Geometric Construction Problem Solving in Computer-Aided Learning

Pascal Schreck, Pascal Mathis, Julien Narboux

Geometric Construction Problem Solving in Computer-Aided Learning

Pascal Schreck
Université de Strasbourg
Strasbourg, France
Email: schreck@unistra.fr

Pascal Mathis
Université de Strasbourg
Strasbourg, France
Email: mathis@unistra.fr

Julien Narboux
Université de Strasbourg
Strasbourg, France
Email: narboux@unistra.fr

***Abstract*—Constraint satisfaction problems related to geometry mostly arise in CAD. But even though they are designed for geometry, none of the methods proposed to solve these problems fully meets the requirements needed by the educational domain. In this paper, we adapt CAD methods to education and show that results must be construction programs in order to take into account particular cases. We present then a framework implemented in Prolog as a knowledge-based system called Progé.**

I. Introduction

Roughly speaking, solving geometric construction problems consists in finding geometric figures which meet a given specification. The domain is essentially known through previous works in Computer-Aided Design (CAD) [1]–[4] and has been viewed as a particular case of Constraint Satisfaction Problems (CSP) under the name of Geometric Constraint Solving (GCS) with a slightly adapted formulation. It remains that geometry features make GCS a special field of research, particularly when considering Computer-Aided Learning (CAL).

In CAD, the problems are given under the form of a dimensioned sketch obeying to a pictorial norm. The expected result consists in numerical values that meet all the metric requirements given on the sketch. Several methods have been developed in CAD for solving geometric constraints. Some are very general and related to equation solving through either numerical methods [5], or formal algebraic methods [6]. The so-called *constructive methods* are more focused on geometry.

Apart from few exceptions [1], [7], all constructive methods for solving geometric constraints in CAD are based on a graph representation of the constraint system where the vertexes correspond to the objects to be found and the edges correspond to the constraints. A degree of freedom is assigned with each vertex (that is, roughly speaking, its number of free coordinates) and a degree of restriction with each edge (roughly speaking, the number of equations it represents). These pieces of information are used to sort the edges using the order of construction. This way, the constraint graph is used to operate a decomposition of the whole systems into smaller ones that can be translated into small equation systems. Various graph algorithms have been used: degree of freedom propagation, search of the tri-connected components or computing some flow on a bipartite graph.

Some constructive methods are closer to geometric constructions (see [1], [7] for instance): they build a plan of construction that is an ordered list of definitions $o_i = f_i(o_{j_1}, \dots o_{j_m})$ where for each o_{j_k}, $i > j_k$. In dynamic geometry programs such as GeoGebra or Cinderella [8], this construction plan is built interactively by the user and numerically evaluated each time a free object changes.

The problematic of CAD constraint solving is slightly different from CAL geometric construction problems. In education, the problems are given in a literal form, sometimes with an illustrative figure, and the teacher want the students to examine all the possible cases and in each case to find all the solutions . In CAD the statement of the problem is a dimensioned sketch and the user wants the solution which looks the most like the sketch (see [9] for more details on this subject).

In this paper, first we introduce the notion of *Program of Construction*, and we present a knowledge-based system which can solve construction problems.

The rest of the paper is organized as follows. Section II explains the problematic in greater details and gives an overview of our approach. Section III shows how the knowledge representation is structured in our system. Section IV describes specific engines that are used to update the working memory. Section V gives some examples whose construction has been found successfully by Progé. And section VI describes some improvements that our approach could make to the dynamic geometry programs used in CAL.

II. Overview of our Approach

A. Program of Construction

In education, solving a geometric construction problem is a process whose input is a literal statement, like the ones given in the example below, and whose output is a way to construct *all* the solutions in *any* case. Let us consider two examples.

Statement 1. *Construct a triangle* `(a,b,c)`*, given its base* `[a,b]`*, a base angle, say* α = $\angle$*(ab, ac) and sum* `k = ac + cb` *of the other two sides* [10].

The locus method gives easily the result: indeed `c` is one of the intersection points of the line `d` making angle α with base `[a, b]`, with the ellipse defined by its foci `a` and `b` and the length `k`. On the other hand, this result is not convenient for a high school student who has to achieve a ruler and

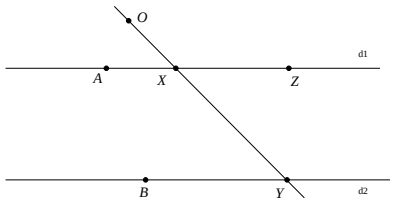

Fig. 1. A construction for statement 2.

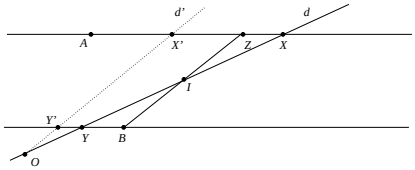

Fig. 2. A case where previous construction fails.

compass construction since an ellipse is in general not RC-constructible.The *construction* given in [10] is more suitable:

```
a, b, α and k are given
Let l be line (ab)
construct line l1 through a making angle α with l
construct circle c1 with center a and radius k
construct (p3, p4) as the intersection of c1 with l1
construct l2 the perpendicular bisector of [b,p3]
construct c as the intersection of l1 and l2
```

Nonetheless, considering this as the result of homework, the teacher could ask some questions like "how many solutions do we have?" or "what happens when `a=b`?".

The following example is more difficult to overcome:

Statement 2. *Given two different parallel lines `l1` and `l2`, and three points `a` on `l1`, `b` on `l2`, and `o` in a general position, construct a line `l` passing through `o` and cutting `l1` at `x` and `l2` at `y` such that `ax + by` equals a given length `k`.*

Using the same trick than in example 1, we construct point `z` at distance `k` from `a` on `l1`. It is easy to see that `(xzby)` is a parallelogram (Fig. 1), and we get the construction:

```
a, b, o, direction d1 and k are given
construct l1 = line through a with direction d1
construct l2 = line through b with direction d1
construct c1 = circle with center a and radius k
construct (z1, z2) = intersection of c1 and l1
construct i = midpoint of z1 and b
construct l = line passing through o and i
```

Unfortunately, it could happen that `(xzby)` is not a parallelogram (see Fig. 2), but that `(xzyb)` is! Different constructions must be done according to the cases. Taking the previous remarks into consideration, a more accurate answer consists in the following program of construction

```
a, b, o, direction d1 and k are given
construct l1 = line through a with direction d1
construct l2 = line through b with direction d1
construct c1 = circle with center a and radius k
construct {z1, z2} = intersection of c1 and l1
foreach z in {z1, z2} do
 foreachcase do
   case((xzby) is a parallelogram) :
      construct i = midpoint of z1 and b
      if (o <> i) then
       construct l = line passing through o and i
       verify_statement
      else fail by under-constriction
   case((xzyb) is a parallelogram) :
      construct l = line passing through o and
                    parallel to line (zb)
      verify_statement
  end cases
end foreach
```

Note that, since we use necessary conditions to build this program, we have to verify, numerically or formally, if the statement is really fulfilled.

This shows that the result of a geometric construction problem should not be a plain sequence of constructions but a *program* with specific control flow structures (`foreach` and `foreachcase`) to deal with case distinctions.

B. Knowledge-based system

We consider the framework of Euclidean geometry with typed variables and parameters: points, lines, circles, distances, angles, directions of lines. Such objects are used with functional and predicative symbols that express constraints and define objects.

The reasoning that leads to a program of construction from a statement is based on geometric rules. These rules are implemented in a knowledge-based system. We find then the usual components of such a system with a *knowledge base*, a *working memory* and an *inference engine*. Several important aspects should be taken into account for the implementation. The construction program must remain readable by a student, but many deductions related to some basic geometric properties (commutativity of some relations, *etc*.) may increase drastically the program size. There are also some useless deductions that should not clutter up the program.

Thus specific mechanisms for geometry must be integrated into the knowledge-base system. We found them in the knowledge base which contains on one hand *low level rules* that express geometric properties which are usually considered as assumptions and on the other hand *high level rules* that state geometrical theorem having the form `if ... then [either] ... [or ...]`. The working memory is also decomposed into two components: a *formal figure* containing useful properties for the construction but that does not appear in it; and a *formal reasoning* which is a dynamic graph to prevent loops. The inference engine relies on *modus ponens* but with an extended notion of unification that we call *unification modulo* the formal figure. Moreover, it contains a *case distinction engine* which tries to detect equalities between objects

C. Example

We describe here the beginning of the resolution for the previous construction problem. The formal figure is initialized with the following facts:

```
a is a point lying on l1, a is known
b is a point lying on l2, b is known
o is a point lying on l, o is known
x is a point lying on l1 (with a degree of freedom = 1)
y is a point lying on l2 (with a dof = 1)
di1 is a known direction related to l1 et l2
l1 is a line containing a and x with direction d1
   it is known (dof = 0)
```

```
l2 is a line containing b and y with direction d
   it is known
l is line containing o, with a dof=1
   x and y are on l
d1 is a length equal to ax
d2 is a length equal to by
k is a length equal to d1+d2, k is known
```

and in the *formal reasoning graph*, we have:

```
k = ax+by
```

Then, the inference engine starts using a data-driven search. A rule that can possibly be applied is the following one:

```
 300 # if [dist(A,M)+dist(B,N)=L]
           and [known line(A,M), known L]
      then  [
           X naming interlc(line(A,M), ccr(A, L),
           dist(X,M) = dist(B,N) ].
```

Or, in natural language, if the sum of the lengths AM and BN and the line (AM) are known, the construction can be helped by considering point X on (AM) at distance L from A, since we get the new fact XM = BN. A special unification is used (see below) giving A=a, M=x, B=b, N=y, L=k, then line(a,x) is identified as l1 since a and x are on l1. The system checks if l1 and k are known, the rule is applied and new knowledge is added into the two databases:

```
c1 is a circle with center a and radius k, dof=0
     c1 contains p1
p1 is a point defined as the intersection of l1 and c1
     its dof=0, it is on l1 and on c1
the knowledge about l1 is completed: l1 contains p1
the knowledge about d2 is completed: d2=dist(p1,x)
```

and for the formal reasoning graph:

```
d2 = dist(p1,x)
```

In a next step, our system considers the following rule:

```
  200 # if [dist(A,B) = dist(C,D),
           dirl(line(A,B)) = dirl(line(C,D))]
and
 [alldifferent [A,B,C,D]]
      then
either [] and [pll(A, B, C, D)]
       or
either [] and [pll(A, B, D, C)].
```

which is selected thanks to our unification mechanism: the working memory contains the fact d2 = dist(p1, x) which does not unify with dist(A,B) = dist(C,D). But using data stored in the formal figure, the system finds out that d2 = dist(b,y) and d2 = dist(p1, x). d2 is then replaced by one of these terms and the unification process is resumed. Considering dist(p1, x) = dist(p1, x) leads to the failure of the control term alldifferent [A, B, C, D] and the second choice is made by matching the fact dist(p1, x) = dist(b, y) giving the unifier A=p1, B=x, C=b and D=y. The inference engine has then to test if this unification verifies dirl(line(A, B)) = dirl(line(C,D)). Again the figure is used to prove that line(p1, x) = l1, line(b, y) = l2, and di1 = dirl(l1) = dirl(l2). Since the test alldifferent [A, B, C, D] succeeds, the rule is launched.

This rule consider two segments [A,B] and [C, D] which are parallel and have the same length,then, either (A, B, C, D) or (A, B, D, C) is a parallelogram. The empty lists before the keyword and indicate that these conclusions are not subject to additional conditions. The inference engine has to deal with this alternative. The two possibilities are explored: two constructions are launched and a foreachcase structure is introduced in the construction program.

The rest of the construction(s) uses classical properties for parallelograms which are expressed as high level rules in our system.

functionnal term	intuitive semantics
interll(l1,l2)	intersection of line l1 and l2
intercl(c1, l1)	list of the 2 intersections of circle c1 and line l1
intercc(c1, c2)	list of the 2 intersections of c1 and c2
mid(p1, p2)	midpoint of points p1 and p2
center(c1)	center of circle c1
line(p1, p2)	line through p1 and p2
lpdir(p1, di1)	line through p1 with direction di1
bis(l1, l2)	list of 2 bisector lines of l1 and l2
ccr(p1, d1)	circle with center p1 and radius d1
ccp(p1, p2)	circle with center p1 passing through p2
dist(p1, p2)	distance between points p1 and p2
dipl(p1, l1)	distance from point p1 to line l1
radius(c1)	radius of circle c1
+, -	simple arithmetic operators
dirl(l1)	direction of line l1

predicative term	intuitive semantics
x = y	equality
x is_on y	incidence relationship
iso(p1, p2, p3	(p1,p2,p3) is isosceles triangle
tangentcc(c1, c2, p1)	circles c1 and c2 are tangent at point p1
pll(p1, p2, p3, p4)	(p1,p2,p3,p4) is a parallelogram

TABLE I
SOMEBASIC CONSTRUCTIONS AND GEOMETRIC CONSTRAINTS

D. Geometric Universe

Usually, only straightedge and compass are authorized in geometric constructions. But, in the classrooms, others tools corresponding to basic constructions are allowed. We consider here such constructions involving points, lines, circles, lengths, directions and angles. Table I gives some constructions and some constraints considered in our system.

If one wants to draw figures or to verify some properties on numerical data, one has to consider numerical semantics where coordinates are given to geometric types and functional and predicative symbols are interpreted by numerical functions. In our case, a "formal semantics" is needed and we use a set of axioms describing the relationships between the notions formalized by the functional and predicative symbols.

III. KNOWLEDGE REPRESENTATION

A. Logic

In this work, we adopted a pragmatic point of view. We use logical statements as an intermediate language to express usual knowledge coming from mathematical textbooks, in

order to write them in Prolog. In this framework, functional symbols represent basic constructions and relational symbols, including equality, the constraints used to describe the figures. The geometric knowledge can be then expressed as an axiom set like the following one (the upper-case letters indicate universally quantified variables):

```
(c1) dist(X,Y) = dist(Y,X)
(c2) mid(X,Y) = mid(Y,X)

(i1) X is-on-line Z ∧ Y is-on-line Z ∧ X ≠ Y ⊃
     Z = lpp(X,Y)
(i2) X = mid(A,B) ∧ A ≠ B ⊃ X is-on-line lpp(A,B)

(d1) X is-on-line lpdir(X, Di)
     ∧ Y is-on-line lpdir(Y, Di)
     ∧ lpdir(X, Di) ≠ lpdir(Y, Di) ⊃ X ≠ Y


(r1) (pll(A, B, C, D) ⊃ mid(A, C) = mid(B, D)

(r2) (pll(A, B, C, D) ⊃ dirl(lpp(A,B)) = dirl(lpp(C,D))
         ∧ dirl(lpp(A,C)) = dirl(lpp(B,D))

(r3) (dist(A,B)=dist(C,D) ∧ dirl(lpp(A,B))=dirl(lpp(C,D))
        ∧ A ≠ B ∧) lpp(A,B) ≠ lpp(C,D))
     ⊃ (pll(A, B, C, D) ∨ pll(A, B, D, C))
```

In the next section, we show how we put a structure on this axiom set in order to implement our framework.

B. Low Level Knowledge

As said above, some axioms seem to be useless when explaining a construction, for instance making clear that dist(a, b) = dist(b, a) in a large construction is pointless. Previous axioms (c1), (c2), (i1) and (i2) given above fall in this category. In addition, there are some pieces of knowledge that are used for controlling the solving process and do not deal with pure syntactic logic, for instance the degree of freedom of points is 2 since we aim the 2D-plane as a model for our geometry, the same way the incidence relationships have a degree of restriction equal to 1. We choose to associate this kind of knowledge with the signature of the theory rather than to explicitly write them as high level rules. This way, this kind of knowledge can be used by very low level mechanisms like unification, search in and modification of the working memory.

Precisely, here are some examples of attributes linked to geometric types and coded as Prolog facts:

```
dmax(point,2).
autom(line,D,[dir :: Di names dirl(D)], [lpdir(nul, Di)]).
```

The first clause indicates that a free point has a degree of freedom equals to 2, and the second one says that when the system introduces a line D in the working memory, it has also to introduce its direction Di, then D has direction Di.

Now, the next piece of Prolog code shows different sorts of attributes linked to the functional symbols.

```
/*1*/ rmult(intercl).
/*2*/ decomp(line,[point/incid, point/incid]).
/*3*/ 'update:' M eq mid(A,B) ==> [M is_on line(A,B)].
/*4*/ Term equiv Term.
/*5*/ mid(A,B) equiv mid(B,A).


/*6*/ deg_titre(incid, 1).

/*7*/ D eq line(A,B) 'except:'
 [[A diff B] >> [], [A eq B] >> [def(D,1,[B/point/incid])]].
```

Clause number 1 indicates that `interlc` is a functional symbol noting a multi-function computing a list of results. When used as a definition in a program of construction, it leads to an iterative structure:

```
li1 := intercl(c1,l1)
foreach p1 in li1 do ...
```

Clauses number 2 and 3 essentially code axioms (i1) and (i2). Clause number 5, precises that the midpoint of A and B is to midpoint of B and A. Clause number 6 defines the degree of restriction of an incidence relationship, for instance, if an unknown line passes through a known point, then its degree of freedom will be decreased by 1. Finally, clause number 7, expresses the preconditions linked to the symbol functional "line"(See below).

C. High Level Axiom Set

The axioms whose use is made visible are translated by rules represented by Prolog clauses such the ones shown at section II. The general syntax is the following:

```
<number> # if <list of the premises>
         and <list of control terms>
         then < list of actions/assertions >
```

where keywords are in boldface and defined as Prolog infix operators. The list of premises has to be instantiated using the facts in the working memory as in usual expert systems. In order to avoid to infer useless facts, the control list is used to guide the application of the rule: through these controls the inference engine can test if a geometric object is yet constructed (`known`) or not (`not_known`), if they are different if the current context (`alldifferent`). The rule is launched if all tests succeed. Finally, the conclusion list contains the new facts to be added in the working memory, in the case of classical geometric inference, pieces of construction with the possibility to introduce auxiliary objects or control actions, like inhibiting some rules.

In addition, in order to take special cases into consideration, we define the notion of disjunctive rule where several conclusions are possible. They follow the syntax:

```
<number> # if <list of the premises>
         and <list of control terms>
         then
           either <list of additional premises>
                       and < list of actions/assertions >
         or
           either <list of additional premises>
                       and < list of actions/assertions >
           ...
```

Each conclusion beginning with keyword `either` indicates a particular case which is defined by the list of additional premises. This list can be empty like for the rule labeled 200 given at section II. The list of actions or assertions after the `and` keyword has the same meaning as a standard rule.

D. Exceptions

Degenerate cases appear everywhere in geometry. For instance the axiom `i1` contains preconditions to functional

terms. We also saw that in our knowledge structure, such preconditions are directly connected to the right functional symbol. We designed a special automatic prover to take these preconditions into account. In fact, when a functional term with precondition acts as a definition in a program of construction, this special prover try to prove either that the preconditions are fulfilled or, if it fails, that the negation of the preconditions are fulfilled. If the prover fails in the two cases, they are both considered and this gives birth to two branches in the program of construction. This special prover uses special knowledge specially to prove that some objects are different. To this end, Progé owns a small base of rules translating axioms like axiom (d1) in the list above.

IV. Knowledge Management and Program Synthesis

In Progé, the knowledge which defines the geometric universe is implemented into a set of Prolog facts that the expert user can modify in order to add high level rules, but also geometric objects (as conics for instance) or functional symbols.

In addition, to this "global" geometric knowledge, Progé has to deal with specific knowledge coming from the current problem of construction. The working memory is divided into two parts: a formal figure and a formal reasoning graph.

Formal Figure. The formal figure is a bookkeeping of all inferred information that a human could read on a graphical figure that he would have drawn. This includes incidence relationships, equalities between objects, the degree of freedom of each object and if the object is known, the term which defines it. As mentioned, the formal figure is used by our unification process but also to propagate the degrees of freedom each time an object is constructed. It can also be inspected for explanation purposes.

Formal figure is queried and updated very often. For instance, each time that a functional or predicative term is considered, it is normalized by a process we called *atomization* and consisting in naming all its sub-terms including itself. For instance, considering the rule labeled 300 and the example given at section II, the system has to consider the directive:

```
X naming interlc(line(a,x), ccr(a, k))
```

The term `interlc(line(a,x), ccr(a,k))` is then recursively *atomized*: the system search a name for `line(a,x)` and since the formal figure contains the fact that points `a` and `x` are on line `l1`, term `line(a,x)` is replaced by `l1`; the system searches then a name for term `ccr(a, k)` and since there is yet not such object stored in the formal figure a new object automatically named `c1` is created, it is of type circle, its center is `a` and is radius `k`, name `c1` is returned. Then, resulting term `interlc(l1, c1)` has to be atomized: a new object is created with name `p1` and two incidence informations about `l1` and `c1`. Finally variable `X` is instantiated by `p1`.

It can also occur that the system discovers that two objects first considered as different are actually equal, for instance by considering a degenerate case. The formal figure has then to be strongly updated by merging the two objects and using a table of synonyms.

Formal Reasoning Graph. The history of the construction is stored by an hypergraph whose nodes are the high level facts and those hyperedges correspond to the high level rules.

Formal reasoning is used to instantiate high level rules through our extended unification mechanism, and it is updated when rules are applied. Keeping the history serves several purposes. First, it can be used as a source of explanations about how a particular construction has been found. Second, it prevents infinite loops during the data-driven search.

Moreover, Prolog follows a deep first search which is known for being an unfair inference strategy. To avoid this behavior, we use a heuristic method that penalizes the facts which have led to many deductions and which have allowed to construct some searched objects. Given a certain level determined by the system, the facts that own a weight greater than this level cannot be used. When all the rules fail at this level, it is incremented and then, another deduction cycle can begin. A maximum level is imposed to avoid any infinite loop.

Exception and Particular Cases. The *case distinction engine* deals with non degeneracy conditions and disjunctive rules. When the inference engine needs to apply a rule, the case distinction engine is invoked to take care of the non-degeneracy conditions. It tries to prove the non-degeneracy conditions or its negation as indicated above. If none of those two conditions can be proved in the current context, a conditional statement will be introduced in the construction program in order to perform case distinction.

The *geometric program synthesizer* generates the construction program from the *formal figure*. At the end, for each case, either the system succeds and we get a construction program or it is a failure. There are many reasons for a failure: either the case is impossible, or the assumptions in this case lead to an infinity of solutions, or also Progé is not powerful enough to find a solution Note that examining degenerated cases, is difficult and often leads to the failure on such peculiar branches, even if the general case is successfully treated.

V. Results and Discussion

Our implementation of the approach described in this paper consists in a prototype, Progé, with about 5000 Prolog lines including the description of the basic geometric universe.

A. Results

Progé has been successfully experimented on about 30 classical exercises taken from textbooks: some examples are given in Table II where letters `a`, `b`, `c`, `e`, `f` and `o` represent points; `l`, `l1`, `l2`, ... lines; `c1`, `c2` ... circles and `r`, `k` lengths.

The algorithms used in Progé are not efficient. For instance, our unification algorithm shares some similarities with AC unification which is NP-hard. However, the complexity remains polynomial but with a high degree. Fortunately, most of the time the considered problems are small and Progé can solve them within about 1s on a standard personal computer.

Given	Searched	Constraints
a, b, c	c1	c1 is circumscribed to (a, b, c)
p1, p2, p3, r	c1	c1 tangent to ccr(p1, r), ccr(p2, r), ccr(p3, r)
a, c1	l	a is on l l tangent to c1
b, c	a	(a, b, c) is isosceles (ab) $\perp$ (bc)
b, c	a	(a, b, c) is isosceles (ab) $\perp$ (ac)
a, b, c1	e, f, l1	a, e and f are on l1 e and f are on c1 bf = be

TABLE II
EXAMPLES OF SOLVED PROBLEMS

B. Related Works

Other researchers have considered geometric constructions in link with the domain of automated proof in geometry.

Matsuda and VanLehn [11] propose a tool for proving automatically geometric theorems whose proof requires basic constructions but they do not focus on the construction problem but on proving theorems. Chou, Gao and Zhang propose an approach based on a heuristic programmed using Prolog to solve the construction problem [12]. Their approach differs from ours in several aspects: they do not take degenerate cases into account and they only consider points. As we are able to manage lines and circle directly, we believe that our approach can be more easily adapted to an interactive setting. Vesna Marinković and Predrag Janičić have proposed a tool which is specialized in triangle constructions [13], but they do not deal with degenerated cases. More recently Gulwani, Anand Korthikanti and Tiwari propose a method to solve geometric construction problems using a program synthesis approach [10]. As Chou, Gao and Zhang, they overlook the fact that there are different cases to consider. Our experiences show that generating constructions for each cases is much harder.

VI. CONCLUSION

In this paper we described an approach for modeling and using geometric knowledge in order to solve geometric construction problems in the e-learning sense: non-degeneracy conditions are taken into account, all particular cases are examined and all the solutions are searched. Moreover, explanations about the construction process can be extracted.

From the point of view of the knowledge representation, one of the main characteristics of our approach consists in distinguishing low level knowledge, used by fundamental operations of the domain, from "readable" knowledge used by a high level inference engine. Preconditions linked to functional symbols are one example of low level knowledge systematically used here to take degenerate cases into consideration and allowing to build robust geometric construction programs.

We believe that the most fruitful applications of this work lie in adding more formal geometric semantics to dynamic geometry and useful tools like the following ones:

- solving or helping: with an *ad hoc* user interface for editing geometric construction statements, our approach could be used for automatically solving them, or, more interesting, to assist a student in solving such problems by giving him some hints for the next move or by putting in light degenerate cases.
- verifying constructions is another kind of help for a student. by using algebraic tools like Wu's method [14] or by using classical methods [16]. These methods could be helped by our constructive approach.
- giving explanations as mentioned earlier in the body of the text.

Progé is a prototype intended as a proof of concept. It succeeded in this task. A significant work remains to transform it into a really usable software.

REFERENCES

[1] J.-F. Dufourd, P. Mathis, and P. Schreck, "Geometric construction by assembling solved subfigures," *Artificial Intelligence Journal*, vol. 99(1), pp. 73–119, 1998.

[2] X.-S. Gao and S.-C. Chou, "Solving geometric constraint systems. i. a global propagation approach," *Computer-Aided Design*, vol. 30, no. 1, pp. 47 – 54, 1998.

[3] C. M. Hoffmann, A. Lomonosov, and M. Sitharam, "Decomposition Plans for Geometric Constraint Problems, part II : New Algorithms," *J. Symbolic Computation*, vol. 31, pp. 409–427, 2001.

[4] C. Jermann, G. Trombettoni, B. Neveu, and P. Mathis, "Decomposition of geometric constraint systems: a survey," *International J. of Computational Geometry and Application*, vol. 16, no. 5-6, pp. 379–414, 2006.

[5] H. Lamure and D. Michelucci, "Solving constraints by homotopy," in *Proceedings of the ACM-Sigraph Solid Modeling Conference*. ACM Press, 1995, pp. 134–145.

[6] K. Kondo, "Algebraic method for manipulation of dimensional relationships in geometric models," *Computer Aided Design*, vol. 24, no. 3, pp. 141–147, 1992.

[7] B. Aldefeld, "Variations of geometries based on a geometric-reasoning method," *Computer-Aided Design*, vol. 20, no. 3, pp. 117–126, 1988.

[8] K. Ulrich and R.-G. Jürgen, "Making the move: The next version of cinderella," in *Proceedings of the First International Congress of Mathematical Software, ICMS 2002*, A. M. Cohen, X.-S. Gao, and N. Takayama, Eds. World Scientific, 2002, pp. 208–216.

[9] C. Essert-Villard, P. Schreck, and F. Dufourd, J., "Sketch-based pruning of a solution space within a formal geometric constraint solver," *Artificial Intelligence*, vol. 124, pp. 139–159, 2000.

[10] S. Gulwani, V. A. Korthikanti, and A. Tiwari, "Synthesizing geometry constructions," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. San Jose, California, USA: ACM Press, 2011, pp. 50–61.

[11] N. Matsuda and K. VanLehn, "Gramy: A geometry theorem prover capable of construction," *Journal of Automated Reasoning*, vol. 32, no. 1, pp. 3–33, 2004.

[12] S.-C. Chou, X.-S. Gao, and J.-Z. Zhang, *Automated reasoning and its applications*. Cambridge, MA, USA: MIT Press, 1997, ch. Automated generation of construction steps for geometric constraint problems, pp. 49–69.

[13] V. Marinković and P. Janicic, "Towards Understanding Triangle Construction Problems," in *CALCULEMUS 2012*, Bremen, Germany, 2012.

[14] S.-C. Chou, *Mechanical Geometry Theorem Proving*. D. Reidel Publishing Company, 1988.

[15] J.-D. Génevaux, J. Narboux, and P. Schreck, "Formalization of wu's simple method in coq," in *Proceedings of the First International Conference on. Certified Programs and Proofs, CPP 2011*, J.-P. Jouannaud and Z. Shao, Eds. Springer-Verlag, 2011, pp. 71–86.

[16] P. Janicic, J. Narboux, and P. Quaresma, "The Area Method : a Recapitulation," *Journal of Automated Reasoning*, vol. 48, no. 4, pp. 489–532, 2012. [Online]. Available: http://hal.inria.fr/hal-00426563