# Interactive handling of a construction program in CAD

C. Essert-Villard, P. Mathis

*LSIIT, UPRES-A CNRS 7005*
*Université Louis Pasteur de Strasbourg, France*
*essert,mathis@dpt-info.u-strasbg.fr*

## ABSTRACT

*In CAD systems, symbolic geometric solvers allow to provide, for a constrained sketch drawn by the designer, a general construction program, that describes how and in which order the objects must be built. Then this program is interpreted to generate the required figure. If multiple solutions are produced, these kind of solvers generally allow to scan the entire space of the solutions found. After briefly recalling our sketch-based selection method, that enables to easily eliminate most of the solutions and to keep the only, or at the worst the few solutions that have the best likeness with the original drawing, we introduce a new step by step interpretation mechanism implemented as a debugger-like tool, that allows to browse the remaining solutions tree in order to help the user choosing the required solution.*

**Keywords**

*symbolic geometric constructions; constraint solving; computer-aided design; tree pruning*

## 1 INTRODUCTION

In Computer-aided design (CAD), a geometric object can be precisely described by constraints. They concern distances between points, angles between lines, tangency of circles and lines, etc. Generally, constraints are declaratively placed on a sketch. If we wish to carry out calculations, simulations or manufacturing, the object must really respect the constraints. Thus, a CAD system must be able to solve them and give the possible solutions. This kind of approach was initiated by I.E. Sutherland [9] with Sketchpad and was then studied by many authors.

Whatever the approach, a constraint system does not usually define a single figure. In the case of a well-constrained system, the exploration of the solutions space is not as easy as it seems. In most cases, CAD users only want one solution figure when they design an object. That's why an important matter of geometric solvers is identifying the solution that is most consistent with the user's expectations, as we can see in [3] and [7]. The most common response to this problem is the use of heuristics to filter the results. When using a numerical method, the constrained figure is compared with each of the numerical solutions. This is generally characterized by slow runtimes, and there is often more than one solution left. Our symbolic approach allows us to take advantage of the construction program to compare the sketch with a solution, and to define an easy-to-use debugger-like tool if the solution space has even so to be explored.

The rest of the paper is structured as follows. Section 2 presents the constraint solving framework. Then, Section 3 explains a basic construction program evaluation. Section 4 shows how the sketch can be used to find a good program's evaluation and Section 5 how using interactive tools. Finally, Section 6 concludes.

## 2 SOLVING WITH *YAMS*

*YAMS* (Yet Another Meta Solver) is the prototype resulting from the merging of the 3D topology-based geometric modeller *TOPOFIL*([2]) and a 2D geometric constraint solver. The construction programs (or construction plans) on which we work are supplied by the solver part of *YAMS*. That's why we present here quickly its functionalities.

The solver belongs to the family of symbolic solvers. The solving process acts in two steps: first, a symbolic phase that produces a *construction program* according to the constraints; then a numerical phase interprets this construction program. The symbolic stage is obviously the most costly.

### 2.1 Solving the constraints

Constraints are predicative terms of the form $P(x1, \ldots, xn)$, where $P$ is a predicative symbol, and $xi$ are typed identifiers of geometric elements. Then, denoting that the distance between a point $p1$ and another point $p2$ is a length $k1$ can be written $distpp(p1, p2, k1)$. About thirteen different kinds of constraints exist in *YAMS*. Among them, we distinguish metric constraints (such as distances, angles) and Boolean constraints (such as incidence or tangency). Note that this way of writing the constraints is quite usual, and can be found, for instance, in [1, 4].

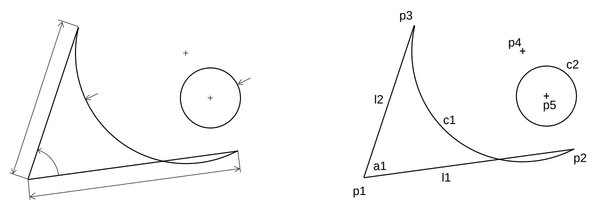


**Figure 1**: A sketch with constraints (left) and identifiers association (right)

During the symbolic solving, the numerical values of distances and angles are not taken into account, whereas they are given by the user with the rest of the constraints. They only appear in a symbolic way in the constraints under the form of typed identifiers (for instance $k1$ in the example above, to represent a length). The numerical values are associated to these identifiers by functional terms, in *definitions* of the form: $x := f(x1, \ldots, xn)$, where $x$ is the defined identifier, $f$ a functional symbol, and $xi$ the parameters that can be either other identifiers or numerical values. For example, if the user imposes a length to be 100 units from point $p1$ to point $p2$, we express it by a constraint $distpp(p1, p2, k1)$ and a definition $k1 := initl(100)$, where $initl$ initializes $k1$ to the value 100.

When capturing the data of a problem, these definitions are the first lines of the construction program (that is a list of definitions), that will be supplemented during the symbolic solving with other definitions. Let's take an example: Fig.1 shows the placing of the constraints on a sketch, and the identifiers association. The symbolic transcription of the constraints and the definitions for this problem are the following:

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

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

Our solver gives a geometric answer to this problem, that has the advantage of producing several solutions. The construction program given above expresses the geometric construction yielded by the solver, and describes, in the right order, the objects to build and the operations to apply so as to obtain a figure.

The numerical interpretation forms the subject of the rest of this paper. For more details on symbolic solving, see article [5] that explains this part more precisely, notably the original general mechanism of decomposition in subfigures and assembling that *YAMS* uses to solve large systems.

### 2.2 Construction program

In the construction program, the list of definitions is presented in *triangular solved form*, i.e. an identifier used as parameter in a definition must have been defined earlier in the program. Note that by switching two definitions in a construction program, it is possible to obtain an equivalent one, as long as the result is still in triangular solved form.

In a general way, a set of definitions can be structured as a Direct Acyclic Graph (DAG), called *dependence graph*. Its vertices are the definitions, and its oriented edges makes a link from a definition $\boldsymbol{x} = f(x1, \ldots, xn)$ to a definition $y = g(y1, \ldots, \boldsymbol{x}, \ldots, ym)$. A *topological sort* of a DAG gives a list of vertices such that a vertex does not appear in the list before its successors. For a DAG, there generally are several possible topological sorts which, in our case, correspond to the different possible construction programs. Note that all these possible programs provide exactly the same solutions, after a numerical interpretation.

Therefore, even if the solver gives a particular construction program, we can choose another order for the definitions, taking into account the dependencies, without affecting the solutions.

## 3 INTERPRETATION

### 3.1 Tree of interpretation

In this stage, the data given by the user are exploited as parameters for the numerical interpretation of the construction program.

Each functional symbol is associated with a numerical function. But interpretation of a functional term may provide multiple results. For example, the intersection between two circles, symbolized by *intercc*, generally produces two points, and *medradcir* that builds a circle through two known points, with a known radius, generally produces two different circles. So these are not simple functions, but what we call *multifunction*, i.e. functions that can give more than one result.

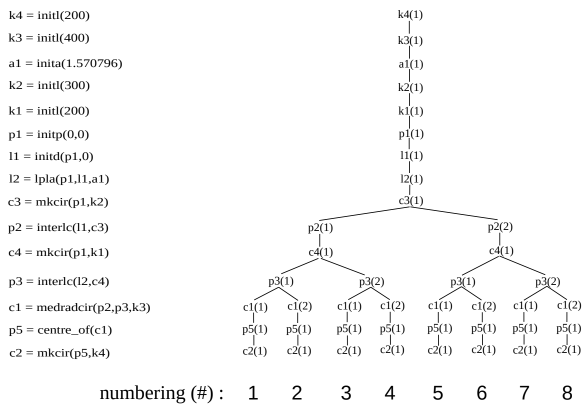
The existence of multifunctions in a construction program introduces choices in the interpretation process. So, we can consider the interpretation as the building of a tree labeled with numerical values. The interpretation of a multifunction that can produce up to $k$ results generates a branching of degree $k$. By giving a numbering to the various solutions produced by each multifunction, we number the branches of the tree. At the end, the tree represents the solution space, and one solution corresponds to the labels of one branch.

We have to distinguish two kinds of trees :

- the tree of the possible solutions, made by only taking into account the degrees of multifunctions, and whose number of branches is maximum. This one is called *tree of possibilities*
- the tree of the effective solutions, made by interpretation with real values parameters, and that may have less branches than the tree of possibilities. This one is called *tree of solutions*

The difference is caused by several kinds of events that may occur during the interpretation process. A multifunction may provide less results because of particular data (for example if two circles are tangent, the intersection has only one result), or even a “failure” (for example if those circles have no intersection). In this last case, the interpretation stops in the branch.

Note that practically, in our prototype, the tree is not really built but explored by a depth-first backtracking.



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

k4(1)
k3(1)
a1(1)
k2(1)
k1(1)
p1(1)
l1(1)
l2(1)
c3(1)
p2(1) p2(2)
c4(1) c4(1)
p3(1) p3(2) p3(1) p3(2)
c1(1) c1(2) c1(1) c1(2) c1(1) c1(2) c1(1) c1(2)
p5(1) p5(1) p5(1) p5(1) p5(1) p5(1) p5(1) p5(1)
c2(1) c2(1) c2(1) c2(1) c2(1) c2(1) c2(1) c2(1)
numbering (#) : 1 2 3 4 5 6 7 8

**Figure 2**: Construction program corresponding to Fig.1 and tree of solutions

## 3.2 Problems due to a high number of solutions

Even if the tree of solutions is lighter than the tree of possibilities, the number of solutions can be very important, and increases with the length of the construction program (that depends on the number of geometric entities of the sketch). That’s why, at first, we would like to minimize the size of the tree, in order to speed up the backtracking used to explore the tree.

A first pruning can be done by eliminating what we call the “false solutions”. Actually, the computed construction program enables to construct all the solutions as well as other figures which are not consistent with the constraints, because the geometric solver only uses necessary conditions to make the construction. This can be done with a simple test, by verifying if the constraints are satisfied.

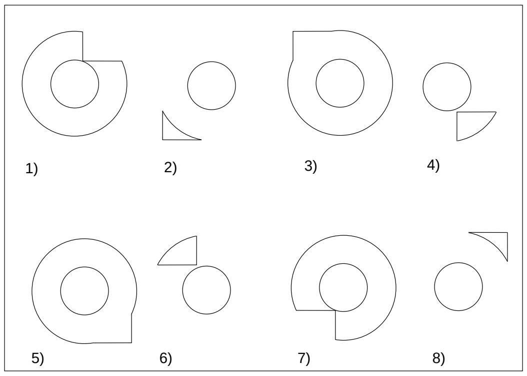


**Figure 3**: The generated solutions

The figures corresponding to the branches of the tree given in the previous section on Fig.2 are shown on Fig.3. Four of these solutions (numbered 3, 4, 5 and 6) can quickly be eliminated because the sign of angle $a_1$ is the opposite of what is given in the constraints. Moreover, among the remaining solutions, we can eliminate #7 and #8 that are identical to #1 and #2 apart from displacements.

But that may be insufficient. In the example presented on Fig.5, there are 32768 different solutions for a geometric object made of 15 equilateral triangles figure, but the solution space can not be reduced because all of the figures are consistent with the constraints. Other heuristics are necessary to drastically prune the tree of solutions, eliminating the figures that does not look like the sketch.

## 4 USING THE SKETCH

### 4.1 Usual criteria of likeness

Likeness is generally defined as conformity in appearance between things. Two figures are usually said to look like each other if some geometric properties are similar, such as:

- orientation of points,
- relative placing of objects,
- angles acuteness,
- convexity of some parts of the figure.

This definition is used in most of the CAD frameworks to eliminate inappropriate solutions.

However, most of these criteria can be held in check by some simple examples. For instance, Fig.4 illustrates that sometimes we are not able to decide between two solutions by only comparing the geometric properties : in this figure, all angles are acute and all points have the same relative placing and orientation.

### 4.2 Freezing of a branch

In order to eliminate a maximum number of solutions that does not look like the sketch, we proposed another definition of likeness (see [6]). This definition is based on the notions of geometric homotopy,

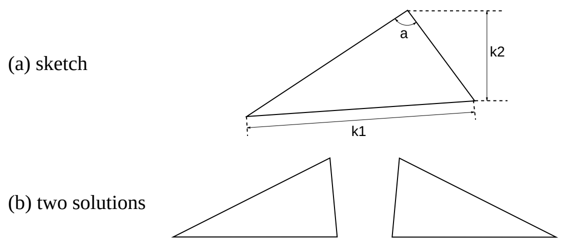


**Figure 4**: Lack of discrimination criterion

continuous deformation of a constrained system, and continuous numbering of the solutions.

For each metric multifunction we use in our solver, we described a particular continuous numbering of its distinct results. This continuous numbering allows us to use an original method to find the figure that has the best likeness with the sketch.

We first make an interpretation of the construction program, using as parameters the data measured on the sketch drawn by the user. This interpretation produces a tree of solutions, among which lies the branch corresponding to the sketch. We memorize the number of this branch. Then, it only remains to make another interpretation, using the user's data as parameters, and to follow the branch which number has been memorized. With the properties we explained before, we are sure that the figure we found has the same geometric characteristics than the sketch, and looks like it in the sense that we defined. We call this process *freezing of a branch*. The branch is selected and its number is kept for further interpretation, with new numerical values for the parameters. The other branches of the tree are not cut, so the other solutions are not lost and can be examined later.

This method give very good results when all the multifunctions used in the construction program are metric. As an example, Fig.5 shows a sketch made up of fifteen adjacent triangles. The lengths of all their sides are asked to be equal to a given dimension. This kind of configuration was studied by Owen [8], and is known to have $2^{p-2}$ distinct solutions, where $p$ is the number of points. In our case, with 17 points, we obtain 32768 solutions (triangles are often superposed, because their sides are equal). Some of them are presented on Fig.6. The corresponding construction program has 80 definitions, and the system contains 94 constraints. It takes more than 1 minute to calculate all possible solutions, whereas our method gives an instantaneous good answer, presented on Fig.7.

However, our method is not appropriate when Boolean constraints (such as tangency or equality between objects) are present in the construction program. Indeed, it is impossible to compare the solutions with the sketch when some information is missing in the sketch. Actually, unlike metric constraints that don't affect topology, these constraints are generally not respected on the sketch.

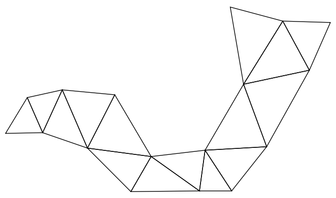


**Figure 5**: 15 triangles configuration: the sketch

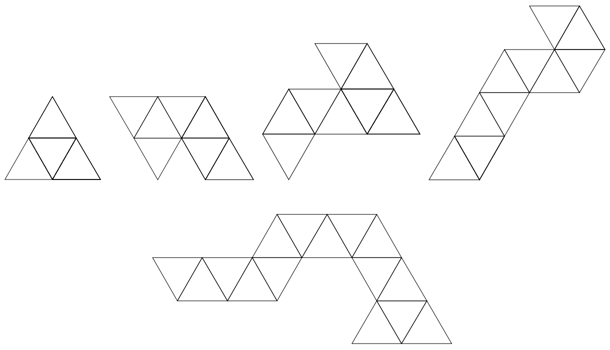


**Figure 6**: Five solutions among 32768 to "15 triangles"

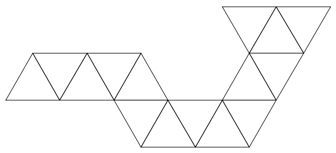


**Figure 7**: The required solution of the sketch given on Fig.5

Because of these Boolean constraints, some systems have a tree of solutions that can not be reduced to a single branch. Its number of branches can be decreased down to a few branches, but there still remains a little subtree to be explored. It may also happen that the user is not satisfied with what the solver found, whether the sketch he drew was not precise enough, or he did not expect such a solution for the constraints he gave. For all these reasons, the user may want to interactively explore, either the subtree of solutions, or the rest of the entire tree of solutions, and to examine solutions that are close to the frozen branch.

## 5 INTERACTIVE SOLUTION REFINING

All the above reasons led us to propose some functionalities to explore the solution space within *YAMS*. Remember this solution space is not simply a set of figures, but a structured space. The solutions tree and the construction program structures we use offer us the possibility to define an exploration tool, inspired by *debug tools* provided by most of the development systems in software engineering.

## 5.1 A step by step interpretation

First, remind that in the case where the user wants to explore the entire solution space, the number of solutions (i.e. of branches in the tree) can be very important. So, viewing the solutions one after the others may be a tedious task.

Suppose that the figure is not yet numerically computed. A good way to browse efficiently the solutions can be to explicitly choose, at each branching of the tree, which branch to follow, thanks to a step by step interpretation.

However, there are two kinds of definitions. Some definitions correspond to objects that can be seen by the user. In the rest of this paper, they will be called *sketch definitions*. Some other definitions correspond to auxiliary objects. For instance circles that are used to find a point, by making an intersection with a line. They will be called *auxiliary definitions*. As the geometric entities defined by auxiliary definitions are not drawn on the screen, it is difficult to choose which values to keep for them. Moreover, the user is not interested in the construction of intermediate objects, that has to be completely transparent to him. So, an idea is to make a step in the interpretation only at the sketch definitions.

At each step, we work on a *layer* (see Fig.8). In the layer, the last definition is a sketch definition, and the others are auxiliary definitions. The different possible values for the concerned object are proposed, and the user can choose one of them. It means that for this operation, a little subtree is explored. This subtree contains a few branchings corresponding to the auxiliary definitions within the layer to which a multi-function of *degree* $> 2$ is assigned. So, a backtracking is done into this layer, but this backtracking is hidden from the user, in order to make it transparent. Then, when a interpretation is chosen for the current sketch definition, the corresponding branch is frozen in this layer. See Fig.8, where the branch that has been frozen so far is in bold, the current studied layer is between dashed lines, and the visible objects in the sketch are framed.

The construction program may not be provided by the solver in the best form for this operation. It can be necessary to perform a topological sort of the program before the step by step interpretation.

Indeed, we need to have the following criterion on the construction program: let $d1$ and $d2$ be two sketch definitions, $d1$ being placed after $d2$ in the construction program, such that no other sketch definition exists between $d1$ and $d2$. Then, all definitions between $d1$ and $d2$, that are obviously auxiliary definitions, are the remaining definitions that are necessary to compute $d2$ and that have not been required before $d1$.

In order to obtain such a form, we have to sort the construction program. The topological sort is

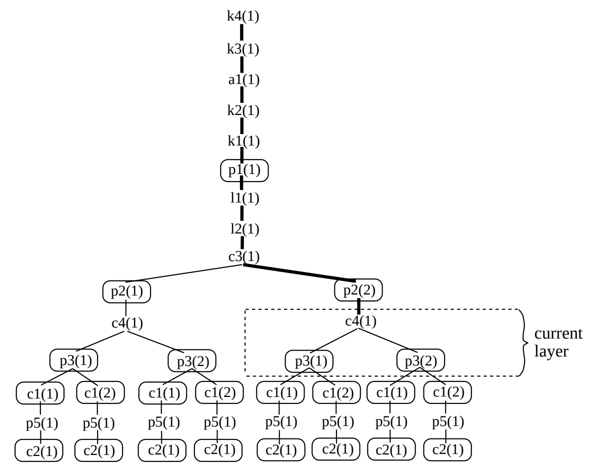


**Figure 8**: Backtracking on a little subtree, included in a layer of the solutions tree

made by placing first the sketch definitions following the current order, and then interleaving the auxiliary definitions just before the first sketch definition that needs it (i.e. that contains it as an argument).

When a construction program verifies the above criterion, the only backtracking to be done is located in the subtree between $d1$ and $d2$, excluding $d1$. If the user is not satisfied with the numerical interpretations proposed for $d2$, and wishes to see other possible solutions, then we are sure that some of the sketch definitions have to be thrown back into question.

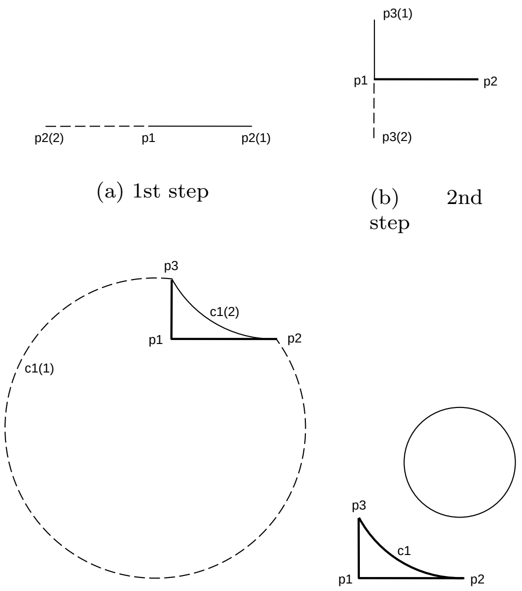
In such a case, we browse the sketch definitions that have been defined earlier, and on which $d2$ depends. We suggest to the user to reconsider some of the values he had chosen for these previous sketch definitions. First, we propose him to review only a few of them, those that are placed closer in the tree. Then, progressively we put into consideration more definitions, including those that were defined a longer time ago.

On Fig.9, we can see a step by step interpretation of the constrained sketch of Fig.1. At each step ((a), (b), or (c)), the user chooses one of the two available results. The part of the figure that has already been frozen is in thick, the chosen value is in thin, and the value that was not accepted is in dashed line.

## 5.2 Our debugger-like tool

The method exposed above is implemented as a module of *YAMS*, and the user has the choice to use it or not, and to start it when he needs.

Practically talking, we draw the solution step by step as the interpretation goes along. For each new object drawn on the solution figure, the corresponding part of the sketch is highlighted. This way, the user can easily follow the construction process. At each step, *YAMS* proposes a set of possible choices for the current object to be drawn. When the user chooses



(a) 1st step

(b) 2nd step

(c) 3rd step

(d) end of interpretation

**Figure 9**: Interpretation in 3 steps, and final result

one, it is constructed on the figure and *YAMS* goes on to the next step.

One of the objectives we want to reach, and that is being developed, is to be able to revise a solution that has already been completely constructed. So, we would be able to make first an interpretation with the "freezing of a branch" technique, and then either to choose between the remaining solutions (if there were Boolean constraints), either to suggest other close solutions if the user is not completely satisfied.

## 6 CONCLUSION

In this paper, we first exposed our symbolic approach of geometric constructions for CAD constraints solving. We explained that our prototype *YAMS* provides a general construction program, that is afterwards numerically interpreted. Then, after showing how we can prune the solution space represented by a tree, we put forward the remaining problems that led us to find a way to easily browse the solutions tree.

As a solution, we proposed a tool that is based on the idea of a step by step numerical interpretation. This debugger-like tool is used in case the pruning method did not manage to find one unique solution because of the presence of Boolean constraints, or in case the user is not satisfied with the solution. This mechanism can be enhanced with several kinds of breakpoint tools. Moreover, it is possible to offer the opportunity to freeze a part of a tree of solutions between two breakpoints, and then to skip this part as if it was a big step.

Our debugger-like tool is a first stage in the conception of a series of exploration tools. We plan to add another one that would be based on the idea of a *magnetic grid*. It would allow a more intuitive approach of the selection problem. On the basis of a solution, a user could drag a misplaced element of the figure towards one of the positions allowed by the tree of solutions.

## 7 REFERENCES

[1] B. Aldefeld. Variations of geometries based on a geometric-reasoning method. *Computer-Aided Design*, 20(3):117-126, 1988.

[2] Y. Bertrand, J.F. Dufourd. Algebraic specification of a 3D-modeller based on hypermaps. *Computer Vision - GMIP*, 56(1):29-60, 1994.

[3] W. Bouma, I. Fudos, C. Hoffmann, J. Cai, and R. Paige. Geometric constraint solver. *Computer-Aided Design*, 27(6):487-501, 1995.

[4] B. Brüderlin. Automatizing geometric proofs and constructions. *Proceedings of Computational Geometry'88*, LNCS 333, Springer-Verlag, Berlin, p.232-252, 1988.

[5] J.-F. Dufourd, P. Mathis, and P. Schreck. Formal resolution of geometric constraint systems by assembling. *Proceedings of the ACM-Siggraph Solid Modelling Conference, Atlanta*, p.271-284, 1997, ACM Press.

[6] C. Essert-Villard, P. Schreck, and J.-F. Dufourd. Sketch-based pruning of a solution space within a formal geometric constraint solver. *Artificial Intelligence*, 99:73-119, 1998.

[7] H. Lamure and D. Michelucci. Solving constraints by homotopy. *Proceedings of the ACM-Siggraph Solid Modelling Conference*, ACM Press, p.134-145, 1995.

[8] J. Owen. Algebraic solution for geometry from dimensional constraints. *Proceedings of the 1st ACM Symposium of Solid Modelling and CAD/CAM Applications*, p.397-407, 1991, ACM Press.

[9] I.E. Sutherland. Sketchpad: A man-machine graphical communication system. *Proceedings of the IFIP Spring Joint Computer Conference*, p.329-36, 1963.