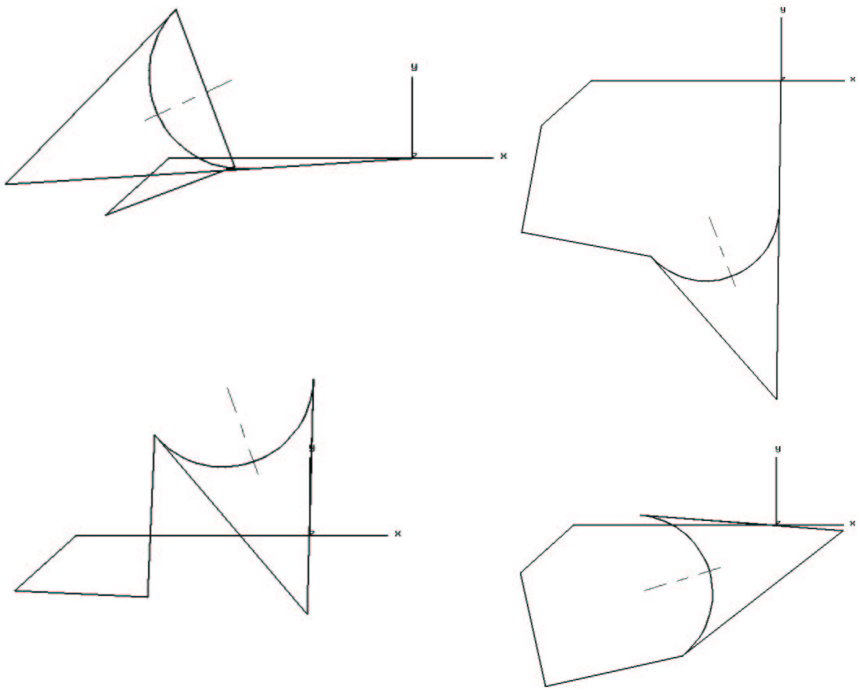


Fig. 2.16: Quatre solutions produites par YAMS dans un autre repère

Chapitre 3

Spécification des principales notions

Avant de pouvoir entreprendre une recherche sur la sélection de solutions, il nous a semblé nécessaire de disposer de notions précises et rigoureusement définies, afin que ce travail puisse reposer sur des bases claires. C'est pourquoi, avant de commencer à étudier le problème des solutions multiples, nous avons préféré prendre le temps de détailler, non pas tous les concepts utilisés dans *YAMS*, mais au moins ceux qui nous seront utiles par la suite.

Ainsi, nous avons choisi le formalisme des spécifications algébriques [EM85], et plus particulièrement le langage *OBJ3*[GW88, GWM+93], pour décrire avec précision notre univers géométrique ainsi que toutes les notions que nous avons jugées indispensables à la poursuite de notre étude. Notons ici que toutes les notions spécifiées grâce à *OBJ3* ont été testées sur divers jeux d'essais. En effet, outre la capacité d'*OBJ3* à permettre une description concise et sans ambiguïté des composants, il permet également d'exécuter sur des exemples les opérations définies.

Ces spécifications concernent donc dans un premier temps la description de l'univers géométrique, ainsi que des opérations de la phase initiale du processus de résolution. Ensuite, nous abordons la phase de l'interprétation numérique et la structure de l'espace des solutions qu'elle produit.

Enfin, le nombre de solutions étant exponentiel, et donc l'exploration de l'arbre des solutions étant forcément très coûteuse, nous nous sommes attardés sur le tri topologique du plan de construction, et son arrangement dynamique.

3.1 De l'univers géométrique aux plans de construction

Le formalisme que nous utilisons pour décrire l'univers géométrique de *YAMS* est particulièrement adapté à notre cadre, mais il introduit une petite difficulté supplémentaire bien connue des logiciens [CDE+99] : puisque l'univers géométrique est une donnée manipulable

par *YAMS*, le cadre géométrique que nous voulons spécifier est lui-même une spécification algébrique comprenant des sortes, des symboles fonctionnels, etc. Nous faisons ainsi une méta-spécification d'un univers géométrique dans laquelle nous prenons soin de distinguer les deux niveaux.

3.1.1 Spécifications algébriques avec *OBJ3*

Une spécification algébrique décrit une théorie en logique équationnelle typée [EM85]. Elle comporte une signature hétérogène indiquant les sortes et les symboles fonctionnels qui sont l'objet de la description, ainsi qu'un ensemble d'équations définissant les relations entre les termes composés sur la signature.

Plus précisément, une signature hétérogène Σ est un couple (S, F) où S est un ensemble de symboles - les éléments de S sont appelés des *sortes* - et F est un ensemble de symboles fonctionnels - les éléments de F sont souvent appelés des *opérations*. Cette signature sert à constituer des termes typés. À cet égard, chaque opération est munie d'un *profil* noté $s_1...s_n \longrightarrow s$ où $s_1...s_n$ est une suite finie de sortes qu'on appelle l'*arité* de l'opération et s est une sorte qu'on appelle la *co-arité* de l'opération. À cette signature, on ajoute un ensemble X de *variables typées* qui sont des symboles associés chacun à une sorte. On impose, évidemment, que les ensembles de symboles S, F et X soient disjoints. Dans l'exemple suivant, qui utilise le langage de spécification *OBJ3*, cette signature prend la forme de déclarations de sortes et d'opérations comme dans un langage de programmation classique.

```
sort Nat .
op 0 : -> Nat .
op s : Nat -> Nat .
op add : Nat Nat -> Nat .
var n m : Nat .
```

Dans cet exemple, `0`, `s(s(0))`, `add(s(0),s(s(0)))`, etc. sont des termes sans variables sur la signature Σ - on parle alors de termes *clos* - et `s(s(n))`, `add(s(s(0)),add(m,s(n)))`, etc. sont des termes avec variables.

Les équations précisent le comportement des sortes et symboles fonctionnels de Σ. Ce sont des couples de termes de même sorte de la forme $t = t'$ (le mot clé `eq` introduit les axiomes équationnels, tandis que `cq` introduit les équations conditionnelles). En reprenant notre exemple en *OBJ3*, on peut ajouter les axiomes classiques de Peano :

```
eq add(0, n) = n .
eq add(s(m), n) = s(add(m, n) .
```

Notons qu'*OBJ3* permet d'utiliser des notations infixées pour les opérations. C'est ainsi

que pour reprendre une notation plus proche de la pratique, on utilisera plutôt que `add` une opération notée + définie par :

```
op _+_ : Nat Nat -> Nat .
eq 0 + n = n .
eq s(m) + n = s(m + n) .
```

Notons également, qu'*OBJ3* permet d'utiliser l'arithmétique classique de l'informatique sans avoir à réécrire l'axiomatique de Peano (par exemple) ou des bibliothèques de calculs sur les flottants. C'est aussi ce que nous ferons dans la suite.

La sémantique souvent associée aux spécifications *OBJ3* est la sémantique du *modèle initial*. Un tel modèle ne contient que des objets que l'on peut construire à l'aide de la spécification et deux objets sont égaux dans le modèle uniquement si les termes qui les définissent sont égaux dans la spécification modulo la relation d'équivalence induite par les équations. Les spécifications "constructives" en *OBJ3* possèdent aussi une *sémantique opérationnelle* : on peut les exécuter à l'aide d'un mécanisme de réécriture sophistiqué. Lorsque le système de réécriture induit par la spécification est convergent, *OBJ3* peut prouver tous les théorèmes équationnels déductifs. Cet aspect opérationnel d'une spécification est très important lorsqu'on veut expérimenter ses idées dans un domaine à explorer.

Parmi les termes bien formés construits sur une signature, il en est certains auxquels on ne souhaite pas donner de signification. Par exemple, dans une spécification des piles d'entiers, on ne peut pas extraire le sommet d'une pile vide. Pour indiquer ceci, les formalismes de spécifications algébriques permettent souvent de définir des préconditions qui permettent de décrire le domaine de validité des arguments d'un terme. La philosophie d'*OBJ3* est de définir ces domaines de manière constructive et non pas restrictive en introduisant un ordre sur les sortes et en obligeant de définir des constructeurs propres aux sous- et sur-sortes. Si cette manière de faire est très élégante, il nous semble parfois plus parlant - et plus facile à définir - d'utiliser des préconditions que nous mettrons, dans le texte *OBJ3*, en commentaire.

La plupart des langages de spécification algébrique permettent de définir des sortes génériques. En *OBJ3*, ceci se traduit par la possibilité de paramétrer une spécification par une autre spécification. Le "type" du paramètre est défini par une spécification particulière qu'on appelle une *théorie* et qu'on interprète par une *sémantique lâche* (*loose semantic*). Ce mécanisme est beaucoup plus général que la paramétrisation par un type dont on donne une *interface* - comme par exemple dans CLU [LAB+81] - ou la paramétrisation par une fonction. Voici, par exemple, une spécification de la théorie des relations d'équivalence que nous utiliserons par la suite.

```
th EQUIV is
sort Elt .
op _~_ : Elt Elt -> Bool .
```

```
var x y z : Elt .
eq x ~ x = true .
eq x ~ y = y ~ x .
eq x ~ z = x ~ y and y ~ z .
endth
```

OBJ3 prédéfinit certaines théories, comme la théorie `TRIV` qui impose uniquement l'existence d'une sorte. Contrairement aux autres spécifications *OBJ3*, les théories ne possèdent pas de caractère constructif et n'ont *a priori* pas de sémantique opérationnelle : elles servent uniquement à décrire les propriétés que doivent avoir les sortes et opérations passées en paramètre. L'exemple suivant montre une spécification des ensembles finis paramétrée par la théorie `EQUIV` donnée ci-dessus. Il faut aussi noter dans cet exemple l'utilisation judicieuse du sous-sortage.

```
obj SET-EQ[X :: EQUIV] is protecting INT .
sort Set .
subsort Elt < Set .
op emptyset : -> Set .
op _ _ : Elt Set -> Set          *** ajout d'un element a un ensemble
op is-in-set : Elt Set -> Bool . *** appartenance d'un element a un
                                                          ensemble
op size : Set -> Int .           *** cardinal de l'ensemble
op is-subset : Set Set -> Bool . *** e1 est-il un sous-ensemble de e2?
var x x1 x2 : Elt .
var e1 e2 : Set .
cq x e1 = e1 if is-in-set(x,e1) . *** un element ne peut etre ajoute
                                      s'il appartient deja a l'ensemble
eq size(emptyset) = 0 .
eq size(x e1) = 1 + size(e1) .
eq is-in-set(x,emptyset) = false .
eq is-in-set(x1,x2 e1) = x1 ~ x2 or is-in-set(x1,e1) .
eq is-subset(emptyset,e1) = true .
eq is-subset(x e1,e2) = is-in-set(x,e2) and is-subset(e1,e2) .
endo
```

La manière de lier une spécification avec le paramètre consiste à définir une *vue* de la spécification suivant la théorie contraignant le paramètre. Cette vue peut être considérée d'un point de vue logique comme une interprétation de la théorie donnée dans le modèle initial décrit par la spécification, ou de manière pragmatique, comme un renommage des sortes et opérations décrites dans la théorie et utilisées dans la spécification paramétrée. Par exemple, la spécification `NAT` des entiers naturels fournie par *OBJ3* respecte la théorie `EQUIV` donnée plus haut lorsque l'opération `~` est interprétée par l'égalité sur les entiers naturels définis par `NAT`. Ceci s'écrit :

```
view NATS from EQUIV to NAT is
sort Elt to Nat .
var x1 x2 : Elt .
op x1 ~ x2 to x1 == x2 .
endv
```

On peut maintenant parler de `SET-EQ[NATS]` comme étant une spécification des ensembles d'entiers naturels (modulo l'égalité). L'instanciation effective se fait en utilisant la commande `make`. Les mécanismes de renommage peuvent être utilisés par ailleurs pour éviter les ambiguïtés. C'est ainsi que nous définissons `NSET` de la manière suivante :

```
make NSET is SET-EQ[NATS]*(sort Set to Nset , op emptyset to n-emptyset)
endm
```

La sorte `Nset` désignant alors les ensembles d'entiers, et `n-emptyset` l'ensemble vide pour les entiers naturels.

3.1.2 Méta-spécification d'univers géométriques

Dans la perspective d'une utilisation par *YAMS* de différents univers géométriques, voire de leur définition par un expert, nous ne souhaitons pas décrire *un* univers géométrique particulier, mais la manière de constituer des univers. Ainsi, cette section présente une description axiomatique, sous forme de spécification algébrique, d'un cadre métathéorique dans lequel nous allons pouvoir considérer par exemple la sorte *sorte géométrique* ou la sorte *symbole fonctionnel*. De cette manière, nous pourrons traiter les symboles fonctionnels et prédicatifs comme des objets sur lesquels nous effectuerons des opérations, comme la construction de termes abordée plus loin.

Tout d'abord, nous montrons comment, dans ce cadre, définir l'univers géométrique. Rappelons que nous travaillons uniquement dans le plan euclidien. La signature comprend une sorte `S` des sortes géométriques, une sorte `F` de symboles fonctionnels et une sorte `P` des symboles prédicatifs. Les symboles fonctionnels servent à décrire des constructions, tandis que les symboles prédicatifs servent à décrire des contraintes. Nous nous focalisons ici sur les symboles et termes fonctionnels qui interviennent directement dans la constitution des plans de construction. Notons à cet égard, que nous interprétons toujours les symboles fonctionnels par des multifonctions. C'est pourquoi, nous employons indifféremment les termes fonction et multifonction dans la suite.

Les sortes géométriques sont vues comme des constantes de sorte `S`. Les sortes géométriques de base sont `point, line, circle, long, angle, depl, et num`. Les symboles fonctionnels et prédicatifs considérés comme des constantes, respectivement de sorte `F` et `P`, sont munis d'une arité et d'une coarité. Les symboles fonctionnels et prédicatifs sont ici spécifiés par les sortes `F` et `P`. Nous avons ainsi l'extrait de spécification suivant :

```
sort S F P .
ops point line circle long angle depl num : -> S .
ops center-of medradcir : -> F .
ops perp distpp : -> P .
```

Un index de symboles fonctionnels et prédicatifs, accompagnés de leurs profils, ainsi que de l'explication de leur signification se trouve en Annexe A.

Le profil d'un symbole fonctionnel `f` est noté `ar(f) -> coar(f)` et celui d'un symbole prédicatif `p` est noté `ar(p) ->` . Le profil est une autre sorte, notée `Profil`, qui possède les formes suivantes :

```
op _=>_ : Slist S -> Profil .
op _=> : Slist -> Profil .
```

où `Slist` est la sorte des listes de sortes géométriques, dans lesquelles les sortes géométriques sont séparées par le symbole ' " '. Il faut définir pour chaque symbole particulier son profil. Le profil d'un symbole est alors déterminé par une équation mettant en jeu l'opération `profil`, qui donne le profil d'un symbole fonctionnel ou prédicatif :

```
op profil : F -> Profil .
op profil : P -> Profil .
```

Par exemple, le profil du symbole `interlc` est défini par l'équation suivante :

```
eq profil(interlc) = Line " Circle => Point .
```

D'autres opérations sur les symboles fonctionnels et prédicatifs ainsi que sur les profils peuvent être définies, comme l'opération `arity` qui renvoie l'arité.

```
op arity : Profil -> Slist .
eq arity(sl -> s) = sl .
op arity : F -> Slist .
eq arity(f) = arity(profil(f)) .
```

où `sl` est une variable de type `Slist` et `s` une variable de type `S`. Alors, par exemple, l'arité de `interlc` peut être retrouvée à partir de l'équation `eq arity(sl -> s) = sl`, comme un théorème équationnel :

```
arity(interlc) = Line " Circle .
```

Un des intérêts du passage au niveau métathéorique est de pouvoir attacher des informations supplémentaires aux symboles fonctionnels. C'est ainsi qu'à la notion de profil d'un symbole fonctionnel comprenant l'arité et la coarité, nous adjoignons deux entiers, correspondant respectivement aux *degrés de multiplicité* et de *risque* des multifonctions qu'ils représentent. Ces deux entiers vont servir de paramètres de choix pour le tri topologique que nous exposons à la Section 3.3.

Le degré de multiplicité M est le cardinal maximal de l'ensemble calculé par une multifonction. Par exemple, la multifonction qui calcule les droites tangentes à deux cercles possède degré de multiplicité de 4, car elle peut produire de 0 à 4 droites selon les positions relatives des cercles. Une vraie fonction, comme `center-of`, a un degré de multiplicité de 1.

Le degré de risque R est un coefficient qui quantifie la probabilité qu'a une multifonction d'échouer, c'est-à-dire de retourner un ensemble vide. Le centre d'un cercle n'ayant aucune chance d'échouer, son coefficient de risque est 0, alors que l'intersection d'une droite et d'un cercle est bien plus hasardeuse et possède un degré de risque de 4. Toutes ces informations sont contenues dans le *profil* du symbole fonctionnel. Nous transformons donc ce profil et lui donnons désormais la forme suivante :

```
op _=>_M_R_ : Slist S Int Int -> Profil .
```

Ainsi, par exemple, le profil du symbole `medradcir` est codé par l'équation :

```
eq profil(medradcir) = Point " Point " Long => Circle M 2 R 3 .
```

Dans notre univers géométrique, nous faisons usage de symboles de variables que nous nommons identificateurs simples. Ces symboles sont typés en leur adjoignant une sorte géométrique. Pour distinguer des objets définis dans des sous-figures différentes, et donc dans des repères différents, nous indiçons les identificateurs typés en leur accolant le numéro du repère dans lequel ils sont définis. Rappelons ici que, comme nous l'avons dit à la Section 2.2, dans la suite de ce document, l'indice pourra être omis dans un souci d'allègement si aucune décomposition n'a été nécessaire lors de la résolution, et si tous les éléments sont définis dans le même repère.

Finalement, nous spécifions dans le module `IDENT` la sorte `I` des *identificateurs indicés*. Les identificateurs indicés sont donc construits à partir d'une sorte géométrique `S`, ce qui permet une vérification de type plus rapide, d'un identificateur simple de sorte `Id` qui est en quelque sorte le nom de l'objet, et d'un indice de sorte `Index` qui est le numéro de la sous-figure à laquelle il appartient. Voici le constructeur en notation infixée :

```
op _#_._ : S Id Index -> I .
```

Ces trois composantes sont séparées les unes des autres par les symboles '#' et '.'. Par exemple, une droite nommée `d1` et appartenant à la sous-figure `1` aura pour identificateur indicé `line#d1.1`.

Nous avons défini un certain nombre d'opérations sur ces identificateurs indicés. Par exemple, `name-id` renvoie la partie identificateur de l'identificateur indicé.

```
op name-id : I -> Id .
eq name-id(s#a.n) = a .
```

La théorie `EQUIV` que nous avons vue à la Section 3.1.1 nous permet d'effectuer une comparaison entre des identificateurs indicés autre que l'égalité classique. En effet, nous souhaitons pouvoir vérifier si le même "nom" n'a pas été attribué à deux objets différents, même s'ils ne font pas partie de la même sous-figure. On considérera donc deux identificateurs indicés comme équivalents lorsque ceux-ci auront la même composante de sorte `Id` (que l'on retrouve grâce à la fonction `name-id`), même si ceux-ci ne sont pas strictement égaux à cause de leur appartenance à deux sous-figures distinctes. Les identificateurs indicés vont donc suivre la théorie `EQUIV`. On aura pris soin dans la vue `IDENTS` de donner une interprétation adéquate à l'opération `~` pour les identificateurs indicés.

```
view IDENTS from EQUIV to IDENT is
sort Elt to I .
var i1 i2 : Elt .
op i1 ~ i2 to name-id(i1) == name-id(i2) .
endv
```

Les symboles fonctionnels, ou prédicatifs, et les identificateurs indicés sont utilisés pour construire des *termes*, comme nous l'expliquons à la section suivante.

3.1.3 Termes et définitions

Les termes sont formés sur le modèle classique des termes du premier ordre. Un terme est donc soit un simple identificateur indicé de sorte `I`, soit un terme fonctionnel de sorte `Termf`, soit un terme prédicatif de sorte `Termp`. La notion de sous-sortage permet de hiérarchiser ces sortes de termes : `subsorts Termf Termp I < Term .` Les termes fonctionnels servent à former les définitions qui décrivent les constructions, tandis que les termes prédicatifs représentent des contraintes. Les termes fonctionnels et prédicatifs ont la forme suivante :

```
F[i1 " ... " in]
P[i1 " ... " in]
```

où F et P sont des symboles fonctionnel et prédicatif et i_1 " ... " i_n est une liste d'identificateurs indicés nommant les arguments du terme, avec l'impératif que les types de i_1,...,i_n correspondent à l'arité de F ou P. Le terme est alors dit *bien formé*. Avec le formalisme d'*OBJ3*, nous décrivons ainsi les termes fonctionnels et prédicatifs :

```
op _[_] : F Ilist -> Termf .
op _[_] : P Ilist -> Termp .
```

où Ilist est une liste d'identificateurs indicés appelée arguments. Par exemple, le terme prédicatif perp[line#d1.1 " line#d2.1] représente la contrainte imposant aux droites d1 et d2 d'être perpendiculaires, dans la sous-figure 1, sachant que perp a pour profil Line Line -> .

Une définition, spécifiée par la sorte Def, est la mise en correspondance d'un identificateur indicé et d'un terme fonctionnel bien formé. Une définition décrit la construction d'un objet géométrique désigné par son identificateur indicé. Le terme fonctionnel associé contient les objets géométriques dont dépend sa construction ainsi que la fonction qui doit leur être appliquée. La forme générale d'une définition est donnée par le constructeur := dont le profil en *OBJ3* est :

```
sort Def .
op _:=_ : I Termf -> Def .
```

Par exemple, line#d1.1 := lpp[point#p1.1,point#p2.1] définit dans la sous-figure 1 la droite d1 passant par les points p1 et p2, lpp ayant pour profil Point Point -> Line.

La définition d'un paramètre est traitée comme un cas particulier, et possède la forme suivante : i := param, où i est un identificateur indicé. Ce type de définition n'a pas d'argument.

3.1.4 Ensemble de définitions et graphe de dépendance

Si l'on fait abstraction de l'ordre d'apparition des définitions dans un plan de construction, on obtient un ensemble de définitions se rapportant chacune à un élément différent. Autrement dit, deux définitions de cet ensemble ne peuvent pas contenir le même identificateur indicé. Nous appelons un tel ensemble un ensemble non ambigu de définitions (NADS pour Non Ambiguous Definition Set). Dans notre spécification des NADS, cette propriété est garantie grâce à l'équivalence entre définitions décrite par la vue DEFS :

```
view DEFS from EQUIV to DEF is
sort Elt to Def .
var d1 d2 : Elt .
```

```
op d1 ~ d2 to name-id(ident(d1)) == name-id(ident(d2)) .
endv
```

Nous avons donc finalement la spécification suivante pour les NADS (pour des questions de surcharge, nous n'avons pas placé ici les équations relatives à la spécification des NADS, mais celle-ci est présentée dans son intégralité en Annexe B) :

```
obj NADS is protecting DEF-SET + ISET .    *** Non Ambiguous Definition Set
op id-set : Def-set -> Iset .              *** ensemble des id. deja definis
op args-set : Def-set -> Iset .            *** ens des id qui sont arguments
op pred : I Def-set -> Iset .              *** ens des predecesseurs d'1 id
op succ : I Def-set -> Iset .              *** ens des ids success. d'1 id
op succ : Def Def-set -> Def-set .         *** ens des defs success. d'1 def
op sources : Def-set -> Iset .             *** ensemble des sources
op sources : Def-set Iset -> Iset .        *** outil
op parameters : Def-set -> Iset .          *** ensemble des parametres
op ncdef : Def-set -> Iset .               *** ensemble des id dont les args
                                                *** ne sont pas tous definis
op ncdef : Def-set Def-set Iset -> Iset . *** outil
op wells : Def-set -> Iset .               *** ensemble des puits
op connected : Def-set -> Bool .           *** connexite
op search : Iset Def-set Iset -> Iset .    *** outil
op has-circuit : Def-set -> Bool .         *** y a-t-il un circuit dans
                                                *** l'ensemble de def?
op depth : Iset Def-set Iset -> Bool .     *** outil
op is-triang : Def-set -> Bool .           *** l'ens est-il triangulaire?
op def : I Def-set -> Def .                *** def corresp. a l'id
op defs : Iset Def-set -> Def-set .        *** defs corresp. aux ids
endo
```

Bien qu'un NADS ne soit qu'un ensemble, il existe une relation de dépendance entre ses définitions : celle-ci est induite par le fait que certains identificateurs indicés sont présents en tant qu'arguments dans d'autres définitions. Nous noterons ce type de relation "dépend de" par $\rightarrow$. Par exemple, la définition $y = f(x_1, \ldots, x_n)$ donne lieu aux relations de dépendance $y \rightarrow x_1, \ \ldots, \ y \rightarrow x_n$.

Une structure de graphe orienté se déduit donc de la relation de dépendance $\rightarrow$ entre ses définitions. Nous appelons ce graphe *graphe de dépendance*. À chaque sommet de ce graphe orienté est associée bijectivement une définition du NADS. Il existe un arc d'une définition à une autre si et seulement si l'identificateur défini dans la première fait partie des arguments de la deuxième. Les définitions sans argument déjà défini, ou qui sont des paramètres, sont des sources, et les définitions qui ne sont pas argument dans une autre définition sont des puits du graphe. Chaque sommet qui n'est ni un puits ni une source a donc au moins un prédécesseur et un successeur.

Exemple 6 *La Fig.3.1 présente un exemple de graphe de dépendance associé au NADS représenté à la Table 3.1.*

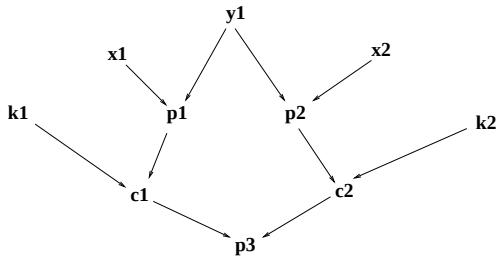


Fig. 3.1: Graphe de dépendance

Tab. 3.1: NADS correspondant au graphe de dépendance de la Fig.3.1

`p1 := initp[x1 y1]`	`y1 := param`	`p3 := intercc[c1 c2]`
`x2 := param`	`k1 := param`	`x1 := param`
`c2 := mkcir[p2 k2]`	`c1 := mkcir[p1 k1]`	
`p2 := initp[x2 y1]`	`k2 := param`	

Pour que le NADS corresponde effectivement à un plan de construction, il faut que le graphe associé soit *triangulaire*, ce qui signifie qu'il doit impérativement remplir les trois conditions suivantes :
(i) toutes ses sources doivent être des paramètres
(ii) les arguments des définitions doivent être tous définis
(iii) il ne doit pas contenir de circuit

En effet, toute source qui ne serait pas un paramètre serait une définition dont les arguments n'auraient pas de définition dans le NADS et donc il manquerait des informations pour construire la figure. Enfin, si le graphe possédait un circuit, cela voudrait dire que plusieurs définitions se font mutuellement référence, et il serait impossible d'en prendre une en compte avant les autres. Les Fig.3.2 et 3.3 présentent des graphes associés à des NADS inexploitables.

Le point (ii) signifie que les relations $\rightarrow$ ont une conséquence sur le placement des définitions. En effet, si $y \rightarrow x_1$, alors la définition produisant x_1 doit être évaluée avant celle produisant y. Il faut donc construire un ordre d'évaluation $<<$ total, tel que $y \rightarrow x$ implique $x << y$ (notons que la réciproque est fausse). La relation d'ordre $<<$ correspond donc à l'expression "est évalué avant".

On peut alors ordonner les définitions d'un NADS en faisant un *tri topologique* du graphe de dépendance, pour retrouver une liste de définitions ordonnée selon un ordre $<<$ vérifiant la propriété ci-dessus, c'est-à-dire un plan de construction. Naturellement, plusieurs tris topologiques sont possibles, et nous verrons à la Section 3.3 des heuristiques de tris pour que le plan produit puisse être interprété plus ou moins efficacement.

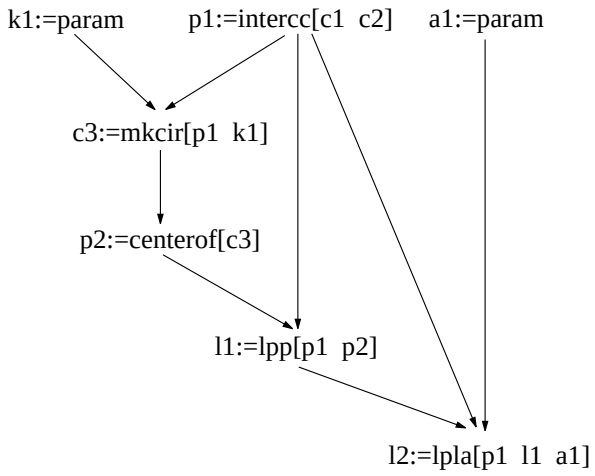


Fig. 3.2: Graphe contenant une source non paramètre

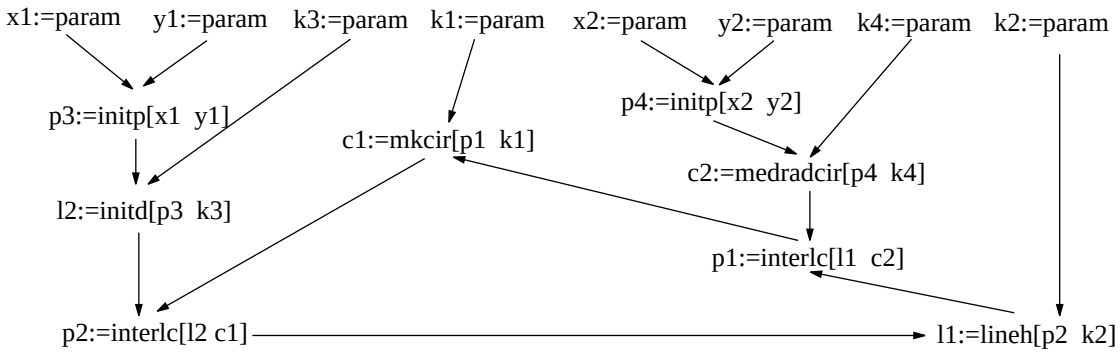


Fig. 3.3: Graphe contenant un circuit

3.1.5 Plan de construction et fonction de construction

Nous spécifions dans cette section l'opération de tri topologique sur un NADS. Cette opération produit une liste de définitions de sorte `Def-list` qui correspond au plan de construction.

Pour créer cette liste, on considère un ensemble P de définitions potentielles dans lequel on va puiser un par un les éléments à placer. Au départ, cet ensemble P est constitué des sources du NADS. Puis, à chaque étape, on sélectionne un élément x de P que l'on place dans la liste. La façon de choisir cet élément x est a priori indifférente ; le seul impératif est de respecter l'ordre des dépendances. Le choix est assuré par une opération nommée `choice` dans la spécification ci-dessous. Nous détaillerons plus loin différentes manières d'instancier la fonction `choice`. On calcule ensuite l'ensemble de ses successeurs S_x, et on réalise l'union $P \bigcup S_x$ qui sert d'ensemble P pour l'étape suivante. Autrement dit, l'ensemble P contient des éléments dont au moins un argument a déjà été traité, et lorsqu'on

sélectionne un élément x de P pour le placer il faut vérifier que tous ses arguments ont déjà été traités. Une spécification précise de cet algorithme en *OBJ3* est donnée ci-après. Notons le polymorphisme de l'opération `topo-sort` ; celle que nous cherchons à spécifier est la troisième, qui transforme un ensemble de définitions en liste de définitions, tandis que les deux autres sont auxiliaires.

```
obj TOPO-SORT[X :: CHOICE] is protecting NADS .
op topo-sort : Def-set Def-set Ilist Def -> Def-list .
op topo-sort : Def-set Def-set Ilist -> Def-list .
op topo-sort : Def-set -> Def-list .
op choice : Def-set Ilist -> Def.

var ds cur-ds : Def-set .
var il : Ilist .
var ch : Def .

eq topo-sort(cur-ds,ds,il,ch) =
   ch " topo-sort(union(del(ch,cur-ds),succ(ch,ds)),
                  ds,
                  ident(ch) " il) .
eq topo-sort(d-emptyset,ds,il) = d-emptylist .
eq topo-sort(cur-ds,ds,il) = topo-sort(cur-ds,ds,il,choice(cur-ds,il)) .
eq topo-sort(ds) = topo-sort(defs(sources(ds),ds),ds,i-emptylist) .
endo
```

Avant d'aborder la phase d'interprétation, nous devons encore effectuer une petite transformation sur la liste de définition de sorte `def-list` ainsi obtenue grâce au tri topologique. En effet, nous souhaitons pour cette spécification isoler les paramètres, afin de leur associer plus facilement des valeurs. Nous préférons donc utiliser la notion de *fonction de construction* de sorte `FUN` :

```
op _ _ : Ilist Def-list -> Fun .
```

qui est la concaténation d'une liste d'identificateurs indicés, qui décrivent des paramètres, et d'une liste de définitions, qui correspond au plan de construction amputé des définitions des paramètres. Ainsi, nous avons une forme adéquate pour entamer le processus d'interprétation.

3.2 Interprétation

Intuitivement, l'interprétation d'une fonction de construction a pour but d'associer à chaque objet géométrique, défini dans cette fonction par sa liste de définitions, une

valeur numérique correspondant à ses coordonnées dans le plan euclidien. Les valeurs de départ, les paramètres effectifs, sont données par l'utilisateur. En fait, l'interprétation de certains symboles fonctionnels par des multifonctions conduit à considérer des ensembles de coordonnées pour chacun des objets calculés. Le résultat d'une interprétation d'une fonction de construction est donc, comme nous l'avons déjà dit, un arbre de valeurs numériques appelé *arbre des solutions*, ou encore *arbre d'interprétation.*

La fonction `interp`, qui réalise cette interprétation, a pour profil :

```
op interp : Fun Numlist -> IT .
```

où `Numlist` désigne la sorte des listes de valeurs numériques, et `IT` la sorte des arbres d'interprétation. Si la sorte `Numlist` n'appelle aucun commentaire particulier, la sorte `IT` doit en revanche être expliquée plus précisément.

3.2.1 Arbre des solutions

Concrètement, un arbre d'interprétation est construit niveau par niveau, en interprétant successivement les définitions de la fonction de construction. A chaque définition correspond un niveau. Chaque nœud contient un système de coordonnées possibles de l'objet défini à ce niveau. La valeur d'un nœud est trouvée en interprétant la définition correspondant au niveau et en utilisant comme arguments les valeurs numériques des ancêtres. Plusieurs fils sont créés dans le cas d'une vraie multifonction.

Exemple 7 *L'interprétation de la liste de définitions donnée à la Table 3.2 avec des valeurs effectives de l'énoncé, produit l'arbre d'interprétation de la Fig.3.4(b). En admettant que les cercles* `c1` *et* `c2` *sont sécants, on obtient une solution comme celle de la Fig.3.5.*

Tab. 3.2: Liste de définitions produisant l'arbre de la Fig.3.4(b)

`x1 := param`	`p2 := initp[x2 y2]`
`y1 := param`	`c1 := mkcir[p1 k1]`
`x2 := param`	`c2 := mkcir[p2 k2]`
`y2 := param`	`i1 := intercc[c1 c2]`
`k1 := param`	`d1 := lpp[p1 p2]`
`k2 := param`	`i2 := interlc[d1 c1]`
`p1 := initp[x1 y1]`	`d2 := lpp[i1 i2]`

La manière de spécifier les arbres des solutions doit refléter leur mode de construction durant l'interprétation. C'est pourquoi nous avons choisi de les spécifier d'une façon particulière que nous appelons *codage par le nombre de frères.* C'est une variante du codage

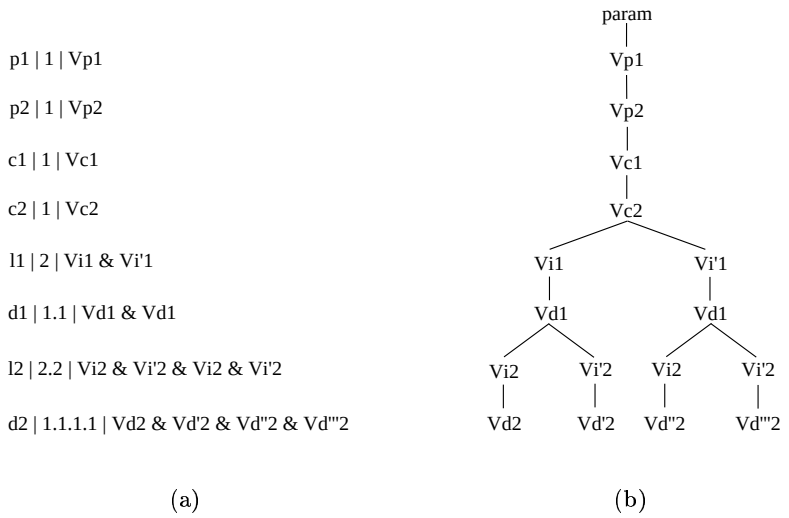


Fig. 3.4: Arbre des solutions

par occurrences, avec la différence que chaque nœud est désigné non pas par son numéro dans la fratrie mais par son nombre de frères.

Nous rappelons ici qu'il ne s'agit pas réellement d'une implantation, mais plutôt d'une description formelle des choses, et que le choix de cette façon de spécifier l'arbre n'implique aucune considération d'efficacité. En effet, dans les programmes ultérieurs, cet arbre est parcouru mais jamais calculé entièrement.

Dans notre spécification en *OBJ3*, un arbre est une liste de *niveaux* (`Floor`). Chaque niveau est étiqueté par un identificateur indicé, et comporte une liste d'entiers représentant le codage par le nombre de frères (`Brothers`) ainsi que la liste des coordonnées calculées pour cet identificateur (`Coordlist`). Ceci se traduit par les lignes *OBJ3*suivantes :

```
sort Floor .
including LIST[Floor]*(sort List to IT , op emptylist to emptytree)
op _|_|_ : I Brothers Coordlist -> Floor .
```

Le codage pour notre exemple est donné à la figure 3.4(a), avec `Vsi` la valeur de la solution `si` pour l'objet `s`. Ceci signifie que `Vi1` et `V'i1` sont deux frères et ont le même père, tandis que les deux `Vd1` n'ont pas le même père et ne sont pas frères, chacun est fils unique. Pourtant, on les duplique car chacun correspond à une branche initialisée par `Vi1`, `V'i1`, etc. Lorsqu'on additionne tous les frères pour un objet, on retrouve le nombre de nœuds présents à la hauteur de l'arbre qui correspond à cet objet.

Comme nous le disions plus haut, cet arbre des solutions est destiné à être parcouru et non calculé entièrement. En effet, chaque solution - qui est une branche de l'arbre -

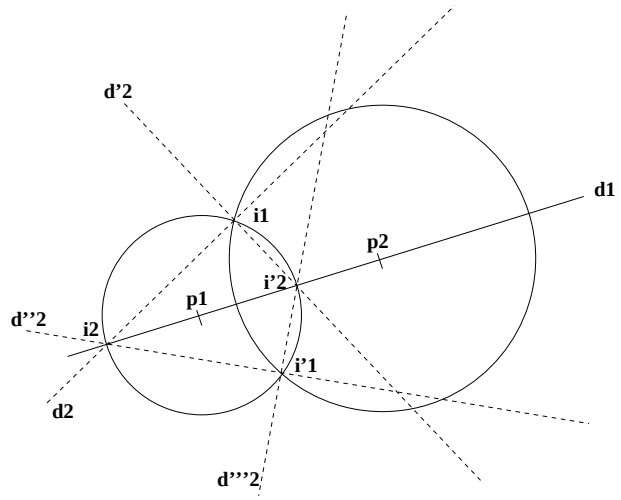
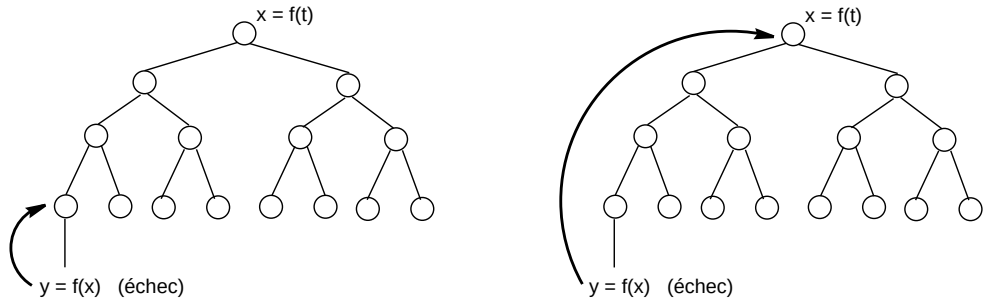


FIG. 3.5: Solutions résultantes

est calculée dès que l'utilisateur souhaite la visualiser. Ainsi, lorsque l'utilisateur passe en revue toutes les solutions pour choisir celle qu'il souhaite, un backtracking est effectué sur l'arbre. Un backtracking naïf permet généralement de passer d'une solution à l'autre en remontant la branche jusqu'au premier nœud rencontré offrant une voie différente, puis en descendant par cette nouvelle voie. Autrement dit, si la définition $y = f(x_1, \ldots, x_k)$ conduit à un échec, le backtracking naïf choisit la solution suivante de la définition $y' = g(z_1, \ldots, z_l)$ qui est située immédiatement au-dessus de celle de y. Si y' n'est pas un argument de y, calculer de nouveau $y = f(x_1, \ldots, x_k)$ conduira alors encore à un échec et donc à une perte de temps. Pour remédier à cela, *YAMS* utilise un backtracking dit "intelligent". Celui-ci permet de tirer parti au mieux des possibilités offertes par la relation $\rightarrow$: dès l'échec de y, le plus proche argument de y dans l'arbre est sélectionné pour continuer l'interprétation (voir Fig.3.6). Différentes recherches ont été menées sur les algorithmes intelligents dans le domaine des contraintes. Parmi ceux-ci, on peut citer par exemple l'algorithme Conflict-Back-Jumping (CBJ) [Pro93], ou le Dynamic-Backtracking (DBT) [Gin93]. C'est en ayant, entre autres, à l'esprit ce type de backtracking que nous allons développer dans la Section 3.3 des heuristiques pour le tri topologique qui permettront une réorganisation du plan de construction.

Revenons à l'interprétation proprement dite. Rappelons qu'une fonction de construction est constituée d'une liste de paramètres suivie d'une liste de définitions. Il faut donc une autre opération qui sera chargée d'interpréter la liste de définitions extraite de la fonction de construction, nous la nommons également `interp` (`interp` sera donc une opération surchargée). Interpréter une liste de définitions veut dire interpréter un à un chaque terme extrait d'une définition puis stocker les valeurs calculées et leur identificateur associé dans une structure (que nous nommons `Interp`) qui est une liste d'affectations des identificateurs avec leurs valeurs et qui permettra de les consulter lors de l'interprétation d'une définition qui en dépend. Nous surchargerons donc encore l'opération `interp` afin de pouvoir interpréter un terme. Enfin, la dernière forme de l'opération `interp` sera celle qui permettra



(a) backtracking naïf

(b) backtracking intelligent

FIG. 3.6: Backtracking intelligent

d'interpréter le symbole fonctionnel présent dans ce terme en faisant appel à la fonction qui lui correspond. Voici donc les quatre formes de l'opération `interp` :

Interprétation d'une fonction de construction : on lance l'interprétation de la liste de définitions en passant comme paramètre la structure `Interp` initiale qui ne contient que les paramètres. La sorte *FFFlist* est celle des listes de listes de listes de réels qui servent à décrire les données de l'esquisse. Elles sont représentées sous la forme : `(x ;y) & ((a1 ;b1 ;c1) | (a2 ;b2 ;c2))` ce qui veut dire qu'il y a deux objets (juxtaposés par `&`) dont un est constitué d'un couple de réels `(x ;y)` et l'autre a deux valeurs possibles, soit le triplet `(a1 ;b1 ;c1)` soit le triplet `(a2 ;b2 ;c2)`.

```
op interp : Fun FFFlist -> Interp .
eq interp(fct,fffl) =
        interp(definitions(fct),affect(parameters(fct),fffl)) .
```

Interprétation d'une liste de définitions : on lance l'interprétation du terme de la définition en tête de la liste, ce qui donne une affectation que l'on place dans la liste des affectations. Signalons ici qu'une affectation d'une valeur `v` à un identificateur `i` est notée `v => i`. Les autres définitions seront interprétées récursivement, puisque cette liste d'affectations nouvellement enrichie sert pour l'appel de l'interprétation de la queue de la liste de définitions. Lorsque la liste de définitions est vide, l'interprétation est égale à la liste d'affectations courante.

```
op interp : Def-list Interp -> Interp .
eq interp(d " dl,al) =
        interp(d1,ident(d) => interp(term(d),al)) " al) .
eq interp(d-emptylist,al) = al .
```

Interprétation d'un terme : on lance l'interprétation du symbole fonctionnel du terme avec la liste des valeurs des arguments extraites de la liste des affectations déjà calculées. Le résultat est une *FFlist* c'est-à-dire plusieurs solutions réelles éventuelles pour l'objet. Par exemple pour un point, `(5 ;3) | (4 ;2)` veut dire qu'il peut avoir pour coordonnées soit `(5 ;3)` soit `(4 ;2)` (un résultat multiple étant le fruit d'une multifonction).

```
op interp : Term Interp -> FFlist .
eq interp(f[il],al) = interp(f,extract(il,il,al)) .
```

Interprétation du symbole fonctionnel : on appelle la fonction adéquate avec la liste des valeurs correspondant aux arguments. Nous ne donnerons pas ici une liste des opérations correspondant à tous les symboles fonctionnels. Le lecteur intéressé peut la trouver à l'Annexe B. En revanche nous donnons un exemple, l'opération correspondant au symbole fonctionnel `lpp` (droite passant par deux points), sachant que les sortes géométriques sont interprétées par les n-uplets présentés à la Table 3.3.

```
op interp : F FFFlist -> FFlist .
op interp(lpp,(x1 ; y1) & (x2 ; y2)) =
                if x1 == x2 and y1 == y2
                then fail
                else (y2-y1 ; x1-x2 ; y1*(x2-x1) - x1*(y2-y1))
                fi .
```

TAB. 3.3: N-uplets correspondant aux sortes géométriques

sorte	n-uplet	commentaire
Point	(x, y)	coordonnées dans le plan, x : abscisse, y : ordonnée
Line	(a, b, c)	coordonnées de la droite dans le plan
Circle	(x, y, r)	coordonnées (x, y) du centre et rayon r
Long	(l)	longueur réelle
Angle	(a)	réel en radians compris entre -π et π
Depl	(x_1, y_1, a, x_2, y_2)	(x_1, y_1) coordonnées de l'origine du repère source, (x_2, y_2) coordonnées de l'origine du repère destination, a angle orienté dans le sens source-destination entre les directions des deux repères
Num	(x)	réel

Les symboles fonctionnels sont interprétés par des fonctions dont le nombre de résultats potentiels est égal au degré de multiplicité du symbole. Ces fonctions peuvent, selon les valeurs passées en argument, ne donner aucun résultat, comme par exemple l'intersection

de deux droites parallèles. Dans ce cas, la valeur renvoyée est `fail`. La constante `fail` est considérée comme une valeur enrichissant l'ensemble des réels.

Les symboles fonctionnels peuvent être classés en deux catégories selon leur degré de multiplicité. La première catégorie regroupe ceux dont la fonction d'interprétation a au maximum une solution, la deuxième regroupe ceux dont la fonction d'interprétation est une multifonction.

Parmi la première catégorie, on peut encore classer les fonctions d'interprétation par familles. Ce sont les fonctions d'entrée (ou d'initialisation), telles que *initp*, les fonctions calculant un déplacement ou en appliquant un, comme *make-depl-pp* ou *transfp*, les fonctions de calcul de distances et d'angles (*distpp, bissect*), ou des fonctions diverses retournant un point, une droite, ou un cercle, comme *interll, mediatrice*, ou encore *mkcir*.

La deuxième catégorie est celle des "vraies" multifonctions, c'est-à-dire des fonctions pouvant renvoyer plusieurs résultats. Le nombre de résultats renvoyés par une multifonction n'est pas fixe, il peut varier en fonction des positions relatives des objets géométriques qui sont ses arguments. Il ne peut cependant pas dépasser le nombre maximum de solutions qui est appelé *degré de multiplicité*. Nous travaillons actuellement sur des multifonctions dont le degré de multiplicité est inférieur ou égal à 4. Parmi celles-ci, on trouve (la liste n'est pas exhaustive) des multifonctions de déplacement (*make-dep-pl*), de construction de cercles (*medradcir*), des multifonctions renvoyant des points (*interlc*), des droites (*bissectdd,ltangent*).

On peut aisément comprendre que le nombre de solutions peut exploser rapidement. En effet, le nombre maximal de solutions potentielles pour une fonction de construction (c'est-à-dire dans l'hypothèse où aucune fonction partielle ne connaîtrait d'échec et où toutes les multifonctions auraient un nombre de solutions égal à leur degré de multiplicité) est égal à $\prod_{i=0...n} d_i$, où n est le nombre de symboles fonctionnels et d_i sont les degrés de multiplicité de chacun. Ainsi, afin de trouver plus rapidement et plus aisément la solution désirée, il pourrait donc être souhaitable, dans un premier temps, d'améliorer le parcours dans l'arbre des solutions. La section suivante propose différentes manières, statiques ou dynamiques, de réorganiser le plan de construction.

3.3 Réorganisation du plan de construction par tri topologique

Le programme de construction tel qu'il est produit par *YAMS* ne se situe pas à un niveau aussi abstrait que le NADS que nous avons vu précédemment. Le passage du NADS abstrait au plan se fait par tri topologique. On peut alors exploiter le fait qu'il y ait plusieurs tris possibles pour essayer de minimiser tel ou tel critère dont on pense qu'il aura un impact sur l'efficacité. Nous proposons à la suite quelques heuristiques (assez naïves), qui ont surtout pour but de montrer que l'on peut manipuler les plans de construction pour essayer d'optimiser un point ou un autre, mais dont l'objectif reste cependant d'optimiser

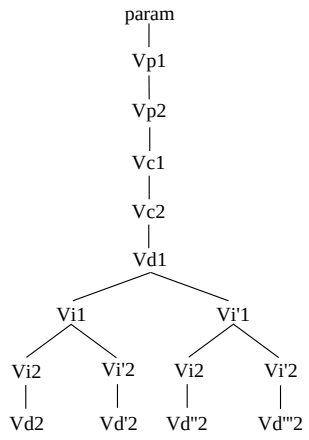


Fig. 3.7: Arbre des solutions réduit

l'efficacité du parcours de l'arbre des solutions.

3.3.1 Heuristiques pour le tri topologique

On remarque tout d'abord dans l'exemple précédent que le nombre de nœuds n'est pas minimal. On aurait en effet pu intervertir les lignes de $d1$ et $i1$ car $d1$ ne dépendant pas de $i1$, le fait que $i1$ ait deux solutions ne change rien au calcul de $d1$. Ainsi, si dans le plan de construction on permute les définitions de $d1$ et de $i1$, alors cela aura pour conséquence de réduire le nombre de noeuds dans l'arbre des solutions qui devient celui de la Fig.3.7.

Réduire le nombre de nœuds peut permettre d'avoir un backtracking plus efficace. Lorsqu'on parcourt une branche qui aboutit à un échec, le backtracking est alors utilisé pour remonter dans l'arbre et ainsi continuer sur une autre branche. En remontant vers le sommet les définitions qui ont les degrés de multiplicité les plus faibles, sans perturber le mécanisme des dépendances, aura pour conséquence de faire descendre vers les feuilles les embranchements, et donc de réduire le nombre de noeuds et les distances à parcourir lors du backtracking. C'est le but de la première heuristique.

Une autre façon de voir est de faire en sorte que les fonctions qui doivent échouer le fassent au plus vite dans l'arbre afin d'éviter de faire des quantités de calculs avant de s'apercevoir que dans tous les cas nous aurons un échec. C'est ici qu'intervient le degré de risque des fonctions. C'est un coefficient arbitraire qui est fixé au départ et qui classe les fonctions les unes par rapport aux autres en fonction des chances qu'elles ont de ne pas aboutir. Les fonctions qui n'ont aucun risque d'échouer ont un degré égal à 0, comme la distance entre deux points. Certaines fonctions sont dites plus "risquées" que d'autres, par exemple l'intersection de deux droites a plus de chances d'aboutir que l'intersection de deux cercles, car dans le premier cas la seule possibilité d'échouer est que les deux droites soient parallèles tandis que dans le deuxième cas il suffit que les deux cercles soient disjoints,

ce qui arrive plus fréquemment. Les degrés de *interll* et *intercc* ont donc respectivement été fixés à 2 et 4. Notons qu'une idée très similaire sera reprise à la Section 3.3.3 pour réorganiser la fonction de construction de manière dynamique, en fonction des échecs avérés des multifonctions en cours d'interprétation.

3.3.2 Tri topologique

Le tri topologique est donc toujours le même, seule l'opération de choix `choice` est modifiée pour prendre en compte les nouvelles heuristiques. De la même façon qu'auparavant, on va donc créer un ensemble d'éléments qui pourront être placés au vu du plan partiel déjà construit, mais au lieu de prendre le premier élément valide de cet ensemble pour le placer on va le choisir selon un critère prédéfini.

Il reste à décrire l'algorithme de choix. Celui-ci peut suivre deux heuristiques dont nous avons parlé au paragraphe précédent. Soit c'est un choix selon le risque, soit c'est un choix selon la multiplicité. Les algorithmes de choix prennent en entrée un ensemble de définitions (les éléments potentiels) ainsi qu'une liste d'identificateurs (ceux des définitions déjà triées) et renvoient une définition (celle qui est choisie).

Choix selon le risque :

```
 obj RISK-CHOICE is protecting NADS .
 op risk-choice : Def-set Ilist -> Def .
 var d1 d2 : Def .
 var ds : Def-set .
 var il : Ilist .
 eq risk-choice(d1,il) = d1 .
 eq risk-choice(d1 d2 ds,il) =
    if ((is-param(d1) or are-in-list(args-term(d1),il))
        and (risk(d1) <= risk(d2)))
    then risk-choice(d1 ds,il)
    else risk-choice(d2 ds,il)
    fi .
endo
```

Choix selon la multiplicité :

```
 obj MULTI-CHOICE is protecting NADS .
 op multi-choice : Def-set Ilist -> Def .
 var d1 d2 : Def .
 var ds : Def-set .
 var il : Ilist .
```

```
 eq multi-choice(d1,il) = d1 .
 eq multi-choice(d1 d2 ds,il) =
    if ((is-param(d1) or are-in-list(args-term(d1),il))
        and (multi(d1) >= multi(d2)))
    then multi-choice(d1 ds,il)
    else multi-choice(d2 ds,il)
    fi .
endo
```

Bien entendu, ce n'est pas une liste exhaustive des algorithmes de choix que nous présentons ici, il est possible d'en implanter d'autres en fonction des besoins. On peut également faire un algorithme hybride qui combinerait ces deux heuristiques, ou d'autres. Par exemple, on peut proposer `COMB-CHOICE`, qui combinerait les deux heuristiques précédentes en considérant qu'une d'elles serait prioritaire, et que l'autre servirait à départager les derniers éléments satisfaisant de façon égale la première. Il est également envisageable, vu le fonctionnement de *YAMS* par assemblage de sous-figures, d'appliquer une heuristique à chaque sous-plan séparément et une autre sur l'ensemble des sous-plans.

3.3.3 Approche dynamique

Les heuristiques que nous venons de voir font partie de ce qu'on appelle les heuristiques statiques. Il existe une autre classe d'heuristiques, qui peut être utilisée pour réduire le nombre de nœuds : celle des heuristiques dynamiques, basées sur l'apprentissage par l'échec. Une collaboration avec Laure Brisoux-Devendeville nous a permis d'étudier une heuristique dynamique inspirée des techniques de SAT, et qui a donné lieu à une publication [BDEVS00].

L'idée de départ est de considérer le plan de construction selon un point de vue différent. On peut remplacer une définition $y = f(x_1, \ldots, x_n)$ par la conjonction de contraintes finies $y \in \{t_1, t_2, \ldots, t_m\}$ et $y \rightarrow x_1, \ldots, y \rightarrow x_n$, où $t_i = F(x_1, \ldots, x_n, i)$, et $\rightarrow$ est toujours la relation de dépendance définie à la Section 3.1.4. Autrement dit, on considère y comme appartenant à un ensemble de valeurs qui sont les résultats des fonctions partielles de f (voir Définition 7), et comme ayant pour contraintes de dépendre directement de $x_1, \ldots, x_n$. Comme conséquence, nous avons également les relations $x_1 << y, \ldots, x_n << y$.

La relation d'appartenance avec la relation d'ordre effectif $<<$ correspond exactement à un plan de construction, tandis que la relation d'appartenance avec la relation de dépendance correspond à tous les ordres d'interprétations possibles, c'est-à-dire au graphe de dépendance. La relation algorithmique entre $\rightarrow$ et $<<$ est que l'ordre des variables définies dans un plan de construction est le résultat d'un tri topologique sur le graphe de dépendance.

Rappelons tout d'abord quelques notions concernant le problème SAT. SAT consiste à vérifier la satisfiabilité d'une formule booléenne de forme conjonctive normale (CNF).

Une formule CNF est un ensemble (interprété comme une conjonction) de clauses, où une clause est une disjonction de littéraux. Un littéral est une variable propositionnelle positive ou négative.

Une interprétation d'une formule booléenne est l'affectation de valeurs de vérité à ses variables. Un modèle est une interprétation qui satisfait la formule. SAT consiste donc à trouver un modèle pour une formule CNF lorsqu'un tel modèle existe, ou à prouver qu'un tel modèle n'existe pas. SAT est un problème NP-complet, ce qui veut dire que tous les algorithmes pour le résoudre pourraient être exponentiels dans le pire des cas. Cependant, toutes les instances de SAT ne font pas preuve de la même difficulté, selon les algorithmes utilisés.

Les techniques complètes les plus efficaces sont basées sur la procédure classique de Davis Logemann Loveland appelée DPLL [MSG97]. DPLL consiste à développer un arbre de recherche binaire, dont les sommets sont étiquetés par un littéral représentant leur affectation dans l'interprétation partielle, et dont les nœuds représentent l'ensemble des clauses qui n'ont pas été satisfaites par l'interprétation partielle. Un des points clés de l'efficacité dans la procédure DPLL tient à sa stratégie d'embranchement. Une de ces règles d'embranchement, proposée par L. Brisoux *et al.* [BGS99], peut être décrite de la façon suivante : lorsque des clauses ont été démontrées comme étant insatisfiables à certaines étapes du processus de recherche, cette information ne doit pas être oubliée pour la suite du processus. Au contraire, dans le développement d'autres branches de l'arbre de recherche il peut être efficace d'essayer de rencontrer de nouveau dès que possible ces situations d'insatisfiabilité. Ceci peut être effectué en sélectionnant en priorité les littéraux apparaissant dans ces clauses.

Cette idée peut être réalisée de façon assez simple en ajoutant un poids β_c à chaque clause. Chaque fois que DPLL sélectionne une variable propositionnelle qui amènerait immédiatement à une inconsistance, chaque clause initiale c de l'instance SAT qui serait montrée insatisfiable à cette étape de l'arbre de recherche voit sont facteur β_c incrémenté d'une valeur γ, et son importance est augmentée dans la suite de la recherche.

Nous avons étendu cette heuristique à notre problème en introduisant un poids à chaque définition. Nous proposons de réorganiser les définitions en prenant en compte ce nouvel élément. La motivation est la même que pour le problème SAT : favoriser les définitions qui ont déjà mené à un échec à certaines étapes du parcours, sans avoir besoin d'autre chose pour réduire l'arbre. Cette idée se rapproche de l'heuristique statique de choix selon le risque. Cependant, la façon de classer le risque des multifonctions n'est plus arbitraire et décidée une fois pour toutes, mais elle s'adapte en fonction du système rencontré, et elle peut évoluer au cours du parcours. Elle est donc mieux adaptée à chaque cas.

Au début, le poid de chaque définition est fixé à une constante c, comme dans SAT. Lorsque la fonction de la définition n'a pas de solution (c'est un échec), son poids est augmenté de 1 et le plan est réorganisé de manière à prendre en compte ce poids, tout en respectant toujours la précédence des arguments puisque cette définition vérifie toujours

Algorithme 1 : Procédure interpretation(P)

```
Préconditions: soit P un plan de construction
Postconditions: affiche toutes les solutions de P si elles existent
 def ← tete(P)
 premiere_boucle ← vrai
 tant que pile_non_vide ou premiere_boucle faire
   tant que def existe faire
     premiere_boucle=faux
     out ← calcule_solutions(@def)
     si out = pas_de_solution alors
       augmente_poids(def)
       reorganise(def,P)
       def ← depile_apres_reorganisation()
     sinon
       def.position_solution ← next(def.position_solution)
       si def.position_solution n'existe pas alors
         def ← depiler
       fin si
       ajouter_aux_solutions_calculees(def.position_solution)
       si out = une_solution alors
         def ← suivant_dans(P)
       sinon
         si out = plus_d_une_solution alors
           empiler(def)
           def ← suivant_dans(P)
         sinon
           ⇝ out = plus_d_une_solution_deja_depile
           def ← suivant_dans(P)
         fin si
       fin si
     fin si
   fin tant que
   afficher_solutions_calculees()
 fin tant que
```

$\rightarrow$. Cette procédure est décrite plus formellement dans l'algorithme 1. Elle n'a cependant pas été spécifiée en *OBJ3*, car le backtracking intelligent utilisé, qui rappelons-le permet de tirer parti au mieux des possibilités offertes par la relation $\rightarrow$, la rend très difficile à spécifier. C'est pourquoi nous avons préféré l'ecrire en pseudo-code.

La procédure `calcule_solutions(@def)` (où @ représente l'adresse d'une variable) met à jour la variable `def.solutions` dans laquelle les solutions sont stockées (la position de la solution courante est indiquée par `def.position_solution`, et le nombre de solutions est mis dans `out`). Selon les valeurs qu'elle retourne, il peut être décidé de réorganiser le plan de construction par empilement/dépilement sur une pile de définitions (pour simuler la récursivité). La fonction `depile_apres_reorganisation()` dépile toutes les valeurs stockées dans la pile, entre l'ancienne position de `def` et la nouvelle. A la fin, le sommet de la pile résultante est retourné.

La fonction `augmente_poids` incrémente de poids d'une définition de 1, tandis que la fonction `reorganise(def,P)` essaie de trouver le meilleur emplacement dans le plan de construction de façon à ce que toutes les définitions soient classées par poids décroissant autant que possible, et que $\rightarrow$ soit respectée.

Soit def la définition à remonter dans le plan P. Si le prédécesseur de def dans P n'est pas un argument de def et que son poids est inférieur, alors la définition def peut être insérée avant lui. Cette procédure peut être répétée jusqu'à ce que la bonne place soit atteinte. Nous obtenons la procédure présentée à l'Algorithme 2.

Algorithme 2 : PROCÉDURE REORGANISE(DEF,P)

Préconditions: def est une définition appartenant au plan de construction P
Postconditions: P est réorganisé en tenant compte les poids
courante $\leftarrow$ def
ici $\leftarrow$ courante
pred $\leftarrow$ courante
fin $\leftarrow$ faux
tant que non fin et (poids(courante) $>$ poids(pred)) et non_sommet_de(P) **faire**
  si respecte_NADS(courante) **alors**
    pred $\leftarrow$ courante
    courante $\leftarrow$ avant(courante)
  sinon
    ici $\leftarrow$ pred
    fin $\leftarrow$ vrai
  fin si
fin tant que
si ici $\neq$ def **alors**
  inserer_(def,ici)
fin si

Les trois procédures auxiliaires utilisées sont : la procédure `poids(def)`, qui retourne

le poids de la définition `def` ; la procédure `respecte_NADS(def)`, qui retourne vrai si et seulement si la relation $\rightarrow$ est toujours respectée lorsque la définition `def` est déplacée ; la procédure `insérer(def,ici)`, qui insère la définition `def` après la définition `ici` dans le plan de construction.

Ces algorithmes sont transparents pour l'utilisateur. Les contraintes finies sont gérées par le logiciel afin de réduire le nombre de calculs durant une éventuelle exploration complète de l'arbre des solutions. Ces techniques ne permettent cependant pas de réduire le nombre de solutions dans cet arbre. Elles seront toutefois utiles dans le cas où les méthodes d'élagage automatique de cet arbre, qui seront abordées au Chapitre 4, ne seraient pas suffisantes pour réduire l'arbre à une seule branche, mais produiraient un petit sous-arbre, ou encore si l'utilisateur souhaite voir toutes les solutions. Dans ces situations, elles permettront de rendre plus efficace le parcours des solutions.

Notre collaboration s'etant achevée un peu trop tôt, cette direction de recherche n'a pas été poursuivie plus avant. Nous avons en effet ensuite privilégié l'étude des méthodes automatiques de sélection de solutions. Cependant, c'est une voie intéressante qui mériterait d'être plus approfondie.

Chapitre 4

Recherche automatique d'une solution

Comme nous l'avons vu précédemment, il peut résulter de l'interprétation du plan de construction une très grande quantité de solutions, présentée sous la forme d'un arbre. Malheureusement, cet arbre ne peut pas toujours être élagué grâce à de simples tests de vérifications de contraintes. C'est pourquoi nous souhaitons trouver un critère de sélection de la "bonne" solution parmi l'arbre des solutions, sans avoir à les passer en revue une par une.

Il s'agit d'une direction de recherche différente de celle vue au chapitre précédent (l'amélioration de l'efficacité du backtracking). Elles peuvent cependant être tout à fait complémentaires dans le cas où, à défaut de trouver la "bonne" solution, on obtiendrait tout de même quelques solutions, c'est-à-dire un arbre réduit dans lequel on aurait éliminé la plus grande partie possible des branches. Dans ce cas, il faudrait tout de même effectuer un backtracking sur le sous-arbre restant.

4.1 Ressemblance

Commençons par définir ce que nous entendons par "la bonne figure". Il s'agit de la figure que l'utilisateur souhaite obtenir, celle qu'il a en tête au moment où il trace son esquisse, celle qui va répondre aux contraintes qu'il a données tout en ressemblant le plus possible à l'esquisse qu'il a tracée. En effet, nous partons de l'idée que l'utilisateur ne dessine pas une figure à la souris totalement au hasard. Il trace une esquisse qui ressemble à ce qu'il souhaite obtenir. Nous nous efforçons donc de chercher quelle est, parmi les solutions, la figure qui ressemble le plus à l'esquisse. Avant d'expliquer la méthode que nous utilisons pour atteindre ce but, nous devons tout d'abord définir notre vision de la ressemblance entre deux figures.

4.1.1 Critère usuel de ressemblance

La similarité est définie par la plupart des dictionnaires comme la conformité en nature ou en apparence entre les choses. Deux figures sont souvent dites semblables si elles ont en commun certaines propriétés géométriques, telles que le placement relatif des points, des droites et des cercles, l'acuité des angles, et la convexité de certaines parties de la figure. Inversement, deux figures ne sont pas similaires si une des caractéristiques est vérifiée par l'une et pas par l'autre.

Cette définition intuitive a souvent été proposée afin d'éliminer des solutions qui ne semblaient pas "intéressantes" dans le cadre de la CAO (voir Section 1.3). De nombreux auteurs l'utilisent comme base à la sélection de solutions, comme J. Owen dans [Owe91] ou C. Hoffmann dans [BFH+95]. Nous remarquons cependant que la plupart de ces comparaisons de propriétés peuvent être mises en échec par des exemples très simples.

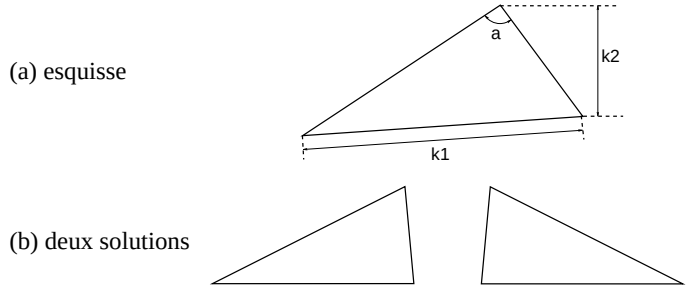


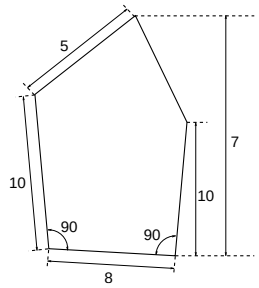
Fig. 4.1: Manque de critère de discrimination

Exemple 8 (manque de critère de discrimination) *Sur la Fig.4.1, tous les angles sont aigus et tous les points ont le même placement relatif. Ces critères de comparaison ne sont donc pas suffisants pour décider automatiquement quelle solution était requise. A notre sens, la plus ressemblante paraît tout de même être celle de gauche.*

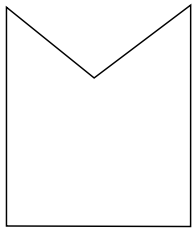
Exemple 9 (défaut de convexité) *Sur la Fig.4.2 l'esquisse a un défaut de convexité par rapport à la solution, alors que la solution proposée est la seule possible. Une comparaison des convexités des deux figures aurait donc éliminé la seule solution possible.*

Exemple 10 (défaut d'acuité) *Sur la Fig.4.3, l'esquisse présente un défaut d'acuité, alors que la solution proposée est la seule possible. Une comparaison des acuités des angles sur les deux figures aurait donc là encore éliminé la seule solution possible.*

Ces critères intuitifs ne sont donc pas satisfaisants. La section suivante propose un meilleur critère, basé sur l'homotopie. Remarquons que la vérification de propriétés géométriques sur une esquisse donnée par l'utilisateur, appelée *vérification sémantique*, a déjà été utilisée dans le cadre des preuves automatiques de théorèmes géométriques [Bun83].

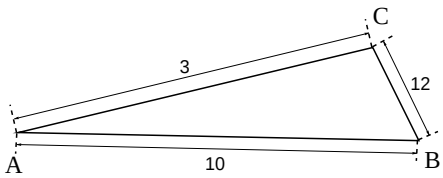


(a) esquisse (polygone convexe)

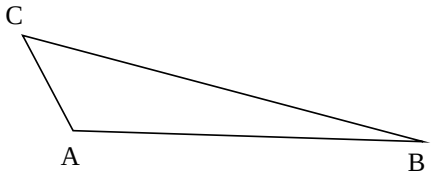


(b) solution (polygone concave)

Fig. 4.2: Défaut de convexité



(a) esquisse (angles en C obtus et en A aigu)



(b) une solution (angles en C aigu et en A obtus)

Fig. 4.3: Défaut d'acuité

4.1.2 L'homotopie en tant que notion de ressemblance

En géométrie, plusieurs notions de ressemblance entre les objets existent, en fonction du degré d'abstraction considéré. Nous essayons de définir un critère prenant exactement en compte notre cadre, c'est-à-dire les figures géométriques qui sont solutions de systèmes de contraintes.

En ne considérant que les propriétés topologiques, la déformation continue appelée homotopie est un concept de ressemblance habituel. Rappelons la définition de l'homotopie, que nous avons déjà évoquée brièvement à la Section 1.2.1 :

Définition 2 *(homotopie) Soient $\mathcal{P}$ un espace topologique, $f_0 : [0,1] \to \mathcal{P}$ et $f_1 : [0,1] \to \mathcal{P}$ deux courbes paramétrées continues. f_0 et f_1 sont dites* homotopes *s'il existe une fonction $\varphi : [0,1] \times [0,1] \to \mathcal{P}$ telle que $\varphi(x,0) = f_0(x)$, $\varphi(x,1) = f_1(x)$, et φ est continue. Alors φ est appelée une* homotopie.

Un point intéressant de cette définition est qu'elle fait un lien entre une notion locale de proximité dans le plan euclidien et une notion globale. Cet aspect nous semble très

intéressant étant donné qu'en CAO l'esquisse peut être très éloignée de la solution (voir Fig.4.2-4.3). Bien sûr, cette définition est bien trop générale pour nous, puisqu'elle ne prend pas en considération les propriétés géométriques élémentaires des objets, et plus particulièrement leur type. Par exemple, un cercle est homotope à un triangle, et ceci n'a pas de sens pour un utilisateur en CAO. Nous devons donc affiner cette caractérisation pour notre cadre de constructions géométriques.

4.1.3 L'homotopie géométrique

Tout d'abord, éclaircissons un peu notre notion de type géométrique. Dans [DMS97], une figure est décrite comme un n-uplet d'objets géométriques tels que les points, les droites, les cercles, etc. Le type d'une figure est donc le produit cartésien de types simples (*point*, *segment*, *cercle*, etc.), auxquels nous ajoutons les relations d'incidence.

Exemple 11 *Le triangle ABC de la Fig.4.3 a pour type point* $\times$ *point* $\times$ *point* $\times$ *segment* $\times$ *segment* $\times$ *segment avec les contraintes d'incidences appropriées.*

À l'aide d'un système de coordonnées, nous pouvons définir une topologie métrique, de laquelle découle une notion de proximité. Nous pouvons désormais donner notre définition de l'homotopie géométrique.

Définition 3 (homotopie géométrique) *Soient deux figures* f_0 *et* f_1 *de même type* $\tau_1 \times \tau_2 \times \ldots \times \tau_n$ *avec les mêmes relations d'incidence, on dit que* f_0 *et* f_1 *sont* géométriquement homotopes *s'il existe une transformation continue* $\varphi : [0,1] \to \tau_1 \times \tau_2 \times \ldots \times \tau_n$ *préservant les relations d'incidence, telle que* $\varphi(0) = f_0$ *and* $\varphi(1) = f_1$.

La transformation φ de cette définition peut être vue comme un n-uplet de transformations continues $(\varphi_1, \ldots, \varphi_n)$ pour les types $\tau_1, \ldots, \tau_n$.

Exemple 12 *Sur la Fig.4.4, la figure composée d'un triangle* $p1p2p3$ *et de son cercle circonscrit* C *est considérée comme ayant le type point* $\times$ *point* $\times$ *point* $\times$ *segment* $\times$ *segment* $\times$ *segment* $\times$ *circle. Nous déformons la figure en translatant* $p3$ *en* $p'3$*, ce qui déforme aussi* C *en* C'*. Les transformations continues appliquées sur les composants sont : identité sur* $p1$ *et* $p2$*, translation* $\overrightarrow{p3p'3}$ *sur* $p3$*, identité sur* $[p1p3]$*, similitudes de centres* $p1$ *et* $p2$ *sur* $[p1p3]$ *et* $[p2p3]$ *respectivement, et une homothétie sur* C *dont le centre est sur la médiatrice de* $[p1p2]$.

Notons que la Définition 3 peut caractériser les déformations géométriques interactives telles que celles utilisées dans *Cabri-Géomètre* pour découvrir des invariants géométriques [SG94]. La continuité de telles déformations est en particulier étudiée dans [Kor99]. Dans

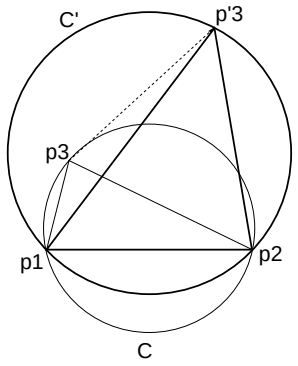


Fig. 4.4: Homotopie géométrique

une optique un peu différente, Kramer [Kra92] évoque ce qui peut être vu comme un cas particulier de déformation continue, où les contraintes correspondent à des articulations.

Cependant, pour notre part, nous ne traitons pas des figures simples : nous devons garder à l'esprit le système de contraintes donné par l'utilisateur, aussi bien que les figures. Nous ne pouvons donc pas considérer uniquement une seule solution, mais aussi sa position dans l'espace des solutions tout entier.

4.2 Déformations contraintes

4.2.1 Déformation continue d'un système de contraintes

Si nous voulons étudier les déformations continues d'une figure cotée, nous devons définir non seulement la déformation continue d'une figure, mais aussi celle de son système de contraintes. Tous les deux doivent être soumis à une déformation continue, depuis une solution particulière du système instancié par un n-uplet de valeurs u, jusqu'à une autre solution particulière du système instancié par un n-uplet de valeurs v. En fait, la notion de déformation d'un système concerne la classe des systèmes équivalents toute entière. Nous proposons donc la formalisation suivante. Notons que cette notion est voisine de la méthode homotopique présentée à la Section 1.2.1, à la différence que dans notre cas, on a une déformation explicite en fonction des paramètres U du système.

Définition 4 (Déformation continue d'un système de contraintes) *Soit $S = (U, X, C)$ un système de contraintes géométriques instancié en S_u et S_v par deux n-uplets de valeurs u et v pour U. Une* déformation continue *de S_u à S_v est une fonction continue $\psi : [0,1] \to \mathbb{R}^s$ telle que :*

(i) $\psi(0) = u$ *et* $\psi(1) = v$

(ii) pour chaque système S' tel que $S' \equiv S$, tout sous-système bien contraint $\sigma' \subseteq S'$, et tout $t \in [0,1]$, $\sigma'_{\psi(t)}$ a autant de solutions réelles distinctes que σ'_u.

Rappelons que S et S' sont dits équivalents, ce qui est noté $S \equiv S'$, s'ils ont exactement les mêmes solutions réelles distinctes quelles que soient les valeurs des paramètres de u (cf. Section 2.1). Le point (*ii*) de la Définition 4 est très fort puisqu'il impose à tout sous-système σ' du système global S (ou d'un système équivalent S') de garder le même nombre de solutions au cours de la déformation. En fait, une fois que les paramètres sont fixés, disons à u, l'espace des solutions peut être vu comme la classe de tous les systèmes équivalents à S_u. Alors, la définition 4 peut être vue comme la déformation continue d'un espace de solutions. Le lemme suivant, découlant directement de la définition, nous sera utile par la suite.

Lemme 1 *Soient S et S' deux systèmes equivalents avec le même ensemble de paramètres, et u et v deux instanciations de ces paramètres. S'il existe une déformation continue de S_u vers S_v, alors il existe une déformation continue de S'_u vers S'_v.*

4.2.2 La S-homotopie et ses propriétés

Finalement, la notion de déformation d'un système de contraintes est liée à celle de déformation géométrique d'une figure de la façon suivante.

Définition 5 (S-homotopie) *Soient S_u et S_v deux instances différentes d'un système de contraintes S, f_u une solution particulière de S_u, et f_v une solution particulière de S_v. Si les conditions suivantes sont satisfaites*

(i) *il existe une déformation continue ψ de S_u vers S_v*

(ii) *il existe une fonction continue $\varphi : [0,1] \rightarrow \tau_1 \times \tau_2 \times \ldots \times \tau_n$ telle que $\varphi(0) = f_u$ et $\varphi(1) = f_v$*

(iii) *$\forall t \in [0,1]$, $\varphi(t)$ est une solution de $S_{\psi(t)}$*

alors on dit qu'il existe une homotopie géométrique de f_u vers f_v par rapport au système de contraintes S, autrement dit une S-homotopie.

En d'autres termes, la déformation par S-homotopie d'un système de contraintes ne doit pas atteindre de cas dégénéré (dont la définition est donnée ci-dessous), car si cela se produit on peut sauter d'une solution à une autre au lieu de suivre la même solution.

Définition 6 (Cas dégénéré) *Soit g une multifonction à n arguments avec un maximum de k résultats. On dit qu'on atteint un* cas dégénéré *si et seulement si les arguments $x_1, \ldots, x_n$ de g sont tels que* $\mathcal{C}ard\Big(g(x_1, \ldots, x_n)\Big) < k$.

Autrement dit, on atteint un cas dégénéré en $(x_1, \ldots, x_n)$ si le nombre de solutions réelles distinctes de g n'atteint pas son maximum pour ces valeurs. Par exemple, la fonction *intercc* qui construit l'intersection entre une droite et un cercle peut fournir jusqu'à 2

solutions, et atteint un cas dégénéré lorsque la droite est tangente au cercle puisqu'il n'y a alors qu'une seule solution possible.

Notons qu'il existe un lien de parenté très fort entre la S-homotopie et la méthode par homotopie utilisée pour résoudre des équations (*cf.* Section 1.2.1). Cependant, ici, la S-homotopie n'est pas utilisée comme méthode de résolution, mais comme critère de ressemblance.

Quelques exemples Nous présentons ici quelques exemples simples, afin de mieux appréhender les notions de S-homotopie et de déformation continue de systèmes de contraintes que nous venons de définir.

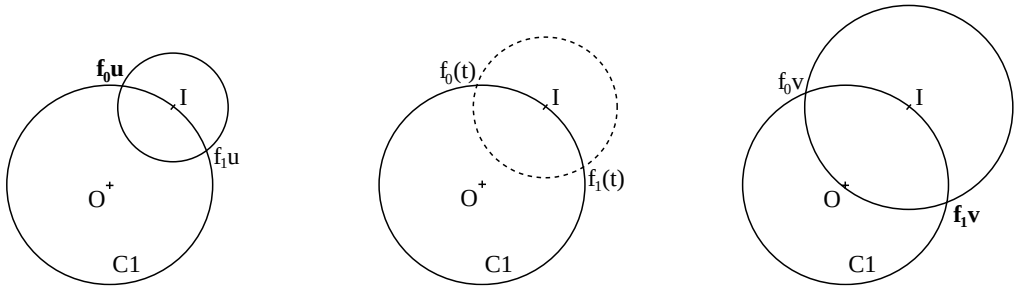
Exemple 13 *Soit le cercle $C1$ de rayon r et de centre O sur lequel se trouve un point I. Soit un point M contraint à se situer sur $C1$ à une distance d de I. M est donc l'intersection entre $C1$ et le cercle $C2$ de centre I et de rayon d. Il existe deux solutions possibles pour M, de part et d'autre de I sur $C1$. Tout ceci constitue un système de contrainte que l'on nommera S et qui est présenté Fig.4.5.*

Soient deux instanciations S_u et S_v de S, pour lesquelles d vaut $k1$ et $k2$ respectivement, et qui possèdent chacune 2 solutions f_0u, f_1u (Fig.4.5(a)) et f_0v, f_1v (Fig.4.5(c)). Il existe une déformation continue de S_u à S_v, car elles possèdent toutes les deux 2 solutions, et chaque système intermédiaire a aussi 2 solutions (Fig.4.5(b)). Alors f_0u et f_0v sont S-homotopes, de même que f_1u et f_1v. Par contre, f_0u n'est pas S-homotope à f_1v par exemple (ces deux solutions sont en gras sur les Fig.4.5(a) et (c)), car on rencontrerait un cas dégénéré lors de la déformation continue de f_0u en f_1v. Il existe en effet un cas dégénéré lorsque $d = 0$ ou $d = 2r$ (cercles tangents) : il n'y a alors qu'une seule solution au lieu de deux.

Exemple 14 *Soit le cercle $C1$ de l'exemple précédent. Cette fois-ci, le point M doit être tel que l'angle OI, OM vaut α. Il n'existe ici qu'une seule solution pour M. Ce système S' peut être instancié en S'_u et S'_v pour deux valeurs différentes a_1 et a_2 de α. S'_u et S'_v possèdent alors chacune une seule solution, g_0u (Fig.4.6(a)) et g_0v (Fig.4.6(c)) respectivement. Il existe une déformation continue de S'_u à S'_v, et g_0u et g_0v sont S-homotopes car on ne rencontre aucun cas dégénéré lors de la déformation de g_0u en g_0v (cf. Fig.4.6(b)).*

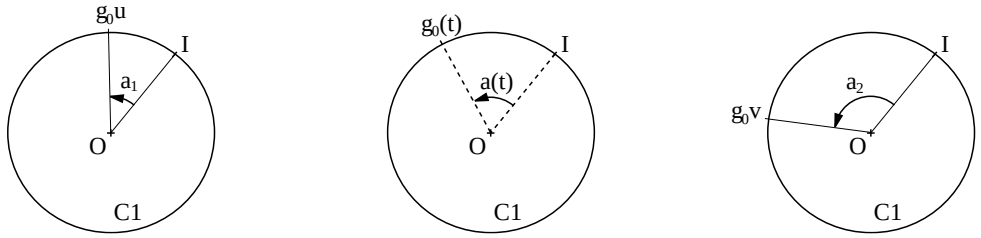
Notons qu'avec ces contraintes il est possible, selon la valeur donnée à α, d'obtenir les mêmes solutions que dans l'exemple 13, par exemple $g_0u = f_0u$ et $g_0v = f_1v$ (cf. Fig.4.6(d), en gras). Dans ce cas, les deux solutions sont S-homotopes.

Exemple 15 *Sur la Fig.4.7(a), le triangle ABC donné comme esquisse possède trois contraintes : la longueur BC égale k_1, la distance entre la droite $l_1 = (BC)$ et le point A égale k_2, et l'angle orienté en A égale a. Les deux solutions possibles, f_0 et f_1, sont présentées aux Figs.4.7(b) et (d). Lorsque B et C sont connus, k_1 étant donc fixe, A peut être construit comme l'intersection entre la droite l_2, qui est à la distance k_2 de l_1, et l'arc associé à a. Il est possible de déformer continûment la première solution f_0 en translatant*

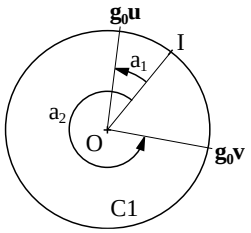


(a) 2 solutions pour S_u (b) figure intermédiaire (c) 2 solutions pour S_v

FIG. 4.5: Déformation continue de S_u en S_v



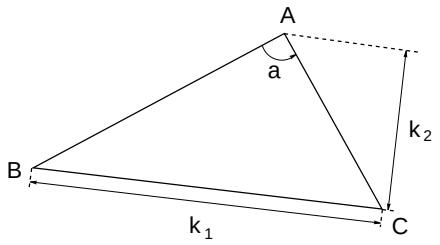
(a) 1 solution pour S'_u (b) figure intermédiaire (c) 1 solution pour S'_v



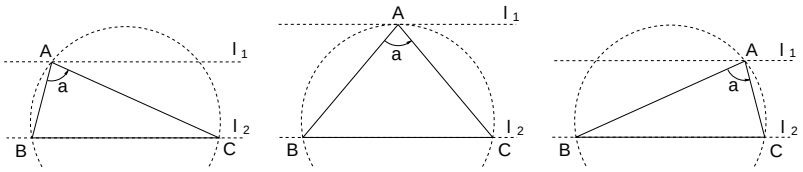
(d) valeurs particulières pour α

FIG. 4.6: Déformation continue de S'_u en S'_v

l_1 (en faisant varier k_2). Mais pour atteindre la seconde solution f_1, on passe par le cas dégénéré que constitue la tangence de l_1, comme le montre la Fig.4.7(c). Donc, f_0 et f_1 ne sont pas S-homotopes. Nous expliquerons plus loin pourquoi la solution que nous garderions probablement est f_1.



(a) esquisse



(b) 1° solution : f_0 (c) cas dégénéré (d) 2° solution : f_1

FIG. 4.7: Déformation continue passant par un cas dégénéré

Puisque la résolution symbolique d'un système de contraintes S consiste à construire un système triangulaire résolu T équivalent à S (voir Section 2.1), les cas dégénérés de S sont les cas dégénérés de T et inversement. Étant donné que, dans notre cas, T est un ensemble de définitions de la forme $x_i = g(u_1, \ldots, u_s, x_1, \ldots, x_{i-1})$, où g est une multifonction, un tel cas dégénéré se produit lorsque g ne donne pas un nombre maximum de résultats, ceci à cause de valeurs particulières des paramètres.

Exemple 16 *La multifonction interlc (voir Annexe A) construisant l'intersection entre une droite et un cercle produit généralement deux solutions, mais dans le cas dégénéré où le cercle est tangent à la droite, on n'obtient qu'une seule solution (Fig.4.7).*

Ceci nous amène d'une part à attribuer une numérotation pour les différents résultats produits par les multifonctions, mais aussi à définir une numérotation qui soit compatible avec la déformation continue des systèmes de contraintes géométriques, et que nous définissons comme une numérotation continue. Commençons par définir la notion de numérotation simple.

Définition 7 (numérotation) *Soit g une multifonction à n arguments avec un maximum de k résultats. Une* numérotation *de g est une fonction G à $n+1$ arguments telle que*

$$g(x_1, \ldots, x_n) = \{G(x_1, \ldots, x_n, 1), \ldots, G(x_1, \ldots, x_n, k)\}$$

où $G(x_1, \ldots, x_n, i)$ et $G(x_1, \ldots, x_n, j)$ sont des fonctions distinctes de $x_1, \ldots, x_n$ si $i \neq j$, appelées fonctions partielles *de g.*

Nous pouvons désormais étendre la définition de la numérotation des solutions des multifonctions, en définissant une numérotation continue, qui tient compte de la déformation continue des systèmes de contraintes géométriques. Cette définition s'applique naturellement à un plan de construction.

Définition 8 (numérotation continue) *Soient g une multifonction et G une numérotation telle que*

$$g(x_1, \ldots, x_n) = \Big\{G(x_1, \ldots, x_n, 1), \ldots, G(x_1, \ldots, x_n, k)\Big\}$$

alors G est une numérotation continue *si G est continue sur tout domaine ne contenant aucun cas dégénéré, i.e. où $Card\Big(g(x_1, \ldots, x_n)\Big) = k$ est toujours vrai.*

Dans ce qui suit, nous supposerons que toutes les multifonctions sont continûment numérotées. Cette numérotation continue est utilisée pour définir l'occurrence d'une solution, qui permet de distinguer précisément les solutions.

Définition 9 (occurrence d'une solution) *Soit T_u une instance d'un plan de construction qui définit les inconnues $x_1, x_2, \ldots, x_n$ grâce aux multifonctions $g_1, g_2, \ldots, g_n$ numérotées par $G_1, G_2, \ldots, G_n$, avec les maxima $k_1, k_2, \ldots, k_n$ respectivement. Une solution particulière de T_u est $\Big(G_1(u, i_1), G_2(u, i_2), \ldots, G_n(u, i_n)\Big)$, où $1 \leq i_1 \leq k_1, \ldots, 1 \leq i_n \leq k_n$. On dit que $(i_1, i_2, \ldots, i_n)$ est l'*occurrence *de cette solution.*

Il s'agit de la notion usuelle d'occurrence utilisée pour les noeuds d'arbres. Toutefois, cette notation étant assez lourde à manipuler, nous utilisons parfois la notion de numéro d'une solution définie ci-dessous, peut-être un peu plus ambiguë que celle d'occurrence, mais qui allège la notation.

Définition 10 (numéro d'une solution) *Soit T_u une instance d'un plan de construction. Si les p solutions de T_u ont pour occurrences respectives $(i_1^1, i_2^1, \ldots, i_n^1), \ldots, (i_1^p, i_2^p, \ldots, i_n^p)$ avec $(i_1^i, i_2^i, \ldots, i_n^i) \leq (i_1^j, i_2^j, \ldots, i_n^j)$ selon l'ordre lexicographique lorsque $i \leq j$, alors on dit que la solution d'occurrence $(i_1^i, i_2^i, \ldots, i_n^i)$ porte le* numéro i.

Exemple 17 *Poursuivons l'exemple du Chapitre 2 dont l'esquisse cotée était donnée à la Fig.2.1. A la Fig.4.8, nous rappelons son plan de construction et son arbre des solutions, en*

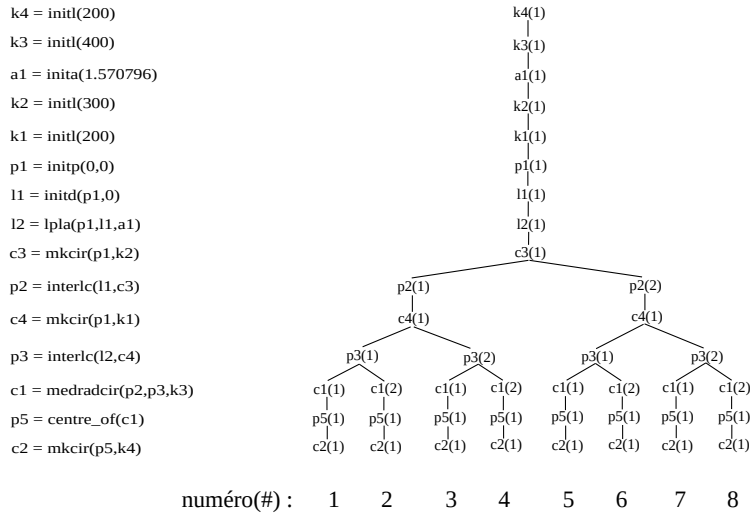


Fig. 4.8: Plan de construction correspondant à la Fig.2.1, et son arbre des solutions numéroté

précisant que les chiffres entre parenthèses déjà évoqués à la Section 2.3.2 correspondaient à cette définition de la numérotation des solutions des multifonctions. Cette numérotation détermine les occurrences des solutions, et donc leurs numéros, que avons ajoutés au schéma. Par exemple, l'occurrence de la solution #6 *est* $(1,1,1,1,1,1,1,1,1,2,1,1,2,1,1)$. *Nous pouvons donc également numéroter (de* #1 *à* #8*) les solutions de ce système sur la Fig.4.9.*

Maintenant que nous avons défini clairement notre vision de la ressemblance et de la numérotation, nous pouvons faire le lien entre ces notions en énonçant le théorème suivant.

Théorème 1 *Soit S un système de contraintes, et deux instanciations u et v des paramètres telles qu'il existe une déformation continue de S_u vers S_v ainsi qu'un système triangulaire résolvant S. Si T désigne ce système triangulaire, deux figures f_u et f_v, respectivement solutions de S_u et S_v, sont S-homotopes si et seulement si elles ont la même occurrence $(i_1, \ldots, i_n)$ dans T_u et T_v respectivement.*

La preuve du théorème précédent est donnée en Annexe D. Nous pouvons tout de même dire ici simplement que cette preuve est basée principalement sur des arguments de continuité. Ce théorème est vrai quelle que soit la résolution symbolique T de S, même si l'occurrence dépend de T. Plus précisément, nous avons le corollaire suivant.

Corollaire 1 *Sous les hypothèses du théorème 1, deux figures f_u et f_v, respectivement solutions de S_u et S_v, sont S-homotopes si et seulement si, pour tout système triangulaire T résolvant symboliquement S, elles ont la même occurrence dans T_u et T_v, respectivement.*

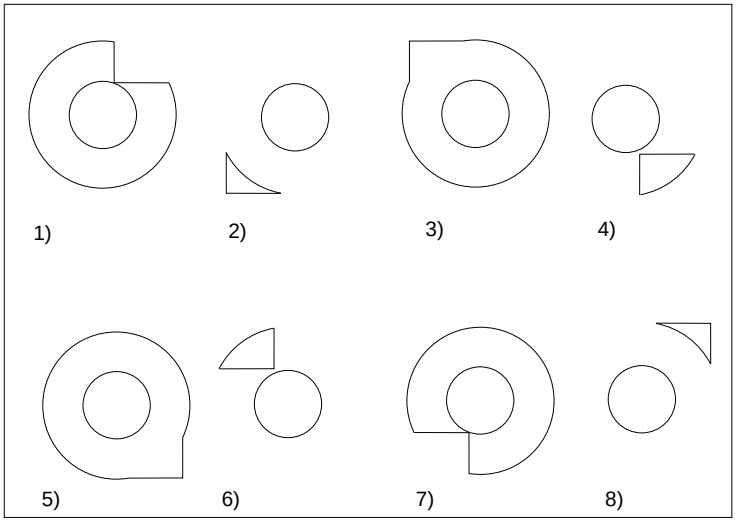


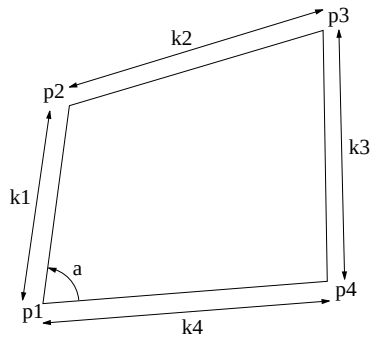
Fig. 4.9: Les solutions engendrées numérotées

De plus, f_u étant une solution de S_u, il n'existe qu'une seule solution f_v de S_v telle que f_u et f_v sont S-homotopes.

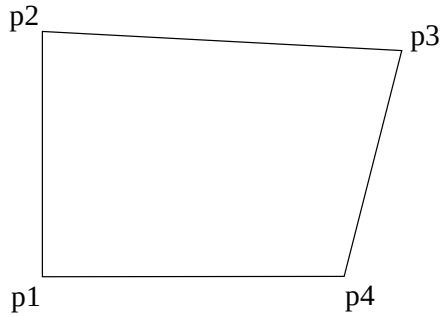
Exemple 18 *Afin d'illustrer le Théorème 1 et le Corollaire 1, la Fig.4.10 présente une esquisse d'un quadrilatère $p1p2p3p4$ dont les contraintes sont les longueurs des côtés $k1, k2, k3$, et $k4$ et l'angle orienté $a = \pi$, constituant un système S. On effectue deux interprétations numériques distinctes en prenant deux instanciations u et v différentes des paramètres, telles que les longueurs $k1$, $k2$, et $k3$ sont différentes, mais $k4$ et a restent inchangés. Les deux interprétations produisent chacune deux solutions, dûes à l'intersection de deux cercles auxiliaires de centres respectifs $p2$ et $p4$ et de rayons $k2$ et $k3$, et mènent respectivement aux Fig.4.10(b) (solution 1) et 4.10(d) (solution 2) pour la première et aux Fig.4.10(c) (solution 1) et 4.10(e) (solution 2) pour la deuxième. Les solutions 1 de S_u et 1 de S_v sont S-homotopes, de même que les solutions 2 de S_u et 2 de S_v. Par contre, les solutions numérotées 1 ne sont pas homotopes aux solutions numérotées 2, car la déformation continue qui permettrait de passer des unes aux autres passerait par le cas dégénéré tel que les deux cercles auxiliaires de construction seraient tangents.*

Le théorème 1 et son Corollaire 1 relient la notion d'homotopie à la numérotation des arbres des solutions. Ils donnent un critère de correspondance homotopique entre l'esquisse et une figure solution, cette dernière n'étant pas calculée en utilisant la méthode homotopique.

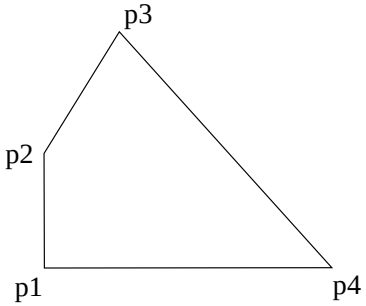
Il reste donc désormais à montrer qu'on peut numéroter correctement les multifonctions, puis comment on les numérote en pratique.



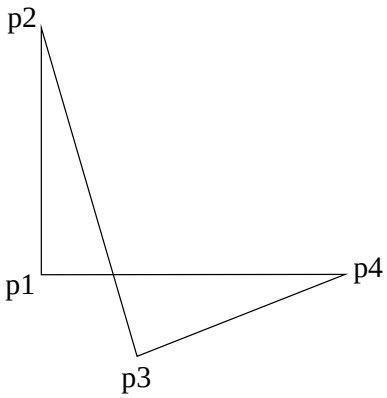
(a) esquisse



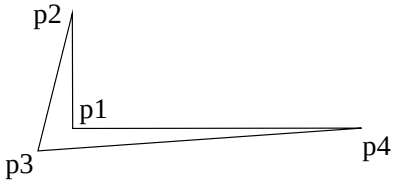
(b) solution 1 de S_u



(c) solution 1 de S_v



(d) solution 2 de S_u



(e) solution 2 de S_v

Fig. 4.10: Illustration du théorème 1 et du Corollaire 1

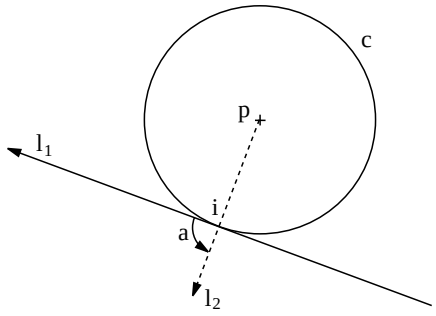
4.3 La numérotation en pratique

Afin de pouvoir utiliser cette théorie dans la pratique, nous avons déterminé pour toutes les multifonctions des propriétés géométriques nous permettant de distinguer leurs résultats et de les numéroter continûment.

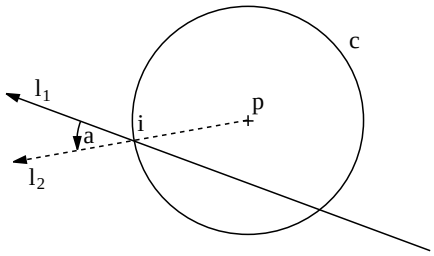
Exemple 19 *Poursuivant l'Exemple 16 avec la multifonction interlc, considérons l'angle a entre la droite donnée l_1 et la droite l_2 passant par le centre du cercle c et une intersection i, dans cet ordre (voir Fig.4.11 résumée à la Fig.4.12 en haut à gauche). Notons que les droites et angles sont orientés. Dans le cas dégénéré (c'est-à-dire si la droite l_1 est tangente à c), a est un angle droit. Lorsque interlc produit deux résultats, ils sont discriminés par l'acuité ou non de a. Ceci nous permet de donner une numérotation continue pour interlc : par exemple, la solution numéro 1 sera obtenue lorsque a est aigu, et la solution numéro 2 correspondra à un angle a obtus. Lorsqu'on calcule numériquement les solutions de cette intersection grâce aux équations faisant intervenir les coordonnées de la droite et du cercle, ces situations, et donc la discrimination des solutions, correspondent au signe positif ou négatif du discriminant Δ. Ce dernier discrimine donc continûment les solutions de interlc.*

Il en va de même pour chacune des multifonctions que nous utilisons actuellement dans notre solveur. Ainsi, pour chacune d'elles, nous décrivons une telle caractéristique géométrique, illustrée ensuite à la Fig.4.12 :

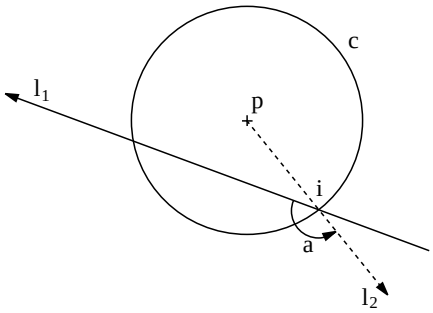
- *interlc* : construit l'intersection i entre une droite l_1 et un cercle c. Soit a l'angle entre l_1 et la droite l_2 qui passe par le centre p de c et par i. Il existe au plus deux solutions pour cette multifonction. Pour une solution a est aigu, pour l'autre a est obtus. Il y a une solution dégénérée lorsque c est tangent à l_1, et alors a est un angle droit.
- *intercc* : construit l'intersection i entre deux cercles c_1 et c_2. Les deux solutions sont différenciées par le placement relatif de trois points (dans le sens des aiguilles d'une montre ou le sens inverse) : les centres p_1 et p_2 des cercles, et i. La solution dégénérée se produit lorsque c_1 et c_2 sont tangents : p_1, p_2, et i sont alignés.
- *mkcir4* : construit un cercle c_2 tangent à un cercle donné c_1. Une solution est située à l'intérieur de c_1, l'autre est en-dehors de c_1. Le cas dégénéré est atteint lorsque c_1 a un rayon égal à zero.
- *medradcir* : construit un cercle c passant par deux points p_1 et p_2, connaissant son rayon k. Comme pour *intercc*, le placement relatif de trois points (dans le sens des aiguilles d'une montre ou le sens inverse) est différent pour les deux solutions. Dans ce cas, les trois points sont p_1, p_2, et le centre p de c. Lorsque $k = \frac{p_1p_2}{2}$, p devient le milieu de $[p_1p_2]$, les trois points sont colinéaires, et il n'y a qu'une seule solution possible.
- *bissectdd* : construit la bissectrice l_3 de deux droites l_1 et l_2. L'angle a entre l_1 et l_3 est soit aigu, soit obtus. Lorsque l_1 et l_3 sont parallèles, nous avons un cas dégénéré.
- *linev* : construit une droite verticale l à une distance donnée k d'un point p. Celui-ci est soit à gauche, soit à droite de l. Si $k = 0$, alors il n'y a qu'une seule possibilité pour l, et elle passe par p.



(a) l_1 tangente à c, angle a droit : cas dégénéré, une seule solution



(b) angle a aigu : solution 1



(c) angle a obtus : solution 2

FIG. 4.11: Caractéristique géométrique pour la multifonction *interlc*

- *lineh* : construit une droite horizontale l à une distance donnée k d'un point p. Celui-ci est soit au-dessus, soit en-dessous de l. Si $k = 0$, alors il n'y a qu'une seule possibilité pour l, et elle passe par p.
- *ldl* : construit une droite l_2 parallèle à une autre droite l_1, à une distance donnée k d'elle. L'angle a entre l_1 et $\overrightarrow{p_1 p_2}$, où $p_1 \in l_1$ et $p_2 \in l_2$, est négatif ou positif. Si $k = 0$, alors $l_1 = l_2$.

Dans chacun des cas, le passage d'une solution à l'autre consiste en une déformation continue passant par une figure intermédiaire, constituant un cas dégénéré, dans laquelle la valeur associée à la propriété caractéristique atteint un extremum. Toutes les propriétés géométriques définies ci-dessus constituent une numérotation pour les multifonctions. Or, elles sont conservées par la déformation continue, tant qu'on n'atteint pas l'extremum. On en déduit donc que la numérotation qu'elles constituent est continue, d'après la Définition 8.

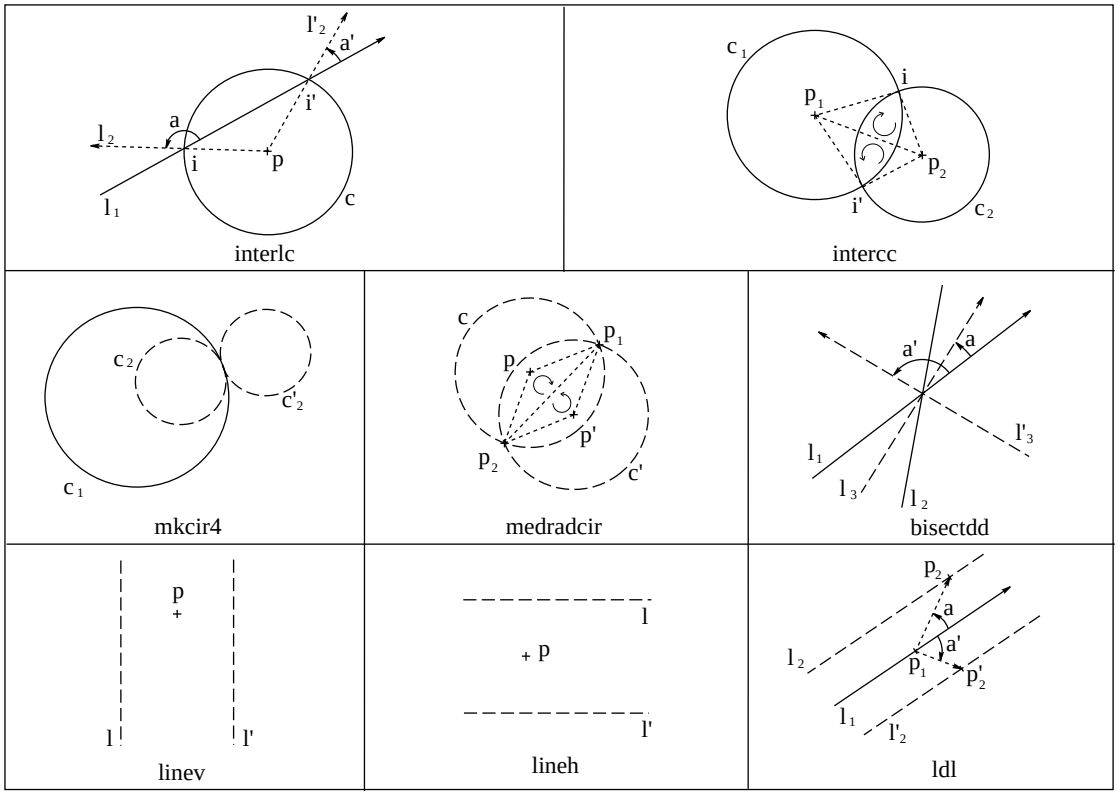


Fig. 4.12: Caractéristiques géométriques des multifonctions

Il est à noter que les critères géométriques définis ci-dessus ne servent que pour obtenir une fois pour toutes une discrimination entre les différentes solutions des multifonctions utilisées dans notre solveur, et donc une numérotation. Une fois cette numérotation obtenue, nous ne nous servons plus de ces critères pour comparer directement l'esquisse avec une figure solution. Seule la numérotation est utilisée par la suite, et nous allons montrer comment dans la section suivante.

4.4 Gel d'une branche

Afin de trouver la solution que l'utilisateur attend, nous mettons en avant l'hypothèse que les contraintes qu'il a données avec l'esquisse reflètent son attente. Plus précisément, en utilisant la notation du Théorème 1, on suppose premièrement qu'il existe une instance u des paramètres telle que l'esquisse f_u est une solution de S_u, et deuxièmement qu'il existe une déformation continue de S_u vers S_v où v est le n-uplet des cotes initialement données dans les contraintes. Dans cette section, nous expliquons comment remplir la première condition dans le cas de contraintes métriques. La deuxième condition est forte, mais semble être habituellement remplie. Pour le moment, nous ne considérerons pas les

cas particuliers. Ils seront abordés dans la Section 4.4.2. Tout ceci nous amène à un moyen simple de trouver la solution demandée, en appliquant le Théorème 1 de la façon suivante.

4.4.1 Technique de gel d'une branche

Nous notons tout d'abord que, comme nous l'avons dit à la Remarque 1 dans la section 2.1, l'esquisse peut être vue comme une solution particulière du système de contraintes S_u instancié par le n-uplet u des valeurs lues sur l'esquisse. Puis, nous supposons qu'il y a une déformation continue entre S_u et S_v, si S_v est l'instanciation du système par le n-uplet v des cotes données. Alors, si l'on peut trouver une solution de S_v ayant la même occurrence que l'esquisse, on peut dire que c'est la solution recherchée. Même si cela peut paraître étrange, nous essayons de retrouver l'esquisse grâce au plan de construction.

Exemple 20 *Poursuivons l'Exemple 1 de la Section 2.1, dont le plan de construction était représenté avec l'arbre des solutions et la numérotation des solutions à la Fig.4.8. Nous représentons à la Fig.4.13 à gauche l'arbre des solutions correspondant au système de contraintes instancié par le n-uplet des valeurs lues sur l'esquisse (S_u), et à droite l'arbre des solutions correspondant au même système de contraintes mais instancié cette fois par le n-uplet des cotes données (S_v). Il existe une déformation continue entre S_u et S_v, donc chaque solution de l'arbre de droite peut être retrouvée par déformation continue de la solution de l'arbre de gauche qui porte le même numéro.*

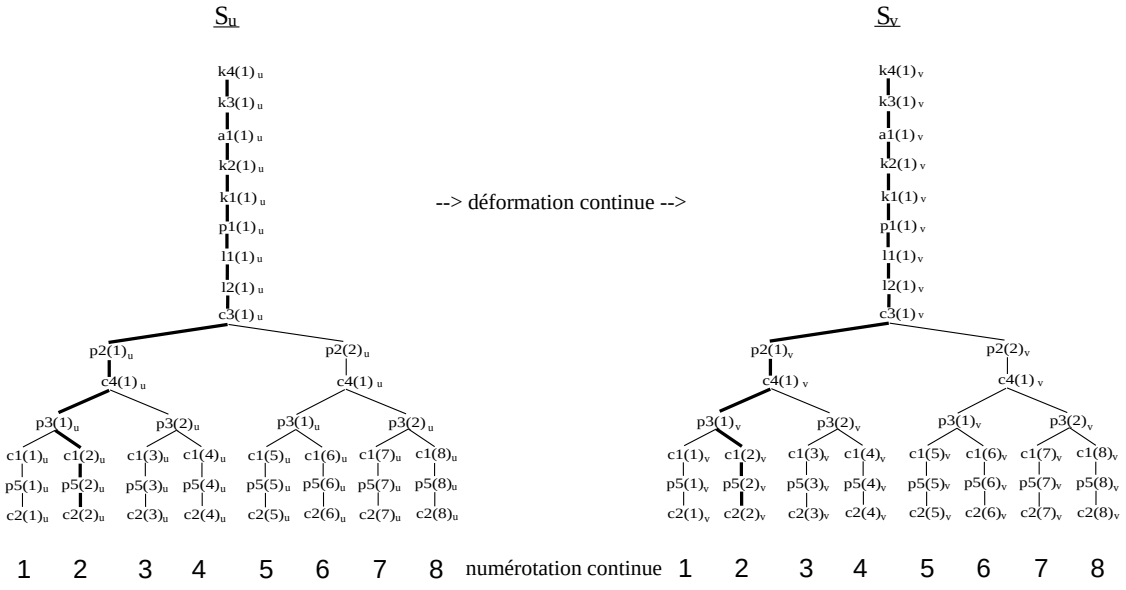


Fig. 4.13: Gel d'une branche : à gauche la branche correspondant à l'esquisse, à droite la solution gelée

En pratique, nous tirons parti de notre approche formelle de résolution de la façon suivante. Dans un premier temps, nous appliquons l'interprétation $\mathcal{I}_u$ à la forme triangulaire résolue T_u du système de contraintes S_u avec des paramètres pris sur l'esquisse. A chaque embranchement (multifonction) de l'arbre des solutions produit par $\mathcal{I}_u$, nous pouvons décider quelle branche suivre en comparant, comme expliqué à la Section 4.3, les propriétés géométriques des résultats de la multifonction avec les valeurs effectivement lues sur l'esquisse. Le but est que la version calculée, c'est-à-dire la branche choisie, doit être identique (à ε près dû aux approximations de calculs sur des réels) à l'esquisse effective. De cette façon, on peut stocker l'occurrence de la branche de l'arbre des solutions correspondant à l'esquisse. Nous appelons cette opération le *gel d'une branche*.

Exemple 21 *Toujours sur la Fig.4.13, nous avons en trait gras sur l'arbre de gauche la branche que nous avons repérée comme étant celle correspondant à l'esquisse. Grâce à notre théorie, nous pouvons en déduire que la branche en trait gras de l'arbre de droite, qui porte le même numéro, est celle que l'on obtient en la déformant continûment. C'est donc celle-ci qui est gelée.*

Dans un deuxième temps, pour des paramètres fixés v, la figure est trouvée grâce à l'interprétation $\mathcal{I}_v$ de la forme triangulaire résolue T_v du système de contraintes S_v, en suivant simplement la branche qui a été gelée durant la première phase et dont l'occurrence avait été stockée. Il est alors possible de recommencer cette interprétation avec autant de paramètres qu'on le souhaite sans avoir à refaire le gel de la branche.

Exemple 22 *Poursuivant cette fois l'Exemple 5 de la Section 2.3, la Fig.4.14 présente l'unique solution trouvée en utilisant notre méthode de gel d'une branche sur l'esquisse cotée donnée à la Fig.2.5(a).*

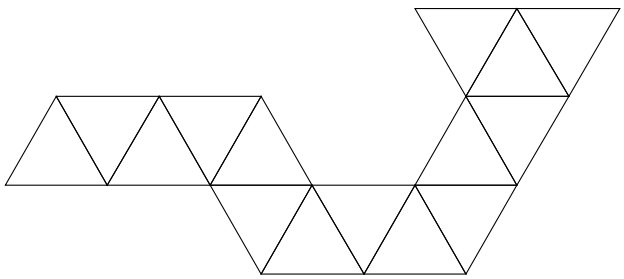


Fig. 4.14: La solution unique correspondant à l'esquisse présentée à la Fig.2.5(a)

Un des avantages de cette méthode est sa vitesse, car une comparaison n'est effectuée que si elle est nécessaire. En fait, au lieu de comparer géométriquement tous les objets de la figure les uns avec les autres, nous comparons uniquement, à chaque embranchement de l'arbre, les objets mis en jeu dans la multifonction concernée. Puisque le traitement est effectué au fur et à mesure de l'interprétation, le temps de calcul est réduit de manière significative comparativement à une méthode systématique.

Exemple 23 *Pour obtenir la solution unique présentée Fig.4.14, notre méthode donne une réponse instantanée, tandis que le calcul de toutes les solutions possibles prend plus d'une minute. Ces tests ont été effectués sur un PC avec un processeur à 266 MHz.*

Dans la pratique, on n'effectue pas le gel d'une branche sur tout l'arbre pour ensuite effectuer une deuxième fois l'interprétation en suivant l'occurrence. Le gel d'une branche est plutôt effectué *au fur et à mesure* de l'interprétation numérique. Lorsqu'on arrive à l'interprétation d'une définition d du plan, on commence par l'interpréter avec les données de l'esquisse, puis on compare le résultat avec le résultat théorique en notant son numéro, et enfin on interprète avec les données de la cotation en sélectionnant le résultat dont on a mémorisé le numéro. On peut alors ensuite passer à une autre définition. L'Algorithme 3 précise ce processus.

Algorithme 3 : Procédure Interprétation_avec_gel(d)

Préconditions: soit d une définition
Postconditions: interprète numériquement d, en utilisant le gel, pour générer la solution unique sol

```
val ← Interprete_avec_valeurs_esquisse(d)
si d.valeur_d_esquisse=vide alors
   d.valeur_d_esquisse ← val
sinon
   i ← Compare(d.valeur_d_esquisse,val)
fin si
⇝ i est alors le numéro du résultat gelé, que l'on passe ensuite en paramètre pour la seconde interprétation
sol ← Interprete_avec_valeurs_esquisse(d,i)
d.valeur_interpretation_numerique ← val
si d.suivant=vide alors
   fin
sinon
   Interprétation_avec_gel(d.suivant)
fin si
```

Finalement, comme nous l'avons dit précédemment, les critères géométriques sur les multifonctions dépendent de l'orientation des droites. Par exemple, avec *interlc*, si l'on change l'orientation de la droite, l'acuité de l'angle est inversée, et les solutions permutent. Ainsi, afin d'assurer la conservation de la numérotation, nous devons vérifier que toutes les droites de l'esquisse sont données avec la même orientation que celles qui seront calculées. Nous devons donc comparer l'angle entre chaque droite et l'axe Ox sur l'esquisse avec ce qu'il devrait être, puisque l'esquisse est considérée comme le résultat d'une interprétation.

Exemple 24 *Considérons une droite l_1 décrite dans le plan de construction par une définition contenant le terme fonctionnel $lpla(p, l_2, a)$. La fonction lpla permet de dessiner*

la droite passant par un point p, faisant un angle orienté a avec une autre droite l_2. Nous calculons, en utilisant les données de l'esquisse, ce que devrait être l'orientation de l_1, connaissant l'orientation de l_2 calculée auparavant. Puis, nous comparons cette orientation théorique avec l'orientation effective. Si elles sont opposées, nous modifions les données de l'esquisse en changeant le sens de l_1, comme sur la Fig.4.15. D'un point de vue formel, la construction reste inchangée. Seule la représentation numérique est corrigée.

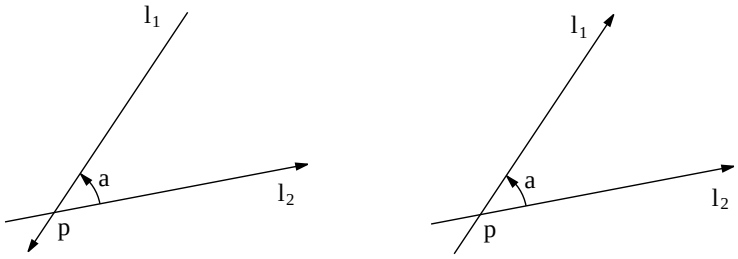


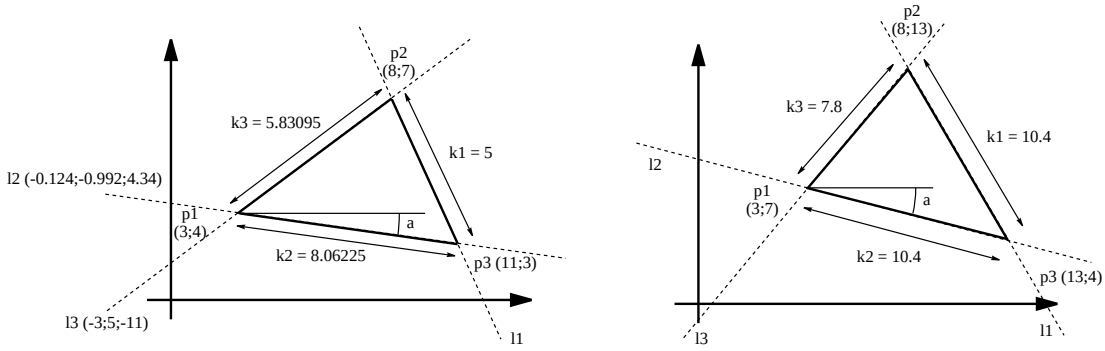
FIG. 4.15: Correction de l'orientation de la droite l_1

Cette méthode a elle aussi été prototypée en *OBJ3*, afin de la valider par la spécification algébrique ainsi que par des séries de tests, avant d'effectuer son implantation dans *YAMS*. C'est ainsi que nous avons défini les opérations suivantes qui permettent d'effectuer le gel d'une branche. Notons que dans l'extrait de spécification du gel d'une branche qui va suivre, ainsi que dans la spécification plus complète qui est présentée en Annexe B, les notations diffèrent légèrement par rapport à tout ce qui a été présenté au Chapitre 3. Ceci est dû à une petite évolution qui a eu lieu entre cette spécification, qui est tirée d'une version antérieure, et celles du Chapitre 3, qui sont tirées d'une version plus récente.

```
*** interpretation avec donnees de l'esquisse
op interp-sketch : Fun Interp -> Nlist .      *** interpretation d'une
                                                *** fonction d'interpretation
op interp-sketch : Def-list Interp -> Nlist . *** interpretation d'une liste
                                                *** de definitions
op interp-sketch : Def Interp -> Int .        *** interpretation d'une
                                                *** definition
*** interpretation avec donnees de la cotation
op interp-values : Fun FFFlist Nlist -> Interp .     *** interpretation
                                                       *** d'une fonction
                                                       *** d'interpretation
op interp-values : Def-list Interp Nlist -> Interp . *** interpretation d'une
                                                       *** liste de definitions
op interp-values : Term Interp Int -> Flist .        *** interpretation d'une
                                                       *** definition

*** operations auxiliaires
op occurrence : Flist FFlist Int -> Int .
op find : I Interp -> Flist .
op near : Flist Flist -> Bool .
op get : FFlist Int -> Flist .
```

Dans cette spécification, l'opération `interp-sketch` permet d'obtenir l'occurrence de la branche gelée, sous la forme d'une liste d'entiers de sorte Nlist. La sorte Interp est une liste d'affectations de réels aux identificateurs des éléments géométriques, affectations symbolisées par `=>`. Une fois l'occurrence de la branche gelée obtenue, on peut alors lancer `interp-values` avec comme paramètres supplémentaires les données de la cotation sous la forme d'une liste de réels (`FFFlist`).

Nous avons donc pu effectuer des séries de tests sur ces spécifications, dont un exemple est donné ci-dessous.



(a) esquisse tracée

(b) solution souhaitée

Fig. 4.16: Triangle utilisé pour tester la spécification

Exemple 25 *Sur la Fig.4.16(a), une esquisse cotée d'un triangle p1p2p3 a été tracée. Ce triangle possède comme contraintes les 3 longueurs de ses côtés k1, k2 et k3, ainsi que l'angle a que forme la doite l2, passant par p1 et p3, avec l'axe Ox. La Table 4.1 représente l'équation nommée b que nous avons écrite afin de pourvoir la réduire grâce à OBJ3. Elle effectue l'interprétation numérique* `interp-values` *avec les données de la cotation, en prenant comme paramètre l'occurrence de la branche gelée trouvée grâce à la réduction de l'interprétation* `interp-sketch` *avec les valeurs de l'esquisse. Dans cette équation, les données de la cotation sont à gauche, entourés par les caractères "&". Les valeurs de l'esquisse sont représentées par les affectations de coordonnées situées à droite. L'exécution de cette réduction de b est montrée à la Table 4.2. La figure solution souhaitée est représentée à la Fig.4.16(b), sur laquelle nous pouvons vérifier l'exactitude du résultat trouvé par la réduction. Sur la Table 4.2, nous avons également ajouté la réduction de la partie* `interp-sketch` *seule, afin d'exhiber le résultat intermédiaire : l'occurrence de la branche, affichée en bas de la table.*

TAB. 4.1: Equation test à réduire

```
eq b = interp-values(f,3 & 7 & -0.27 & 7.8 & 10.4 & 10.4 & interp-sketch(f, (l3 => (-3 ;5 ;-11)) "
                                                                            (p2 => (8 ;7)) "
                                                                            (c3 => (11 ;3 ;5)) "
                                                                            (p3 => (11 ;3)) "
                                                                            (l2 => (-0.124 ;-0.992 ;4.34)) "
                                                                            (c2 => (3 ;4 ;8.06225)) "
                                                                            (c1 => 3 ;4 ;5.83095)) "
                                                                            (p1 => (3 ;4)) "
                                                                            (k2 => 8.06225) "
                                                                            (k1 => 5) "
                                                                            (k3 => 5.83095) "
                                                                            (a => -0.12435) "
                                                                            (y1 => 4) "
                                                                            (x1 => 3)).
```

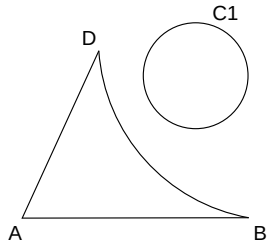
TAB. 4.2: Réduction de l'équation test

```
OBJ> red b.
reduce in TEST : b
rewrites : 18828
result Interp : ((line # 'l3 . 1) => (6.1886389046323291 ;-4.7477098171719767 ;14.668052006306844)) "
                ((point # 'p2 . 1) => (7.7477098171719767 ;13.188638904632329)) "
                ((circle # 'c3 . 1) => (13.023217322205262 ;4.2259930584361562 ;10.4)) "
                ((point # 'p3 . 1) => (13.023217322205262 ;4.2259930584361562) "
                ((line # 'l2 . 1) => (-0.26673143668883115 ;-0.96377089636589053 ;7.5465905846277277)) "
                ((circle # 'c2 . 1) => (3.0 ;7.0 ;10.4)) "
                ((circle # 'c1 . 1) => (3.0 ;7.0 ;7.7999999999999998)) "
                ((point # 'p1 . 1) => (3.0 ;7.0)) "
                ((num # 'x1 . 1) => 3.0) "
                ((num # 'y1 . 1) => 7.0) "
                ((num # 'a . 1) => -0.27000000000000002) "
                ((num # 'k1 . 1) => 10.4) "
                ((num # 'k2 . 1) => 10.4) "
                ((num # 'k3 . 1) => 7.7999999999999998)

OBJ> red interp-sketch(f, (l3 => (-3 ;5 ;-11)) "
                           (p2 => (8 ;7)) "
                           (c3 => (11 ;3 ;5)) "
                           (p3 => (11 ;3)) "
                           (l2 => (-0.124 ;-0.992 ;4.34)) "
                           (c2 => (3 ;4 ;8.06225)) "
                           (c1 => 3 ;4 ;5.83095)) "
                           (p1 => (3 ;4)) "
                           (k2 => 8.06225) "
                           (k1 => 5) "
                           (k3 => 5.83095) "
                           (a => -0.12435) "
                           (y1 => 4) "
                           (x1 => 3)).

reduce in TEST : interp-sketch(f,(l3 => (-3 ;5 ;-11)) " (p2 => (8 ;7)) " (c3 => (11 ;3 ;5)) "
        (p3 => (11 ;3)) " (l2 => (-0.124 ;-0.992 ;4.34)) " (c2 => (3 ;4 ;8.06225)) " (c1 => 3 ;4 ;5.83095)) "
        (p1 => (3 ;4)) " (k2 => 8.06225) " (k1 => 5) " (k1 => 5) " (k3 => 5.83095) " (a => -0.12435) "
        (y1 => 4) " (x1 => 3))
rewrites : 9691
result Nlist : 1 " 1 " 1 " 1 " 1 " 1 " 1 " 1 " 1
OBJ>
```

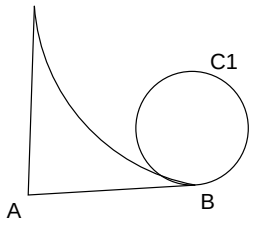
4.4.2 Limites de la méthode

La méthode exposée dans la section précédente donne d'excellents résultats lorsque toutes les contraintes sont des contraintes métriques. Par contre, l'autre type de contraintes utilisé (à moindre échelle cependant) dans *YAMS*, les contraintes booléennes, pose problème. En effet, dans le cas de contraintes booléennes, il nous manque des informations pour trouver la solution demandée. Nous avons vu à la Remarque 1 donnée à la Section 2.1 que, si dans le cas où les contraintes sont toutes métriques, l'esquisse était elle-même solution de S_u où u est le n-uplet des valeurs lues sur l'esquisse, dans le cas contraire où ne serait-ce qu'une contrainte booléenne est présente dans le plan, l'esquisse n'est plus solution de S_u.

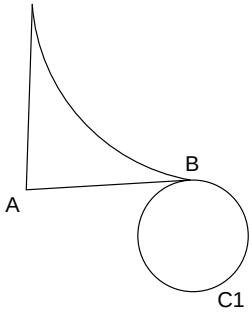
Exemple 26 *Comme le montre la Fig.4.17, les contraintes booléennes ne sont pas toujours respectées sur l'esquisse. Sur l'esquisse de la Fig.4.17(a), une contrainte demande que le cercle $C1$ soit tangent en B au segment $[AB]$, or ce n'est pas le cas sur le dessin tracé par l'utilisateur. Il est donc impossible de faire une comparaison entre cette esquisse et les solutions présentées sur la Fig.4.17(b) et (c).*



(a) esquisse



(b) solution 1



(c) solution 2

Fig. 4.17: Problème des contraintes booléennes : exemple avec la tangence

Remarquons que ce problème peut être ramené à celui des cas dégénérés. En effet, la tangence peut être vue comme une distance nulle entre un cercle et une droite, ce qui est un cas limite. Un exemple de cette situation est donné ci-dessous.

Exemple 27 *Afin d'illustrer à la fois l'efficacité notre méthode sur des contraintes métriques, mais également ses limites sur des contraintes booléennes, nous montrons ici un exemple assez représentatif. L'esquisse de la Fig.4.18, représentant un levier, possède* 106 *contraintes dont* 2 *contraintes de tangence,* $tgcl(c1, l1)$ *et* $tgcl(c3, l2)$*. Ces contraintes sont présentées à la Table 4.3. Notons qu'afin d'alléger la figure, nous avons évité de représenter les contraintes par des flèches comme nous l'avions fait dans les exemples précédents. En utilisant ces contraintes, le solveur produit un plan de construction comprenant* 252 *définitions. Ce plan étant donc très volumineux, nous ne le présentons qu'en Annexe C. Etant donné que, parmi ces définitions,* 29 *ont une arité de* 2*, l'arbre des possibilités de ce système de contraintes possède* $2^{29} = 536870912$ *branches. En réalité, un grand nombre de branches conduisent à un échec lors de l'interprétation ou sont éliminées grâce à une simple vérification de contraintes. Mais l'arbre des solutions possède tout de même* 160 *branches, correspondant à* 160 *solutions à examiner. Notre méthode permet de réduire de façon très significative l'arbre des solutions, en laissant un sous-arbre gelé de* 4 *branches seulement. Les solutions restantes sont présentées Fig.4.19, tandis que quelques solutions parmi celles qui ont été éliminées sont présentées à la Fig.4.20. Parmi les quatre figures, la partie inchangée correspond aux contraintes métriques, et la partie "incertaine" aux contraintes de tangence. Il n'est pas possible de n'obtenir qu'une solution car les contraintes de tangence ne sont pas respectées sur l'esquisse. Afin de réduire ce petit sous-arbre restant à une branche unique, nous pouvons utiliser dans ce cas précis des heuristiques de comparaison de placements relatifs de cercles et de droites, qui nous conduisent à l'obtention de la figure située en bas à gauche de la Fig.4.19.*

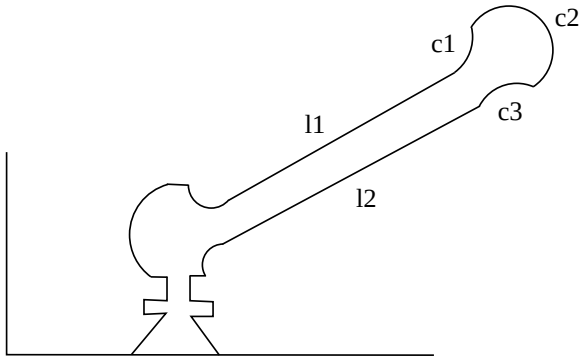


Fig. 4.18: Esquisse d'un levier

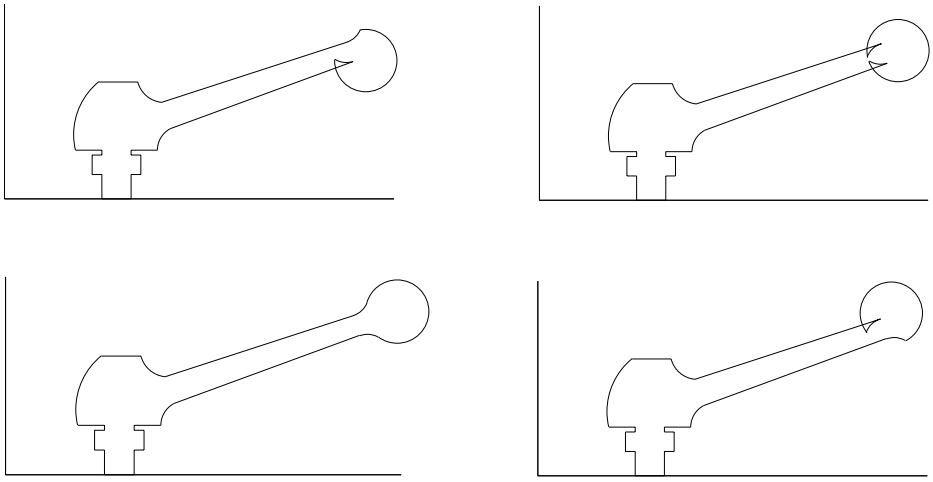


Fig. 4.19: Solutions pour le levier

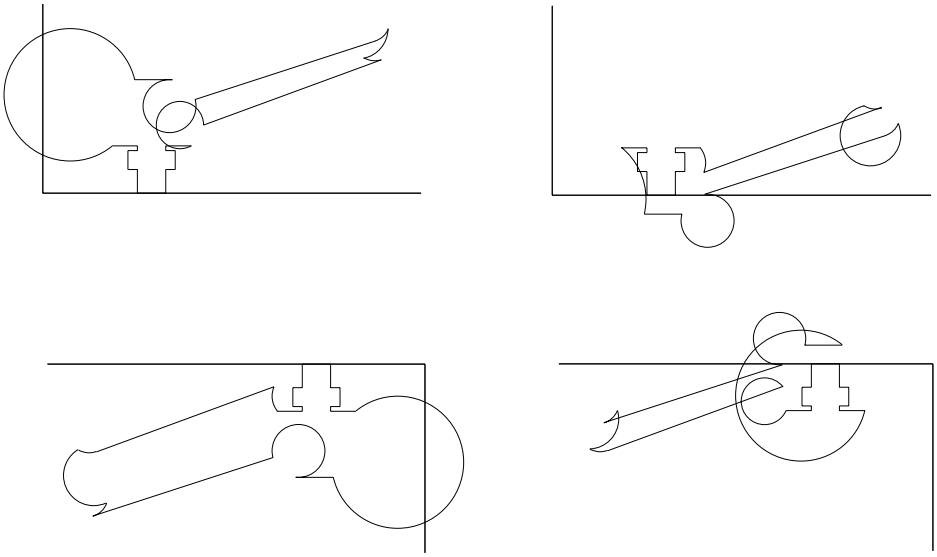


Fig. 4.20: Quelques solutions éliminées pour le levier

TAB. 4.3: Contraintes du levier

angarc(p28, p3, p25, a18)	angarc(p26, p24, p2, a17)	centre(c6, p31)
centre(c4, p29)	centre(c3, p28)	centre(c2, p27)
radius(c5, k31)	radius(c6, k30)	radius(c4, k29)
centre(c5, p30)	centre(c1, p26)	radius(c2, k28)
radius(c3, k27)	radius(c1, k26)	angle(p13, p16, p13, p11, a16)
angle(p17, p16, p17, p18, a14)	angle(p15, p16, p15, p12, a13)	angle(p18, p17, p18, p19, a12)
angle(p10, p12, p10, p9, a10)	angle(p19, p18, p19, p20, a9)	angle(p20, p21, p20, p19, a8)
angle(p21, p20, p21, p22, a6)	angle(p8, p9, p8, p7, a5)	angle(p22, p21, p22, p23, a4)
angle(p16, p17, p16, p15, a15)	angle(p12, p15, p12, p10, a11)	angle(p9, p10, p9, p8, a7)
angle(p17, p14, p4, p3, a2)	angle(p17, p14, p1, p2, a1)	angle(p5, p6, p22, p23,a3)
distpp(p11, p13, k25)	distpp(p13, p14, k24)	distpp(p15, p16, k20)
distpp(p13, p16, k23)	distpp(p16, p17, k22)	distpp(p18, p17, k21)
distpp(p12, p15, k19)	distpp(p19, p18, k18)	distpp(p19, p20, k17)
distpp(p12, p10, k16)	distpp(p9, p8, k12)	distpp(p5, p8, k8)
distpp(p21, p20, k15)	distpp(p9, p10, k14)	distpp(p21, p22, k13)
distpp(p23, p22, k11)	distpp(p7, p8, k10)	distpp(p5, p6, k9)
distpp(p7, p5, k7)	distpp(p1, p22, k6)	distpp(p1, p23, k5)
distpp(p14, p4, k3)	distpp(p4, p3, k2)	distpp(p1, p2, k1)
distpp(p23, p4, k4)	fixorgpl(p13, l10, p16)	onc(p7, c5)
onc(p1, c6)	onc(p6, c6)	onc(p5, c5)
onc(p4, c4)	onc(p23, c4)	onc(p3, c3)
onc(p25, c2)	onc(p24, c2)	onc(p24, c1)
onc(p25, c3)	onc(p2, c1)	onl(p21, l16)
onl(p23, l17)	onl(p22, l17)	onl(p22, l16)
onl(p21, l15)	onl(p20, l15)	onl(p20, l14)
onl(p19, l13)	onl(p18, l13)	onl(p18, l12)
onl(p17, l10)	onl(p16, l10)	onl(p16, l11)
onl(p15, l9)	onl(p14, l10)	onl(p13, l10)
onl(p12, l9)	onl(p12, l7)	onl(p11, l8)
onl(p10, l6)	onl(p9, l6)	onl(p9, l5)
onl(p8, l4)	onl(p7, l4)	onl(p6, l3)
onl(p4, l2)	onl(p3, l2)	onl(p2, l1)
onl(p19, l14)	onl(p17, l12)	onl(p15, l11)
onl(p13, l8)	onl(p10, l7)	onl(p8, l5)
onl(p5, l3)	onl(p1, l1)	tgcl(c1, l1)
tgcl(c3, l2)		

Dans ces situations, deux approches générales peuvent être considérées : celles qui corrigent l'esquisse afin de remplir la condition, et celles produisant plusieurs branches, donc un sous-arbre, qui reste à explorer.

La première approche n'a pas encore été étudiée, mais pourrait consister à interpréter le plan de construction avec les données lues sur l'esquisse tout en corrigeant au fur et à mesure dans ce plan les objets inconsistants. Par exemple, sur la Fig.4.17, l'esquisse pourrait être modifiée de telle façon que le cercle devienne tangent à la droite. Parmi les deux cercles possibles donnés par le plan de construction, nous choisirions le plus proche du cercle original, au sens de la distance euclidienne sur les coordonnées.

La deuxième approche consiste à faire un gel maximum. Toutes les solutions pour lesquelles il n'est pas possible de choisir sont conservées. Le résultat est un *sous-arbre gelé*,

qui peut être exploré grâce à des outils fournis par le logiciel. Ces outils peuvent être de différentes sortes, dont voici quelques exemples :

- des outils automatiques pour élaguer et/ou classifier les branches du sous-arbre restant. Les heuristiques classiques peuvent être utilisées, telles que des comparaisons de certaines propriétés de l'esquisse et des solutions, mais d'une façon plus spécifique à notre cas. Nous pouvons tirer parti du plan de construction pour trouver quels objets sont liés entre eux, et ainsi comparer leurs positions relatives. De cette façon, nous pensons que ces critères sont plus pertinents. Dans l'Exemple 26, les deux possibilités pour le cercle $C1$ peuvent être triées suivant leur position (droite ou gauche) du cercle par rapport à la droite orientée.
- des outils algorithmiques pour rendre plus rapide l'exploration partielle ou totale de l'espace restant des solutions. Parmi ces outils, qui sont bien connus dans d'autres domaines, nous avons expérimenté une table de hachage dans le même esprit que dans certains langages fonctionnels tels que *OBJ3*, un backtracking intelligent, ainsi que des permutations dans le plan de construction (par tri topologique) pour réduire la complexité. Ces dernières heuristiques sont celles expliquées à la Section 3.3 : choix par degré de multiplicité, degré de risque, heuristiques venant du problème SAT.
- une interface conviviale : nous pensons que l'utilisateur peut avoir une idée incertaine de ce qu'il désire obtenir, et que ses souhaits – exprimés grâce à l'esquisse et aux contraintes – peuvent parfois être contradictoires. C'est pourquoi nous proposons une interface utilisateur pour explorer l'espace restant des solutions grâce à la structure du plan de construction. Cette démarche est générale : elle s'applique aussi si l'utilisateur veut explorer tout l'espace des solutions. Nous l'avons donc étudiée plus en détail. Nous exposons au chapitre suivant quelques outils que nous avons développés en vue d'une interface d'exploration conviviale.

Chapitre 5

Interface homme-machine

Comme nous l'avons vu au chapitre précédent, à cause des contraintes booléennes, certains systèmes de contraintes donnent lieu à un arbre des solutions qui ne peut pas être réduit à une branche unique. Il peut être réduit très substantiellement, mais il reste un petit sous-arbre à explorer. Il se peut également que l'utilisateur ne soit pas totalement satisfait de la solution trouvée par le solveur pour les contraintes posées. Pour toutes ces raisons, l'utilisateur peut vouloir explorer soit le sous-arbre de solutions, soit le reste de l'espace des solutions, et examiner les solutions qui sont proches de la branche gelée.

C'est pourquoi nous avons été conduits à proposer quelques fonctionnalités pour explorer l'espace des solutions dans *YAMS*. Rappelons que cet espace de solutions n'est pas un simple ensemble de figures, mais un espace structuré. Les structures d'arbre des solutions et de plan de construction que nous utilisons nous offrent la possibilité de définir des outils d'exploration, inspirés par les outils de débogage fournis par la plupart des environnements de développement en génie logiciel. Ils sont un peu apparentés également à la méthode "incrémentale" ébauchée par C. Hoffmann *et al.* dans [BFH+95], et dont nous avons parlé dans la Section 1.3.

5.1 Une interprétation pas à pas

Tout d'abord, rappelons que dans le cas où l'utilisateur souhaite explorer l'espace entier des solutions, le nombre de solutions (c'est-à-dire de branches dans l'arbre) peut être très important. Visualiser les solutions une par une peut donc se révéler très fastidieux. Il serait donc utile de trouver par exemple un moyen de nous déplacer dans l'arbre en direction des feuilles, en nous dirigeant le plus rapidement possible vers la bonne solution.

Pour cela, un premier type d'outil est inspiré de l'idée d'exécution pas à pas d'un programme, qui est très souvent proposée par les débogueurs, en particulier par le langage *Prolog* où la notion de backtracking est aussi prise en compte [SS90]. Supposons que la

figure n'est pas encore interprétée numériquement. Une bonne manière de parcourir les solutions efficacement peut être de choisir, à chaque embranchement de l'arbre, quelle branche suivre. Voici comment nous procédons.

5.1.1 Approche

Dans le plan de construction, il existe deux sortes de définitions. Certaines correspondent à des objets qui peuvent être visualisés par l'utilisateur. Dans le reste de ce mémoire, elles seront appelées *définitions d'esquisse*. D'autres correspondent à des objets auxiliaires utilisés pour les besoins de la construction. Par exemple, des cercles qui sont utilisés simplement pour le calcul d'un point de la figure, par intersection avec une droite existante. Celles-ci seront appelées *définitions auxiliaires*. Puisque les entités géométriques définies par des définitions auxiliaires ne sont pas dessinées à l'écran, il est difficile de choisir quelles valeurs on doit conserver pour elles. De plus, l'utilisateur n'est pas intéressé par la construction des objets intermédiaires, qui doit être totalement transparente pour lui. Alors, l'idée est de ne faire une pause dans l'interprétation que lorsqu'on atteint une définition d'esquisse impliquant un choix.

A chaque pas, on travaille sur une *couche* (*cf.* Fig.5.1). Une couche peut être définie comme un petit sous-arbre de l'arbre des solutions dans lequel la dernière définition (c'est-à-dire la plus basse dans le sous-arbre) est une définition d'esquisse, et toutes les autres définitions sont des définitions auxiliaires. Les différentes valeurs possibles pour l'objet visible concerné par ce pas doivent être proposées, et l'utilisateur doit pouvoir choisir l'une d'entre elles. Cela signifie que pour cette opération, ce petit sous-arbre doit être exploré.

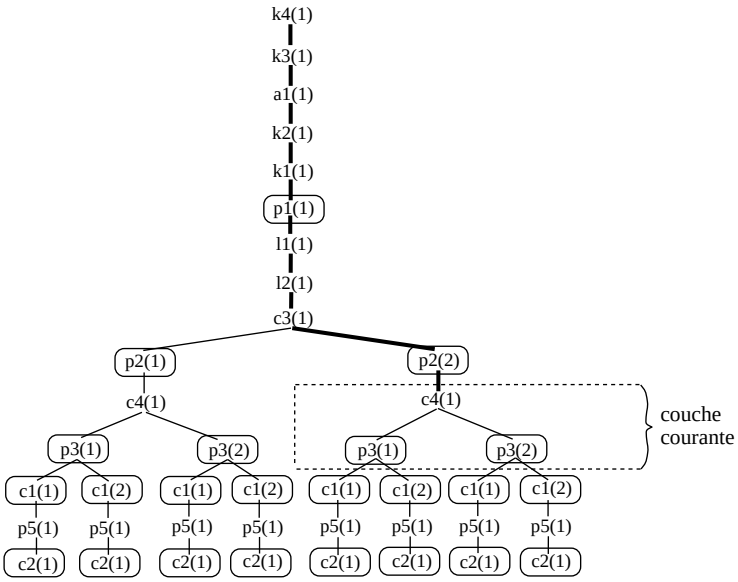


Fig. 5.1: Backtracking sur un petit sous-arbre, inclus dans une couche de l'arbre des solutions

5.1.2 Pré-traitement

Ce sous-arbre contient quelques embranchements, qui correspondent aux définitions auxiliaires de la couche auxquelles une multifonction de *degré* ≥ 2 est associée. Un backtracking doit alors être effectué à l'intérieur de la couche. Puis, lorsqu'un des résultats sera choisi pour la définition courante, la branche correspondante sera gelée dans cette couche. Sur l'exemple de la Fig.5.1, la branche gelée jusqu'ici est en gras, la couche courante est entre pointillés, et les objets visibles sont encadrés.

Le plan de construction a cependant pu être généré par le solveur dans une forme inadéquate pour le bon déroulement de cette opération. Il peut alors être nécessaire d'effectuer un tri topologique du plan avant l'interprétation pas à pas. En effet, le critère 1 suivant doit impérativement être respecté par le plan de construction.

Critère 1 *Soient $d1$ et $d2$ deux définitions d'esquisse, $d1$ étant placée avant $d2$ dans le plan de construction, et telles qu'il n'existe pas d'autre définition d'esquisse entre elles. Alors, toutes les définitions placées entre $d1$ et $d2$, qui sont par conséquent des définitions auxiliaires, sont des définitions qui sont nécessaires au calcul de $d2$ et qui n'ont pas encore été requises avant $d1$.*

Tab. 5.1: Un plan de construction non trié

```
  x1 :=param
  y1 :=param
  x2 :=param
  y2 :=param
  k1 :=param
  k2 :=param
  k3 :=param
  k4 :=param
* p3 :=initp[x1 y1]
* p4 :=initp[x2 y2]
  c1 :=mkcir[p3 k1]
* c2 :=medradcir[p4 k4]
  l2 :=initd[p3 k3]
* p2 :=interlc[l2 c1]
  l1 :=lineh[p2 k2]
* p1 :=interlc[l1 c2]
```

Nous devons donc trier ce plan de construction pour être sûrs qu'il soit de la bonne forme, c'est-à-dire qu'il respecte le Critère 1. Un tri topologique est effectué en plaçant

d'abord les définitions d'esquisse en suivant l'ordre courant, puis en interclassant les définitions auxiliaires, en plaçant chacune juste avant la première définition d'esquisse qui la nécessite (c'est-à-dire dans laquelle elle figure parmi les arguments).

Lorsqu'un plan de construction vérifie le Critère 1, le seul backtracking qui doit être fait est situé dans le sous-arbre entre $d1$ et $d2$, $d1$ exclue. Si l'utilisateur n'est pas satisfait avec les interprétations numériques proposées pour $d2$, et qu'il désire voir d'autres solutions possibles, alors on est sûr que d'autres définitions d'esquisse doivent être remises en question. Un exemple de ce tri topologique est montré aux Tables 5.1 et 5.2. Dans la Table 5.1, les définitions d'esquisse sont précédées d'une astérisque.

TAB. 5.2: Tri topologique du plan de construction

1ère phase	2ème phase	3ème phase
`x1 :=param`	`x1 :=param`	`x1 :=param`
`y1 :=param`	`y1 :=param`	`y1 :=param`
`p3 :=initp[x1 y1]`	`p3 :=initp[x1 y1]`	`p3 :=initp[x1 y1]`
	`x2 :=param`	`x2 :=param`
	`y2 :=param`	`y2 :=param`
	`p4 :=initp[x2 y2]`	`p4 :=initp[x2 y2]`
		`k4 :=param`
		`c2 :=medradcir[p4 k4]`

4ème phase	5ème phase
`x1 :=param`	`x1 :=param`
`y1 :=param`	`y1 :=param`
`p3 :=initp[x1 y1]`	`p3 :=initp[x1 y1]`
`x2 :=param`	`x2 :=param`
`y2 :=param`	`y2 :=param`
`p4 :=initp[x2 y2]`	`p4 :=initp[x2 y2]`
`k4 :=param`	`k4 :=param`
`c2 :=medradcir[p4 k4]`	`c2 :=medradcir[p4 k4]`
`k3 :=param`	`k3 :=param`
`l2 :=initd[p3 k3]`	`l2 :=initd[p3 k3]`
`k1 :=param`	`k1 :=param`
`c1 :=mkcir[p3 k1]`	`c1 :=mkcir[p3 k1]`
`p2 :=interlc[l2 c1]`	`p2 :=interlc[l2 c1]`
	`k2 :=param`
	`l1 :=lineh[p2 k2]`
	`p1 :=interlc[l1 c2]`

Dans un tel cas, on revient sur les définitions d'esquisse qui ont été définies plus tôt, et desquelles dépend $d2$. On suggère à l'utilisateur de reconsidérer certaines des valeurs qu'il avait choisies pour ces définitions. On lui propose d'abord de revoir seulement un petit nombre d'entre elles, celles qui sont placées à la plus petite distance de $d2$ dans l'arbre. Puis, progressivement, on remet en cause plus de définitions, en incluant petit à petit celles qui ont été définies beaucoup plus tôt.

5.1.3 Exemple d'utilisation de notre outil

Notre outil propose de tracer la solution pas à pas, au fur et à mesure de l'interprétation. Pour chaque nouvel objet dessiné sur la figure solution, la partie de l'esquisse correspondante est mise en surbrillance. De cette manière, l'utilisateur peut aisément suivre le processus de la construction. A chaque pas, *YAMS* propose un ensemble de résultats disponibles pour l'objet courant. Lorsque l'utilisateur en choisit un, il est construit sur la figure, et *YAMS* poursuit alors l'interprétation jusqu'au pas suivant.

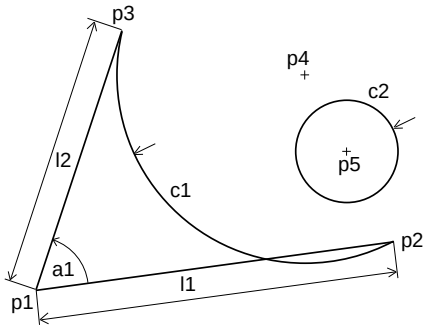


Fig. 5.2: Esquisse cotée

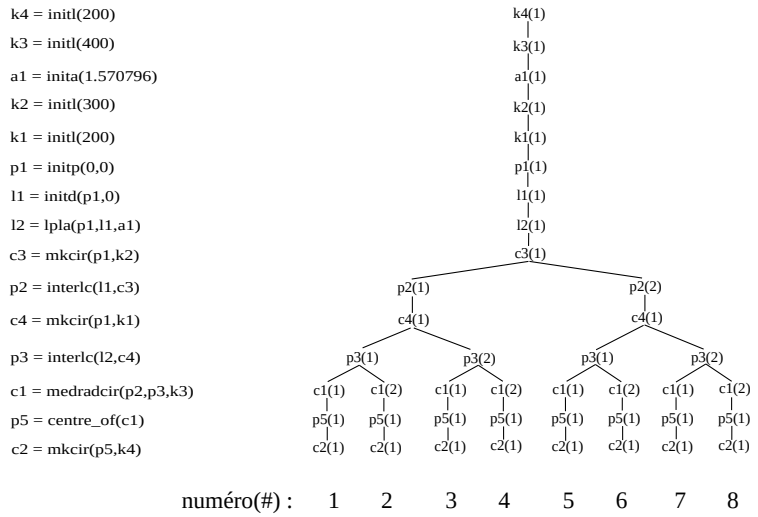
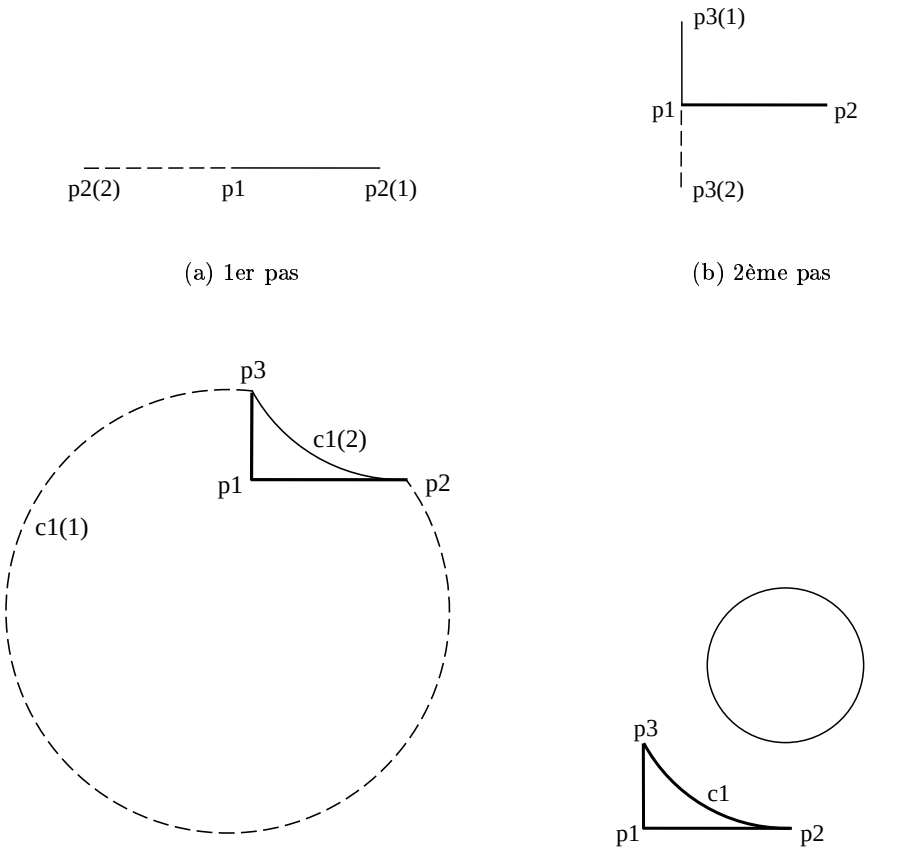


Fig. 5.3: Plan de construction correspondant à la Fig.5.2, et son arbre des solutions numéroté

Exemple 28 *Reprenons l'exemple de la Section 2.1, dont nous rappelons l'esquisse cotée à la Fig.5.2 ainsi que le plan de construction et l'arbre d'interprétation à la Fig.5.3. Sur la Fig.5.4, on peut voir le déroulement d'une interprétation pas à pas de ce système contraint. Comme on peut le voir sur son arbre des solutions, trois définitions du plan contiennent des multifonctions qui impliquent des embranchements, c'est-à-dire des choix. Il s'agit de*



(a) 1er pas

(b) 2ème pas

(c) 3ème pas

(d) fin de l'interprétation

Fig. 5.4: Interprétation en 3 pas, et résultat final

$p2 = interlc(l1, c3)$, $p3 = interlc(l2, c4)$ et $c1 = medradcir(p2, p3, k3)$. Les définitions des points $p2$ et $p3$ et de l'arc de cercle $c1$ provoquent donc une pause dans l'interprétation. A chaque pas, (représenté sur la Fig.5.4 en (a) pour $p2$, en (b) pour $p3$, et en (c) pour $c1$), l'utilisateur choisit un des deux résultats disponibles. Nous avons représenté en gras la partie de la figure qui a déjà été gelée, en trait normal la valeur choisie, et en pointillés celle qui n'a pas été retenue. Sur la Fig.5.4(d), on peut observer la solution obtenue après ces choix successifs.

Cependant, construire une figure entièrement avec cet outil peut être une tâche relativement longue lorsque les objets sont volumineux et les multifonctions nombreuses. C'est pourquoi, dans un premier temps, nous avons ajouté une fonctionnalité inspirée des points d'arrêt des débogueurs. Elle permet de disposer des marqueurs sur le plan de construction. Ces marqueurs délimitent les zones dans lesquelles une interprétation pas à pas devra être effectuée, et les zones dans lesquelles on aura une interprétation automatique. Par exemple,

l'utilisateur peut décider d'effectuer automatiquement l'interprétation du début du plan, et d'interpréter pas à pas uniquement quelques définitions à la fin du plan. Il peut également définir une zone dans le plan, correspondant par exemple à une sous-figure qui lui convient, qui sera "sautée" par l'interprétation pas à pas. A l'intérieur de cette zone, l'interprétation sera automatique. Ceci permet de concentrer les choix uniquement sur la partie litigieuse d'une figure.

Toutefois, il est parfois difficile pour un utilisateur de repérer dans le plan de construction les parties correspondant aux éléments qu'il souhaite modifier. C'est pourquoi, dans un deuxième temps, nous proposons un autre type d'outil, plus intuitif que le précédent, sous la forme d'une interface utilisateur de manipulation de solutions.

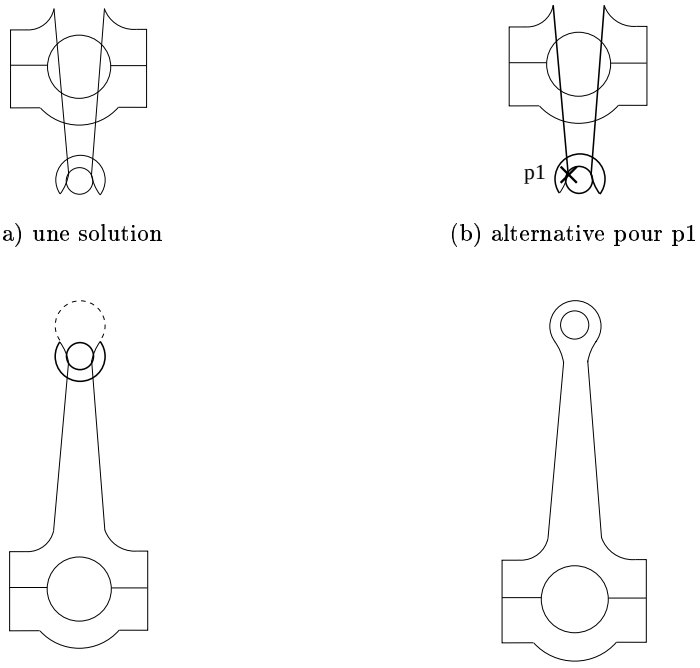
5.2 Manipulation de solutions

L'idée est de permettre à l'utilisateur de réviser une solution qui a déjà été complètement calculée et construite par le solveur. Pour cela, on peut effectuer une première interprétation, par exemple en utilisant le gel d'une branche, et suggérer ensuite d'autres solutions proches si l'utilisateur n'est pas entièrement satisfait. Notons que cette partie de notre étude est encore en cours de développement.

Le processus commence à partir d'une solution déjà calculée. L'utilisateur sélectionne et déplace à l'aide de sa souris l'élément de la figure qu'il ne juge pas à la bonne place. Les autres positions possibles qui lui sont proposées doivent aussi respecter les contraintes. C'est pourquoi nous utilisons le plan de construction pour les calculer. L'utilisateur ne peut déplacer l'élément sélectionné que vers une des solutions proposées.

Il peut arriver que l'utilisateur souhaite déplacer un élément auquel d'autres sont liés par une dépendance $\rightarrow$. Dans ce cas, l'ensemble des éléments liés tout entier sera déplacé en même temps que l'élément sélectionné. Lorsqu'un utilisateur sélectionne un élément, nous utilisons donc le graphe de dépendance pour savoir quels autres éléments en dépendent, et ce groupe d'éléments lui est signalé. De cette façon, l'utilisateur peut aisément visualiser la partie de la figure qui sera modifiée, et quelles conséquences aura le déplacement de l'élément choisi.

Exemple 29 *Sur la Fig.5.5(a), on peut voir une solution produite pour la conception d'une bielle. Dans cette solution, la partie allongée de la bielle est retournée, c'est pourquoi l'utilisateur souhaite manipuler cette partie de la figure. Il sélectionne le point* $p1$ *représenté par une croix. Puis, la partie de la figure qui est liée à ce point est affichée en traits gras. La Fig.5.5(b) montre l'alternative proposée automatiquement pour* $p1$*, représentée par une croix en pointillés. On suppose que l'utilisateur a choisi l'autre point proposé en faisant glisser* $p1$ *à la souris. Le résultat est présenté Fig.5.5(c). Ensuite, l'utilisateur sélectionne l'arc, qu'il souhaite voir au-dessus de la partie allongée de la bielle. Le petit cercle ayant le même centre que l'arc doit bouger avec lui, donc les deux éléments sont en gras. L'alter-*



(a) une solution (b) alternative pour p1

(c) alternative pour l'arc (d) résultat final

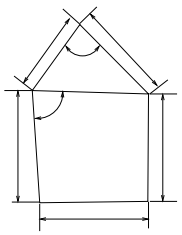
Fig. 5.5: Manipulation d'une solution pour une bielle

native proposée pour l'arc est en pointillés. Si l'utilisateur choisit l'arc proposé, on obtient finalement le résultat présenté Fig.5.5(d).

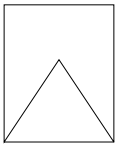
Cette méthode tire parti de la structure de l'espace des solutions. L'arbre des solutions nous permet de trouver facilement les autres solutions possibles pour un élément en sautant d'une branche à l'autre. Notons qu'elle nécessite la même hypothèse que la méthode d'interprétation pas à pas, c'est-à-dire la vérification par le plan du Critère 1. Le même tri topologique doit donc être effectué avant, pour des raisons identiques. Les définitions auxiliaires ne sont placées dans le plan de construction que lorsqu'elles sont nécessaires au calcul de la prochaine définition d'esquisse, et pas avant. De cette façon, si l'on change d'embranchement dans une couche, on peut continuer à suivre la branche portant la même numérotation relative dans les couches suivantes si celles-ci n'en dépendent pas.

Cependant, on reste dépendant du plan de construction. En effet, dans certains cas, manipuler une figure peut ne pas être aussi intuitif qu'on le souhaiterait. L'Exemple 30 illustre ce cas de figure.

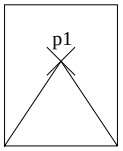
Exemple 30 *Sur la Fig.5.6(a), l'utilisateur a dessiné une esquisse contrainte représentant un triangle et un quadrilatère. Il semble clair d'après l'esquisse qu'il a tracée qu'il souhaite que le triangle soit en-dehors du quadrilatère. Supposons que la figure solution donnée est celle représentée à la Fig.5.6(b). Il voudra intuitivement déplacer le point* p1 *vers la droite. Mais dans cette configuration, avec les contraintes particulières qu'il a données, le solveur a fourni un plan de construction qui démarre par la construction de* p1*, donc* p1 *est situé au sommet de l'arbre des solutions. Il est donc impossible de bouger ce point, car aucune solution ne peut être proposée, comme illustré Fig.5.6(c). Par contre, il peut déplacer* p2*, comme on peut le voir Fig.5.6(d), qui implique le déplacement de* 3 *des côtés du quadrilatère (en gras), afin d'obtenir la figure présentée Fig.5.6(e) qui est celle qu'il souhaitait.*



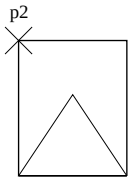
(a) esquisse



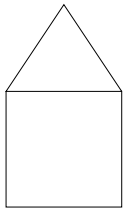
(b) solution calculée



(c) pas de proposition pour p1



(d) alternative pour p2



(e) résultat final

Fig. 5.6: Impossibilité de manipuler certains points

Bien sûr, il serait possible de tenter d'effectuer un autre résolution symbolique, produisant un plan de construction différent (mais produisant toujours les mêmes solutions) qui ne commencerait pas par la définition de l'élément que l'on souhaite modifier. Mais en admettant qu'on puisse en trouver un, ce qui n'est pas toujours le cas puisqu'il est possible que cet élément soit une base obligatoire au reste de la construction, cela retirerait l'avantage qu'offrait une résolution symbolique.

On peut remarquer cependant que les deux opérations différentes de l'Exemple 30 (déplacer le point $p1$ ou le groupe de points lié à $p2$) aboutissent à deux figures équivalentes à un déplacement près. Une solution pourrait donc être, lorsqu'un tel cas est détecté, de déduire automatiquement la partie alternative de la figure qui peut être modifiée, ainsi que le déplacement associé, qui produiraient ensemble le même résultat que la modification de l'élément choisi, qui est ici impossible à effectuer. Ceci éviterait de recommencer la phase de résolution symbolique. Cette théorie n'en est pour l'instant qu'au stade de projet, et mériterait d'être poussée plus avant.

Chapitre 6

Le prototype *SAMY*

Les méthodes que nous avons définies tout au long des chapitres précédents, le gel d'une branche, la navigation ou la manipulation de solutions, ont été concrétisées par différents modules satellites, ajoutés au noyau de *YAMS* pour former un prototype. Ces modules ainsi que le noyau de *YAMS* sont regroupés sous le nom de *SAMY* (Sélection Automatique : Modules pour Yams).

Dans ce chapitre, nous présentons ce groupe de modules au travers de quelques exemples, illustrés de captures d'écran prises aux différents stades de leur traitement, après avoir détaillé son interface et ses fonctionnalités générales.

6.1 Présentation générale de *SAMY*

Avant de détailler quelques exemples, nous allons tout d'abord présenter le fonctionnement général de *SAMY*. Une capture d'écran du logiciel dans l'état où il se trouve dès son ouverture, en attente de chargement d'un fichier d'exemple, nous permet d'observer son allure globale. Elle est présentée à la Fig.6.1.

Sur cette capture d'écran, nous pouvons voir que l'interface de *SAMY* se compose de trois parties principales : un panneau d'outils (boutons d'exécution, listes d'édition, outils de changements de paramètres) sur la gauche, et deux fenêtres de visualisation sur la droite, l'une au-dessus de l'autre.

Les deux fenêtres de droite permettent de visualiser en même temps l'esquisse fournie par l'utilisateur (en haut) et la solution courante que l'utilisateur est en train d'examiner (en bas). Leur disposition offre l'avantage d'avoir toujours l'esquisse sous les yeux, à des fins de comparaison visuelle, lorsqu'une solution est proposée, lorsque l'utilisateur navigue parmi les solutions, lorsqu'il effectue une interprétation pas à pas, ou encore lorsqu'il manipule les solutions. La souris, grâce à laquelle on peut effectuer des translations sur les figures, permet également de sélectionner une ou plusieurs parties d'une des deux figures. Les

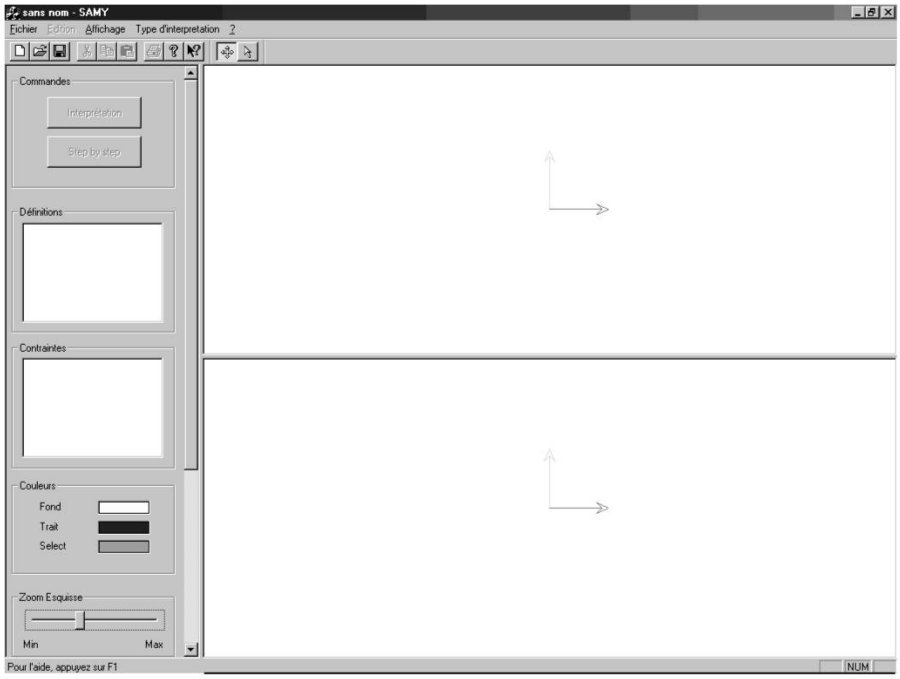


Fig. 6.1: Capture d'écran : ouverture de *SAMY*

parties correspondantes sur l'autre figure sont alors également sélectionnées. On peut ainsi aisément situer un élément de l'esquisse sur la figure solution, et inversement.

Le panneau de gauche contient différents types d'outils qui sont détaillés ci-dessous.

- un premier bouton intitulé "Interprétation" permet de lancer le processus d'interprétation. Il est disponible dès qu'un fichier d'exemple a été chargé. Le type d'interprétation qui est lancé lorsqu'on appuie sur ce bouton est déterminé par l'option qui a été choisie dans le menu "Type d'interprétation" de la barre de menu. Deux types d'interprétation sont disponibles dans ce menu : "Classique" ou "Gel d'une branche" ;
- un deuxième bouton intitulé "Step by step" permet d'effectuer une interprétation pas à pas ;
- une liste déroulante "Définitions" dans laquelle est affiché le plan de construction dès qu'un fichier d'exemple est chargé ; lorsqu'une partie des figures est sélectionnée, les définitions des éléments correspondants sont mises en surbrillance dans cette liste.
- une liste déroulante "Contraintes" dans laquelle est affichée la liste des contraintes dès qu'un fichier d'exemple est chargé ; lorsqu'une partie des figures est sélectionnée, les contraintes dans lesquelles les éléments correspondants sont impliqués sont mises en surbrillance dans cette liste.
- un ensemble de boutons "Couleurs" qui permet de modifier les couleurs du fond, du trait, et des éléments sélectionnés ;
- deux barres de défilement, contrôlant le zoom à appliquer sur chacune des figures affichées.

Notons que dans l'état actuel des choses, *SAMY* ne permet pas à lui seul de saisir des esquisses cotées, puisqu'il n'est pas associé à un modeleur. En effet, seul le noyau de *YAMS* a été repris. *SAMY* prend donc pour l'instant en entrée des fichiers ASCII contenant un plan de construction, une liste de contraintes, ainsi qu'une description des objets constituant la figure. De tels types de fichiers sont générés automatiquement par *YAMS*.

6.2 Exemple 1 : l'heptagone

Commençons par un exemple simple. Nous présentons à la Fig.6.2 une capture d'écran effectuée juste après la lecture d'un exemple intitulé *heptagone*. L'esquisse de l'heptagone se place dans la fenêtre de visualisation du haut. L'heptagone possède 51 définitions et 26 contraintes, dont 7 contraintes de longueurs (une par côté), et 4 contraintes d'angles, et dont on peut voir une partie sur les listes déroulantes.

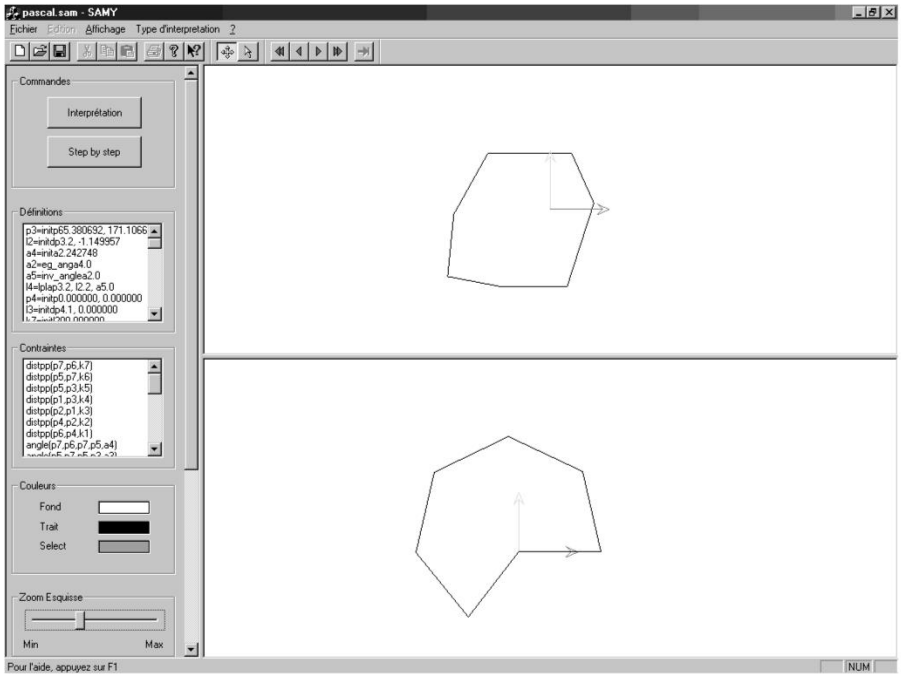


Fig. 6.2: Heptagone : une figure solution

La Fig.6.2 est obtenue en appuyant sur le bouton "Interprétation" après avoir choisi l'option "Classique" pour le type d'interprétation. Sur cette figure, on peut voir dans la fenêtre de visualisation du bas une des figures que l'on peut obtenir en effectuant une interprétation classique. On constate que cette figure ne semble pas être celle qu'attendait l'utilisateur. En effet, elle présente un défaut évident de convexité par rapport à l'esquisse. Notons au passage que lorsqu'on lance une interprétation, une nouvelle barre d'outils apparaît en haut. Celle-ci comporte différents types de flèches qui permettent d'avancer ou de

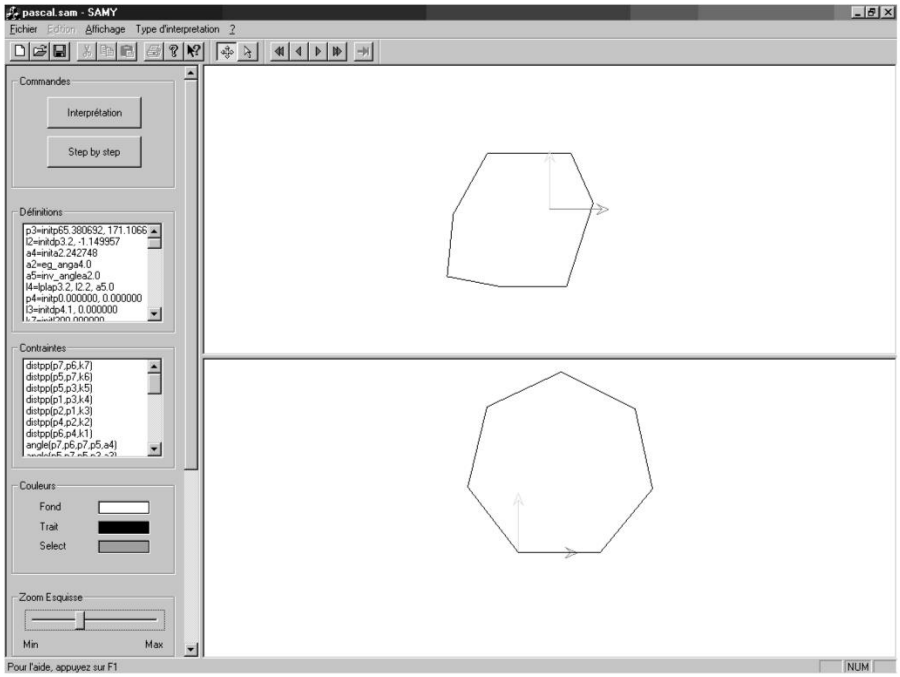


FIG. 6.3: Heptagone : figure obtenue par gel d'une branche

reculer parmi les solutions, de passer directement à la dernière ou de revenir directement à la première solution, ou encore de voir la solution dont on donne le numéro.

En effectuant une interprétation faisant intervenir la méthode de gel d'une branche (menu "Type d'interprétation", choix "Gel d'une branche", puis bouton "Interprétation"), on obtient directement la bonne solution, qui est présentée sur la capture d'écran de la Fig.6.3. Dans ce cas, une seule solution est pour l'instant proposée après l'exécution du gel d'une branche. L'utilisateur aura encore cependant la possiblilté par la suite de choisir de naviguer parmi les solutions qui avaient été mises de côté par ce processus.

6.3 Exemple 2 : un pavage de triangles

Prenons maintenant un exemple représentant un pavage d'une portion de l'espace composé de triangles. Cet exemple est présenté à la Fig.6.4. La figure comprend en tout 34 triangles adjacents, que l'on souhaite équilatéraux pour former un pavage régulier, de côtés égaux à 150. Le plan de construction produit pour cet exemple contient 156 définitions, et 175 contraintes, que nous ne donnerons pas ici afin déviter de surcharger cette section. Toutes les contraintes sont métriques. Cet exemple rappelle celui des 15 triangles que nous avons déjà étudié dans les chapitres précédents, mais celui-ci est de taille un peu plus importante.

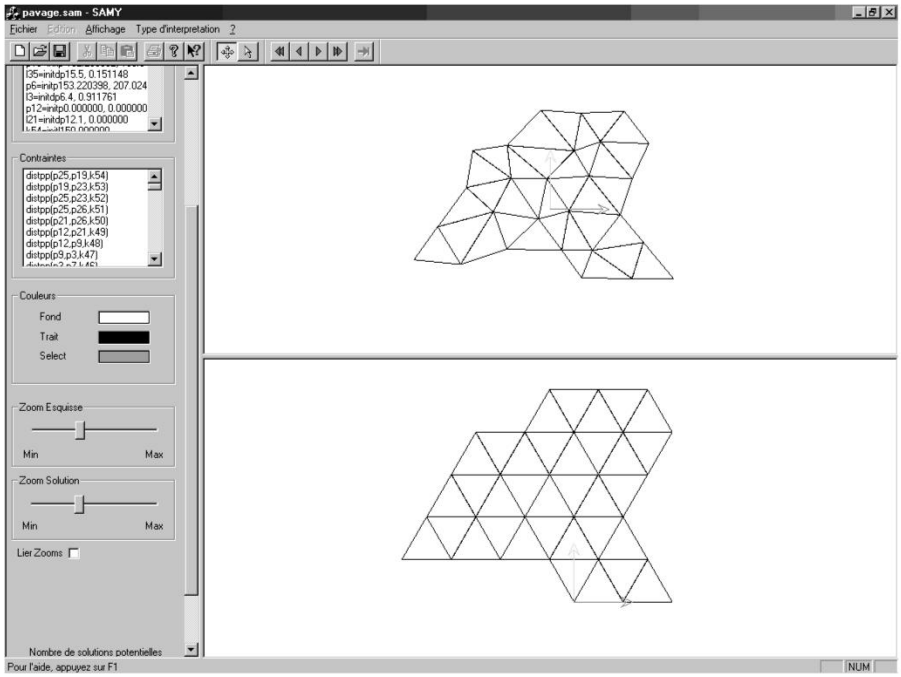


FIG. 6.4: Pavage régulier : figure obtenue par gel d'une branche

Comme nous l'avons déjà évoqué, ce genre de cas de figure possède 2^{p-2} solutions distinctes, où p est le nombre de points. Dans notre cas, avec 27 points, nous obtiendrions 33554432 solutions potentielles. La Fig.6.4 présente, dans la fenêtre de visualisation du bas, la solution trouvée grâce à la méthode de gel d'une branche.

Sur un exemple comme celui-ci, les performances sont extrêmement satisfaisantes. La phase de chargement du fichier immédiatement suivie du tri topologique du plan de construction prend environ 2 secondes. Le temps nécessaire à l'interprétation numérique avec la méthode de gel d'une branche est de l'ordre de la fraction de seconde. Le résultat escompté est donc trouvé quasi instantanément.

Le même genre d'exemple peut être également réalisé avec des pavages irréguliers, avec des longueurs différentes pour tous les côtés des triangles, avec un résultat tout aussi rapide puisque celui-ci ne dépend pas des valeurs numériques. Sur la Fig.6.5, on peut voir un pavage similaire à l'exemple précédent, mais comportant 3 contraintes de longueur différentes. Nous présentons ici la solution sélectionnée par le gel d'une branche sur la fenêtre du bas de la Fig.6.5, ainsi qu'une des solutions rejetées par cette méthode sur la Fig.6.6.

Si l'utilisateur juge un élément mal placé, il a la possibilité de le déplacer grâce à la technique de manipulation interactive, dont cet exemple va nous permettre d'illustrer les avantages et les inconvénients.

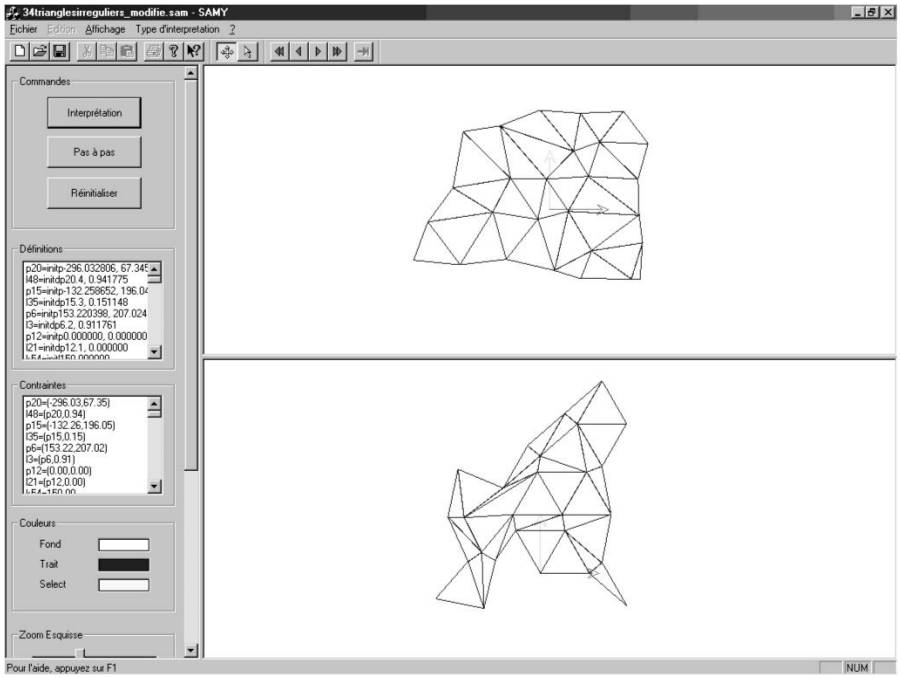


FIG. 6.5: Pavage irrégulier : figure obtenue par gel d'une branche

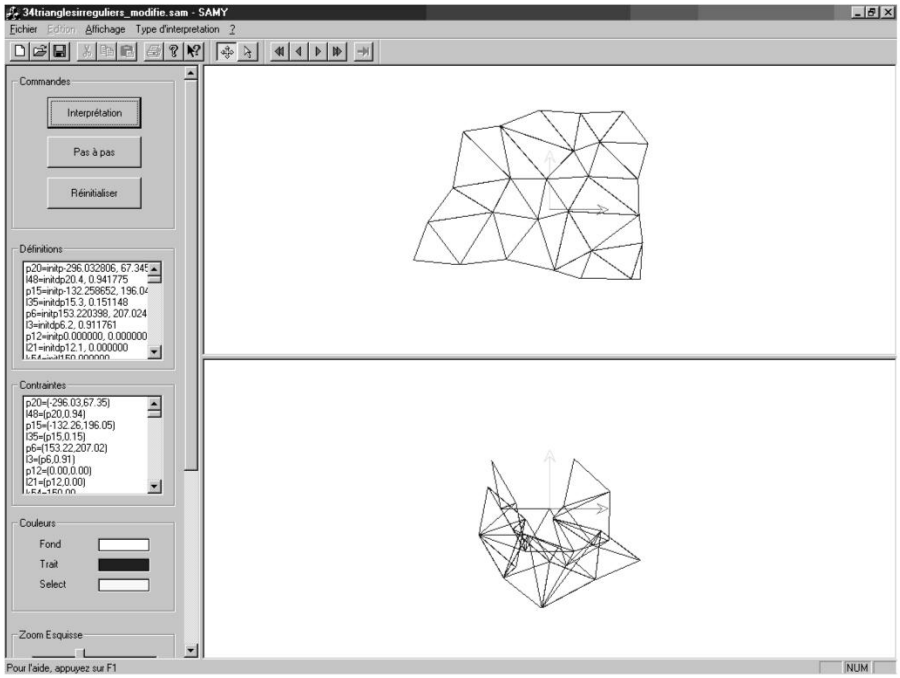


FIG. 6.6: Pavage irrégulier : une des figures rejetées par gel d'une branche

La Fig.6.7 montre quelles alternatives sont proposées lorsque l'utilisateur a choisi le point en bas à gauche. Ce dernier est accentué par un carré vert clair, de la même façon que le point auquel il correspond sur l'esquisse dans la fenêtre du haut. L'autre point possible proposé est représenté par un carré rouge. La Fig.6.8 représente la figure obtenue après mise à jour lorsque l'utilisateur a accepté l'alternative proposée.

Mais comme nous l'avons expliqué un peu plus tôt, dans certains cas la mise à jour peut produire un résultat moins intuitif. Par exemple, si l'utilisateur souhaite cette fois déplacer le point matérialisé par le carré clair sur la Fig.6.9, et qu'il choisit le point proposé par *SAMY*, alors le résultat après mise à jour est celui qui est présenté à la Fig.6.10. Ceci est du au fait que la définition concernant la construction de ce point est placée très tôt dans le plan de construction, et que sa remise en cause provoque une remise en cause également d'un grand nombre d'autres définitions et donc d'éléments.

6.4 Exemple 3 : le levier

Nous allons cette fois reprendre un exemple déjà évoqué dans les chapitres précédents, celui du levier. En effet, celui-ci nous semble assez significatif, puisqu'il contient à la fois des contraintes métriques mais aussi des contraintes booléennes, et que sa taille est assez importante. Il nous permet donc d'exploiter toutes les fonctionnalités de *SAMY*.

La Fig.6.11 montre l'état de *SAMY* lorsque le fichier de l'exemple du levier est chargé. Rappelons que celui-ci comporte 252 définitions ainsi que 106 contraintes, qui sont données en Annexe C. Rappelons également que l'interprétation numérique classique génère 160 solutions.

Tout d'abord, l'utilisateur peut effectuer une interprétation pas à pas de ce système de contraintes. Lorsqu'il appuie sur le bouton "Step by step", *SAMY* entre en mode *pas à pas*, et une nouvelle boîte de dialogue s'affiche en haut à gauche de l'écran. Comme nous pouvons le voir sur la Fig.6.12, celle-ci contient à droite une liste déroulante, à gauche quatre boutons représentant des flèches directionnelles, un bouton "OK", et une case contenant un numéro.

La liste déroulante contient le plan de construction. A chaque étape du processus d'interprétation pas à pas, la définition d'esquisse sur laquelle on s'est arrêté est mise en surbrillance. On peut ainsi voir de quel élément de la figure on est en train de décider l'emplacement.

Le premier résultat possible fourni par la multifonction de la définition sur laquelle on s'est arrêté est affiché à l'écran. Son numéro s'affiche donc dans la case du bas : $1/n$, s'il existe n résultats possibles pour cette multifonction. S'il s'agit de la définition d'un point, celui-ci est représenté dans la fenêtre d'édition par un carré. Si de plus le segment dont il fait partie est complet, c'est-à-dire si le point constituant l'autre extrémité du segment a déjà été calculé, alors le segment est tracé.

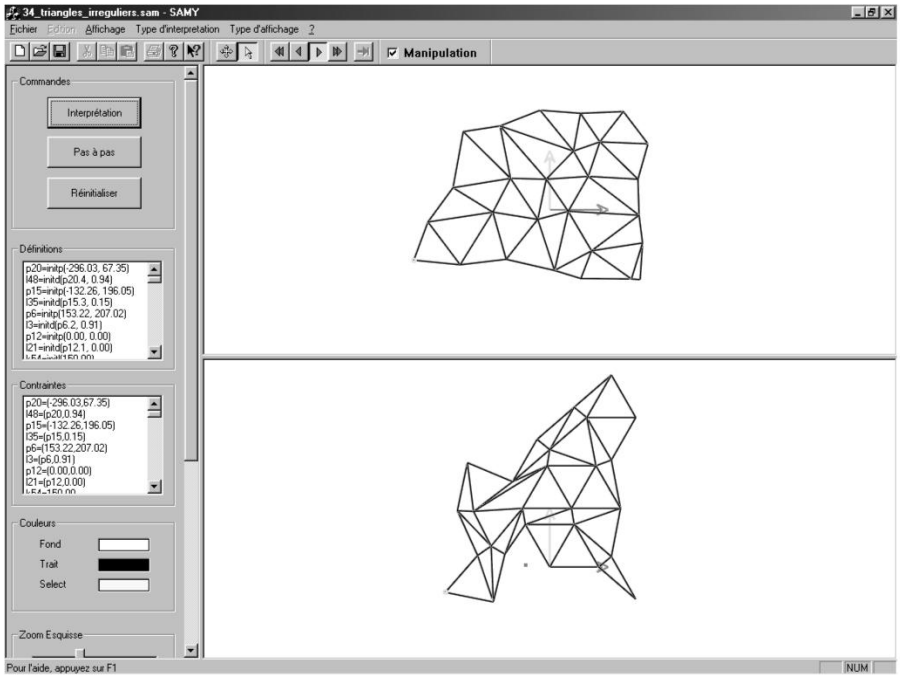


Fig. 6.7: Pavage irrégulier : 1 alternative proposée pour le point

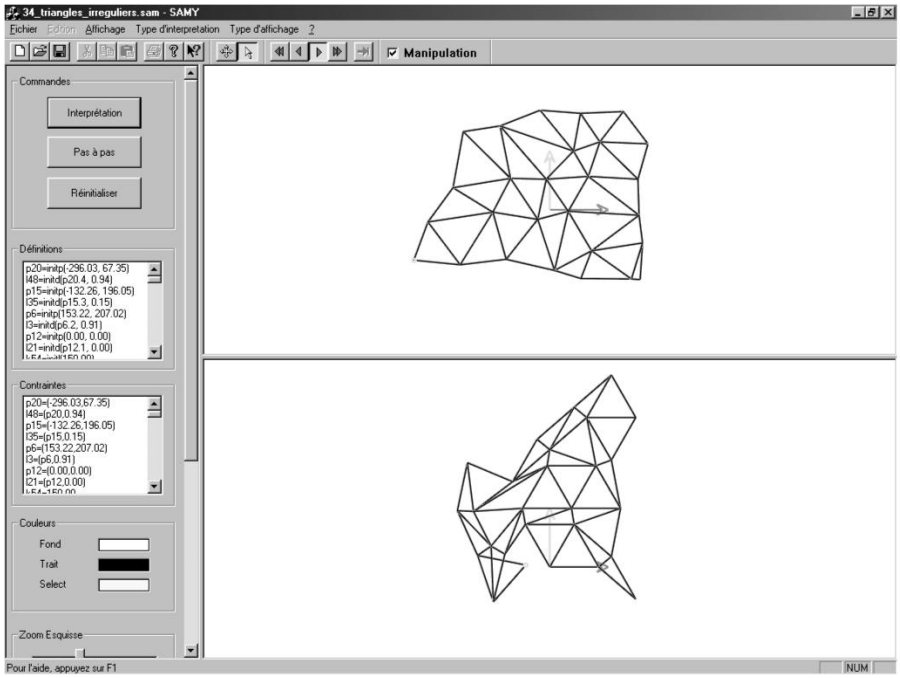


Fig. 6.8: Pavage irrégulier : mise à jour après choix de l'alternative

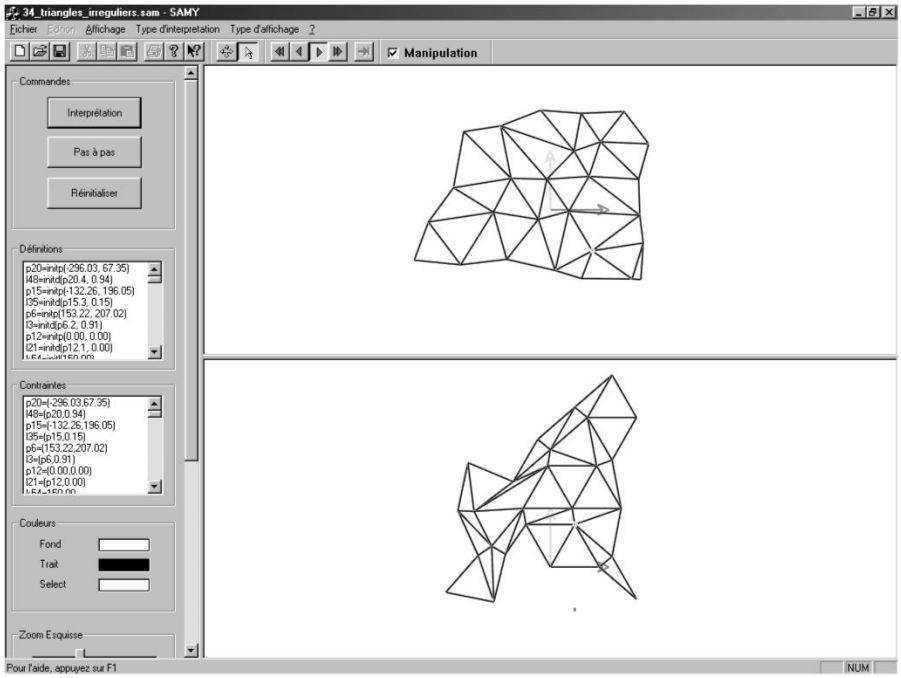


Fig. 6.9: Pavage irrégulier : point à déplacer

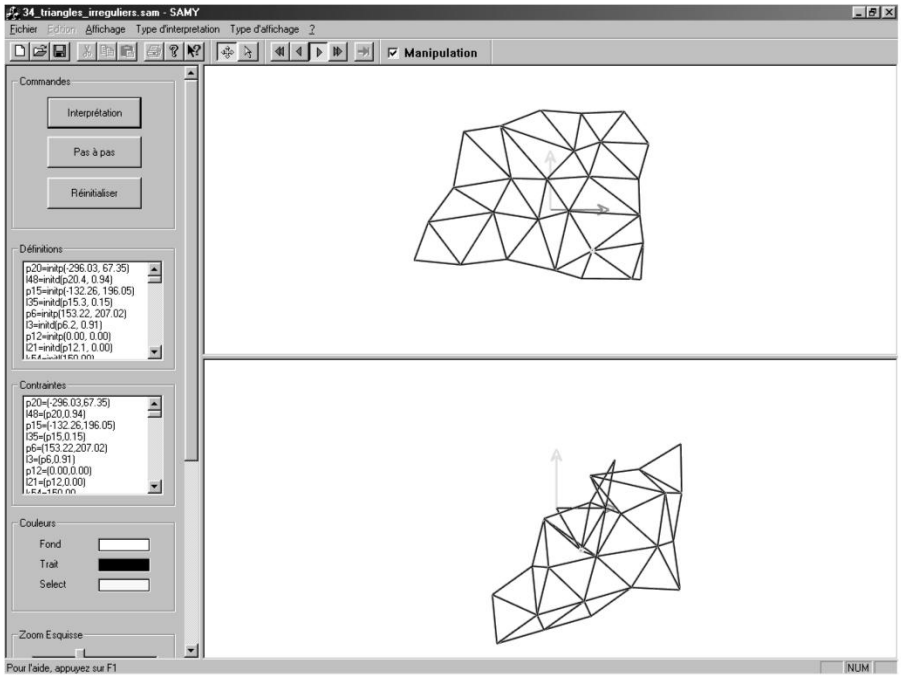


Fig. 6.10: Pavage irrégulier : mise à jour moins intuitive que pour la Fig.6.8

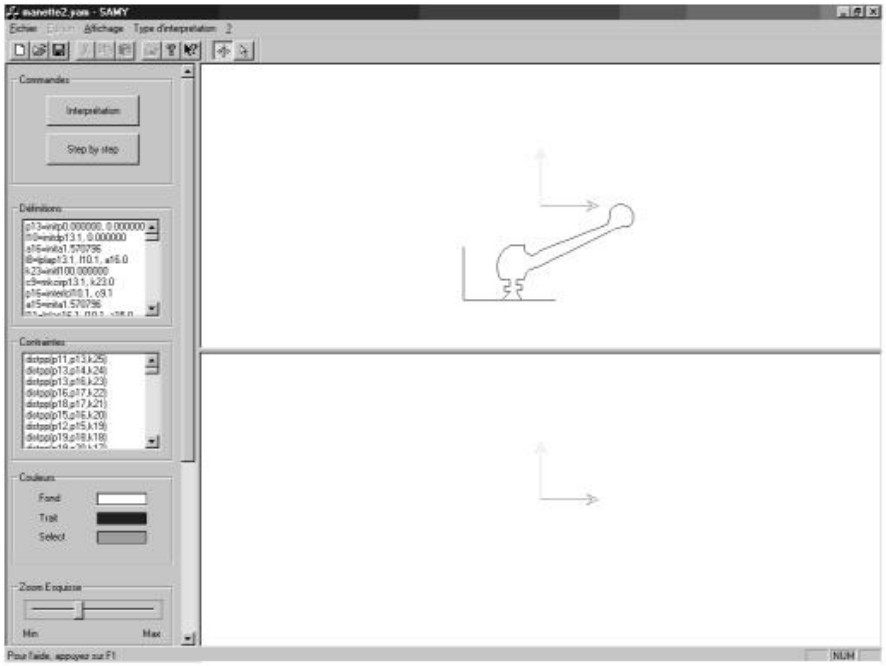


Fig. 6.11: Exemple du levier

Si cette proposition faite pour l'élément courant ne convient pas à l'utilisateur, il peut alors choisir de visualiser un autre résultat possible en appuyant sur la flèche droite. Le numéro de solution est alors actualisé dans la case du bas $(2/n)$. Il peut effectuer cette opération plusieurs fois, jusqu'à atteindre le dernier résultat proposé pour cet élément. Il peut également effectuer l'opération inverse, c'est-à-dire revenir à un résultat précedent, en appuyant sur la flèche gauche. Il navigue ainsi parmi les résultats qui lui sont proposés.

Lorsque l'utilisateur a choisi un résultat qui lui convient, il passe alors à l'étape suivante en appuyant sur la flèche bas. Il peut ainsi naviguer parmi les résultats de la prochaine définition d'esquisse qui implique un choix. S'il veut réviser un choix qu'il a effectué pour une définition antérieure, il peut remonter dans le plan de construction avec la flèche haut.

La Fig.6.12 montre une interprétation pas à pas en cours d'exécution. On peut voir dans la boîte de dialogue que l'utilisateur est en train de choisir un résultat pour le point $p23$, et qu'il examine actuellement le premier sur 2 possibles.

Cependant, l'interprétation totale en mode pas à pas de ce genre d'exemple est une opération assez longue à effectuer. La méthode de gel d'une branche, qui donne de très bons résultats avec des contraintes métriques, est ici gênée par la présence de contraintes booléennes. Celles-ci ne sont cependant qu'au nombre de deux, ce qui nous permet d'avoir tout de même un élagage assez conséquent de l'arbre des solutions, puisqu'à l'issue de cette interprétation avec gel il ne reste que 4 solutions.

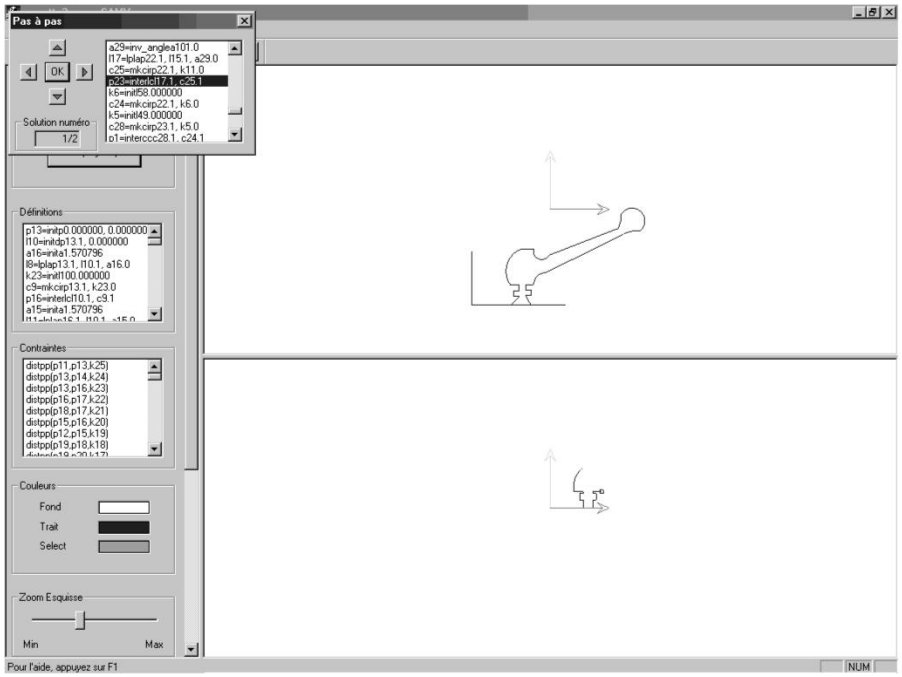


Fig. 6.12: Examen de la solution 1/2 pour la définition de $p23$

Lorsqu'on lance l'interprétation avec gel, la première de ces 4 solutions est d'abord présentée à l'utilisateur. On peut l'observer sur la Fig.6.13. La barre d'outils "flèches" permet alors de visualiser les autres solutions (comme lors d'une interprétation classique). C'est ainsi que l'on découvre, parmi ces 4 solutions possibles, la solution attendue qui est présentée à la Fig.6.14. Celle-ci aura donc été trouvée en un temps extrêmement court, puisque la totalité de l'opération, constituée de la lecture du fichier exemple, du tri topologique, de l'interprétation avec gel suivie du parcours de 4 solutions (au pire) ne prend qu'environ 10 secondes en tout.

6.5 Exemple 4 : le support de rail

L'exemple suivant présente une vue du dessus d'un support de rail, pièce de quincaillerie utilisée dans l'industrie du bâtiment. Cet exemple va nous permettre d'illustrer une des limitations de l'interprétation pas à pas, en même temps que de représenter un aspect plus appliqué dans le monde de l'usinage.

La Fig.6.15 montre plusieurs étapes de la construction pas à pas du support de rail, dont l'esquisse est représentée sur les fenêtres supérieures de chaque écran. On remarque assez distinctement qu'il se produit un "saut" au cours de cette construction. Ce type de phénomène est du à la décomposition en plusieurs sous-figures qui a été faite pour résoudre le système. La construction se fait alors en plusieurs parties, chacune dans un

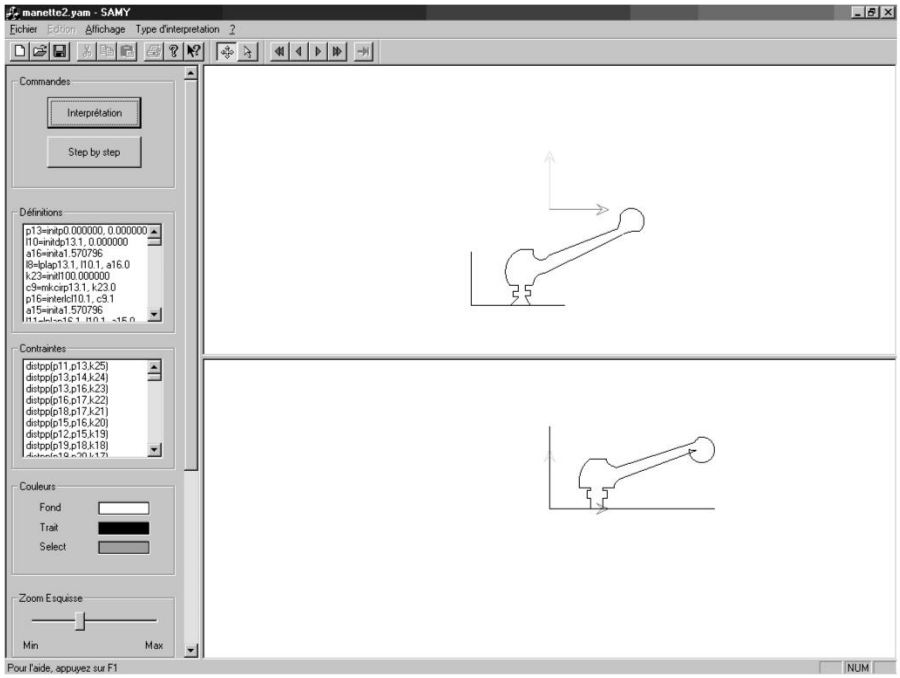


Fig. 6.13: Première solution proposée parmi 4 pour le levier, par interprétation avec gel

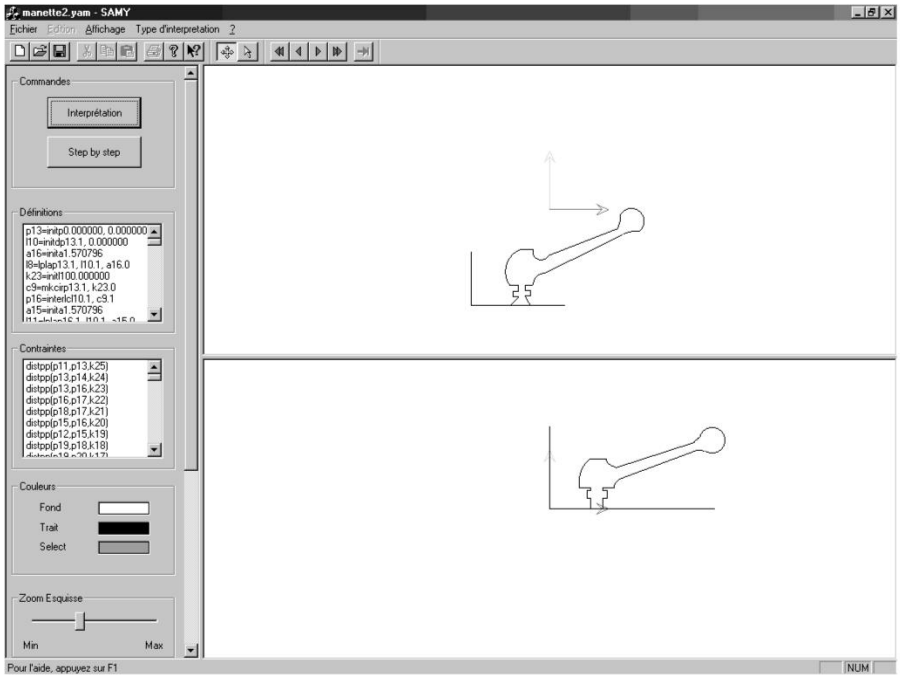


Fig. 6.14: La bonne solution pour le levier

repère différent, et qui sont ensuite assemblées en une seule. C'est cet assemblage qui provoque un changement de repère pour certains éléments de la figure.

Dans le cas de ce support de rail, la résolution a produit un plan de construction contenant 2 sous-figures. Il y a donc un seul saut, ce qui rend encore raisonnable la visualisation de l'interprétation pas à pas. En revanche, lorsque les sous-figures sont en plus grand nombre, la lecture des étapes devient beaucoup plus difficile.

Notons enfin que cet exemple a été volontairement compliqué pour les besoins de l'illustration, puisqu'il est également soluble sans recourir à une décomposition, et que dans ce cas la solution souhaitée (représentée à la Fig.6.16) est trouvée instantanément par la méthode de gel d'une branche.

Les différentes techniques que nous avons proposées sont donc utilisables pour résoudre des problèmes dans des domaines très différents, et pour des exemples de natures variées. Dans un grand nombre de cas, une ou l'association de plusieurs de ces techniques permet d'aboutir à la solution désirée en un temps immédiat ou très raisonnable. Le module *SAMY* apporte a *YAMS* une grande convivialité dans la sélection de cette solution.

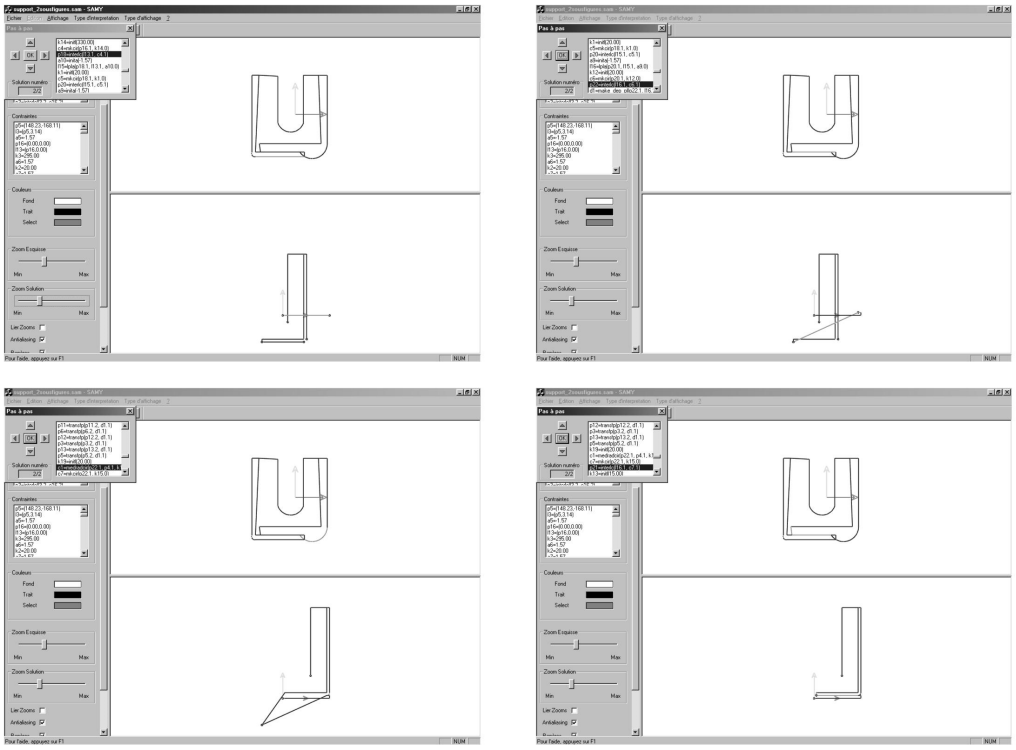


Fig. 6.15: 4 étapes de la construction pas à pas du support de rail

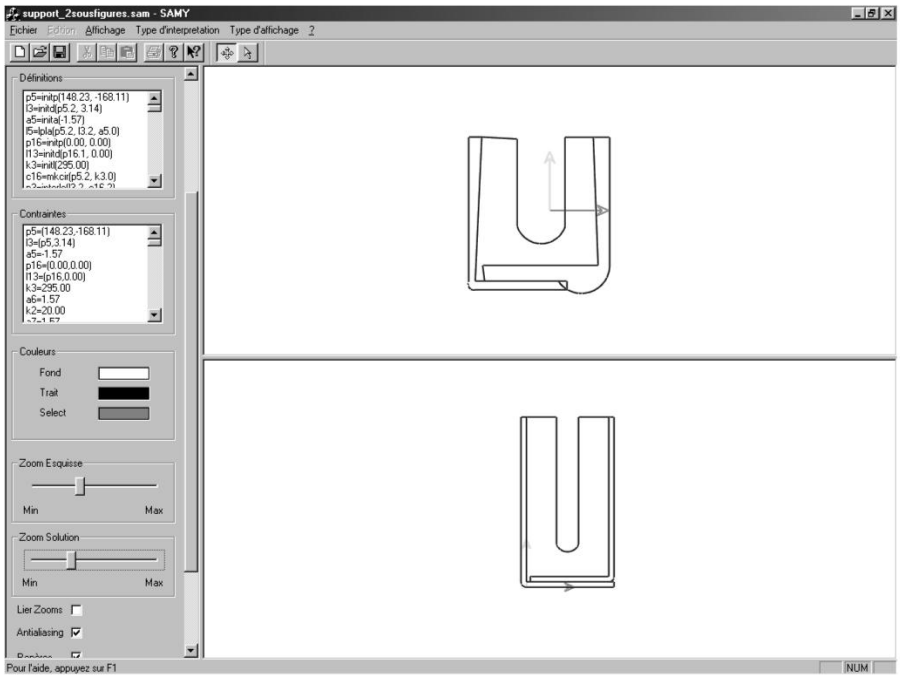


Fig. 6.16: La bonne solution pour le support

Conclusion

Bilan

Nous avons traité dans cette thèse des problèmes liés à la sélection de solutions parmi un espace de solutions dans le cadre des résolutions symboliques de systèmes de contraintes géométriques. Il s'agit d'un travail exploratoire, dans un domaine où beaucoup de bonnes intentions ont été émises, mais pas de véritable réponse. Nous avons donc apporté plusieurs approches complémentaires, comme la réorganisation statique ou dynamique des données, la sélection automatique de solution grâce à la méthode de gel d'une branche, ou encore des outils d'exploration interactifs.

Nous avons travaillé sur un espace de solutions produit par un solveur géométrique formel 2D existant nommé *YAMS*. *YAMS* est un prototype qui permet de résoudre des problèmes géométriques se présentant sous la forme de systèmes de contraintes géométriques. Il permet à un utilisateur souhaitant concevoir une pièce quelconque de saisir une esquisse de sa pièce, puis de poser diverses contraintes supplémentaires sur cette esquisse (contraintes de longueur, d'angles, de tangence, etc.). *YAMS* résout ensuite ce système de contraintes qui lui est soumis pour former un *plan de construction* général. Pour obtenir une figure solution, la dernière phase consiste alors à interpréter numériquement le plan de construction avec les valeurs de la cotation. Cette façon de faire offre l'avantage de ne pas nécessiter une deuxième résolution lorsque l'on souhaite modifier les valeurs numériques. Seule la phase d'interprétation doit dans ce cas être recommencée.

C'est à cette phase que nous nous sommes plus précisément intéressés. Le résultat qu'elle fournit se présente sous la forme d'un *arbre de solutions*. Cette structure est due à la présence dans les définitions du plan de construction de *multifonctions* géométriques, c'est-à-dire des fonctions pouvant renvoyer plusieurs résultats, selon les valeurs particulières données. La présence d'une multifonction dans une définition induit un choix représenté par un embranchement de l'arbre des solutions. Lorsque de nombreuses multifonctions sont présentes dans le plan de construction, la taille de l'arbre peut prendre des proportions gigantesques, rendant le parcours des solutions extrêmement difficile. C'est pourquoi nous avons cherché d'une part à sélectionner automatiquement la "bonne" solution en nous basant sur l'esquisse tracée, puis d'autre part à faciliter le parcours des solutions.

Avant toute chose, nous avons introduit une spécification algébrique de toutes les no-

tions existantes, afin de pouvoir constituer une base solide à ce travail. La spécification algébrique en *OBJ3* nous a permis de décrire avec rigueur et précision l'univers géométrique, ainsi que les notions de base utilisées par *YAMS* comme le *plan de construction*, le *graphe de dépendance*, le *tri topologique*, l'*arbre des solutions*, etc. Nous avons également pu vérifier cette spécification grâce à la possibilité qu'offre le langage *OBJ3* de réduire des termes par réécriture. Nous avons donc effectué des séries de tests qui ont validé notre spécification.

Une fois les données bien clarifiées, nous avons pu chercher tout d'abord une réponse au problème de sélection de solutions parmi l'espace de solutions généré par *YAMS*. Nous pensons que l'utilisateur ne trace pas l'esquisse totalement au hasard, mais qu'il a au contraire une idée bien précise d'allure générale de la figure qu'il souhaite obtenir et donc qu'il trace une esquisse ressemblant à ce que sera la solution, si elle existe. Nous avons donc premièrement donné une définition de la ressemblance entre deux figures, basée sur la définition de nouvelles notions comme la déformation continue d'un système de contraintes, la S-homotopie, et la numérotation continue des solutions d'un système. Nous avons ensuite prouvé que deux figures correspondant à des branches de même numéro dans deux arbres d'interprétation produits avec le même plan de construction mais des valeurs de paramètres différentes se ressemblaient au sens où nous l'avions défini. Cela nous a permis d'élaborer une technique, que nous avons appelée *gel d'une branche*, et qui est basée sur le fait que l'esquisse fait partie des solutions générées par le solveur lorsqu'on lance l'interprétation avec comme paramètres des valeurs lues sur cette esquisse. On peut donc repérer le numéro de la branche correspondant à l'esquisse dans l'arbre des solutions ainsi produit, et mémoriser ce numéro pour ensuite pouvoir sélectionner la branche de même numéro sur l'arbre des solutions produit cette fois avec les valeurs des cotes. Nous obtenons grâce à cette technique de gel d'une branche des résultats extrêmement satisfaisants dans tous les cas de systèmes de contraintes ne contenant que des contraintes dites *métriques*, c'est-à-dire des contraintes qui font intervenir des cotes (comme les contraintes de distances, d'angles, etc.).

Cependant, il se pose un problème lorsque des contraintes dites *booléennes*, qui affectent directement la topologie (comme les contraintes de tangence, d'égalité de points, etc.), sont présentes dans le système. En effet, dans ce cas l'esquisse ne fait pas partie de l'arbre des solutions généré par l'interprétation à partir de données lues sur l'esquisse. Une solution est alors de faire un *gel maximum*, c'est-à-dire de geler les parties de l'arbre liées à des contraintes métriques, et de laisser des embranchements là où il est impossible de décider. On obtient ainsi un petit sous-arbre qui doit être exploré.

Afin d'explorer ce sous-arbre, ou bien afin de permettre à l'utilisateur de visualiser les autres solutions qui avaient été mises de côté par la méthode de gel, nous avons ensuite souhaité améliorer l'efficacité du parcours des solutions, autrement dit des branches de l'arbre. Pour cela, nous avons tout d'abord défini une nouvelle façon d'effectuer l'interprétation, dénommée *interprétation pas à pas*, en marquant une pause à chaque embranchement correspondant à la définition d'un élément visible de la figure, pour laisser l'utilisateur décider du résultat qui doit être choisi, c'est-à-dire de la branche à suivre. Cette opération néces-

sitant une configuration particulière du plan de construction, qui doit vérifier un certain critère, nous avons également introduit un tri topologique spécifique, qui est effectué dès la lecture d'un fichier d'exemple.

Toutefois, cette interprétation pas à pas peut être assez longue si l'exemple est de taille importante. Nous avons donc défini une autre méthode plus intuitive qui permet de manipuler à la souris des solutions déjà calculées. Elle permet à l'utilisateur de sélectionner un ou plusieurs éléments qu'il juge mal positionnés sur la figure que propose le solveur, et de les déplacer sur un emplacement qui lui convient, tout en respectant toujours les contraintes de départ. Pour cela, le solveur propose les emplacements disponibles, parmi lesquels l'utilisateur peut choisir. Le déplacement d'un élément peut cependant induire le déplacement d'autres éléments qui lui seraient liés par une relation de dépendance dans le plan de construction. Cette opération est donc parfois délicate, d'autant plus que restant dépendant du plan de construction, il reste des cas où l'on ne peut pas forcément effectuer de déplacement sur l'élément souhaité.

Pour la réalisation, nous avons conçu un prototype nommé *SAMY*, basé sur le noyau de *YAMS*, et composé de modules correspondant chacun à une des méthodes que nous avons élaborées.

Perspectives

Jusqu'à présent, nous avons proposé des méthodes efficaces qui sont complémentaires, mais qui possèdent chacune des limitations plus ou moins importantes. Tout au long de cette thèse, nous avons déjà évoqué certaines perspectives qu'offre ce travail, tant pour remédier à ces restrictions que pour apporter d'autres méthodes.

Comme nous l'avons déjà évoqué, la méthode d'interprétation par gel d'une branche fonctionne très bien, et possède des performances extrêmement satisfaisantes si toutes les contraintes présentes dans le système sont métriques. Mais elle est mise en échec dès que des contraintes booléennes interviennent. Afin de pallier ce problème, une première alternative serait de corriger l'esquisse au vol afin qu'elle remplisse la condition qui fait défaut, et que l'on puisse ainsi appliquer le gel d'une branche. Une deuxième méthode consisterait à effectuer toujours un gel maximum, complété par divers outils automatiques de sélection de solutions. Ceux-ci pourraient être basés sur des heuristiques classiques telles que des comparaisons des propriétés géométriques de l'esquisse et des solutions, mais d'une façon plus adaptée à notre cas, rendant ainsi ces critères plus pertinents. Ils pourraient également être basés sur des contraintes d'inégalités implicites déduites de l'esquisse, comme par exemple l'appartenance ou non d'un élément à un demi-plan.

L'exploration de l'espace des solutions peut elle aussi être améliorée. Bien que nos deux nouvelles approches, par interprétation pas à pas et par manipulation directe de solutions, apportent déjà une contribution à son efficacité, celles-ci ont cependant leurs inconvénients. L'interprétation pas à pas, bien qu'allant plus directement au but qu'une simple énumération visuelle des solutions, reste tout de même assez fastidieuse pour des exemples de

grande ampleur. Même si les marqueurs inspirés des points d'arrêt des débogueurs offrent une aide significative, ceux-ci ne peuvent pas toujours être utilisés.

La manipulation de solutions est une opération moins lourde à utiliser, et plus intuitive. Elle permet à l'utilisateur de parcourir les solutions de façon transparente. Déplacer des éléments de la figure revient en effet à passer d'une branche de l'arbre des solutions à une autre, mais ce saut reste invisible pour lui. Nous avons cependant souligné les quelques limites de cette méthode, liés au plan de construction. Si la définition de l'élément que l'on veut déplacer se situe dans le début du plan, on ne pourra pas le déplacer. Si l'on souhaite déplacer un élément auquel d'autres sont liés par une relation de dépendance, alors on risque d'être obligé de déplacer avec lui une partie de la figure que l'on ne désirait pas modifier. Ainsi, quelques contraintes pèsent sur cette méthode. Pour remédier à la première, nous envisageons de calculer, lorsque le cas de figure se présente, l'association entre la modification du reste de la figure (c'est-à-dire la figure moins l'élément sélectionné) et le déplacement qui ensemble formeraient l'opération équivalente au déplacement de l'élément choisi. Il resterait encore à définir avec plus de précision la notion d'opération équivalente. Cette approche permettrait de résoudre le problème sans avoir recours à une deuxième résolution du système pour placer la définition de l'élément choisi plus loin dans le plan de construction.

D'autres sujets de recherche concernent par exemple l'amélioration de l'interface homme-machine. Plus celle-ci est conviviale et "intelligente", plus l'utilisation du prototype est facile pour l'utilisateur. Ils peuvent concerner également la réalisation de méthodes hybrides, utilisant certaines des notions et structures de *YAMS* qui n'ont pas encore été exploitées au maximum. Nous pouvons par exemple tirer parti des sous-figures pour effectuer des opérations différentes selon la sous-figure considérée. Il serait possible de geler une branche correspondant à une sous-figure, puis d'interpréter pas à pas la partie du plan de construction correspondant à une deuxième sous-figure, puis de manipuler des éléments d'une troisième sous-figure que l'on aurait isolée du reste de la figure, et ainsi de suite.

Toutes ces améliorations pourraient être expoitées dans le but d'animer des figures contraintes. En effet, l'approche symbolique qui a été suivie permet d'effectuer des calculs en temps réel lors de mises à jour des paramètres, ce qui offrirait la possibilité d'actualiser la visualisation d'une figure suffisament rapidement pour obtenir une animation fluide. Il reste à étudier les "sauts" d'une solution à une autre qui peuvent intervenir lorsque des valeurs particulières des paramètres sont atteintes. Si la figure animée passe par un cas dégénéré, le passage d'une branche de l'arbre des solutions à une autre devra être transparent pour l'utilisateur.

Un autre grand sujet d'étude reste encore l'intégration à *SAMY* d'un modeleur, de plus ou moins grande ampleur, qui nous affranchirait de l'utilisation préalable de l'ancienne version de *YAMS*, dont nous restons pour l'instant dépendants pour la saisie des esquisses cotées. *SAMY* pourrait ainsi être totalement indépendant. Ce modeleur pourrait, selon l'ampleur que l'on souhaite lui accorder, soit permettre simplement une saisie minimum d'entités géométriques et de leurs propriétés et contraintes à la façon de *Cabri-Géomètre*, soit intégrer entièrement la fonction de résolution de systèmes de contraintes de *YAMS* en

plus de la fonction de saisie de l'esquisse et de la cotation.

Nous pensons d'autre part que certains utilisateurs peuvent parfois être amenés à concevoir des pièces qui soient souvent du même type, ou qui comportent des éléments identiques. Il pourrait donc être judicieux de proposer à l'utilisateur de mémoriser des morceaux de constructions pré-résolus, éventuellement paramétrés, qu'il suffirait ensuite d'ajouter à une construction, sous une forme pouvant rappeler par exemple celle des "features" de la modélisation géométrique. Des recherches apparentées à cette idée sont déjà en cours dans notre équipe.

Pour finir, une des ambitions envisagées pour *YAMS* étant de passer à la construction d'objets en 3 dimensions, *SAMY* devra pouvoir s'adapter aussi. En réalité, *YAMS* permet déjà d'obtenir des objets 3D grâce à des fonctions qui permettent l'extrusion d'objets 2D. Mais il n'intègre pas encore le traitement de contraintes 3D. Il faudra donc prévoir une collaboration étroite dans ce sens.

Annexe A

Signature de l'univers géométrique

Symboles prédicatifs

`nom`	*profil* commentaire
`angarc`	$point \times point \times point \times angle \longrightarrow$ rayon angulaire
`angle`	$point \times point \times point \times point \times angle \longrightarrow$ angle entre deux bipoints
`distpp`	$point \times point \times longueur \longrightarrow$ distance entre deux points
`distv`	$point \times point \times longueur \longrightarrow$ distance verticale entre deux points
`disth`	$point \times point \times longueur \longrightarrow$ distance horizontale entre deux points
`distpl`	$point \times droite \times longueur \longrightarrow$ distance point-droite
`distlc`	$droite \times cercle \times longueur \longrightarrow$ distance droite-centre d'un cercle
`centre`	$cercle \times point \longrightarrow$ centre d'un cercle
`egal_p`	$point \times point \longrightarrow$ égalité de deux points
`egal_l`	$droite \times droite \longrightarrow$ égalité de deux droites
`egal_c`	$cercle \times cercle \longrightarrow$ égalité de deux cercles

`egdist` $point \times point \times point \times point \longrightarrow$
égalité de la longueur de deux bipoints

`fixpl` $point \times droite \times point \longrightarrow$
fixer un point et une droite comme sur l'esquisse

`fixorgpl` $point \times droite \times point \longrightarrow$
fixer un point à l'origine et une droite à l'axe Ox

`fixpp` $point \times point \times longueur \longrightarrow$
fixer deux points à une distance donnée

`onl` $point \times droite \longrightarrow$
incidence point-droite

`onc` $point \times cercle \longrightarrow$
incidence point-cercle

`radius` $cercle \times longueur \longrightarrow$
rayon d'un cercle

`tgcl` $cercle \times droite \longrightarrow$
tangence cercle-droite

`tgcc` $cercle \times cercle \longrightarrow$
tangence cercle-cercle

Symboles fonctionnels

`bissect` $angle \longrightarrow angle$
renvoie la moitié de la mesure d'un angle

`bissectdd` $droite \times droite \longrightarrow droite$
renvoie les bissectrices des deux angles formés par les deux droites

`centre_of` $cercle \longrightarrow point$
renvoie le centre d'un cercle

`eg_p` $point \longrightarrow point$
renvoie un point de mêmes coordonnées

`eg_l` $droite \longrightarrow droite$
renvoie une droite de même localisation

`eg_c` $cercle \longrightarrow cercle$
renvoie un cercle de même localisation

`fangarc` $point \times point \times angle \longrightarrow point$
connaissant le centre, une extrémité et la mesure de l'angle d'une arc de cercle, renvoie la deuxième extrémité

`fdist` $point \times point \longrightarrow longueur$
renvoie la distance entre deux points

`fdistpl` $point \times droite \longrightarrow longueur$
renvoie la distance entre un point et une droite

`fdistlc` $droite \times cercle \longrightarrow longueur$
renvoie la distance entre une droite et un cercle

`fangle` $point \times point \times point \times point \longrightarrow angle$
renvoie la mesure de l'angle entre deux bipoints

`inita` $réel \longrightarrow angle$
création d'une mesure d'angle

`initd` $point \times réel \longrightarrow droite$
localisation d'une droite par sa pente et un point incident

`initl` $réel \longrightarrow longueur$
création d'une longueur

`initp` $réel \times réel \longrightarrow point$
localisation d'un point par ses coordonnées

`initpp` $point \times longueur \longrightarrow point$
localisation d'un point à partir d'un autre, placé sur le même axe horizontal, et de la distance qui les sépare

`interll` $droite \times droite \longrightarrow point$
intersection droite-droite

`interlc` $droite \times cercle \longrightarrow point$
intersection droite-cercle

`intercc` $cercle \times cercle \longrightarrow point$
intersection cercle-cercle

`lpla` $point \times droite \times angle \longrightarrow droite$
renvoie la droite passant par un point donné et faisant un angle donné avec une autre droite

`lppa` $point \times point \times angle \longrightarrow droite$
renvoie la droite passant par un point donné et faisant un angle donné avec le bipoint, le point appartenant au bipoint

`lppa2` $point \times point \times point \times angle \longrightarrow droite$
renvoie la droite passant par un point donné et faisant un angle donné avec le bipoint, le point n'appartenant pas au bipoint

`ldl` $droite \times longueur \longrightarrow droite$
renvoie la droite parallèle à une droite donnée et étant à une certaine distance de cette droite

`lpp` $point \times point \longrightarrow droite$
renvoie la droite passant par deux points donnés

`lineh` $point \times longueur \longrightarrow droite$
renvoie une droite horizontale dont la distance à un point donné est connue

`linev` $point \times longueur \times droite$
renvoie une droite verticale dont la distance à un point donné est connue

`ltangent` $cercle \times cercle \longrightarrow droite$
calcul des tangentes

`make_dep_pp` $point \times point \times point \times point \longrightarrow deplacement$
calcul du déplacement permettant de passer d'un couple point-point à un autre

`make_dep_pl` $point \times droite \times point \times droite \longrightarrow deplacement$
idem avec un couple point-droite

`make_dep_pc` $point \times cercle \times point \times cercle \longrightarrow deplacement$
idem avec un couple point-cercle

`make_dep_ll` $droite \times droite \times droite \times droite \longrightarrow deplacement$
idem avec un couple droite-droite

`make_dep_cl` $cercle \times droite \times cercle \times droite \longrightarrow deplacement$
idem avec un couple cercle-droite

`make_dep_cc` $cercle \times cercle \times cercle \times cercle \longrightarrow deplacement$
idem avec un couple cercle-cercle

`mediatrice` $point \times point \longrightarrow droite$
médiatrice de deux points

`medradcir` $point \times point \times longueur \longrightarrow cercle$
calcul d'un cercle dont on connaît le rayon et deux points incidents

`mkcir` $point \times longueur \longrightarrow cercle$
cercle de centre et de rayon connus

`mkcir2` $point \times point \longrightarrow cercle$
cercle dont on connaît le centre et un point incident

`mkcir3` $point \times droite \longrightarrow cercle$
cercle dont on connaît le centre et une droite tangente

`mkcir4` $point \times cercle \longrightarrow cercle$
cercle dont on connaît le centre et un cercle tangent

`transfp` $point \times deplacement \longrightarrow point$
application d'un déplacement à un point

`transfl` $droite \times deplacement \longrightarrow droite$
application d'un déplacement à une droite

`transfc` $cercle \times deplacement \longrightarrow cercle$
application d'un déplacement à un cercle

Annexe B

Spécifications algébriques en *OBJ3*

```
th EQUIV is
sort Elt .
op _ ~ _ : Elt Elt -> Bool .
var x y z : Elt .
eq x ~ x = true .
eq x ~ y = y ~ x .
eq x ~ z = x ~ y and y ~ z .
endth
--- ---------------------------------------------------
obj LIST[X :: TRIV] is protecting INT .
sort List .
subsort Elt < List .
op emptylist : -> List .
op _"_ : List List -> List [assoc gather (e E) id: emptylist] .
op is-in-list : Elt List -> Bool . *** appartenance d'un element a la liste
op are-in-list : List List -> Bool . *** appartenance de plusieurs elts a la liste
op size : List -> Int . *** longueur de la liste
op elem : List Int -> Elt . *** nieme element
op is-sublist : List List -> Bool . *** l1 est-elle une sous-liste de l2?
op number : Elt List -> Int . *** numero de l'element dans la liste
op number : Elt List Int -> Int . *** outil
op del : Elt List -> List . *** suppression d'un element d'un ensemble
op diff : List List -> List . *** difference liste1-liste2
op empty : List -> Bool . *** la liste est-elle vide?
--- precondition(elem(l, n)) = n > 0 and n <= size(l) .
--- precondition(number(x,l)) = is-in-list(x,l) .
var x x1 x2 : Elt .
var l1 l2 : List .
var n : Int .
eq size(emptylist) = 0 .
eq size(x " l1) = 1 + size(l1) .
eq is-in-list(x,emptylist) = false .
eq is-in-list(x1,x2 " l1) = x1 == x2 or is-in-list(x1,l1) .
eq are-in-list(emptylist,l2) = true .
eq are-in-list(x1 " l1,l2) = is-in-list(x1,l2) and are-in-list(l1,l2) .
eq elem(x " l1,1) = x .
cq elem(x " l1,n) = elem(l1,n - 1) if n =/= 1 .
*** sous-liste non consecutive ***
eq is-sublist(emptylist,l1) = true .
eq is-sublist(l1,emptylist) = l1 == emptylist .
eq is-sublist(x1 " l1,x2 " l2) = (x1 == x2 and is-sublist(l1,l2))
  or (x1 =/= x2 and is-sublist(x1 " l1,l2)) .
eq number(x,x1 " l1,n) = if x == x1 then n else number(x,l1,n + 1) fi .
```

```
eq number(x,l1) = number(x,l1,1) .

eq del(x,emptylist) = emptylist .
eq del(x1,x2 " l1) = if x1 == x2 then del(x1,l1) else x2 " del(x1,l1) fi .

eq diff(l1,emptylist) = l1 .
eq diff(l1,x " l2) = diff(del(x,l1),l2) .

eq empty(l1) = if (l1 == emptylist) then true else false fi .

endo

--- ---------------------------------------------------

obj SET-EQ[X :: EQUIV] is protecting INT + LIST[X] .

sort Set .
subsort Elt < Set .

op emptyset : -> Set .
op _ _ : Elt Set -> Set [prec 33 gather (e E) id: emptyset] .
op is-in-set : Elt Set -> Bool . *** appartenance
op size : Set -> Int . *** longueur de l'ensemble
op is-subset : Set Set -> Bool . *** e1 est-il un sous-ens de e2?
op del : Elt Set -> Set . *** suppression d'un element d'un ensemble
op diff : Set Set -> Set . *** difference ens1-ens2
op list-to-set : List -> Set . *** transformation liste->ensemble
op set-to-list : Set -> List . *** transformation ensemble->liste
op union : Set Set -> Set . *** union de 2 ensembles
op union : Set Set Set -> Set . *** union de 3 ensembles
op inter : Set Set -> Set . *** intersection de 2 ensembles
op head : Set -> Elt . *** premier element de l'ensemble
op _===_ : Set Set -> Bool . *** egalite
op comp : Set Set -> Bool . *** outil

var x x1 x2 : Elt .
var e1 e2 e3 : Set .
var n : Int .
var l : List .

--- precondition(diff(e1,e2)) = size(e1) > size(e2) and is-subset(e2,e1) .
--- precondition(del(x,e)) = is-in-set(x,e) .

eq size(emptyset) = 0 .
eq size(x e1) = 1 + size(e1) .

eq is-in-set(x,emptyset) = false .
eq is-in-set(x1,x2 e1) = x1 ~ x2 or is-in-set(x1,e1) .

cq x e1 = e1 if is-in-set(x,e1) .

eq is-subset(emptyset,e1) = true .
eq is-subset(x e1,e2) = is-in-set(x,e2) and is-subset(e1,e2) .

eq del(x,emptyset) = emptyset .
eq del(x1,x2 e1) = if x1 == x2 then e1 else x2 del(x1,e1) fi .

eq diff(e1,emptyset) = e1 .
eq diff(e1,x e2) = diff(del(x,e1),e2) .

eq list-to-set(emptylist) = emptyset .
eq list-to-set(x " l) = if is-in-list(x,l) then list-to-set(l)
else x list-to-set(l) fi .
eq set-to-list(emptyset) = emptylist .
eq set-to-list(x e1) = x " set-to-list(e1) .

eq union(emptyset,e1) = e1 .
eq union(x1 e1,e2) = if is-in-set(x1,e2) then union(e1,e2) else x1 union(e1,e2) fi .
eq union(e1,e2,e3) = union(e1,union(e2,e3)) .

eq inter(emptyset,e1) = emptyset .
eq inter(x1 e1,e2) = if is-in-set(x1,e2) then  x1 inter(e1,e2) else inter(e1,e2) fi .

eq head(x e1) = x .

eq comp(emptyset,emptyset) = true .
eq comp(x1 e1,e2) = if is-in-set(x1,e2) then comp(e1,del(x1,e2)) else false fi .
eq e1 === e2 = if size(e1) == size(e2) then comp(e1,e2) else false fi .

endo

--- ---------------------------------------------------

obj SORT is protecting INT .

sort S . *** sorte geometrique

op num : -> S .
op long : -> S .
op point : -> S .
op line : -> S .
```

```
op circle : -> S .
op angle : -> S .
op depl : -> S .

op dof : S -> Int . *** degre de liberte
op dofc : S -> Int . *** degre de liberte dependant du repere
op dm : S -> Int . *** degre de liberte modep
op s-metric : S -> Bool .

var s : S .

eq dof(num) = 1 .
eq dof(long) = 1 .
eq dof(point) = 2 .
eq dof(line) = 2 .
eq dof(circle) = 3 .
eq dof(angle) = 1 .
eq dof(depl) = 3 .

eq dofc(num) = 0 .
eq dofc(long) = 0 .
eq dofc(point) = 2 .
eq dofc(line) = 2 .
eq dofc(circle) = 2 .
eq dofc(angle) = 0 .
eq dofc(depl) = 2 .

eq dm(s) = dof(s) - dofc(s) .

eq s-metric(s) = dofc(s) =/= 0 .

endo

--- ---------------------------------------------------

view SORTS from EQUIV to SORT is

sort Elt to S .

var s1 s2 : Elt .

op s1 ~ s2 to s1 == s2 .

endv

--- ---------------------------------------------------

make SSET is SET-EQ[SORTS]*(sort Set to Sset , sort List to Slist ,
    op emptyset to s-emptyset , op emptylist to s-emptylist)
endm

--- ---------------------------------------------------

obj SIGN is protecting INT + SSET .

sorts F P . *** symb. fonctionnel et symb. predicatif
sort Profil . *** profil d'un symbole fonctionnel ou predicatif

op _=>_M_R_ : Slist S Int Int -> Profil .  *** profil d'un symbole fonctionnel
op _=> : Slist -> Profil . *** profil d'un symbole predicatif
op arity : Profil -> Slist . *** arite d'un profil
op arity : F -> Slist . *** arite d'un symbole fonctionnel
op arity : P -> Slist . *** arite d'un symbole predicatif
op coarity : Profil -> S . *** coarite d'un profil
op coarity : F -> S . *** coarite d'un symbole fonctionnel
op multi : Profil -> Int . *** degre de multiplicite d'un profil
op multi : F -> Int . *** degre de multiplicite d'un symbole fonctionnel
op risk : Profil -> Int . *** risque d'echec d'un profil
op risk : F -> Int . *** risque d'echec d'un symbole fonctionnel
op profil : F -> Profil . *** profil d'un symbole fonctionnel
op profil : P -> Profil . *** profil d'un symbole predicatif

op initp : -> F . *** initialisation d'un point
op initl : -> F . *** initialisation d'une droite
op initd : -> F . *** initialisation d'une distance
op inita : -> F . *** initialisation d'un angle
op initpp : -> F . *** initialisation d'un point a partir d'un autre de
*** meme x a la distance d
op eg-p : -> F . *** renvoie un point de memes coordonnees
op eg-l : -> F . *** renvoie une droite de meme localisation
op eg-c : -> F . *** renvoie un cercle de meme localisation
op make-dep-pp : -> F . *** deplacement reliant 2 points
op make-dep-pl : -> F . *** deplacement reliant un point a une droite
op make-dep-pc : -> F . *** deplacement reliant un point a un cercle
op make-dep-ll : -> F . *** deplacement reliant 2 droites
op make-dep-cl : -> F . *** deplacement reliant un cercle a une droite
op make-dep-cc : -> F . *** deplacement reliant 2 cercles
```

```
op transfp : -> F . *** transforme d'un point par un deplacement
op transfl : -> F . *** transforme d'une droite par un deplacement
op transfc : -> F . *** transforme d'un cercle par un deplacement
op fdistpp : -> F . *** distance entre 2 points
op fdistpl : -> F . *** distance entre un point et une droite
op fdistlc : -> F . *** distance entre une droite et le centre d'un cercle
op fangle : -> F . *** angle forme par 4 points
op bissect : -> F . *** angle representant la moitie de l'angle donne
op fangarc : -> F . *** tranforme de p2 par rotation d'angle a de centre p1
op interll : -> F . *** intersection de 2 droites
op interlc : -> F . *** intersection d'un cercle et d'une droite
op intercc : -> F . *** intersection de deux cercles
op centre-of : -> F . *** centre d'un cercle
op lpla : -> F . *** droite qui passe par le 1er point et fait un angle
*** alpha avec la droite
op lppa : -> F . *** droite qui passe par le 1er point et fait un angle
*** alpha avec le bipoint
op ldl : -> F . *** droite dont on connait la distance avec une autre
*** droite
op lpp : -> F . *** droite qui passe par deux points
op mediatrice : -> F . *** mediatrice de 2 points
op lineh : -> F . *** droite horizontale a une distance d du point p
op linev : -> F . *** droite verticale a une distance d du point p
op ltangent : -> F . *** tangente a 2 cercles
op bissectdd : -> F . *** bissectrice de 2 droites
op mkcir : -> F . *** cercle de centre p et de rayon r
op mkcir2 : -> F . *** cercle de centre p1 et passant par p2
op mkcir3 : -> F . *** cercle de centre p et tangent a une droite d
op mkcir4 : -> F . *** cercle de centre p et tangent a un cercle c
op medradcir : -> F . *** cercle passant par p1 et p2 et de rayon r
op add-angle : -> F . *** somme de 2 angles
op min-angle : -> F . *** difference de 2 angles
op inv-angle : -> F . *** oppose d'1 angle
op supp-angle : -> F . *** supplementaire d'1 angle
op mod-angle : -> F . *** angle modulo pi
op eg-angle : -> F . *** angle egal
op add-lg : -> F . *** somme de 2 longueurs
op mul-lg : -> F . *** produit de 2 longueurs
op min-lg : -> F . *** difference de 2 longueurs
op div-lg : -> F . *** quotient de 2 longueurs
op eg-lg : -> F . *** longueur egale
op mil : -> F . *** milieu de 2 points

op distpp : -> P . *** distance entre 2 points
op distpl : -> P . *** distance entre un point et une droite
op distlc : -> P . *** distance entre une droite et un cercle
op distv : -> P . *** distance verticale entre 2 points
op disth : -> P . *** distance horizontale entre 2 points
op egdist : -> P . *** egalite entre les longueurs de 2 bipoints
op angle : -> P . *** angle entre 2 bipoints
op radius : -> P . *** rayon d'un cercle
op onl : -> P . *** incidence point-droite
op onc : -> P . *** incidence point-cercle
op fixpl : -> P . *** fixer un point a l'origine et une droite comme sur
*** l'esquisse
op fixplv : -> P . *** fixer un point a l'origine et une droite a l'axe Oy
op fixpp : -> P . *** fixer deux points a une distance donnee
op fixorgpl : -> P . *** fixer un point a l'origine et une droite a l'axe Ox
op tgcl : -> P . *** tangence cercle-droite
op tgcc : -> P . *** tangence cercle-cercle
op centre : -> P . *** centre d'un cercle
op angarc : -> P . *** angle d'un arc de cercle
op egal-p : -> P . *** egalite entre 2 points
op egal-l : -> P . *** egalite entre 2 droites
op egal-c : -> P . *** egalite entre 2 cercles

eq profil( initp ) = num " num => point M 1 R 0 .
eq profil( initl ) = point " num => line M 1 R 0 .
eq profil( initd ) = num => long M 1 R 0 .
eq profil( inita ) = num => angle M 1 R 0 .
eq profil( initpp ) = point " num => point M 1 R 0 .
eq profil( eg-p ) = point => point M 1 R 0 .
eq profil( eg-l ) = line => line M 1 R 0 .
eq profil( eg-c ) = circle => circle M 1 R 0 .
eq profil( make-dep-pp ) = point " point " point " point => depl M 1 R 0 .
eq profil( make-dep-pl ) = point " line " point " line => depl M 2 R 0 .
eq profil( make-dep-pc ) = point " circle " point " circle => depl M 1 R 0 .
```

```
eq profil( make-dep-ll ) = line " line " line " line => depl M 2 R 2 .
eq profil( make-dep-cl ) = circle " line " circle " line => depl M 2 R 0 .
eq profil( make-dep-cc ) = circle " circle " circle " circle => depl M 1 R 0 .
eq profil( transfp ) = point " depl => point M 1 R 0 .
eq profil( transfl ) = line " depl => line M 1 R 0 .
eq profil( transfc ) = circle " depl => circle M 1 R 0 .
eq profil( fdistpp ) = point " point => long M 1 R 0 .
eq profil( fdistpl ) = point " line => long M 1 R 0 .
eq profil( fdistlc ) = line " circle => long M 1 R 0 .
eq profil( fangle ) = point " point " point " point => angle M 1 R 0 .
eq profil( bissect ) = angle => angle M 1 R 0 .
eq profil( fangarc ) = point " point " angle => point M 1 R 0 .
eq profil( interll ) = line " line => point M 1 R 2 .
eq profil( interlc ) = line " circle => point M 2 R 4 .
eq profil( intercc ) = circle " circle => point M 2 R 0 .
eq profil( centre-of ) = circle => point M 1 R 0 .
eq profil( lpla ) = point " line " angle => line M 1 R 0 .
eq profil( lppa ) = point " point " angle => line M 1 R 0 .
eq profil( ldl ) = line " long => line M 2 R 0 .
eq profil( lpp ) = point " point => line M 1 R 1 .
eq profil( mediatrice ) = point " point => line M 1 R 1 .
eq profil( lineh ) = point " long => line M 2 R 0 .
eq profil( linev ) = point " long => line M 2 R 0 .
eq profil( ltangent ) = circle " circle => line M 4 R 3 .
eq profil( bissectdd ) = line " line => line M 2 R 0 .
eq profil( mkcir ) = point " long => circle M 1 R 0 .
eq profil( mkcir2 ) = point " point => circle M 1 R 0 .
eq profil( mkcir3 ) = point " line => circle M 1 R 0 .
eq profil( mkcir4 ) = point " circle => circle M 2 R 0 .
eq profil( medradcir ) = point " point " long => circle M 2 R 3 .
eq profil( add-angle ) = angle " angle => angle M 1 R 0 .
eq profil( min-angle ) = angle " angle => angle M 1 R 0 .
eq profil( inv-angle ) = angle => angle M 1 R 0 .
eq profil( supp-angle ) = angle => angle M 1 R 0 .
eq profil( mod-angle ) = angle => angle M 1 R 0 .
eq profil( eg-angle ) = angle => angle M 1 R 0 .
eq profil( add-lg ) = long " long => long M 1 R 0 .
eq profil( mul-lg ) = long " long => long M 1 R 0 .
eq profil( min-lg ) = long " long => long M 1 R 0 .
eq profil( div-lg ) = long " long => long M 1 R 0 .
eq profil( eg-lg ) = long => long M 1 R 0 .
eq profil( mil ) = point " point => point M 1 R 0 .

eq profil( distpp ) =  point " point " long => .
eq profil( distpl ) =  point " line " long => .
eq profil( distlc ) =  line " circle " long => .
eq profil( distv ) =  point " point " long => .
eq profil( disth ) =  point " point " long => .
eq profil( egdist ) =  point " point " point " point => .
eq profil( angle ) =  point " point " point " point " angle => .
eq profil( radius ) =  circle " long => .
eq profil( onl ) =  point " line => .
eq profil( onc ) =  point " circle => .
eq profil( fixpl ) =  point " line " point => .
eq profil( fixplv ) =  point " point " line => .
eq profil( fixpp ) =  point " point " long => .
eq profil( fixorgpl ) =  point " line " point => .
eq profil( tgcl ) =  circle " line => .
eq profil( tgcc ) =  circle " circle => .
eq profil( centre ) =  circle " point => .
eq profil( angarc ) =  point " point " point " angle => .
eq profil( egal-p ) =  point " point => .
eq profil( egal-l ) =  line " line => .
eq profil( egal-c ) =  circle " circle => .

var sl : Slist .
var s : S .
var f : F .
var p : P .
var x1 x2 : Int .

eq arity(sl => s M x1 R x2) = sl .
eq coarity(sl => s M x1 R x2) = s .
eq multi(sl => s M x1 R x2) = x1 .
```

```
eq risk(sl => s M x1 R x2) = x2 .
eq arity(f) = arity(profil(f)) .
eq coarity(f) = coarity(profil(f)) .
eq arity(p) = coarity(profil(p)) .
eq multi(f) = multi(profil(f)) .
eq risk(f) = risk(profil(f)) .

endo

--- ---------------------------------------------------

obj IDENT is protecting QID + SORT .

sorts Index I . *** I: identificateurs indices
subsort Int < Index .

op _#_._ : S Id Index -> I . *** constructeur
op sort-id : I -> S . *** sorte geometr. de l'identificateur
op index-id : I -> Index . *** indice de l'identificateur
op name-id : I -> Id . *** nom de l'identificateur
op dof : I -> Int . *** degre de liberte de l'id. ind.

var i : I .
var s : S .
var n : Index .
var a : Id .

eq sort-id(s # a . n) = s .
eq index-id(s # a . n) = n .
eq name-id(s # a . n) = a .

eq dof(i) = dof(sort-id(i)) .

endo

--- ---------------------------------------------------

view IDENTS from EQUIV to IDENT is

sort Elt to I .

var i1 i2 : Elt .

op i1 ~ i2 to name-id(i1) == name-id(i2) .

endv

--- ---------------------------------------------------

make ISET is SET-EQ[IDENTS]*(sort Set to Iset , sort List to Ilist ,
     op emptyset to i-emptyset , op emptylist to i-emptylist)
endm

--- ---------------------------------------------------

obj TERM is protecting ISET + SIGN .

sorts Term Termf Termp .
subsorts Termf Termp I < Term .

op _[_] : F Ilist -> Termf .
op _[_] : P Ilist -> Termp .
op arity : Term -> Slist .
op coarity : Termf -> S .
op args-term : Term -> Ilist .
op F-term : Termf -> F . *** donne le symbole fonctionnel du terme
op P-term : Termp -> P . *** donne le symbole predicatif du terme
op is-F : Term -> Bool . *** le terme est-il fonctionnel?
op is-P : Term -> Bool . *** le terme est-il predicatif?
op is-I : Term -> Bool . *** le terme est-il un identificateur?
op appear-in-term : I Term -> Bool . *** l'id. apparait-il dans le terme?
op wf : F Ilist -> Bool . *** le terme fonctionnel est-il bien forme?
op wf : P Ilist -> Bool . *** le terme predicatif est-il bien forme?
op corresp : Slist Ilist -> Bool . *** les identificateurs (arguments) sont-ils
*** de la bonne sorte?

var f : F .
var p : P .
var li : Ilist .
var i i1 i2 : I .
var t : Term .
var ls : Slist .
var s : S .

--- precondition(f[li]) = wf(f,li) .
--- precondition(p[li]) = wf(p,li) .

eq arity(f[li]) = arity (f) .
eq coarity(f[li]) = coarity(f) .
eq arity(p[li]) = arity (p) .

eq args-term(f[li]) = li .
```

```
eq args-term(p[li]) = li .
eq F-term(f[li]) = f .
eq P-term(p[li]) = p .
eq is-F(f[li])= true .
eq is-F(p[li]) = false .
eq is-F(i) = false .
eq is-P(p[li]) = true .
eq is-P(f[li]) = false .
eq is-P(i) = false .
eq is-I(i) = true .
eq is-I(f[li]) = false .
eq is-I(p[li]) = false .

eq appear-in-term(i1,i2) =  i1 == i2 .
eq appear-in-term(i,t) = is-in-list(i,args-term(t)) .

eq wf(f,li) = corresp(arity(f),li) .
eq wf(p,li) = corresp(arity(p),li) .
eq corresp(s-emptylist,i-emptylist) = true .
eq corresp(s " ls,i " li) = (s == sort-id(i)) and corresp(ls,li) .

endo

--- ---------------------------------------------------

obj DEF is protecting TERM .

sort Def .

op _:= param : I -> Def .
op _:=_ : I Termf -> Def .
op args-term : Def -> Ilist . *** liste des arguments
op appear-in-def : I Def -> Bool . *** l'id. apparait-il dans les arguments?
op ident : Def -> I . *** donne l'identificateur defini
op term : Def -> Termf . *** donne le terme fonctionnel d'un id.
op multi : Def -> Int . *** degre de multiplicite du symbole fonctionnel
op risk : Def -> Int . *** risque du symbole fonctionnel
op is-param : Def -> Bool . *** la definition est-elle un parametre?

var i i1 i2 : I .
var t : Termf .

--- precondition(i := t) = sort-id(i) == coarity(t) and not appear-in-term(i,t) .
--- precondition(term(d)) = not (is-param(d)) .

eq args-term(i := param) = i-emptylist .
eq args-term(i := t) = args-term(t) .
eq appear-in-def(i1,i2 := param) = false .
eq appear-in-def(i1,i2 := t) = appear-in-term(i1,t) .
eq ident(i := param) = i .
eq ident(i := t) = i .
eq term(i := t) = t .
eq multi(i := param) = 1 .
eq multi(i := t) = multi(F-term(t)) .
eq risk(i := param) = 0 .
eq risk(i := t) = risk(F-term(t)) .
eq is-param(i := param) = true .
eq is-param(i := t) = false .

endo

--- ---------------------------------------------------

view DEFS from EQUIV to DEF is

sort Elt to Def .

var d1 d2 : Elt .

op d1 ~ d2 to ident(d1) == ident(d2) .

endv

--- ---------------------------------------------------

make DEF-SET is SET-EQ[DEFS]*(sort Set to Def-set , sort List to Def-list ,
      op emptyset to d-emptyset , op emptylist to d-emptylist)
endm

--- ---------------------------------------------------

obj NADS is protecting DEF-SET + ISET .  *** Non Ambiguous Definition Set

op id-set : Def-set -> Iset . *** ensemble des id. deja definis
op args-set : Def-set -> Iset . *** ensemble des id qui sont arguments
op pred : I Def-set -> Iset . *** ensemble des predecesseurs d'un id.
op succ : I Def-set -> Iset . *** ensemble des ids. successeurs d'un id.
```

```
op succ : Def Def-set -> Def-set . *** ensemble des defs. successeurs d'une def.
op sources : Def-set -> Iset . *** ensemble des sources
op sources : Def-set Iset -> Iset . *** outil
op parameters : Def-set -> Iset . *** ensemble des parametres
op ncdef : Def-set -> Iset . *** ensemble des id. dont les args. ne sont
*** pas tous definis
op ncdef : Def-set Def-set Iset -> Iset . *** outil
op wells : Def-set -> Iset . *** ensemble des puits
op connected : Def-set -> Bool . *** connexite
op search : Iset Def-set Iset -> Iset . *** outil
op has-circuit : Def-set -> Bool . *** y a-t-il un circuit dans l'ensemble de def?
op depth : Iset Def-set Iset -> Bool . *** outil
op is-triang : Def-set -> Bool . *** l'ensemble est-il triangulaire?
op def : I Def-set -> Def . *** definition correspondant a l'identificateur
op defs : Iset Def-set -> Def-set . *** definitions corresp. aux identificateurs

var i : I .
var d d1 : Def .
var ds cur-ds : Def-set .
var is cur-is : Iset .

--- precondition(is-triang(ds)) = not (has-circuit(ds)) .
--- precondition(tri-top(ds)) = is-triang(ds) .

*** recherches d'identificateurs dans l'ensemble de definition ***

eq id-set(d-emptyset) = i-emptyset .
eq id-set(d ds) = ident(d) id-set(ds) .

eq args-set(d-emptyset) = i-emptyset .
eq args-set(d ds) = union(list-to-set(args-term(d)),args-set(ds)) .

eq pred(i,d-emptyset) = i-emptyset .
eq pred(i,d ds) = if i == ident(d) then list-to-set(args-term(d)) else pred(i,ds) fi .

eq succ(i,d-emptyset) = i-emptyset .
eq succ(i,d ds) = if is-param(d) then succ(i,ds)
  else if appear-in-def(i,d) then ident(d) succ(i,ds)
       else succ(i,ds) fi fi .

eq succ(d,d-emptyset) = d-emptyset .
eq succ(d,d1 ds) = if is-param(d1) then succ(d,ds)
   else if appear-in-def(ident(d),d1) then d1 succ(d,ds)
else succ(d,ds) fi fi .

eq sources(d-emptyset,is) = i-emptyset .
eq sources(d ds,is) = if inter(list-to-set(args-term(d)),is) === i-emptyset
      then ident(d) sources(ds,is)
      else sources(ds,is) fi .
eq sources(ds) = if ds === d-emptyset then i-emptyset else sources(ds,id-set(ds)) fi .

eq parameters(d-emptyset) = i-emptyset .
eq parameters(d ds) = if is-param(d) then ident(d) parameters(ds)
      else parameters(ds) fi .

eq ncdef(d-emptyset,ds,is) = i-emptyset .
eq ncdef(d cur-ds,ds,is) = if not (is-subset(pred(ident(d),ds),is))
   then ident(d) ncdef(cur-ds,ds,is)
   else ncdef(cur-ds,ds,is) fi .
eq ncdef(d-emptyset) = i-emptyset .
eq ncdef(ds) = ncdef(ds,ds,id-set(ds)) .

eq wells(ds) = diff(id-set(ds),args-set(ds)) .

*** connexite ***

eq search(i-emptyset,ds,is) = is .
eq search(i cur-is,ds,is) = search(union(cur-is,diff(succ(i,ds),inter(succ(i,ds),is)),
  diff(pred(i,ds),inter(pred(i,ds),is)))
  ,ds,i is) .
eq connected(ds) = search(head(sources(ds)),ds,i-emptyset) == id-set(ds) .

*** presence de circuits ***

eq depth(i-emptyset,ds,is) = true .
eq depth(i cur-is,ds,is) = if succ(i,ds) == i-emptyset then true
    else if inter(succ(i,ds),is) =/= i-emptyset then false
 else depth(cur-is,ds,is)
 and depth(succ(i,ds),ds,i is) fi fi .
eq has-circuit(d-emptyset) = false .
eq has-circuit(ds) = depth(sources(ds),ds,i-emptyset) .

*** triangularite ***

eq is-triang(ds) = (sources(ds) == parameters(ds))
       and (not (has-circuit(ds)))
```

```
       and (ncdef(ds) == i-emptyset) .
eq def(i,d ds) = if i == ident(d) then d else def(i,ds) fi .
eq defs(i-emptyset,ds) = d-emptyset .
eq defs(i is,ds) = def(i,ds) defs(is,ds) .

endo

--- ---------------------------------------------------

th CHOICE is protecting NADS .

op choice : Def-set Ilist -> Def .

endth

--- ---------------------------------------------------

obj TOPO-SORT[X :: CHOICE] is protecting NADS .

*** tri topologique ***

op topo-sort : Def-set Def-set Ilist Def -> Def-list .
op topo-sort : Def-set Def-set Ilist -> Def-list .
op topo-sort : Def-set -> Def-list .

var ds cur-ds : Def-set .
var il : Ilist .
var ch : Def .

eq topo-sort(cur-ds,ds,il,ch) = ch " topo-sort(union(del(ch,cur-ds),succ(ch,ds)),
  ds,ident(ch) " il) .
eq topo-sort(d-emptyset,ds,il) = d-emptylist .
eq topo-sort(cur-ds,ds,il) = topo-sort(cur-ds,ds,il,choice(cur-ds,il)) .
eq topo-sort(ds) = topo-sort(defs(sources(ds),ds),ds,i-emptylist) .

endo

--- ---------------------------------------------------

obj RISK-CHOICE is protecting NADS .

op risk-choice : Def-set Ilist -> Def .

var d1 d2 : Def .
var ds : Def-set .
var il : Ilist .

eq risk-choice(d1,il) = d1 .
eq risk-choice(d1 d2 ds,il) = if ((is-param(d1) or are-in-list(args-term(d1),il))
     and (risk(d1) <= risk(d2)))
      then risk-choice(d1 ds,il)
      else risk-choice(d2 ds,il)
      fi .

endo

--- ---------------------------------------------------

make RISK-TOPO-SORT is TOPO-SORT[view from CHOICE to RISK-CHOICE is
op choice to risk-choice . endv]
endm

--- ---------------------------------------------------

obj C-FUN is protecting RISK-TOPO-SORT .

sort Fun .

op _ _ : Ilist Def-list -> Fun .
op c-fun : Def-set -> Fun .
op parameters : Fun -> Ilist .
op definitions : Fun -> Def-list .

var ds : Def-set .
var dl : Def-list .
var il : Ilist .
var f : Fun .

eq c-fun(ds) = set-to-list(parameters(ds))
       diff(topo-sort(ds),set-to-list(defs(parameters(ds),ds))) .
eq parameters(il dl) = il .
eq definitions(il dl) = dl .

endo

--- ---------------------------------------------------

view INTS from EQUIV to INT is

sort Elt to Int .

var x1 x2 : Elt .

op x1 ~ x2 to x1 == x2 .
```

```
endv

--- ---------------------------------------------------
make NUMLIST is LIST[FLOAT]*(sort List to Numlist , op emptylist to n-emptylist)
endm
--- ---------------------------------------------------
make BROTHERS is LIST[INT]*(sort List to Brothers , op emptylist to b-emptylist ,
    op _"_ to _._)
endm
--- ---------------------------------------------------
make COORD is LIST[FLOAT]*(sort List to Coord , op emptylist to c-emptylist ,
   op _"_ to _;_ , op size to size1 , op elem to elem1)
endm
--- ---------------------------------------------------
make COORDLIST is LIST[view to COORD is sort Elt to Coord . endv]
*(sort List to Coordlist , op emptylist to cl-emptylist , op _"_ to _&_)
endm
--- ---------------------------------------------------

obj INTERP is protecting C-FUN + NUMLIST + BROTHERS + COORDLIST .

sorts Floor Code . *** etage de l'arbre

protecting LIST[Floor]*(sort List to IT , op emptylist to emptytree) .

op fail : -> Coord . *** detection d'erreur
op _|_ : Brothers Coordlist -> Code . *** code d'un etage
op _|_ : I Code -> Floor . *** etage de l'arbre
op emptyfloor : -> Floor . *** etage vide
op emptycode : -> Code . *** code vide
op id : Floor -> I . *** donne l'identificateur affecte

op interp : Fun Numlist -> IT . *** interprete une fun
op interp : Def-list IT -> IT . *** interprete une Def-list
op interp : Term IT -> Code . *** interprete un terme
op interp : F Coordlist -> Coordlist . *** interprete un symbole fonctionnel
op nodes : F Ilist IT Int Code -> Code .

op root : Ilist Numlist -> IT .
op countlastnodes : IT -> Int .
op add : Brothers -> Int .
op extract : Ilist IT -> Coordlist .
op extract : Ilist IT IT -> Coordlist .
op branch : IT Int -> IT .
op father : Brothers Int Int -> Int .
op countfails : Coordlist Int -> Int .

op angle : Float Float Float Float -> Float . *** + ou - angle entre les vecteurs
op projx : Float Float Float Float Float -> Float . *** x du proj de p1 sur D
op projy : Float Float Float Float Float -> Float . *** y du proj de p1 sur D
op distpp : Float Float Float Float -> Float . *** distance entre 2 points
op distpl : Float Float Float Float Float -> Float . *** + ou - distance point-droite
op rt-ext : Float Float Float Float Float Float -> Float . *** rayon pour les tgtes ext
op rt-int : Float Float Float Float Float Float -> Float . *** rayon pour les tgtes int

var fct : Fun .
var d : Def .
var dl : Def-list .
var i i1 i2 : I .
var il : Ilist .
var f : F .
var x y k a b c w : Float .
var x1 y1 x2 y2 x3 y3 x4 y4 : Float .
var k1 k2 k3 k4 : Float .
var a1 b1 c1 a2 b2 c2 a3 b3 c3 a4 b4 c4 : Float .
var w1 w2 : Float .
var n n1 n2 n3 : Int .
var nl : Numlist .
var it it1 it2 : IT .
var code : Code .
var br : Brothers .
var cd : Coord .
var cl : Coordlist .

*** etage vide ***

eq emptycode = b-emptylist | cl-emptylist .
eq size(fail) = 0 .

*** interpretation des c-fun, des def-set, et des termes ***

eq interp(fct,nl) = interp(definitions(fct),root(parameters(fct),nl)) .
eq interp(d-emptylist,it) = it .
eq interp(d " dl,it) = interp(dl,it " (ident(d) | interp(term(d),it)) ) .
eq interp(f[il],it) = nodes(f,il,it,countlastnodes(it),emptycode) .
eq nodes(f,il,it,n,br | cl) = if n == 0
      then br | cl
      else if branch(it,n) == emptytree
   then nodes(f,il,it,n - 1,br | cl)
```

```
   else nodes(f,il,it,n - 1, size(interp(f,
extract(il,branch(it,n)))) . br | interp(f,
extract(il,branch(it,n))) & cl)
   fi
     fi .
eq root(i-emptylist,n-emptylist) = emptytree .
eq root(i " il,x " nl) = (i | (1 | x)) " root(il,nl) .
eq add(b-emptylist) = 0 .
eq add(n . br) = n + add(br) .
eq countlastnodes(it " i | (br | cl)) = add(br) .
eq father(n1 . br,n2,n3) = if n2 <= n1 then n3 else father(br,n2 - n1,n3 + 1) fi .
eq countfails(cl-emptylist,n) = 0 .
eq countfails(cd & cl,n) = if cd == fail then 1 + countfails(cl,n)
   else if n == 1 then 0
else countfails(cl,n - 1)
fi
   fi .
eq branch(emptytree,n) = emptytree .
eq branch(it " i | (br | cl),n) = if cl == fail then emptytree
  else branch(it,father(br,n + countfails(cl,n),1))
" i | (1 | elem(cl,n + countfails(cl,n))) fi .
eq extract(il,it) = extract(il,it,it) .
eq extract(i-emptylist,it1,it2) = cl-emptylist .
eq extract(i1 " il,it1 " (i2 | (br | cd)),it2) = if i1 == i2
       then cd & extract(il,it2,it2)
       else extract(i1 " il,it1,it2)
       fi .
*** outils numeriques ***
eq angle(x1,y1,x2,y2) =
if x1 * x2 + y1 * y2 == 0
then if x1 * y2 - y1 * x2 > 0
     then pi / 2
     else - pi / 2 fi
else atan((x1 * y2 - y1 * x2) / (x1 * x2 + y1 * y2)) fi .
eq projx(x,y,a,b,c) = (b * b * x - a * b * y - a * c) / (a * a + b * b) .
eq projy(x,y,a,b,c) = if b == 0 then y
else - (a * ((b * b * x - a * b * y - a * c) / (a * a + b * b)) + c) / b fi .
eq distpp(x1,y1,x2,y2) = sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1)) .
eq distpl(x,y,a,b,c) = (a * x + b * y + c) / sqrt(a * a + b * b) .
eq rt-ext(x1,y1,k1,x2,y2,k2) = sqrt(distpp(x1,y1,x2,y2) * distpp(x1,y1,x2,y2)
     + (2 * k1 * k2 - k2 * k2)) .
eq rt-int(x1,y1,k1,x2,y2,k2) = sqrt(distpp(x1,y1,x2,y2) * distpp(x1,y1,x2,y2)
     - (2 * k1 * k2 + k2 * k2)) .

*** interpretation des symboles fonctionnels dont multi == 1 ***
eq interp(initp,x & y) = x ; y .
eq interp(initl,(x ; y) & w) = sin(w) ; (- cos(w)) ; (y * cos(w) - x * sin(w)) .
eq interp(initd,k) = k .
eq interp(inita,w) = w .
eq interp(initpp,(x ; y) & k) = (x + k) ; y .
eq interp(eg-p,x ; y) = x ; y .
eq interp(eg-l,a ; b ; c) = a ; b ; c .
eq interp(eg-c,x ; y ; k) = x ; y ; k .
eq interp(make-dep-pp,(x1 ; y1) & (x2 ; y2) & (x3 ; y3) & (x4 ; y4)) =
if (x2 - x1) * (y4 - y3) - (y2 - y1) * (x4 - x3) > 0
then (x1 ; y1 ; x3 ; y3 ; angle(x2 - x1,y2 - y1,x4 - x3,y4 - y3))
else (x1 ; y1 ; x3 ; y3 ; - angle(x2 - x1,y2 - y1,x4 - x3,y4 - y3)) fi .
eq interp(make-dep-pc,(x1 ; y1) & (x2 ; y2 ; k1) &
      (x3 ; y3) & (x4 ; y4 ; k2)) =
interp(make-dep-pp,(x1 ; y1) & (x2 ; y2) & (x3 ; y3) & (x4 ; y4)) .
eq interp(make-dep-cc,(x1 ; y1 ; k1) & (x2 ; y2 ; k2) &
      (x3 ; y3 ; k3) & (x4 ; y4 ; k4)) =
interp(make-dep-pp,(x1 ; y1) & (x2 ; y2) & (x3 ; y3) & (x4 ; y4)) .
eq interp(transfp,(x ; y) & (x1 ; y1 ; x2 ; y2 ; w)) =
x2 + ((x - x1) * cos(w) - (y - y1) * sin(w)) ;
y2 + ((x - x1) * sin(w) + (y - y1) * cos(w)) .
eq interp(transfl,(a ; b ; c) & (x1 ; y1 ; x2 ; y2 ; w)) =
a * cos(w) - b * sin(w) ;
b * cos(w) + a * sin(w) ;
c + (a * x1
  + (b * y1
  - ((a * cos(w) - b * sin(w)) * x2
```

```
  + ((b * cos(w) + a * sin(w)) * y2)))) .
eq interp(transfc,(x ; y ; k) & (x1 ; y1 ; x2 ; y2 ; w)) =
(x - x1) * cos(w) - ((y - y1) * sin(w) + x2) ;
(x - x1) * sin(w) + ((y - y1) * cos(w) + y2) ; k .
eq interp(fdistpp,(x1 ; y1) & (x2 ; y2)) = distpp(x1,y1,x2,y2) .
eq interp(fdistpl,(x ; y) & (a ; b ; c)) = abs(distpl(x,y,a,b,c)) .
eq interp(fdistlc,(a ; b ; c) & (x ; y ; k)) = interp(fdistpl,(x ; y) & (a ; b ; c)) .
eq interp(fangle,(x1 ; y1) & (x2 ; y2) & (x3 ; y3) & (x4 ; y4)) =
angle(x2 - x1,y2 - y1,x4 - x3,y4 - y3) .
eq interp(bissect,w) = w / 2 .
eq interp(fangarc,(x1 ; y1) & (x2 ; y2) & w) =
x2 + ((x1 - x2) * cos(w) - (y1 - y2) * sin(w)) ;
y2 + ((x1 - x2) * sin(w) + (y1 - y2) * cos(w)) .
eq interp(interll,(a1 ; b1 ; c1) & (a2 ; b2 ; c2)) =
if abs(a2 * b1 - a1 * b2) < 0.005
then fail
else
if a1 =/= 0
then (- (b1 * ((c1 * a2 - c2 * a1) / (a1 * b2 - a2 * b1)) + c1) / a1 ;
     (c1 * a2 - c2 * a1) / (a1 * b2 - a2 * b1))
else (- (b2 * ((c1 * a2 - c2 * a1) / (a1 * b2 - a2 * b1)) + c2) / a2 ;
     (c1 * a2 - c2 * a1) / (a1 * b2 - a2 * b1)) fi fi .
eq interp(centre-of,(x ; y ; k)) = (x ; y) .
eq interp(lpla,(x ; y) & (a ; b ; c) & w) =
(c / a) * sin(w) - (c / b) * cos(w) ;
- (c / a) * cos(w) - (c / b) * sin(w) ;
y * ((c / a) * cos(w) + (c / b) * sin(w))
      - x * ((c / a) * sin(w) - (c / b) * cos(w)) .
eq interp(lppa,(x1 ; y1) & (x2 ; y2) & w) =
(x2 - x1) * sin(w) + (y2 - y1) * cos(w) ;
- (x2 - x1) * cos(w) + (y2 - y1) * sin(w) ;
y1 * ((x2 - x1) * cos(w) - (y2 - y1) * sin(w))
      - x1 * ((x2 - x1) * sin(w) + (y2 - y1) * cos(w)) .
eq interp(lpp,(x1 ; y1) & (x2 ; y2)) =
if x1 == x2 and y1 == y2
then fail
else
y2 - y1 ;
x1 - x2 ;
y1 * (x2 - x1) - x1 * (y2 - y1)
fi .
eq interp(mediatrice,(x1 ; y1) & (x2 ; y2)) =
if x1 == x2 and y1 == y2
then fail
else
interp(lpla,(x1 + (x2 - x1) / 2 ; y1 + (y2 - y1) / 2)
  & interp(lpp,(x1 ; y1) & (x2 ; y2))
  & (pi / 2))
fi .
eq interp(mkcir,(x ; y) & k) = (x ; y ; k) .
eq interp(mkcir2,(x1 ; y1) & (x2 ; y2)) = x1 ; y1 ; distpp(x1,y1,x2,y2) .
eq interp(mkcir3,(x ; y) & (a ; b ; c)) = x ; y ;
interp(fdistpl,(x ; y) & (a ; b ; c)) .
eq interp(add-angle,w1 & w2) = if ((w1 + w2) >= 1.0 or (w1 + w2) <= (- 1.0))
       then (w1 + w2) rem (2 * pi)
       else (w1 + w2) fi .
eq interp(min-angle,w1 & w2) = interp(add-angle,w1 & (- w2)) .
eq interp(inv-angle,w) = - w .
eq interp(supp-angle,w) = 2 * pi - w .
eq interp(mod-angle,w) = w rem pi .
eq interp(eg-angle,w) = w .
eq interp(add-lg,k1 & k2) = k1 + k2 .
eq interp(min-lg,k1 & k2) = k1 - k2 .
eq interp(mul-lg,k1 & k2) = k1 * k2 .
eq interp(div-lg,k1 & k2) = k1 / k2 .
eq interp(eg-lg,k) = k .
eq interp(mil,(x1 ; y1) & (x2 ; y2)) = (((x1 + x2) / 2) ; ((y1 + y2) / 2)) .
*** interpretation des symboles fonctionnels dont multi =/= 1 ***
eq interp(make-dep-pl,(x1 ; y1) & (a1 ; b1 ; c1) & (x2 ; y2) & (a2 ; b2 ; c2)) =
  if (abs(a1 * projx(x1,y1,a1,b1,c1) + b1 * projy(x1,y1,a1,b1,c1) + c1) < 0.05)
  and (abs(a1 * projx(x1,y1,a1,b1,c1) + b1 * projy(x1,y1,a1,b1,c1) + c1) > 0.05)
  then ((x1 ; y1 ; x2 ; y2
 ; angle(projx(x1,y1,a1,b1,c1) - x1,projy(x1,y1,a1,b1,c1) - y1
,projx(x2,y2,a2,b2,c2) - x2,projy(x2,y2,a2,b2,c2) - y2))
```

```
      & (x1 ; y1 ; x2 ; y2
 ; - angle(projx(x1,y1,a1,b1,c1) - x1,projy(x1,y1,a1,b1,c1) - y1
  ,projx(x2,y2,a2,b2,c2) - x2,projy(x2,y2,a2,b2,c2) - y2)))
  else (x1 ; y1 ; x2 ; y2
; angle(projx(x1,y1,a1,b1,c1) - x1,projy(x1,y1,a1,b1,c1) - y1
       ,projx(x2,y2,a2,b2,c2) - x2,projy(x2,y2,a2,b2,c2) - y2)) fi .
eq interp(make-dep-ll,(a1 ; b1 ; c1) & (a2 ; b2 ; c2)
    & (a3 ; b3 ; c3) & (a4 ; b4 ; c4)) =
if a2 * b1 - a1 * b2 == 0
then fail
else
(elem(interp(interll,(a1 ; b1 ; c1) & (a2 ; b2 ; c2)),1) ;
 elem(interp(interll,(a1 ; b1 ; c1) & (a2 ; b2 ; c2)),2) ;
 elem(interp(interll,(a3 ; b3 ; c3) & (a4 ; b4 ; c4)),1) ;
 elem(interp(interll,(a3 ; b3 ; c3) & (a4 ; b4 ; c4)),2) ;
 angle(- b1,a1,- b3,a3))
      & (elem(interp(interll,(a1 ; b1 ; c1) & (a2 ; b2 ; c2)),1) ;
 elem(interp(interll,(a1 ; b1 ; c1) & (a2 ; b2 ; c2)),2) ;
 elem(interp(interll,(a3 ; b3 ; c3) & (a4 ; b4 ; c4)),1) ;
 elem(interp(interll,(a3 ; b3 ; c3) & (a4 ; b4 ; c4)),2) ;
 angle(- b1,a1,- b3,a3) - pi)
fi .
eq interp(make-dep-cl,(x1 ; y1 ; k1) & (a1 ; b1 ; c1)
    & (x2 ; y2 ; k2) & (a2 ; b2 ; c2)) =
interp(make-dep-pl,(x1 ; y1) & (a1 ; b1 ; c1) & (x2 ; y2) & (a2 ; b2 ; c2)) .
eq interp(interlc,(a ; b ; c) & (x ; y ; k)) =
if abs(distpl(x,y,a,b,c)) > k
then fail
else
    if (distpl(x,y,a,b,c) - k < 0.05) and (distpl(x,y,a,b,c) - k > - 0.05)
    then (  x - (distpl(x,y,a,b,c) * a +
    sqrt(- distpl(x,y,a,b,c) * distpl(x,y,a,b,c) + k * k) * b)
           / sqrt(a * a + b * b)
  ; y - (distpl(x,y,a,b,c) * b -
    sqrt(- distpl(x,y,a,b,c) * distpl(x,y,a,b,c) + k * k) * a)
           / sqrt(a * a + b * b) )
    else
(  x - (distpl(x,y,a,b,c) * a +
sqrt(- distpl(x,y,a,b,c) * distpl(x,y,a,b,c) + k * k) * b)
       / sqrt(a * a + b * b)
 ; y - (distpl(x,y,a,b,c) * b -
sqrt(- distpl(x,y,a,b,c) * distpl(x,y,a,b,c) + k * k) * a)
       / sqrt(a * a + b * b) )
      & (  x - (distpl(x,y,a,b,c) * a -
sqrt(- distpl(x,y,a,b,c) * distpl(x,y,a,b,c) + k * k) * b)
       / sqrt(a * a + b * b)
 ; y - (distpl(x,y,a,b,c) * b +
sqrt(- distpl(x,y,a,b,c) * distpl(x,y,a,b,c) + k * k) * a)
       / sqrt(a * a + b * b) )
    fi
fi .
eq interp(intercc,(x1 ; y1 ; k1) & (x2 ; y2 ; k2)) =
interp(interlc,(2 * (x2 - x1) ; 2 * (y2 - y1)
; x1 * x1 + (y1 * y1 + (k2 * k2
  - (x2 * x2 + (y2 * y2 + k1 * k1)))))
    & (x1 ; y1 ; k1)) .
eq interp(ldl,(a ; b ; c) & k) =
if k == 0
then (a ; b ; c)
else (a ; b ; c - k * sqrt(a * a + b * b))
   & (a ; b ; c + k * sqrt(a * a + b * b)) fi .
eq interp(lineh,(x ; y) & k) =
if k == 0
then (0 ; 1 ; - y)
else
(0 ; 1 ; - y + k)
      & (0 ; 1 ; - y - k)
fi .
eq interp(linev,(x ; y) & k) =
if k == 0
then (1 ; 0 ; - x)
else
(1 ; 0 ; - x + k)
      & (1 ; 0 ; - x - k)
fi .
eq interp(bissectdd,(a1 ; b1 ; c1) & (a2 ; b2 ; c2)) =
if a2 * b1 - a1 * b2 == 0
then (a1 ; b1 ; (c2 + c1) / 2)
else
```

```
interp(lpla,interp(interll,(a1 ; b1 ; c1) & (a2 ; b2 ; c2))
  & (a1 ; b1 ; c1)
  & angle(- b1,a1,- b2,a2) / 2)
      & interp(lpla,interp(interll,(a1 ; b1 ; c1) & (a2 ; b2 ; c2))
  & (a1 ; b1 ; c1)
  & angle(- b1,a1,b2,- a2) / 2)
fi .
eq interp(mkcir4,(x1 ; y1) & (x2 ; y2 ; k)) =
(x1 ; y1 ; abs(distpp(x1,y1,x2,y2) - k))
      & (x1 ; y1 ; distpp(x1,y1,x2,y2) + k) .
eq interp(medradcir,(x1 ; y1) & (x2 ; y2) & k) =
if ((distpp(x1,y1,x2,y2) / 2) > k) or (x1 == x2 and y1 == y2)
then fail
else interp(mkcir,interp(interlc,interp(mediatrice,(x1 ; y1) & (x2 ; y2))
       & (x1 ; y1 ; k)) & k)
fi .

*** 1er cas: pas solution, 2eme cas: 1 solution, 2eme cas: 2 solutions, etc...
*** (
eq interp(ltangent,(x1 ; y1 ; k1) & (x2 ; y2 ; k2)) =
if (distpp(x1,y1,x2,y2) == 0
    or (distpp(x1,y1,x2,y2) + k1) < k2
    or (distpp(x1,y1,x2,y2) + k2) < k1)
then fail
else
if (distpp(x1,y1,x2,y2) + k1) == k2 or (distpp(x1,y1,x2,y2) + k2) == k1
then (if k1 < k2
     then (x2 - x1 ; y2 - y1 ;
   k2 * distpp(x1,y1,x2,y2) + (x1 * x2 + (y1 * y2
    - (x2 * x2 + y2 * y2))))
     else (x1 - x2 ; y1 - y2 ;
   k1 * distpp(x2,y2,x1,y1) + (x2 * x1 + (y2 * y1
    - (x1 * x1 + y1 * y1)))) fi)
else
if distpp(x1,y1,x2,y2) < (k1 + k2)
then interp(lpp,elem(interp(intercc,(x1 ; y1 ; rt-ext(x1,y1,k1,x2,y2,k2))
  & (x2 ; y2 ; k2)),1)
      & elem(interp(intercc,(x2 ; y2 ; rt-ext(x2,y2,k2,x1,y1,k1))
  & (x1 ; y1 ; k1)),2))
   & interp(lpp,elem(interp(intercc,(x1 ; y1 ; rt-ext(x1,y1,k1,x2,y2,k2))
  & (x2 ; y2 ; k2)),2)
      & elem(interp(intercc,(x2 ; y2 ; rt-ext(x2,y2,k2,x1,y1,k1))
  & (x1 ; y1 ; k1)),1))
else
if distpp(x1,y1,x2,y2) == (k1 + k2)
then (x2 - x1 ; y2 - y1
      ; k2 * distpp(x1,y1,x2,y2) + (x1 * x2 + (y1 * y2 - (x2 * x2 + y2 * y2))))
   & interp(lpp,elem(interp(intercc,(x1 ; y1 ; rt-ext(x1,y1,k1,x2,y2,k2))
  & (x2 ; y2 ; k2)),1)
      & elem(interp(intercc,(x2 ; y2 ; rt-ext(x2,y2,k2,x1,y1,k1))
  & (x1 ; y1 ; k1)),2))
   & interp(lpp,elem(interp(intercc,(x1 ; y1 ; rt-ext(x1,y1,k1,x2,y2,k2))
  & (x2 ; y2 ; k2)),2)
      & elem(interp(intercc,(x2 ; y2 ; rt-ext(x2,y2,k2,x1,y1,k1))
  & (x1 ; y1 ; k1)),1))
else interp(lpp,elem(interp(intercc,(x1 ; y1 ; rt-ext(x1,y1,k1,x2,y2,k2))
  & (x2 ; y2 ; k2)),1)
      & elem(interp(intercc,(x2 ; y2 ; rt-ext(x2,y2,k2,x1,y1,k1))
  & (x1 ; y1 ; k1)),2))
   & interp(lpp,elem(interp(intercc,(x1 ; y1 ; rt-ext(x1,y1,k1,x2,y2,k2))
  & (x2 ; y2 ; k2)),2)
      & elem(interp(intercc,(x2 ; y2 ; rt-ext(x2,y2,k2,x1,y1,k1))
  & (x1 ; y1 ; k1)),1))
   & interp(lpp,elem(interp(intercc,(x1 ; y1 ; rt-int(x1,y1,k1,x2,y2,k2))
  & (x2 ; y2 ; k2)),1)
      & elem(interp(intercc,(x2 ; y2 ; rt-int(x2,y2,k2,x1,y1,k1))
  & (x1 ; y1 ; k1)),1))
   & interp(lpp,elem(interp(intercc,(x1 ; y1 ; rt-int(x1,y1,k1,x2,y2,k2))
  & (x2 ; y2 ; k2)),2)
      & elem(interp(intercc,(x2 ; y2 ; rt-int(x2,y2,k2,x1,y1,k1))
  & (x1 ; y1 ; k1)),2))
fi fi fi fi .
)
endo
--- ---------------------------------------------------------
```

Spécification algébrique de l'interprétation par gel d'une branche (version antérieure) :

```
make NSET is SET-EQ[INTS]*(sort Set to Nset , sort List to Nlist ,
   op emptyset to n-emptyset , op emptylist to n-emptylist)
endm

--- ---------------------------------------------------

obj BEST is protecting INTERP + NSET .

op near : Flist Flist -> Bool . *** 2 solutions sont-elles proches?
op select-in-fflist : FFlist Flist Int -> Int . *** selectionne une possibilite
op select-in-affect : Affect Interp -> Nlist . *** selectionne une possibilite
op select-in-interp : Interp Interp -> Nlist . *** selectionne une branche
op best : Interp Nlist -> Interp .

var in1 in2 : Interp .
var i1 i2 : I .
var n : Int .
var fl1 fl2 fl : Flist .
var ffl : FFlist .
var f1 f2 : Float .
var nl : Nlist .

eq near(f-emptylist,f-emptylist) = true .
eq near(f1 ; fl1,f2 ; fl2) = ((f1 - f2) < 0.05 or (f1 - f2) > - 0.05)
     and near(fl1,fl2) .

--- selection de la branche correspondant a l'esquisse

eq select-in-fflist(fl1 | ffl,fl2,n) = if near(fl1,fl2) then n
       else select-in-fflist(ffl,fl2,n + 1) fi .
eq select-in-affect(i1 => (fl1 | ffl),(i2 => fl2) " in2) = if i1 == i2
then select-in-fflist(fl1 | ffl,fl2,1)
else select-in-affect(i1 => (fl1 | ffl),in2)
fi .
eq select-in-interp(a-emptylist,in2) = n-emptylist .
eq select-in-interp((i1 => (fl | ffl)) " in1,in2) =
if empty(ffl)
then select-in-interp(in1,in2)
else select-in-affect(i1 => (fl | ffl),in2) " select-in-interp(in1,in2)
fi .

--- selection de cette branche dans l'interpretation avec les nouvelles donnees

eq best(in1,n-emptylist) = in1 .
eq best((i1 => (fl | ffl)) " in1,n " nl) =
if empty(ffl)
then (i1 => (fl | ffl)) " best(in1,n " nl)
else if n == 1
     then (i1 => fl) " best(in1,nl)
     else best((i1 => ffl) " in1,(n - 1) " nl)
     fi
fi .

endo

--- ---------------------------------------------------

obj OCCURRENCES is protecting INTERP + NSET .

op interp-sketch : Fun Interp -> Nlist .
op interp-sketch : Def-list Interp -> Nlist .
op interp-sketch : Def Interp -> Int .
op occurrence : Flist FFlist Int -> Int .
op find : I Interp -> Flist .
op near : Flist Flist -> Bool .
op interp-values : Fun FFFlist Nlist -> Interp .
op interp-values : Def-list Interp Nlist -> Interp .
op interp-values : Term Interp Int -> Flist .
op get : FFlist Int -> Flist .

var fct : Fun .
var al : Interp .
var d : Def .
var dl : Def-list .
var i i1 i2 : I .
var il : Ilist .
var f : F .
var f1 f2 : Float .
var fl fl1 fl2 : Flist .
var ffl : FFlist .
var fffl : FFFlist .
var n : Int .
var nl : Nlist .

eq near(f-emptylist,f-emptylist) = true .
eq near(f1 ; fl1,f2 ; fl2) = ((f1 - f2) < 0.05 or (f1 - f2) > - 0.05)
```

```
     and near(fl1,fl2) .
eq find(i1,(i2 => fl) " al) = if i1 == i2 then fl else find(i1,al) fi .
eq occurrence(fl1,fl2 | ffl,n) = if near(fl1,fl2)
 then n
 else occurrence(fl1,ffl,n + 1)
 fi .
eq interp-sketch(fct,al) = interp-sketch(definitions(fct),al) .
eq interp-sketch(d-emptylist,al) = n-emptylist .
eq interp-sketch(d " dl,al) = interp-sketch(d,al) " interp-sketch(dl,al) .
eq interp-sketch(i := f[il],al) = occurrence(find(i,al),
     interp(f,extract(il,al,al)),1) .
eq get(fl | ffl,n) = if n == 1 then fl else get(ffl,n - 1) fi .
eq interp-values(fct,fffl,nl) = interp-values(definitions(fct),
      affect(parameters(fct),fffl),
      nl) .
eq interp-values(d-emptylist,al,nl) = al .
eq interp-values(d " dl,al,n " nl) =
interp-values(dl,(ident(d) => interp-values(term(d),al,n))" al,nl) .
eq interp-values(f[il],al,n) = get(interp(f,extract(il,al,al)),n) .
endo
```

Annexe C

Contraintes et plan de construction pour l'exemple d'un levier

Tab. C.1: Contraintes du levier

angarc(p28, p3, p25, a18)	angarc(p26, p24, p2, a17)	centre(c6, p31)
centre(c4, p29)	centre(c3, p28)	centre(c2, p27)
radius(c5, k31)	radius(c6, k30)	radius(c4, k29)
centre(c5, p30)	centre(c1, p26)	radius(c2, k28)
radius(c3, k27)	radius(c1, k26)	angle(p13, p16, p13, p11, a16)
angle(p17, p16, p17, p18, a14)	angle(p15, p16, p15, p12, a13)	angle(p18, p17, p18, p19, a12)
angle(p10, p12, p10, p9, a10)	angle(p19, p18, p19, p20, a9)	angle(p20, p21, p20, p19, a8)
angle(p21, p20, p21, p22, a6)	angle(p8, p9, p8, p7, a5)	angle(p22, p21, p22, p23, a4)
angle(p16, p17, p16, p15, a15)	angle(p12, p15, p12, p10, a11)	angle(p9, p10, p9, p8, a7)
angle(p17, p14, p4, p3, a2)	angle(p17, p14, p1, p2, a1)	angle(p5, p6, p22, p23,a3)
distpp(p11, p13, k25)	distpp(p13, p14, k24)	distpp(p15, p16, k20)
distpp(p13, p16, k23)	distpp(p16, p17, k22)	distpp(p18, p17, k21)
distpp(p12, p15, k19)	distpp(p19, p18, k18)	distpp(p19, p20, k17)
distpp(p12, p10, k16)	distpp(p9, p8, k12)	distpp(p5, p8, k8)
distpp(p21, p20, k15)	distpp(p9, p10, k14)	distpp(p21, p22, k13)
distpp(p23, p22, k11)	distpp(p7, p8, k10)	distpp(p5, p6, k9)
distpp(p7, p5, k7)	distpp(p1, p22, k6)	distpp(p1, p23, k5)
distpp(p14, p4, k3)	distpp(p4, p3, k2)	distpp(p1, p2, k1)
distpp(p23, p4, k4)	fixorgpl(p13, l10, p16)	onc(p7, c5)
onc(p1, c6)	onc(p6, c6)	onc(p5, c5)
onc(p4, c4)	onc(p23, c4)	onc(p3, c3)
onc(p25, c2)	onc(p24, c2)	onc(p24, c1)
onc(p25, c3)	onc(p2, c1)	onl(p21, l16)
onl(p23, l17)	onl(p22, l17)	onl(p22, l16)
onl(p21, l15)	onl(p20, l15)	onl(p20, l14)
onl(p19, l13)	onl(p18, l13)	onl(p18, l12)
onl(p17, l10)	onl(p16, l10)	onl(p16, l11)
onl(p15, l9)	onl(p14, l10)	onl(p13, l10)
onl(p12, l9)	onl(p12, l7)	onl(p11, l8)
onl(p10, l6)	onl(p9, l6)	onl(p9, l5)
onl(p8, l4)	onl(p7, l4)	onl(p6, l3)
onl(p4, l2)	onl(p3, l2)	onl(p2, l1)
onl(p19, l14)	onl(p17, l12)	onl(p15, l11)
onl(p13, l8)	onl(p10, l7)	onl(p8, l5)
onl(p5, l3)	onl(p1, l1)	tgcl(c1, l1)
tgcl(c3, l2)		

Tab. C.2: Plan de construction du levier

k25 = initl(200.000000)
k24 = initl(400.000000)
a16 = inita(1.570796)
k23 = initl(100.000000)
k22 = initl(30.000000)
k21 = initl(25.000000)
k20 = eg_lg(k21)
a15 = inita(1.570796)
a14 = inita(-1.570796)
a13 = inita(-1.570796)
a12 = inita(1.570796)
a11 = inita(1.570796)
a10 = inita(1.570796)
a9 = inita(-1.570796)
a8 = inita(1.570796)
a7 = inita(-1.570796)
a6 = inita(1.570796)
a5 = inita(-1.570796)
a4 = inita(1.570796)
k19 = initl(10.000000)
k18 = eg_lg(k19)
k17 = initl(20.000000)
k16 = eg_lg(k17)
k15 = eg_lg(k19)
k14 = eg_lg(k15)
k13 = initl(5.000000)
k12 = eg_lg(k13)
k11 = initl(27.000000)
k10 = eg_lg(k11)
k9 = initl(40.000000)
k8 = initl(70.000000)
k7 = initl(74.000000)
k31 = initl(70.000000)
a3 = inita(0.000000)
k6 = initl(58.000000)
k5 = initl(49.000000)
k30 = initl(28.000000)
k4 = initl(26.000000)
k3 = initl(241.000000)
k29 = initl(25.000000)
a2 = inita(0.349066)
k2 = initl(200.000000)
a18 = inita(-0.900000)
a17 = inita(-0.900000)
k28 = initl(32.000000)
k1 = initl(200.000000)
a1 = inita(0.314159)
k27 = initl(22.000000)
k26 = eg_lg(k27)
p13 = initp(0.000000,0.000000)
l10 = initd(p13,0.000000)
a33 = supp_angle(a14)
a19 = inv_angle(a33)
a34 = supp_angle(a14)
a35 = supp_angle(a13)
a36 = inv_angle(a35)
a37 = add_angle(a15,a36)
a38 = add_angle(a34,a37)
a20 = mod_angle(a38)
a39 = inv_angle(a15)
a40 = supp_angle(a14)
a41 = inv_angle(a40)
a42 = supp_angle(a12)
a43 = inv_angle(a42)
a44 = add_angle(a41,a43)
a45 = add_angle(a39,a44)
a46 = mod_angle(a45)
a47 = supp_angle(a46)
a21 = inv_angle(a47)
a48 = supp_angle(a13)
a49 = inv_angle(a48)
a50 = supp_angle(a11)
a51 = inv_angle(a50)
a52 = add_angle(a49,a51)
a53 = mod_angle(a52)
a54 = supp_angle(a53)
a22 = inv_angle(a54)
a55 = supp_angle(a12)
a56 = inv_angle(a55)
a57 = supp_angle(a9)
a58 = inv_angle(a57)
a59 = add_angle(a56,a58)
a60 = mod_angle(a59)
a61 = supp_angle(a60)
a23 = inv_angle(a61)
a62 = supp_angle(a11)
a63 = inv_angle(a62)
a64 = supp_angle(a10)
a65 = inv_angle(a64)
a66 = add_angle(a63,a65)
a67 = mod_angle(a66)
a68 = supp_angle(a67)
a24 = inv_angle(a68)
a69 = supp_angle(a9)
a70 = inv_angle(a69)
a71 = supp_angle(a8)
a72 = add_angle(a70,a71)
a73 = mod_angle(a72)
a74 = supp_angle(a73)
a25 = inv_angle(a74)
a75 = supp_angle(a10)
a76 = inv_angle(a75)
a77 = supp_angle(a7)
a78 = inv_angle(a77)
a79 = add_angle(a76,a78)
a80 = mod_angle(a79)
a81 = supp_angle(a80)
a26 = inv_angle(a81)
a82 = supp_angle(a8)
a83 = supp_angle(a6)
a84 = inv_angle(a83)
a85 = add_angle(a82,a84)
a86 = mod_angle(a85)
a87 = supp_angle(a86)
a27 = inv_angle(a87)
a88 = supp_angle(a7)
a89 = inv_angle(a88)
a90 = supp_angle(a5)
a91 = inv_angle(a90)
a92 = add_angle(a89,a91)
a93 = mod_angle(a92)
a94 = supp_angle(a93)
a28 = inv_angle(a94)
a95 = supp_angle(a6)
a96 = inv_angle(a95)
a97 = supp_angle(a4)

a98 = inv_angle(a97)
a99 = add_angle(a96,a98)
a100 = mod_angle(a99)
a101 = supp_angle(a100)
a29 = inv_angle(a101)
a102 = supp_angle(a13)
a103 = inv_angle(a102)
a104 = supp_angle(a11)
a105 = inv_angle(a104)
a106 = supp_angle(a10)
a107 = inv_angle(a106)
a108 = supp_angle(a7)
a109 = inv_angle(a108)
a110 = supp_angle(a5)
a111 = inv_angle(a110)
a112 = add_angle(a109,a111)
a113 = add_angle(a107,a112)
a114 = add_angle(a105,a113)
a115 = add_angle(a103,a114)
a116 = add_angle(a15,a115)
a117 = inv_angle(a116)
a118 = supp_angle(a14)
a119 = inv_angle(a118)
a120 = supp_angle(a12)
a121 = inv_angle(a120)
a122 = supp_angle(a9)
a123 = inv_angle(a122)
a124 = supp_angle(a8)
a125 = supp_angle(a6)
a126 = inv_angle(a125)
a127 = supp_angle(a4)
a128 = inv_angle(a127)
a129 = inv_angle(a3)
a130 = add_angle(a128,a129)
a131 = add_angle(a126,a130)
a132 = add_angle(a124,a131)
a133 = add_angle(a123,a132)
a134 = add_angle(a121,a133)
a135 = add_angle(a119,a134)
a136 = add_angle(a117,a135)
a137 = mod_angle(a136)
a138 = supp_angle(a137)
a30 = inv_angle(a138)
l8 = lpla(p13,l10,a16)
c7 = mkcir(p13,k25)
p11 = interlc(l8,c7)
c8 = mkcir(p13,k24)
p14 = interlc(l10,c8)
c9 = mkcir(p13,k23)
p16 = interlc(l10,c9)
c10 = mkcir(p16,k22)
p17 = interlc(l10,c10)
c11 = mkcir(p17,k21)
c12 = mkcir(p16,k20)
c13 = mkcir(p14,k3)
l11 = lpla(p16,l10,a15)
l12 = lpla(p17,l10,a19)
p15 = interlc(l11,c12)
l9 = lpla(p15,l12,a20)
p18 = interlc(l12,c11)
c14 = mkcir(p15,k19)
p12 = interlc(l9,c14)
l13 = lpla(p18,l11,a21)
l7 = lpla(p12,l11,a22)
c15 = mkcir(p18,k18)
p19 = interlc(l13,c15)
l14 = lpla(p19,l12,a23)
c16 = mkcir(p12,k16)
p10 = interlc(l7,c16)
l6 = lpla(p10,l9,a24)
c17 = mkcir(p19,k17)
p20 = interlc(l14,c17)
l15 = lpla(p20,l13,a25)
c18 = mkcir(p10,k14)
p9 = interlc(l6,c18)
l5 = lpla(p9,l7,a26)
c19 = mkcir(p20,k15)
p21 = interlc(l15,c19)
l16 = lpla(p21,l14,a27)
c20 = mkcir(p9,k12)
p8 = interlc(l5,c20)
l4 = lpla(p8,l6,a28)
c21 = mkcir(p21,k13)
p22 = interlc(l16,c21)
l17 = lpla(p22,l15,a29)
c22 = mkcir(p8,k8)
c23 = mkcir(p8,k10)
p7 = interlc(l4,c23)
c24 = mkcir(p22,k6)
c25 = mkcir(p22,k11)
p23 = interlc(l17,c25)
c26 = mkcir(p7,k7)
p5 = intercc(c26,c22)
l3 = lpla(p5,l4,a30)
c5 = medradcir(p7,p5,k31)
c27 = mkcir(p23,k4)
p4 = intercc(c27,c13)
c28 = mkcir(p23,k5)
p1 = intercc(c28,c24)
c4 = medradcir(p23,p4,k29)
c29 = mkcir(p5,k9)
p6 = interlc(l3,c29)
p30 = centre_of(c5)
c30 = mkcir(p4,k2)
l2 = lpla(p4,l10,a2)
c31 = mkcir(p1,k1)
l1 = lpla(p1,l10,a1)
c6 = medradcir(p1,p6,k30)
p29 = centre_of(c4)
p3 = interlc(l2,c30)
p2 = interlc(l1,c31)
p31 = centre_of(c6)
a31 = inita(1.570796)
l18 = lpla(p3,l2,a31)
c32 = mkcir(p3,k27)
p28 = interlc(l18,c32)
c3 = mkcir2(p28,p3)
p25 = fangarc(p28,p3,a18)
l19 = lpla(p2,l1,a31)
c33 = mkcir(p2,k26)
p26 = interlc(l19,c33)
a32 = inv_angle(a17)
p24 = fangarc(p26,p2,a32)
c1 = medradcir(p2,p24,k26)
c2 = medradcir(p25,p24,k28)
p27 = centre_of(c2)

Tab. C.3: Plan de construction du levier trié

p13.1 = initp(0.000000,0.000000)
k23.0 = initl(100.000000)
c9.1 = mkcir(p13.1, k23.0)
l10.1 = initd(p13.1,0.000000)
p16.1 = interlc(l10.1, c9.1)
a15.0 = inita(1.570796)
l11.1 = lpla(p16.1, l10.1, a15.0)
k21.0 = initl(25.000000)
k20.0 = eg_lg(k21.0)
c12.1 = mkcir(p16.1, k20.0)
p15.1 = interlc(l11.1, c12.1)
k22.0 = initl(30.000000)
c10.1 = mkcir(p16.1, k22.0)
p17.1 = interlc(l10.1, c10.1)
a14.0 = inita(-1.570796)
a33.0 = supp_angle(a14.0)
a19.0 = inv_angle(a33.0)
l12.1 = lpla(p17.1, l10.1, a19.0)
a13.0 = inita(-1.570796)
a35.0 = supp_angle(a13.0)
a36.0 = inv_angle(a35.0)
a37.0 = add_angle(a15.0, a36.0)
a34.0 = supp_angle(a14.0)
a38.0 = add_angle(a34.0, a37.0)
a20.0 = mod_angle(a38.0)
l9.1 = lpla(p15.1, l12.1, a20.0)
k19.0 = initl(10.000000)
c14.1 = mkcir(p15.1, k19.0)
p12.1 = interlc(l9.1, c14.1)
k17.0 = initl(20.000000)
k16.0 = eg_lg(k17.0)
c16.1 = mkcir(p12.1,k16.0)
a11.0 = inita(1.570796)
a48.0 = supp_angle(a13.0)
a49.0 = inv_angle(a48.0)
a50.0 = supp_angle(a11.0)
a51.0 = inv_angle(a50.0)
a52.0 = add_angle(a49.0,a51.0)
a53.0 = mod_angle(a52.0)
a54.0 = supp_angle(a53.0)
a22.0 = inv_angle(a54.0)
l7.1 = lpla(p12.1,l11.1,a22.0)
p10.1 = interlc(l7.1,c16.1)
a10.0 = inita(1.570796)
a64.0 = supp_angle(a10.0)
a65.0 = inv_angle(a64.0)
a62.0 = supp_angle(a11.0)
a63.0 = inv_angle(a62.0)
a66.0 = add_angle(a63.0, a65.0)
a67.0 = mod_angle(a66.0)
a68.0 = supp_angle(a67.0)
a24.0 = inv_angle(a68.0)
l6.1 = lpla(p10.1, l9.1, a24.0)
k15.0 = eg_lg(k19.0)
k14.0 = eg_lg(k15.0)
c18.1 = mkcir(p10.1, k14.0)
p9.1 = interlc(l6.1, c18.1)
a7.0 = inita(-1.570796)
a77.0 = supp_angle(a7.0)
a78.0 = inv_angle(a77.0)
a75.0 = supp_angle(a10.0)
a76.0 = inv_angle(a75.0)
a79.0 = add_angle(a76.0, a78.0)
a80.0 = mod_angle(a79.0)
a81.0 = supp_angle(a80.0)
a26.0 = inv_angle(a81.0)
l5.1 = lpla(p9.1, l7.1, a26.0)
k13.0 = initl(5.000000)
k12.0 = eg_lg(k13.0)
c20.1 = mkcir(p9.1, k12.0)
p8.1 = interlc(l5.1, c20.1)
a5.0 = inita(-1.570796)
a90.0 = supp_angle(a5.0)
a91.0 = inv_angle(a90.0)
a88.0 = supp_angle(a7.0)
a89.0 = inv_angle(a88.0)
a92.0 = add_angle(a89.0, a91.0)
a93.0 = mod_angle(a92.0)
a94.0 = supp_angle(a93.0)
a28.0 = inv_angle(a94.0)
l4.1 = lpla(p8.1, l6.1, a28.0)
k11.0 = initl(27.000000)
k10.0 = eg_lg(k11.0)
c23.1 = mkcir(p8.1, k10.0)
p7.1 = interlc(l4.1, c23.1)
k8.0 = initl(70.000000)
c22.1 = mkcir(p8.1, k8.0)
k7.0 = initl(74.000000)
c26.1 = mkcir(p7.1, k7.0)
p5.1 = intercc(c26.1, c22.1)
a3.0 = inita(0.000000)
a129.0 = inv_angle(a3.0)
a4.0 = inita(1.570796)
a127.0 = supp_angle(a4.0)
a128.0 = inv_angle(a127.0)
a130.0 = add_angle(a128.0, a129.0)
a6.0 = inita(1.570796)
a125.0 = supp_angle(a6.0)
a126.0 = inv_angle(a125.0)
a131.0 = add_angle(a126.0, a130.0)
a8.0 = inita(1.570796)
a124.0 = supp_angle(a8.0)
a132.0 = add_angle(a124.0, a131.0)
a9.0 = inita(-1.570796)
a122.0 = supp_angle(a9.0)
a123.0 = inv_angle(a122.0)
a133.0 = add_angle(a123.0, a132.0)
a12.0 = inita(1.570796)
a120.0 = supp_angle(a12.0)
a121.0 = inv_angle(a120.0)
a134.0 = add_angle(a121.0, a133.0)
a118.0 = supp_angle(a14.0)
a119.0 = inv_angle(a118.0)
a135.0 = add_angle(a119.0, a134.0)
a110.0 = supp_angle(a5.0)
a111.0 = inv_angle(a110.0)
a108.0 = supp_angle(a7.0)
a109.0 = inv_angle(a108.0)
a112.0 = add_angle(a109.0, a111.0)
a106.0 = supp_angle(a10.0)
a107.0 = inv_angle(a106.0)
a113.0 = add_angle(a107.0, a112.0)
a104.0 = supp_angle(a11.0)
a105.0 = inv_angle(a104.0)
a114.0 = add_angle(a105.0, a113.0)
a102.0 = supp_angle(a13.0)

a103.0 = inv_angle(a102.0)
a115.0 = add_angle(a103.0, a114.0)
a116.0 = add_angle(a15.0, a115.0)
a117.0 = inv_angle(a116.0)
a136.0 = add_angle(a117.0, a135.0)
a137.0 = mod_angle(a136.0)
a138.0 = supp_angle(a137.0)
a30.0 = inv_angle(a138.0)
l3.1 = lpla(p5.1, l4.1, a30.0)
k9.0 = initl(40.000000)
c29.1 = mkcir(p5.1, k9.0)
p6.1 = interlc(l3.1, c29.1)
k31.0 = initl(70.000000)
c5.1 = medradcir(p7.1, p5.1, k31.0)
c11.1 = mkcir(p17.1, k21.0)
p18.1 = interlc(l12.1, c11.1)
a42.0 = supp_angle(a12.0)
a43.0 = inv_angle(a42.0)
a40.0 = supp_angle(a14.0)
a41.0 = inv_angle(a40.0)
a44.0 = add_angle(a41.0, a43.0)
a39.0 = inv_angle(a15.0)
a45.0 = add_angle(a39.0, a44.0)
a46.0 = mod_angle(a45.0)
a47.0 = supp_angle(a46.0)
a21.0 = inv_angle(a47.0)
l13.1 = lpla(p18.1, l11.1, a21.0)
k18.0 = eg_lg(k19.0)
c15.1 = mkcir(p18.1, k18.0)
p19.1 = interlc(l13.1, c15.1)
a57.0 = supp_angle(a9.0)
a58.0 = inv_angle(a57.0)
a55.0 = supp_angle(a12.0)
a56.0 = inv_angle(a55.0)
a59.0 = add_angle(a56.0, a58.0)
a60.0 = mod_angle(a59.0)
a61.0 = supp_angle(a60.0)
a23.0 = inv_angle(a61.0)
l14.1 = lpla(p19.1, l12.1, a23.0)
c17.1 = mkcir(p19.1, k17.0)
p20.1 = interlc(l14.1, c17.1)
a71.0 = supp_angle(a8.0)
a69.0 = supp_angle(a9.0)
a70.0 = inv_angle(a69.0)
a72.0 = add_angle(a70.0, a71.0)
a73.0 = mod_angle(a72.0)
a74.0 = supp_angle(a73.0)
a25.0 = inv_angle(a74.0)
l15.1 = lpla(p20.1, l13.1, a25.0)
c19.1 = mkcir(p20.1, k15.0)
p21.1 = interlc(l15.1, c19.1)
a83.0 = supp_angle(a6.0)
a84.0 = inv_angle(a83.0)
a82.0 = supp_angle(a8.0)
a85.0 = add_angle(a82.0, a84.0)
a86.0 = mod_angle(a85.0)
a87.0 = supp_angle(a86.0)
a27.0 = inv_angle(a87.0)
l16.1 = lpla(p21.1, l14.1, a27.0)
c21.1 = mkcir(p21.1, k13.0)
p22.1 = interlc(l16.1, c21.1)
a97.0 = supp_angle(a4.0)
a98.0 = inv_angle(a97.0)
a95.0 = supp_angle(a6.0)
a96.0 = inv_angle(a95.0)
a99.0 = add_angle(a96.0, a98.0)
a100.0 = mod_angle(a99.0)
a101.0 = supp_angle(a100.0)
a29.0 = inv_angle(a101.0)
l17.1 = lpla(p22.1, l15.1, a29.0)
c25.1 = mkcir(p22.1, k11.0)
p23.1 = interlc(l17.1, c25.1)
k6.0 = initl(58.000000)
c24.1 = mkcir(p22.1, k6.0)
k5.0 = initl(49.000000)
c28.1 = mkcir(p23.1, k5.0)
p1.1 = intercc(c28.1, c24.1)
a1.0 = inita(0.314159)
l1.1 = lpla(p1.1, l10.1, a1.0)
k1.0 = initl(200.000000)
c31.1 = mkcir(p1.1, k1.0)
p2.1 = interlc(l1.1, c31.1)
a17.0 = inita(-0.900000)
a32.1 = inv_angle(a17.0)
k27.0 = initl(22.000000)
k26.0 = eg_lg(k27.0)
c33.1 = mkcir(p2.1, k26.0)
a31.1 = inita(1.570796)
l19.1 = lpla(p2.1, l1.1, a31.1)
p26.1 = interlc(l19.1, c33.1)
p24.1 = fangarc(p26.1, p2.1, a32.1)
c1.1 = medradcir(p2.1, p24.1, k26.0)
k30.0 = initl(28.000000)
c6.1 = medradcir(p1.1, p6.1, k30.0)
k24.0 = initl(400.000000)
c8.1 = mkcir(p13.1, k24.0)
p14.1 = interlc(l10.1, c8.1)
k3.0 = initl(241.000000)
c13.1 = mkcir(p14.1, k3.0)
k4.0 = initl(26.000000)
c27.1 = mkcir(p23.1, k4.0)
p4.1 = intercc(c27.1, c13.1)
a2.0 = inita(0.349066)
l2.1 = lpla(p4.1, l10.1, a2.0)
k2.0 = initl(200.000000)
c30.1 = mkcir(p4.1, k2.0)
p3.1 = interlc(l2.1, c30.1)
a18.0 = inita(-0.900000)
c32.1 = mkcir(p3.1, k27.0)
l18.1 = lpla(p3.1, l2.1, a31.1)
p28.1 = interlc(l18.1, c32.1)
p25.1 = fangarc(p28.1, p3.1, a18.0)
c3.1 = mkcir2(p28.1, p3.1)
k28.0 = initl(32.000000)
c2.1 = medradcir(p25.1, p24.1, k28.0)
k29.0 = initl(25.000000)
c4.1 = medradcir(p23.1, p4.1, k29.0)
k25.0 = initl(200.000000)
c7.1 = mkcir(p13.1, k25.0)
a16.0 = inita(1.570796)
l8.1 = lpla(p13.1,l10.1,a16.0)
p11.1 = interlc(l8.1, c7.1)
p30.1 = centre_of(c5.1)
p29.1 = centre_of(c4.1)
p31.1 = centre_of(c6.1)
p27.1 = centre_of(c2.1)

Annexe D

Démonstration du théorème 1

Rappel :

$S = (U, X, C)$ et $|U| = s$.
Il existe une déformation continue de S_u vers S_v signifie que $\exists \psi$ continue : $\psi : [0,1] \rightarrow I\!R^s$
(i) $\psi(0) = u$ et $\psi(1) = v$ (c'est-à-dire on passe continûment du vecteur de paramètres u au vecteur de paramètres v)
(ii) $\forall S'$ tel que $S' \equiv S$, $\sigma \subseteq S'$ bien contraint, et $\forall t \in [0,1]$, $\sigma'_{\psi(t)}$ a autant de solutions que σ'_u

Note :

Si $T \equiv S$, et $S_u \hookrightarrow S_v$ (avec $\hookrightarrow$ signifiant "se déforme continûment"), alors $T_u \hookrightarrow T_v$.

Théorème :

$S \equiv T$, $S_u \hookrightarrow S_v$ (et $T_u \hookrightarrow T_v$), $f_u \in \mathcal{F}(S_u)$ et $f_v \in \mathcal{F}(S_v)$.
f_u et f_v sont S-homotopiques si et seulement si elles ont la même occurrence dans T_u et T_v.

Preuve de $\Rightarrow$:

f_u et f_v sont S-homotopiques signifie que l'on a les hypothèses (1) suivantes :
(i) $S_u \hookrightarrow S_v$ par ψ
(ii) $f_u \hookrightarrow f_v$ par φ
(iii) $\forall t \in [0,1]$ tel que $\varphi(t)$ solution de $S_{\psi(t)}$
On peut supposer sans perte de généralité que f_u et f_v sont réduits à 1 objet, c'est-à-dire : T est de la forme $X = g(U)$, g étant une multifonction, avec $f_u \in g(u)$ et $f_v \in g(v)$.

Comme $T_u \hookrightarrow T_v$, $\forall t \in [0,1]$ le degré de $g(\psi(t))$ (le cardinal) reste égal à une constante K, c'est-à-dire $|g(u)| = |g(v)| = [g(\psi(t))| = K$.

f_u et f_v ont la même occurrence (dans une numérotation continue de g) signifie que $\exists k$ tel que, en posant $g(u) = \Big\{G(u,1), G(u,2), \ldots, G(u,K)\Big\}$, $f_u = G(u,k)$ et $f_v = G(v,k)$.

On supposera que $f_u = G(u,1)$.

Le sens $\Rightarrow$ du théorème consiste donc à montrer que sous les hypothèses (1), $f_v = \varphi(1) = G(v,1)$.

Lemme :

On pose $\varphi(t) = G(\psi(t), k(t))$ avec $k : [0,1] \to [\![1,k]\!]$; on appelle k la numérotation de φ par rapport à G.

On a : $\forall t_0 \in [0,1]$, il existe $\epsilon > 0$ et $k_0 \in [\![1,k]\!]$ tels que $\forall t \in [0,1]$, $|t_0 - t| < \epsilon \Rightarrow k(t) = k_0$.

Preuve : ceci découle de :

(1) les fonctions $d_{l,m} : t \mapsto dist\Big(G(\psi(t),l), G(\psi(t),m)\Big)$ sont continues sur [0,1] ($\forall l, m \in [\![1,k]\!]$) à valeurs positives, donc $\exists \alpha > 0$ $\forall l, m$ et $\forall t \in [0,1]$ $dist\Big(G(\psi(t),l), G(\psi(t),m)\Big) \geq \alpha$

(2) $G_{\psi,l} : t \mapsto G(\psi(t),l)$ est continue sur $[0,1]$ donc $\forall t_0$, $\exists \epsilon$ tel que $|t - t_0| < \epsilon \Rightarrow dist\Big(G(\psi(t_0),l), G(\psi(t),m)\Big) < \frac{\alpha}{2}$

Donc : $|t_0 - t| < \epsilon \Rightarrow dist\Big(G(\psi(t),l), G(\psi(t_0),m)\Big) \geq \frac{\alpha}{2}$ Autrement dit, si $k(t_0) = k_0$, alors $\forall l \neq k_0$, $dist\Big(G(\psi(t),l), \varphi(t_0)\Big) \geq \frac{\alpha}{2}$

Donc, $G(\psi(t),l) \neq \varphi(t)$ (sinon, ça contredirait la continuité de φ).

Revenons au **théorème** :

On suppose que $f_u = \varphi(0) = G(u,1)$ et $f_v = \varphi(1) = G(v,l)$ avec $l \neq 1$.

On construit deux suites $(t_i)_{i \in I\!N}$ et $(t'_i)_{i \in I\!N}$ à valeurs dans [0,1] comme suit :

$t_0 = 0$, $t'_0 = 1$

$$t_n = \begin{cases} \frac{t_{n-1}+t'_{n-1}}{2} & \text{si } \; k\left(\frac{t_{n-1}+t'_{n-1}}{2}\right) = 1 \\ t_{n-1} & \text{sinon} \end{cases}$$

$$t'_n = \begin{cases} \frac{t_{n-1}+t'_{n-1}}{2} & \text{si } \; k\left(\frac{t_{n-1}+t'_{n-1}}{2}\right) \neq 1 \\ t'_{n-1} & \text{sinon} \end{cases}$$

Alors on a : $(t_i)_{i \in I\!N}$ est croissante, $(t'_i)_{i \in I\!N}$ est décroissante, et $\lim_i (t_i - t'_i) = 0$ (les suites sont adjacentes).

Donc, $\lim_i t_i = \lim_i t'_i = t_\infty$.

Lorsque i est suffisamment grand, on a $k(t'_i) = \text{cste} \neq 1$, d'après le lemme, par construction.

$$\begin{aligned}
\text{Or, } \lim_i(\varphi(t_i)) &= \lim_i \Big(G(\psi(t_i),1)\Big) \\
&= G(\psi(t_\infty),1) \\
&= \varphi(t_\infty) \\
\text{Par continuité, } \lim_i(\varphi(t'_i)) &= \lim_i \Big(G(\psi(t'_i),k(t'_i))\Big) \\
&= \lim_i \Big(G(\psi(t'_i),k_\infty)\Big) \\
&= G(\psi(t_\infty),k_\infty) \\
&= \varphi(t_\infty)
\end{aligned}$$

Donc $k_\infty = 1$, ce qui mène à une contradiction, car $k_\infty = l(t'_{i_0}) \neq 1$.

Preuve de $\Leftarrow$:

On suppose cette fois que $f_u = G(u,1)$ et $f_v = G(v,1)$
La notion de déformation continue de système que nous avons adoptée est très forte ; elle impose qu'à chaque instant t :

$$g(\psi(t)) = \Big\{G(\psi(t),1),\ldots,G(\psi(t),k)\Big\}$$

On a de plus ajouté dans les hypothèses que G est continue, donc $\exists\varphi_1,\ldots,\varphi_K$ telles que $\varphi_i = G(\psi(t),i)$ est continue.
$\varphi_1(0) = G(\psi(0),1) = f_u$
$\varphi_1(1) = G(\psi(1),1) = f_v$
et $\varphi_1(t)$ solution de $S_{\psi(t)}$

Bibliographie

[AA94] S. Ait-Aoudia. *Modélisation géométrique par contraintes : quelques méthodes de résolution.* PhD thesis, Ecole nationale supérieure des mines de Saint-Étienne, Saint-Étienne, 1994.

[AAJM93] S. Ait-Aoudia, R. Jegou, and D. Michelucci. Reduction of constraint systems. In *Proceedings of the Compugraphics Conference*, pages 83–92, 1993.

[Ald88] B. Aldefeld. Variations of geometries based on a geometric-reasoning method. *Computer-Aided Design*, 20(3) :117–126, 1988.

[AM95] R. Anderl and R. Mendgen. Parametric design and its impact on solid modeling applications. In *Proceedings of the Third Symposium on Solid Modeling and Applications*, pages 1–12. ACM Press, 1995.

[AMRV92] B. Aldefeld, H. Malberg, H. Richter, and K. Voss. *Rule-based variational geometry in Computer-Aided Design*, pages 27–46. Artificial Intelligence in Design. (D.T. Pham ed.), Springer-Verlag, 1992.

[BDEVS00] L. Brisoux-Devendeville, C. Essert-Villard, and P. Schreck. Exploration of a solution space structured by finite constraints. In *Proceedings of the 14th European Conference on Artificial Intelligence, Workshop on Modelling and Solving Problems with Constraints*, volume F, pages 1–18, 2000.

[BFH+95] W. Bouma, I. Fudos, C. Hoffmann, J. Cai, and R. Paige. Geometric constraint solver. *Computer-Aided Design*, 27(6) :487–501, 1995.

[BGS99] L. Brisoux, E. Grégoire, and L. Saïs. Improving backtrack search for sat by means of redundancy. In *Proceedings of the Foundations of Intelligent Systems, Eleventh International Symposium on Methodologies for Intelligent Systems, ISMIS'99*, volume 1609, pages 301–309. LNCS 204, Springer-Verlag, 1999.

[Bor81] A. H. Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory, 1981.

[Brü88] B. Brüderlin. Automatizing geometric proofs and constructions. In *Proceedings of Computational Geometry'88*, pages 232–252. LNCS 333, Springer-Verlag, 1988.

[Brü93] B. Brüderlin. Using geometric rewrite rules for solving geometric problems symbolically. *Theoritical Computer Science*, pages 291–303, 1993.

[Buc85] B. Buchberger. *Multidimensionnal Systems Theory*, pages 184–232. N.K. Bose, 1985.

[Bun83] Alan Bundy. *The computer modelling of mathematical reasoning*, pages 115–131. Academic Press, Inc., 1983.

[But75] Michel Buthion. *Un programme qui résout formellement des problèmes de constructions géométriques.* PhD thesis, Université Pierre et Marie Curie, Paris VI, 1975.

[CDE+99] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M. O. Stehr. Maude as a formal meta-tool. In *Proceedings of World Congress on Formal Methods (FM'99)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1703. Springer-Verlag, 1999.

[CH01a] M. Sitharam C. Hoffman, A. Lomonosov. Decomposition of geometric constraints part i : performance measures. *Journal of Symbolic Computation*, 31(4), 2001.

[CH01b] M. Sitharam C. Hoffman, A. Lomonosov. Decomposition of geometric constraints part ii : new algorithms. *Journal of Symbolic Computation*, 31(4), 2001.

[Che92] G. Chen. Les constructions à la règle et au compas par une méthode algébrique. Technical Report Rapport de DEA, Université Louis Pasteur, 1992.

[Cho88] S. C. Chou. *Mechanical geometry theorem proving.* Mathematics and its Applications. D. Reidel, Dordrecht, 1988.

[CRS98] A. Clément, A. Rivière, and P. Serré. A declarative approach for geometry and topology : a new paradigm for cad-cam systems. In *Proceedings of 2nd International Conference on Integrated Design an Manufacturing in Mechanical Engineering (IDMME'98)*, pages 199–206. Kluwer Academic Publishers, 1998.

[Dcu93] D-Cubed Ltd, 68 Castle Street, Cambridge, CB3 0AJ, England. The Dimensional Constraint Manager*, Version 2.5*, May 1993.

[Dcu95] D-cubed. the 3d dcm part positioning product technical overview. Technical report, D-Cubed Ltd., January 1995.

[DMS95] J.-F. Dufourd, P. Mathis, and P. Schreck. Constructions géométriques dans un modeleur à base topologique. *Revue Internationale de CFAO et d'Informatique Graphique*, 10(4) :398–411, 1995.

[DMS97] J.-F. Dufourd, P. Mathis, and P. Schreck. Formal resolution of geometric constraint systems by assembling. In *Proceedings of the ACM-Siggraph Solid Modelling Conference*, pages 271–284. ACM Press, 1997.

[DMS98] J.-F. Dufourd, P. Mathis, and P. Schreck. Geometric construction by assembling solved subfigures. *Artificial Intelligence*, 99 :73–119, 1998.

[Dur98] Cassiano Durand. *Symbolic and Numerical Techniques for Constraint Solving.* PhD thesis, Purdue University, 1998.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1 : Equations and Initial Semantics*, volume 6. Springer-Verlag, Berlin, Germany, 1985.

[EY88] L.W. Ericson and C.K. Yap. The design of linetool, a geometric editor. In *Proceedings of the Computational Geometry Conference, LNCS 333*, pages 83–92. Springer-Verlag, 1988.

[Fud93] I. Fudos. *Editable Representations for 2D Geometric Design.* MSc thesis, Computer Science, Purdue University, 1993.

[Gin93] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1 :25–46, 1993.

[GW88] J. A. Goguen and T. Winkler. *Introducing OBJ3.* Technical report, SRI International, SRI-CSL-88-9, 1988.

[GWM+93] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and Jouannaud J.-P. *Introducing OBJ,* Tutorial and Manual, Computing Laboratory, Oxford University, 1993.

[JAMSR01] Robert Joan-Arinyo, Nuria Mata, and Antoni Soto-Riera. A constraint solving-based approach to analyze 2d geometric problems with interval parometers. In *Proceedings of the sixth ACM symposium on Solid Modeling and Applications*, pages 11–17. ACM Press, 2001.

[Kal91] K. Kalkbrenner. Elimination theory. Technical report, Research Institute for Symbolic Computation, Johannes Kepler Universität Linz, Austria, 1991.

[Kor99] Ulrich Kortenkamp. *Foundations of Dynamic Geometry.* PhD thesis, ETH Zürich, Institut für Theoretische Informatik, November 1999.

[Kra92] Glenn A. Kramer. *Solving geometric constraint systems : a case study in kinematics.* MIT Press, Cambridge, Mass., 1992. ISBN 0262111640, 1992.

[LAB+81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, Schaffert J. C., Scheifler R., and A. Snyder. Clu reference manual. *Volume 114 of Lecture Note in Computer Science*, 1981.

[Leb50] H. Lebesgue. *Leçon sur les constructions géométriques.* Gauthier-Villars, 1950.

[LK98] J. Y. Lee and K. Kim. A 2-d geometric constraint solver using dof-based graph reduction. *Computer-Aided Design*, 30(11) :883–896, 1998.

[LM95] H. Lamure and D. Michelucci. Solving constraints systems by homotopy. In *Proceedings of 3rd ACM/IEEE Symposium on Solid Modeling and Applications*, pages 263–269, 1995.

[LM97] H. Lamure and D. Michelucci. Qualitative study of geometric constraints. In B. Brüderlin and D. Roller Editors, editors, *Proceedings of Workshop on Geometric Constraint Solving and Applications, Technical University of Ilmenau, Germany*, pages 134–145, 1997.

[Mat97] P. Mathis. *Un système de résolution de contraintes par assemblage en modélisation géométrique.* PhD thesis, Université Louis Pasteur de Strasbourg, 1997.

[Mau80] C. R. F. Maunder. *Algebraic Topology.* Cambridge University Press, 1980.

[MSG97] B. Mazure, L. Saïs, and E. Grégoire. An efficient technique to ensure the logical consistency of interacting knowledge bases. *International Journal of Cooperative Information Systems*, 6(4) :27–36, 1997.

[Owe91] J. Owen. Algebraic solution for geometry from dimensional constraints. In *Proceedings of the 1st ACM Symposium of Solid Modelling and CAD/CAM Applications*, pages 397–407. ACM Press, 1991.

[PFTV88] W. H. Press, B. P. Flannery, S. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1988.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3) :268–299, 1993.

[SABC98] Lluis Solano Albajes and Pere Brunet Crosa. Geometric relaxation for solving constraint-based models. *Geometric Constraint Solving and Applications*, pages 405–425, 1998.

[SB91] W. Sohrt and B. Brüderlin. Interaction with constraints in 3d modeling. In J Rossignac and J Turner, editors, *Proceedings of Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 387–396. ACM Press, 1991.

[Sch93] P. Schreck. *Automatisation des constructions géométriques à la règle et au compas.* PhD thesis, Université Louis Pasteur de Strasbourg, 1993.

[SG94] H. Schumann and D. Green. *Discovering Geometry with a Computer using Cabri-Geometre.* Chartwell-Bratt, 1994.

[SS90] L. Sterling and E. Shapiro. *L'art de Prolog.* Masson, 1990.

[Sun86] G. Sunde. A cad system with declarative specification of shape. In *Proceedings of Eurographics Workshop on Intelligent CAD Systems*, pages 90–105, 1986.

[Sut63] I. E. Sutherland. Sketchpad : A man-machine graphical communication system. In *Proceedings of the IFIP Spring Joint Computer Conference*, pages 329–36, 1963.

[VSR92] A. Verroust, F. Schonek, and D. Roller. Rule-oriented method for parametrized computer-aided design. *Computer-Aided Design*, 24(10) :531–540, 1992.

[Wan01] D. Wang. *Elimination Methods.* Springer-Verlag, Wien New York, 2001.