EDITABLE REPRESENTATIONS FOR 2D GEOMETRIC DESIGN

A Thesis
Submitted to the Faculty

of

Purdue University

by

Ioannis Fudos

In Partial Fulfillment of the
Requirements for the Degree

of

Master of Science

December 1993

ACKNOWLEDGMENTS

I would like to thank my advisor Prof. Christoph Hoffmann for his valuable help and support, Bill Bouma for developing an excellent user interface for the 2D sketcher, Jianchen Cai and Robert Paige for many helpful discussions, and the members of my committee Elias Houstis and Jorg Peters for their help. Also special thanks to Evaggelia Pitoura for her comments and suggestions regarding this thesis.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

LIST OF FIGURES v

ABSTRACT vii

1. INTRODUCTION AND RELATED WORK 1

1.1 Trends in two dimensional sketching 2
1.1.1 The descriptive approach 2
1.1.2 The constructive approach 2
1.1.3 The declarative approach 3
1.2 Constraint solving methods 5
1.2.1 Numerical constraint solvers 5
1.2.2 Constructive constraint solvers 6
1.2.3 Propagation methods 7
1.2.4 Symbolic constraint solvers 9
1.2.5 Solvers using hybrid methods 9
1.2.6 Other methods 10
1.3 The repertoire of constraints 10

2. THE STRUCTURE OF THE EREP 2D SKETCHER 12

2.1 System architecture 12
2.2 Representation 14
2.2.1 Representation of geometries : points, lines and circles 16
2.2.2 Topologies : vertices, tlines, tcircles, segments, arcs and rays . 17
2.2.3 The overall textual representation of a cross section 18
2.2.4 More on geometries and topologies 20
2.2.5 Constraints 23
2.2.6 Display of constraints 25
2.2.7 Higher level of topology 26
2.3 The user interface 28

Page

3. THE CONSTRAINT SOLVING IN EREP 35

3.1 Basics 35
3.1.1 The algebraic equations describing distances and angles 37
3.1.2 Some more definitions 38
3.2 The constraint solving method 39
3.2.1 Finding the sequentially constructible subgraphs 39
3.2.2 Recursion 40
3.2.3 Construction steps 41
3.2.4 Graph transformations 43

4. ROOT IDENTIFICATION 44

4.1 Root identification for iterative methods 47
4.2 Specifying the correct solution by imposing general rules 47
4.3 Specifying the correct solution by over-constraining 48
4.4 The rules for root identification in EREP 50
4.5 Advantages and disadvantages of the approach adopted in EREP . . 55
4.6 An interactive framework for root identification 60

5. CONCLUSIONS AND FUTURE WORK 69

BIBLIOGRAPHY 71

LIST OF FIGURES

Figure Page

1.1 The parallel evolution of three areas 4

2.1 The overall architecture of the 2D sketcher 12

2.2 The structure of the constraint solver 13

2.3 The hierarchy of objects in EREP 15

2.4 The UI panel 28

3.1 A sketch with 10 geometries constrained by 17 constraints 36

3.2 The constraint graph of the previous sketch 37

3.3 Finding the clusters of the graph 40

3.4 Point placement rules 41

3.5 Incidence of `A` with `a` and `C` with `c` and resulting constraint graph. 43

4.1 A well constrained sketch with 3 lines and 3 points 45

4.2 This solution is consistent but not intuitive 45

4.3 A well constrained sketch with 2^{n-2} solutions 46

4.4 Constraining three parallel lines with distances 49

4.5 An instance of the n parallel lines problem 49

4.6 Two lines constrained to lie at distance 5 51

4.7 Two lines at 45° angle 52

4.8 Root identification in placing three points 53

4.9 Root identification in placing a point and two lines 54

Figure Page

4.10 Root identification in placing two points and one line 55

4.11 A well constrained, serially constructible sketch 57

4.12 A degenerate configuration 57

4.13 Setting the parameters for an instance of the problem 59

4.14 The user changes the angle from 90° to 190° 59

4.15 The user gets back a rather unintuitive solution 59

4.16 The initial sketch with two 70° angles 62

4.17 After making the two angles 30° 63

4.18 Selecting solution 4 (instead of 2) from level 7 65

4.19 Selecting solution 1 (instead of 3) from level 4 66

4.20 Complementing the arc Ar_{10} 67

ABSTRACT

Fudos Ioannis. M.S., Purdue University, December 1993. Editable Representations for 2D Geometric Design. Major Professor: Christoph M. Hoffmann.

The user of a 2D sketching system would like to be able to specify with minimum effort what he has in mind. For this reason, we developed the notion of a representation, which is a high-level, concise, constraint-based description that contains all the information needed to specify a sketch uniquely. In addition, this representation is naturally capable of modifications and is supported by a fast constraint solving algorithm and an interactive framework for root selection. We call this representation EREP (Editable REPresentation).

1. INTRODUCTION AND RELATED WORK

In the past few years, there has been a strong trend in the research community ([Lel88]) for developing a system that performs automated design and has the following properties:

1. The design process is natural for the user even if the user is not aware of the underlying representation and technical aspects of the system.
2. A designed object is subject to modification and interrogation by the user.
3. An object is reusable, in the sense that the user may want to use the object as a functional part of a larger object.
4. An object can be used in subsequent phases (analysis, robot motion planning, assembly), which implies that information related to the user's intentions has to be preserved (see also [SKW90]).

The above trend is related to "design by features". Design by features is a notion rather than a formally defined process. It means that certain areas have a specific semantic description that makes them capable of being modified or of functioning in a predetermined (but not fixed) way when linked in a certain environment [Woo90, Rol89a]. The traditional Boundary representation (Brep) and Constructive Solid Geometry (CSG) represenation impose major limitations and exhibit unacceptable performance when used in a system that tries to accomplish the above goals. For conclusions regarding the use of CSG and Brep in such systems, the reader may refer to [Woo90], [Rol89a], [HJ93], [Hof92], [Woo88].

One powerful method for constructing three dimensional features is to create a two dimensional sketch and then use a number of standard operations with predefined

semantics such as extrusion or sweep. In this work we concentrate on defining an Editable REPresentation for two dimensional sketches. We restrict ourselves to a minimal repertoire of geometric objects that can be easily extended to the most general case. Such objects are points, lines, circles, line segments and arcs.

We conclude this chapter by introducing some concepts useful for the next chapters and by presenting the related work in two dimensional sketching. Chapter 2 presents an Editable REPresentation for two dimensional sketches and describes how the 2D sketcher is organized around this representation. Chapter 3 is concerned with the algorithmic issues of constraint solving in EREP, its limitations and extensions. Chapter 4 discusses the problem of root selection and presents a heuristic approach and an interactive framework for root identification. Finally Chapter 5, presents the conclusions from this work and highlights future directions for research.

1.1 Trends in two dimensional sketching

1.1.1 The descriptive approach

In the descriptive approach the user determines the exact coordinates of the objects using a low level language or an interactive system. This approach is used in drawing systems and is the counterpart of boundary representation. Usually everything is defined in terms of point coordinates, and hierarchical structuring of objects supports grouping. Clearly this approach is not appropriate for the goals stated above, since the objects cannot be easily modified and their description does not capture any functionality.

1.1.2 The constructive approach

The constructive approach is the counterpart of CSG and relies on the user for specifying the sequence of operations that will construct the sketch. We may either create programming languages that have high level commands for operations like draw, intersect, merge (union) etc, or embed such constructs in a conventional

programming language like PASCAL or C. Researchers had early realized that this approach could not fulfill the above objectives, and they enhanced this approach with parameterized operations [RBN88] or with some kind of constraints [Wyk82]. A complete, purely constructive system based on parameterized operations is MAMOUR described in [RBN88]. This approach attains all the above goals except the first: the user may be involved in designing parameterized transformations that may be hard to invent and difficult to understand. Another constructive approach is the one described in [GZS88], which is a 3D CSG representation, with the capability of parameterizing the relative placement of the geometric objects using what is called relative position operators.

1.1.3 The declarative approach

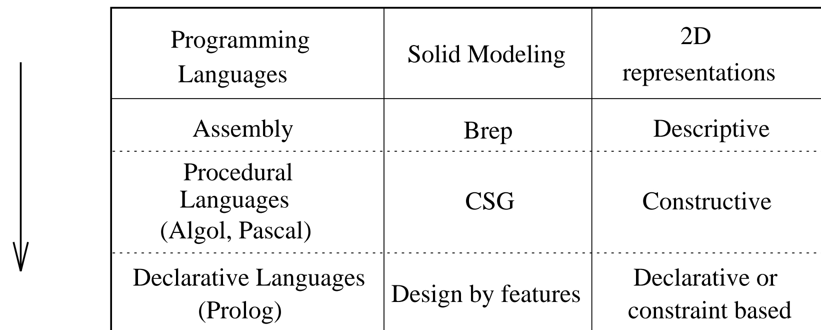
The declarative approach is characterized by a number of constraints that the user imposes on the sketch. In this method, the user does not determine how the constraints will be satisfied.

In one version of this approach, the user implicitly or explicitly determines an order in which the constraints are to be satisfied [Ros86]. We call that approach parametric declarative or simply parametric. Roller in [Rol90, Rol91] describes such a system in which the order that the constraints are satisfied is determined from the design steps made by the user to construct the drawing. In the same system rules are used for determining a unique solution. PIGMOD is a system that follows the same approach, but it provides an additional mechanism for inserting and deleting constraints that is based on a propagation method. The approach used in PIGMOD has been extended to 3D. It is not easy to distinguish between systems using the parametric declarative approach and systems using a constructive approach enhanced with arameterizing. In PIGMOD and Roller's system it seems that the levels of abstraction provided by the systems make the user view the underlying constructive representations as declarative.

For the pure approach where the user does not impose any order the terms variational declarative or simply variational have been employed [BA91, HJ93]. The idea of using constraints for determining a sketch was first used in Sketchpad [Sut63a], and it remains very attractive because of its simplicity from the viewpoint of the user. Hillyard and Braid [HB78] were the first to develop a general theory in which dimensions are used to specify algebraic constraints on the coordinates of vertices of a wire-frame representation. A theory for defining tolerance constraints as regions in the vector space spanned by the model's variables was developed in [Hof82] and extended in [Tur88]. Another framework that tries to overcome the problems in [Hof82] was introduced in [Req83]. A survey of these attempts can be found in [Jus92].

The declarative approach may capture most of the desired functionality, but it had two major drawbacks: until now, the constraint solving methods used were quite inefficient, and the user had no standard way of editing a solution that was not intuitive.

In figure 1.1, we can see the parallel evolution of programming languages, solid modeling and 2D representations. Of course, each area has its own peculiarities but there are fundamental conceptual similarities as indicated by the above discussion.



Programming Languages	Solid Modeling	2D representations
Assembly	Brep	Descriptive
Procedural Languages (Algol, Pascal)	CSG	Constructive
Declarative Languages (Prolog)	Design by features	Declarative or constraint based

Figure 1.1 The parallel evolution of three areas

The vast majority of the systems use some kind of constraints for specifying the sketches. Each system is characterized by the constraint solving method that it uses, and the repertoire of constraints that it provides to the user.

1.2 Constraint solving methods

1.2.1 Numerical constraint solvers

The constraints are translated in a system of algebraic equations and are solved using iterative methods. Numerical solvers are inappropriate for interactive debugging and until now no research has been made on using iterative methods navigated by extra information that is derived from the topology. The exponential number of solutions and the large number of parameters are making the geometric constraint solving problem ill conditioned for numerical methods. Typically the convergence depends on the starting point (see also section 4.1). The advantage of these methods is that they have the potential to solve configurations that may be non solvable using one of the other methods. All constraint solvers more or less switch to iterative methods when the given configuration is not solvable by their method. This fact emphasizes the need for further research in the area of numerical constraint solvers.

Sketchpad, described in [Sut63a] was the first system to use the method of relaxation as an alternative, relaxation is a slow but quite general method. Many systems like ThingLab [Bor81] and Magritte [Gos83] kept the relaxation as an alternative to other methods. In [Bar87] a method called projection method is presented for finding a new solution that minimizes the Euclidean distance between the old and the new solution. The Newton-Raphson method has been used in various systems, and it proved to be faster that relaxation but it has the problem that it may converge to local minima, for that reason Juno [Nel85] uses as initial state the original sketch. A modification of the Newton-Raphson was developed in [LG82], where an improved way for finding the inverse Jacobi matrix is presented. Furthermore, the idea of dividing the matrix of constraints into submatrices as presented in the same work, has the

potential of providing the user with useful information regarding the constraint structure of the sketch. Though this information is usually quantitative and non specific, it may help the user in basic modifications. A method that represents constraints by an energy function and then searches for a local minimum using the energy gradient is presented in [WFB87].

1.2.2 Constructive constraint solvers

This class of constraint solvers is based on the fact that most configurations in an engineering drawing are solvable by ruler, compass and protractor. In these methods the constraints are satisfied in a constructive fashion, which makes them natural for the user and suitable for interactive debugging (see chapter 4). We have two approaches in this direction.

The first approach uses rewrite rules for the discovery and execution of the construction steps, we call this approach rule-constructive solving. In this approach it is easier to express complicated constraints (like tangency), Although it is a good approach for prototyping and experimentation, the extensive search and match algorithms that it uses (borrowed from Logic Programming), make it inherently slow.

A method that guarantees termination, ruler and compass completeness and uniqueness using the Knuth-Bendix critical pair algorithm is presented in [Bru86, Soh91]. This method can be proved to confirm theorems that are provable under the system of axioms given, the rules for determining automatically the desired unique solution are not discussed in these papers. A system based on this method was implemented in Prolog. Aldefeld in [Ald88] uses a forward chaining inference mechanism, the notion of direction of lines is imposed with additional rules, restricting the number of solutions. A similar method is presented in [SAK90], where handling of overconstrained and underconstrained cases is given special consideration. Sunde in [Sun88] uses a similar rule-constructive method but uses different rules for representing directed distance and non directed distance, giving flexibility for dealing with the root selection problem. In [YK90] the problem of nonunique solutions is

handled by imposing an order on three geometries. A recent detailed work describing a complete set of rules for 2D design can be found in [VSR92], in this work the scope of the particular set of rules is characterized. Finally, a technique called Meta-level Inference is introduced in [BW81], this approach combined with multiple sets of rules and selective application is said to cut down on search. This applied has been applied in PRESS, a program for algebraic manipulation.

Much of the recent research on general-purpose languages with constraints has used logic programming as a base. The research in Constraint Logic Programming (CLP) Languages has some interesting results to exhibit. In [BMMW89] a scheme called CLP(D) is described for CLP languages, which is parameterized by D, the domain of constraints. In place of unification, constraints are accumulated and tested for satisfiability over D, using techniques appropriate for the domain. In this work a scheme for Hierarchical CLP languages is also introduced. A review of CLP languages can be found in [Coh90].

The other constructive approach has two phases, during the first phase the graph of constraints is analyzed and a sequence of construction steps is derived, during the second phase these construction steps are followed to derive the geometries. We call this approach graph-constructive solving. This approach is fast, more methodical and proved to be sound. The problem is that as the repertoire of constraints increases the graph-analyze algorithm needs to be modified.

Fitzerald [Fit81] follows the approach of dimensioned trees by Requicha [Req77], only horizontal and vertical distances are allowed in this method and it was useful for simple engineering drawings. Todd in [Tod89] first generalized the dimension trees of Requicha. Owen in [Owe91] presents an extension of this principle to include circularly dimensioned sketches and DCM [D-C93] is a system using this method.

1.2.3 Propagation methods

This was the obvious approach in the early constraint solving systems. Here we first translate the constraints into a system of equations involving variables and

constants and then create an undirected graph having as nodes the equations, the variables and the constants, and having the edges represent whether a variable or constant appears in an equation. Then we try to direct the graph so as to satisfy all the equations starting from the constants. To succeed this, various propagation techniques have been used, but none of them guarantees to derive a solution or to has a reasonable worst case running time. For a review of these methods see [Soh91].

Sketchpad [Sut63a] uses propagation of degrees of freedom and propagation of known values. Pro/ENGINEER [BA91, Pro] uses propagation of known values. Propagation of known values is the inverse process of propagation of degrees of freedom, both methods are global, unstable and do not work for cyclic dimensioned sketches. CONSTRAINTS [SS80] uses retraction, which is a localized version of propagation of known values, that stores some information for each variable, about the variable's premises and dependents. A similar technique is used in [Li88], where an algorithm for local propagation of known values is given, for the remaining simultaneous constraint solving, a system of linear equations is attempted. In general, retraction is faster but less powerful than propagation of known values. ThingLab uses the Blue and Delta Blue algorithms described in [FBMB90, Bor81], they are faster than other algorithms because they are based on a local propagation of degrees of freedom within the constraint graph, the Delta Blue algorithm is just an incremental version of the Blue algorithm where the constraint graph need not be recomputed from scratch with each removal or addition of a constraint. The algorithm presented in [Wil91] has a better conflict detection that is based on information, provided explicitly by the user. Magritte [Gos83] employs a propagation method to transform the undirected constraint graph and then breadth first search is used to derive all solutions. Constraint Kernels, a system described in [Fuq87] uses a somewhat different approach, it transforms the original undirected graph to a directed acyclic dependency graph.

Actually the Constructive Constraint Solvers is a subcase of this method (fixed geometries for constants and variable geometries for variables), that has the advantage that we can derive almost linear algorithms, for solving a considerable class of configurations and we can tell when something is solvable or not in almost linear time.

1.2.4 Symbolic constraint solvers

Here the constraints are again transformed to a system of algebraic equations that is solved with methods known from algebraic manipulation, such as Grobner's basis [Buc85] or Wu's method [Wen86]. Both methods provide some means for solving a system of algebraic equations. These methods have also been used in mechanical geometry theorem proving in [Wen86], [Cho88, CS86, Cho87] and [Kap88].

Kondo in [Kon92] deals with the addition and deletion of constraints (see also [Kon90]), by using the Buchberger's Algorithm [Buc85] to derive a polynomial that shows the relationship between the deleted and added constraints.

We don't know any other sketching system using symbolic methods for constraint solving.

1.2.5 Solvers using hybrid methods

The existing constraints solvers usually use a combination of the above methods. They try one method if it does not succeed then they try another one, the problem is that some methods may take exponential time for giving a negative response. As already mentioned many systems use a numerical approach as alternative to some other method, such systems are DCM [D-C93], Pro/ENGINEER [BA91, Pro], Sketchpad [Sut63a, Sut63b], ThingLab [FBMB90, Bor81], and Magritte [Gos83].

Some solvers like the one used in Pro/ENGINEER try to isolate a number of reduced sets of core equations that must be solve simultaneously, then for solving these systems they use numerical methods.

1.2.6 Other methods

The AI community has some interesting work to exhibit. In [Mac77] a class of algorithms is discussed for discovering a situation that satisfies a set of simultaneous constraints. This method has never been used in the context of geometric modeling.

The GIPS system described in [CFV88], is using macros, that is family of objects, where to define an object the user must supply the values for a number of predefined constraints. How the overall placement of object is made is not discussed. We can characterize this method as a hybrid of the constructive and declarative approach.

A comparative review of six implemented 2D sketching systems can be found in [Rol89b].

1.3 The repertoire of constraints

The repertoire of constraints differs from system to system. There is although a common kernel of constraints that consists of distances and angles. Also the parallel, perpendicular and on constraints are usually being treated as separate constraints for reasons that we will explain later. Constraints like tangent, distances between complicated objects, concentric, midpoint, coincident etc are either expressed in terms of distances and angles, or treated as separate cases. As we already mentioned the first approach reduces the number cases when we have a graph-constructive constraint solver, while the second carries more information about the peculiarities of the sketch and the intention of the user, and is well suited for rule constructive solving. For example a tangency constraint could be translated to a distance constraint but this does not capture the intent of the user that may wanted an arc to interpolate smoothly two line segments.

Another issue is whether a constraint solver allows relations between the dimensional constraints, for example can we define d_1 from $d_1 = {d_2}^2 + 12$? Many constraint solvers support this feature, under the assumption that the user has previously defined all the dimensions involved in an expression. Another approach is to allow only

linear relation between dimensional constraints and use a linear solver for deriving the dimensions before constraint solving.

Some systems like Pro/ENGINEER and [Rol90, Rol91], support the notion of implied constraints, for example two lines that are almost perpendicular or parallel are considered to be perpendicular or parallel, without needing the user to impose explicitly this relation. The method as implemented in these systems seems to be convenient for small size designs, but is becoming very difficult to control in larger sketches. The reason for this is that the user may get constraints that he did not desired resulting to an overconstrained or erroneously constrained sketch. For example in Pro/ENGINEER when you want to constraint the angle between two lines to be $5°$, you must initially place the lines at a larger angle, because otherwise it will consider them parallel. More research is needed in this area, we could have a hierarchy of constraints (like the one presented in [BMMW89]), where the explicit constraints override the implicit constraints.

2. THE STRUCTURE OF THE EREP 2D SKETCHER

In this chapter we shall describe the system architecture and information flow of the 2D EREP sketcher, the representation and its rationale, and the User Interface semantics that determine how the representation is being edited as the user interacts graphically with the system.

2.1 System architecture

The architecture of the 2D sketcher of Erep, is shown in figure 2.1

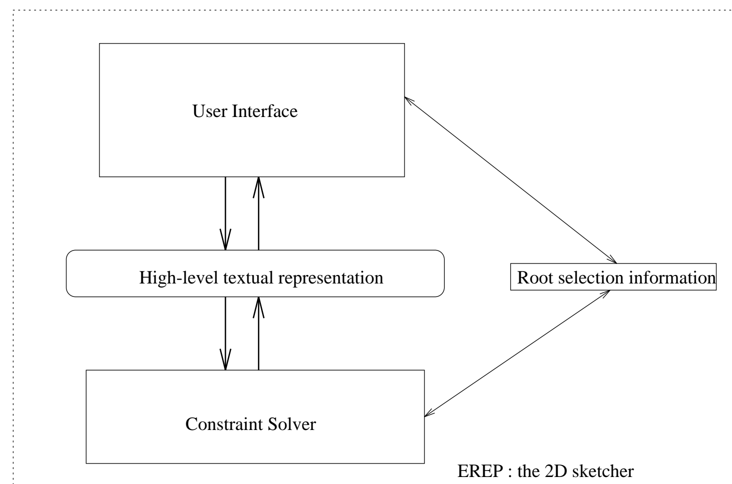


Figure 2.1 The overall architecture of the 2D sketcher

The user sketches a geometric figure and annotates it with constraints using the User Interface that is described in section 2.3, the sketch is transformed to a high-level textual description that captures all the information needed. Then the constraint solver takes as input this high level description, and returns the solution in the same

high level description. The textual description is independent from the constraint solving method used, and determines a unique solution by its own, we shall call this textual representation EREP (Editable REPresenation). The information that has to do with the root identification, allows the user to select another solution consistent to the dimensioning imposed. The root identification problem will be discussed in chapter 4.

Our system supports lines, points, circles, segments, arcs and rays. Constraints are explicit dimensions of distances and angles as well as constraints of parallelism, incidence, perpendicularity, tangency, concentricity, collinearity and prescribed radii. For now we exclude relations on dimension variables and inequality constraints. The user specifies a rough sketch and adds to it geometric and dimensional constraints that are normally not yet satisfied by the sketch. The sketch has only to be topologically correct.

The structure of the constraint solver is shown in figure 2.2.

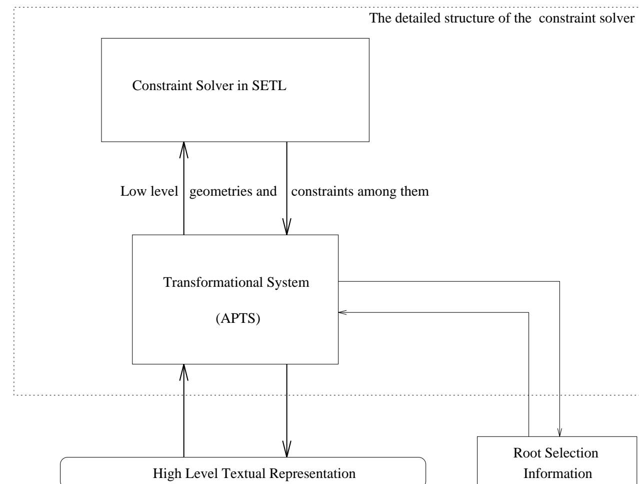


Figure 2.2 The structure of the constraint solver

The constraint solver consists of two parts:

- A part written in the high level transformational programming system APTS [Pai93], this reads the Erep file and transforms it into a normal form where we have only points, lines, circles, distances and angles. This normal form is fed to the solver, and then the solution is read back in, and is incorporated into an EREP file.

- The SETL2 [Sny90] program analyzes the constraint graph and if the sketch is well constrained, a unique solution is being computed in a constructive fashion. The constraint solving algorithm is being explained in chapter 3. SETL2 is a robust, high-level conventional language, with powerful set operations that make it suitable for expressing and extending complicated algorithms. It's implementation is reasonably fast with respect to both compile and run time.

When an error is detected during any phase, the constraint solving process is interrupted and the user is informed by an error window for the nature of the error.

2.2 Representation

The objects in EREP are described using a three level hierarchy shown in figure 2.3.

We have chosen the hierarchical representation instead of a flat representation for the following reasons :

- The sharing of geometries by higher level structures gives the potential for more efficient and flexible representation. For example, instead of forcing two segments to be collinear the user can have them share the same underlying line. This is nicely supported by the User Interface where the user can cut a line into a number of segments.

- Inheritance of attributes that correspond to fix/nonfix and display information is now inherent. Later we will discuss the rules for resolving the conflicts.

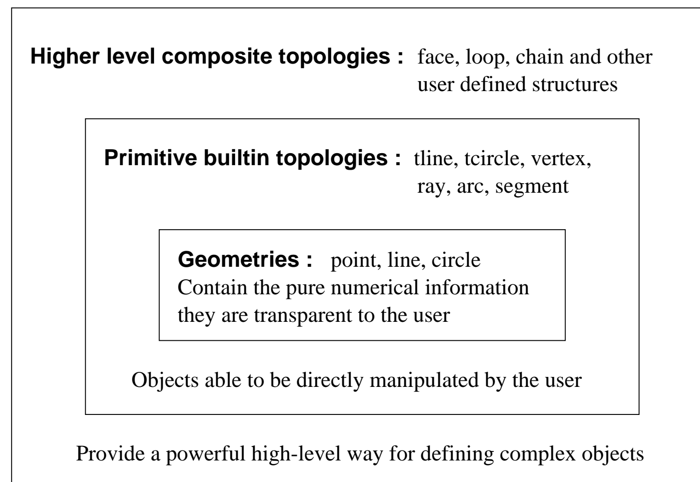


Figure 2.3 The hierarchy of objects in EREP

- If we had a flat representation scheme we would have forced geometry to be bound to irrelevant information (fix/nonfix, display information).
- The constraint solver is fed only with geometry, geometry constraints and translated topological information which expresses topologically implied constraints (e.g. the end points of a segment have to lie on the underlying line).
- Consistency of topology, since a shared underlying geometry can be represented by the same object. In this way we are avoiding replication and consistency problems.
- Our choice is supported by suitable, well defined User Interface semantics that eliminate the potential for ambiguity and confusion.

In terms of a 3D design we shall call a dimensioned sketch cross section. We will use the term cross section for the 2D concept, and we will define another entity called cross section use where we will have a cross section mapped to a sketching plane in 3D. We will have to fix the sketch on the sketching plane by transforming it to a prescribed position. We implement the transformation as a rotation and translation from our original sketch assuming that the target sketch lies at the plane z=0 (all z-coordinates zero).

2.2.1 Representation of geometries : points, lines and circles

An internal representation of a geometry will be considered well conditioned when a small change to the position of a geometry invokes small changes to the values used for representing the geometry. The representation of the geometries in our system is well conditioned.

The internal representation of points, lines and circles is as follows :

1. Point (P_x, P_y), the Cartesian coordinates.
2. Line (d_0, N_x, N_y), where d_0 is Euclidean distance of the origin from the line (signed), (N_x, N_y) is a normal vector of the line, normalized, so it holds :

$$N_x^2 + N_y^2 = 1$$

3. Circle (C_x, C_y, C_r) where (C_x, C_y) is the circle center, and C_r is the circle radius.

Lines and circles are oriented.

In the case of lines the orientation is derived from the normal. That is, when the normal is turned 90 degrees clockwise (CW) it is the line orientation. We make the convention that circles are always oriented counter-clockwise (CCW), so the positive region is the one that is enclosed in the circle.

2.2.2 Topologies : vertices, tlines, tcircles, segments, arcs and rays

1. Vertices: A vertex declaration contains only the underlying point.

2. Topologies Tcircle and Tline: These topologies correspond to an infinite line and a whole circle respectively. The orientation agrees or is opposite from the orientation of the underlying geometry. This is specified with a field called `<orient>` in the textual representation, it can take two values: `POS` for the same orientation or `NEG` for different orientation.

3. Arcs: The arc declaration specifies the underlying circle and the two end vertices. The intended arc is always the one that is derived if we go from the first point to the second CCW. As far as the orientation field is concerned `POS` means follow the orientation of the underlying circle (CCW), and `NEG` means it must be opposite (CW).

4. Segments: The segment declaration specifies the underlying line and the two end vertices. The orientation is implied by the order in which the two vertices are specified (`FROM` vertex `TO` vertex).

5. Rays: The ray begins at the specified vertex and lies on a given line. The part of the line that forms the ray, is determined by the field `<part>`, and is independent

from the orientation of the ray. The ray orientation agrees or is opposite to the underlying line orientation, this is recorded in the textual representation by the `<orient>` entry.

2.2.3 The overall textual representation of a cross section

After having presented the representation of the geometries and topologies we will now describe the structure of an EREP file defining a cross section. Then we will elaborate on each of the remaining representation issues.

The overall structure is described by the following grammar :

```
<representation> ::= CROSS_SECTION  id
                      <geometries>
                      <topologies>
                      <objects>
                      <geometry constraints>
                      <implied constraints>
                      <display of constraints>
                     END_CROSS_SECTION

<geometries>     ::= GEOMETRY
                      <geometry> ; { <geometry> ; }
                     END_GEOMETRY

<topologies>     ::= COMPONENTS
                      <topology> ; { <topology> ; }
                     END_COMPONENTS

<auxiliary>      ::= AUX_COMPONENTS
                      { (<basic topology> | <const def>) ; }
```

```
                     END_AUX_COMPONENTS

<objects>        ::= OBJECTS
                      { <object> ; }
                     END_OBJECTS

<geometry constraints> ::=
                     GEOMETRY_CONSTRAINTS
                      { <constraint> ; }
                     END_CONSTRAINTS

<implied constraints>  ::=
                     IMPLIED_CONSTRAINTS
                      { <implied constraint> ; }
                     END_IMPLIED_CONSTRAINTS


<display of constraints> ::=
                     DISPLAY_OF_CONSTRAINTS
                      { <display info> ;}
                     END_DISPLAY_OF_CONSTRAINTS
```

Here is a brief explanation of the above grammar :

<geometries> Geometries contain the pure geometric information as this is described in the section 2.2.1.

<topologies> Topologies contain the information described in section 2.2.2, some additional information about the displaying of the object, and a field that determines whether the topology is fixed or not.

Cross sections are composed from topologies, which are built on geometries, (points, lines and circles with possible extension to other closed or open curves). Their construction may require in addition auxiliary entities which may be one of the three basic topologies : vertex, tline and tcircle, other than this, auxiliary entities are treated exactly as topologies. The auxiliary entities are used as reference points, lines of symmetry, tangent circles etc. So the `<auxiliary>` entities are a subset of the topologies, they are restricted to vertices, tlines and tcircles, (used to construct symmetries, alignments etc) and constants (needed to express constraints).

`<objects>` are higher level composite topologies. They are intended to give a short, informative and powerful way of representing composite objects.

Geometric constraints are either explicitly defined(`<geometry constraints>`) or inferred implicitly(`<implied constraints>`). Implied constraints may be overridden by explicit constraints.

`<display of constraints>` contains information for the displaying of each dimensional constraint. Displaying of geometric constraints is not yet supported. The issue of displaying constraints is discussed in 2.2.6.

2.2.4 More on geometries and topologies

Here is the rest of the grammar for topologies and geometries.

```
<geometry>       ::= POINT  id = <p_value> {, POINT  id = <p_value> }
                   | CIRCLE id = <c_value> {, CIRCLE id = <c_value> }
                   | LINE   id = <l_value> {, LINE   id = <l_value> }

<basic topology> ::= VERTEX  id ON point_id <additional info>
                         { , id ON point_id <additional info> }
                   | TLINE , id ON line_id [<orient>]
                                             <additional info>
                         { , id ON line_id [<orient>]
```

```
                                        <additional info> }
                    |  TCIRCLE id ON circle_id [<orient>]
                                        <additional info>
                        { , id ON circle_id [<orient>]
                                        <additional info> }

<topology>         ::= <basic topology>
                    |  SEGMENT <seg_decl> {, <seg_decl> }
                    |  RAY <ray_decl> {, <ray_decl> }
                    |  ARC (<arc_decl_c> {, <arc_decl_c> }

<seg_decl>         ::= id  ON line_id FROM  vertex_id  TO  vertex_id
                       <additional info>

<ray_decl>         ::= id  ON line_id FROM  vertex_id [<orient>]
                       <part> <additional info>

<arc_dec_c>        ::= id  ON circle_id FROM vertex_id TO vertex_id
                       [<orient>] <additional info>

<const def>        ::= NUMBER  id = <n_value> { , id = <n_value> }

<additional info>::= [<fix_opt>] [ <visible_opt> ]

<fix_opt>          ::= fixed | nonfixed

<visible_opt>      ::= visible | nonvisible

<orient>           ::= (POS | NEG)
```

```
<part>          ::= 0 | 1

<c_value>     ::= "(" n_value, n_value, n_value ")"

<p_value>     ::= "(" n_value, n_value ")"

<l_value>     ::= "(" n_value, n_value, n_value ")"
```

Inheritance is supported for the visible/nonvisible and the fixed/nonfixed attributes of the topologies.

The information about the displaying of a topology is currently restricted in determining whether the topology is visible or not. Later we may want to increase the options by including colors or patterns.

Why using a fixed/nonfixed field ? The reason is that constraints are really determining the values of the geometries. By fixing a geometry we mean that it it will have to keep the absolute position given interactively by the user.

Structural inheritance is supported for the visible/nonvisible and the fixed/nonfixed attributes of the topologies. That is, if we define a topology to be fixed all the underlying topologies and geometries will become fixed unless one of the descendants is defined as nonfixed. When there are two conflicting ancestors, the fixed and visible win over their counterparts. Formally, we can assign attributes to each symbol of the grammar, and associate each rule with an assignment, building in this way an attribute grammar with the inherited attributes visible and fixed.

Note that the orientation, fixed/nonfixed and display fields are optional, so there must be a default. For regular topologies is reasonable to define as default `POS`, nonfixed, visible respectively. For auxiliary topologies is reasonable to have as default `POS`, fixed, nonvisible respectively.

2.2.5 Constraints

Constraints are dimensional (angles and distances), or geometric (parallel, perpendicular etc). The exact semantics of the angular and linear constraints are described in chapter 4. In determining a constraint we often have to refer to a geometric entity in the opposite orientation, this is done by writing a minus sign.

```
<constraint> ::= <dimensional constraint>
              |  <geometric constraint>

<dimensional constraint> ::= distance ldim_id "(" [<pm>] <d_object> ,
                                                  [<pm>] <d_object> ")"
                          |  angle adim_id "(" [<pm>] <l_object> ,
                                               [<pm>] <l_object> ")"
                          |  radius ldim_id"(" <c_object> ")"
                          |  id = <constr_value>

<pm>              ::= + | -

<constr_value>    ::=     "(" n_value, <exp> , <change_flag> ")"

<change_flag>     ::= 0 | 1
```

The linear constraints can be placed between any combination of objects. For now, angle constraints are between lines. Later, we will include angles between circles and lines, and angles defined by three points.

A dimensional constraint is always associated to a tuple via an assignment statement. A `<constr_value>` tuple contains the current value of the angle, distance or radius, an arithmetic expression that determines the value that it should take after

constraint solving, and a flag that determines whether the current value has been updated (1 for updated, 0 for non updated).

`<exp>` is an arithmetic expression involving numbers and dimensional constraint identifiers. The trivial case in which the right hand side is an expression involving only constants and numbers, assigns a value to a dimension. In the case that the RHS has dimensional constraint identifiers the situation is becoming more complicated. For now we support only RHS expressions that involve either an expression with numbers and constants or a single identifier. Under this format we cannot have shared dimensions, but we can make a dimension to have the same value with another by setting as RHS the identifier representing the other dimension.

Let a `<c_object>` be a circle, a `<l_object>` be a line, and a `<p_object>` be a point (remember that the constraints are imposed only between geometries).

```
<geometric constraint> ::=
               TANGENT "(" ( <l_object> | <c_object> ),
                            ( <l_object> | <c_object> ) ")"
            |  ( PARALLEL | PERPEND ) "(" <l_object>, <l_object> ")"
            |  CONCENTRIC "(" <c_object> | <p_object>,
                               <c_object> | <p_object>")" <orient>
            |  COLLINEAR "(" <p_object> , <p_object> , <p_object> ")"
            |  ON  "(" <object> , <object> ")"
            |  ALIGN  "(" <object> , <object> , <orient> ")"


<implied constraints> ::= ( PARALLEL | PERPEND )
                           "(" <l_object>, <l_object> ")"


<orient>         ::= POS | NEG
```

Since lines are oriented, parallel and perpendicular have not the same meaning as $0°$ and $90°$ angles. For example specifying two lines as parallel is not the same as

enforcing a zero angle between them, a zero angle between L1 and L2 means that L1 and L2 should be parallel having the same direction. L1 and L2 parallel means that they may be parallel or antiparallel.

Incidence (`ON`), always refers to the underlying line or circle, not the segment. Aligning segments and lines means that they are on the same line. Aligning circles and arcs means that they are on the same circle. Since the semantics of the other geometric constraints are obvious, we won't elaborate on them.

2.2.6 Display of constraints

For each dimensional constraint there is one entry in the section `<display of constraints>`. Each such entry has the following form :

```
<display info> ::= ldim_id <cr_valu> [reference point_id]

<cr_value>     ::= "(" n_value, n_value, n_value ")"
```

The first element of `<cr_valu>` determines whether the dimensional constraint will be visible. The second element of `<cr_valu>` is the one that determines the ratio. This determines where between the two objects we want our dimension, to be placed (in the case of radius where between the center and the circumference). The third element determines the distance that specifies the line (for the case of linear dimension) or the circle (for the case of angle dimension) on which the dimension label will be displayed. This distance is always measured with respect to a reference point.

Choosing the reference point is difficult process. A suggestion that works in most of the cases is the following:

- Linear dimension

- line to line : We could keep the distance from a perpendicular line passing through the origin, the disadvantage with this is that the origin is fixed, instead we could carry with a line a point on the line (determined implicitly via the user interface), this point should be updated when the line is moved.
- segment to line/ray/segment: use the midpoint of the (first) segment.
- ray to line/ray: use the starting point of the (first) ray.
- ray/line to segment: use the midpoint of the segment
- line to ray: use the starting point of the ray
- radius : here we have the same problem as in the line-line case, we don't want the display position to remain fixed, now the distance field is interpreted as an angle but from where ? One possible solution is to carry along a point on the circle which should be updated accordingly.

- Angular dimension

  In most of the cases the intersection point is a good choice, but in the case that we have two almost parallel lines we will have to use another reference point. The selection of the reference point is similar to that for the linear dimensions.

2.2.7 Higher level of topology

This corresponds to the `<objects>` section of the grammar, it refers to higher level topologies. There are two basic categories of topologies : chains (an open sequence of segments and arcs) and loops (a closed sequence of segments and arcs). A tline, an arc or a segment are chains by themselves, and a tcircle is a loop by itself, so tline, segment, arc are subtypes of chain, and tcircle is a subtype of loop. Furthermore, we can extend chains to include rays as the first and/or last components of the list.

For purposes of compatible terminology we shall call an arc or a segment of a loop/chain, edge of the loop/chain.

The orientation of a higher level topology is determined based on agreement or disagreement with the orientation of the first topology of the sequence.

Another useful higher level topology is the face :

A face represents a closed connected 2D surface. A face is determined by a loop (outer loop) and a list of loops (inner loops) that have to lie inside the first loop. A face corresponds to the area inside the outer loop that it is not enclosed by any of the inner loops. A face can have an empty list of loops being essentially the area enclosed in the outer loop.

Here is the corresponding grammar :

```
<object>     ::= ( <loop decl> | <chain decl> | <face decl> )

<chain decl> ::= CHAIN id OF
                 [<orient>] ( tline_id |
                              ( <edge_id> {, <edge_id>} ) )

<loop_decl>  ::= LOOP  id OF
                 [<orient>] ( tcircle_id |
                              ( <edge_id> {, <edge_id>} ) )

<face_decl>  ::= FACE id INSIDE loop_id
                         OUTSIDE loop_id {, loop_id}

<edge_id>    ::= segment_id | arc_id
```

The orientation of a chain will be determined by the user, the orientation of a loop will always be CCW initially and it will be adjusted for use as outer or inner loop later.

2.3 The user interface

An instance of the User Interface panel is depicted in figure 2.4.

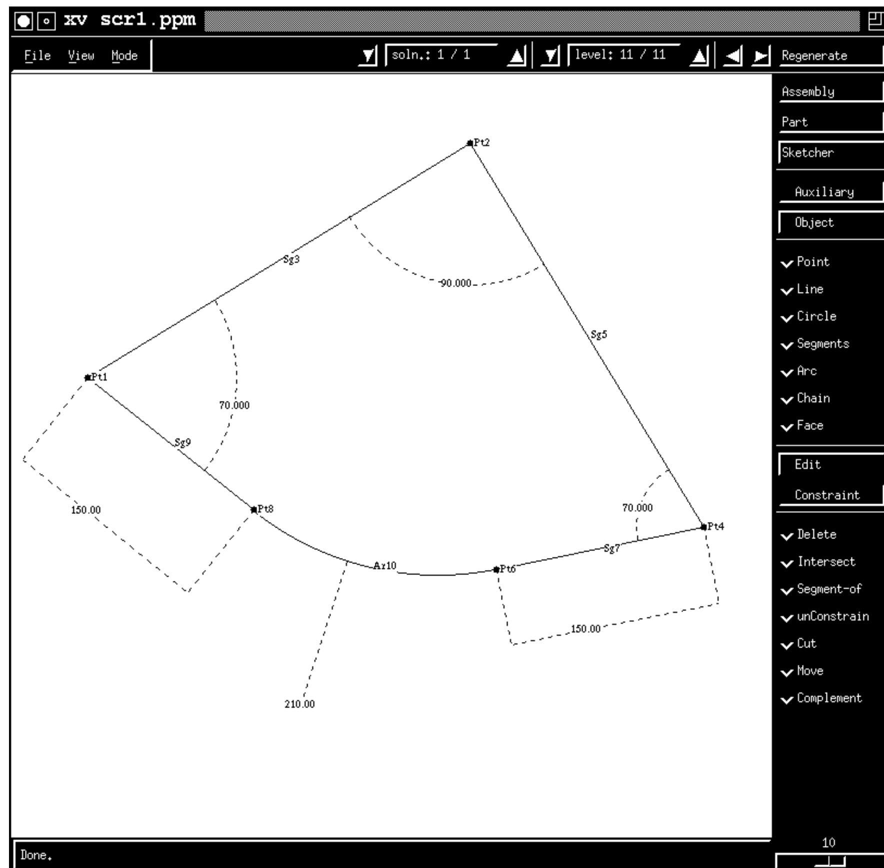


Figure 2.4 The UI panel

When the user selects an operation, a brief description of this operation will be displayed at the bottom of the panel.

1. The File menu, provides the following operations:

   Save, saves a sketch in an Erep file

   Load, loads an Erep file

   Quit, exits Erep

Hardcopy, sends the current sketch to printer

Clear, clears the drawing area

2. The View menu, provides operations for hiding/displaying the Auxiliary Objects, the Dimensions, the Dimension Values, the Object Labels, and the Dimension Labels.

3. The Mode, Assembly and Part menus are all related to the 3D part of the system.

4. The Arrow buttons have to do with the root identification and their functionality is explained in chapter 4.

5. The Regenerate button in the upper right corner of the panel calls the constraint solver. If the sketch is well constrained and ruler and compass constructible the constraint solver will return a sketch that satisfies the given constraints. Otherwise, the user will be informed by an error window for the problem. The user may use the horizontal arrow buttons for retrieving the configuration of the sketch before Regenerate.

6. Selecting an Object: In many cases the user will need to select a topology or a dimension for using it in some operation. This is accomplished using a commit-when-release the mouse-button protocol. The user presses the corresponding button and by keeping the button pressed approaches the desired object until it becomes highlighted, then he releases the button and the highlighted object is selected. If an object is close to other objects then the other objects may become highlighted as well. We have two ways for dealing with this problem. The first is to adjust the sensitivity of the mouse by using the button in the lower right corner of the panel. The second way, is to press the Shift key at the same time with the mouse button, by doing so a menu will appear upon releasing the button containing all the highlighted objects, then one can select

the intended object from that menu. Recall that objects have names and their names can be displayed using the operations in the View menu.

7. The Object menu provides operations for constructing topologies :

   Line, the user presses the left button on the first point that wishes the line to pass from, and releases the button on the second point. The two points define an infinite line. The representations for the corresponding tline and the underlying line are constructed automatically.

   Point, with the left mouse button the user creates a new vertex, with the middle button the user creates a vertex on a pre-existing line, circle, segment or arc. The UI creates the vertex entry and the underlying point. When a vertex is constructed(using the middle mouse-button) on a line, segment, arc or circle an `ON` constraint is inserted between the underlying point and the underlying line or circle.

   Circle, with the left button the user constructs a new circle by pressing on the intended center and releasing anywhere on the desired circumference, with the middle button the user constructs a circle with center a pre-existing vertex. The UI constructs the underlying circle and the tcircle entry. It will also construct a vertex (unless it already exists), and it will constrain the underlying point to be concentric to the circle. For creating a circle with no center point, the user must use the right mouse-button.

   Segments, constructs a sequence of incident segments, using the left mouse-button the user can press on the starting vertex of the first segment of the sequence, and then release on any position to place the next vertex and so on. A vertex can be placed on a line, circle, segment or arc, using the left button, this will introduce an `ON` constraint between the underlying line or circle and the point.

   To select a pre-existing vertex as the first vertex the one has to use the middle button, to select a pre-existing vertex as the next vertex one can use the left

button. The operation is terminated by pressing the right mouse-button. This operation will result in creating an underlying line, and two end vertices (if they do not already exist), with the underlying points. There is no need to constrain the end vertices to be on the line, this constraint is implicit. The reason for a segment being associated with vertices rather than with points is that we want the vertices to be visible and capable of being manipulated by the user.

Arc, constructs an arc, using the left button the user picks the two end vertices by releasing the button on the desired positions, and then selects a third point on the arc in the same way. The user can select pre-existing vertices as the end vertices using the left button again. For choosing a vertex that has to lie on a line, circle, segment or arc the user must use the middle button. This operation will create an underlying circle, and two end vertices (if they do not already exist), with the underlying points. Again there is no need to constrain the vertices to be on the circle, this constraint is implicit. If the end vertices have been selected to lie on a line, segment, arc or circle an `ON` constraint is inserted.

A ray cannot be constructed from this menu, the user has to use the Cut or Delete operations described below, to construct this topology.

Chain, to construct a chain the user will have to select all the edges (segments and arcs) in the appropriate order. The UI is responsible for letting the user select only incident segments/arcs and for making sure that the user never selects the same object twice. The UI will give a POS or NEG orientation to the chain based on the relative position of the first and second edge and on the orientation of the first edge.

Face, to construct a face the user will have to select the outer loop and the inner loops. In selecting a loop a user just clicks on one of its edges, in case of ambiguity the UI asks the user to select a second edge to clarify which loop

he/she wanted. The orientations of the loops are maintained by the UI. Note that there is no button for making a single loop that is not a part of a face.

8. The Auxiliary menu will provide just a subset of the operations available under the Object menu, namely Line, Point and Circle, in addition to a special operation called Axis, that constructs two perpendicular fixed infinite lines. The auxiliary topologies are displayed with different color and they can be hidden using the View operation described above.

9. The Constraint menu provides operations for imposing constraints:

   Distance, the user will select two topologies and will click at the place where he/she wants the label to be placed, this will determine the display of constraints information. The distance constraints will be imposed on the major underlying geometries.

   Angle, same as Distance

   Radius, the user selects the circle or the arc and the place of the label. A radius constraint is imposed on the underlying circle geometry.

   Tangent, On, Concentric, Parallel, Perpendicular and Align, the user selects the two involved topologies. The corresponding constraint is imposed on the underlying geometries. No display of constraints information is recorded for this type of constraints.

   Fixed, by using this operation the user can fix any topology. This means that the corresponding field in the definition of the topology will become fixed.

10. The Edit menu provides the following operations :

    Delete, The user selects the dimensional constraint (distance or angle), or the topology that wants to delete. For deletion semantics we have to decide about the following :

    How do we treat references to the deleted object ?

How do we treat references from the deleted object to other objects?

For the second problem it is reasonable to assume that we can purge a topology or a geometry from the representation only when there are no references from other topologies to it. Since the representation is hierarchical we cannot have cyclic references. To implement this policy we can either keep a reference count or we can keep a list with all the objects that refer to this object. We have chosen the second approach for efficiency.

For dealing with the first problem we have defined the following deletion semantics :

- Deletion of an end vertex of a segment: This will make the segment become a ray.
- Deletion of the starting vertex of a ray : The ray is extended to tline. The orientation remains the same in both cases. In both cases the underlying line remains untouched, this is one of the advantages of using hierarchical representation.
- Deletion of an end vertex of an arc : This will make the arc, tcircle. We apply conservative deletion of the other end vertex as well.
- Deletion of the center vertex of a circle : Since a tcircle has no reference to the center vertex, deleting the center point affect the tcircle.
- Deletion and editing of chains, loops and faces Deletion and editing of chains, loops and faces is done using the Face, and Chain buttons, from the Edit menu. With the middle button of the mouse we can delete a loop from a face, by deleting all loops the face is deleted as well. Similarly with the chains.

Intersect, Intersects a line/segment with a line/segment and creates a vertex on the intersection point.

Segment_of, The user creates a segment (arc) on a line (circle), by placing two end vertices on that object. A segment (arc) is constructed having the same underlying line. Since the line (or circle) can be auxiliary the user has an alternative way of constructing collinear segment (or arcs on the same circle).

Unconstrain, with this button the user can delete non dimensional constraints, one just selects a topology and a menu with all the nondimensional constraints involving the major underlying geometry will appear, now the user can select any of them for deletion.

Cut, with this button the user can split a segment into two segments by placing a vertex anywhere on the segment, the old segment is replaced by two new segments that have the same underlying line. This operation works also for lines (a line is split in two rays), for rays (a ray is split in a ray and a segment), for arcs (an arc is split in two arcs), and for circles (the circle is replaced by one arc where the two end vertices are the same).

Move, with this button the user is able to move the placement of the dimension labels, further more with the middle button the user can move an angle constraint between two lines to any of the four regions formed by the two lines. When the angle constraint is imposed on the underlying lines of two segments only two regions are available.

Complement, with this button the user can substitute an arc by its complementary arc. This is done by simply switching the order of the end vertices.

The UI was implemented in C++ and uses the Motif widgets. The parser for the UI was constructed using the lex++ and yacc++ tools.

3. THE CONSTRAINT SOLVING IN EREP

First we will give some basic definitions useful for understanding our method. Then we will present our constraint solving method.

3.1 Basics

In this chapter we will be concerned with configurations involving lines, points and circles with prescribed radii. It follows from the discussion in Chapter 2 that arcs, segments, rays and other higher level topologies can be supported by those three basic geometries. For the time being we shall assume that we have no fixed geometries.

It is straightforward to prove that circles with prescribed radii can be treated as points by transforming the constraints in which they are involved (tangency, on constraints, distance, concentric). Also, by doing some preprocessing all constraints are being translated to angles and distances, so the number of cases that we have to deal with is reduced, significantly.

So finally we have the following problem: Given a set V of n geometries (points and lines), and a number of distance and angle constraints among them, determine the values that we must assign to the points and lines to satisfy the given constraints.

We allow the following types of angles and distances :

- line to line (parallel) distance
- line to point distance
- point to point distance
- line to line angle

The details of the semantics of these constraints are explained in chapter 4. Since the relative positioning of two lines is determined completely by a parallel distance we could either preprocess the input identifying groups of lines constrained by parallel

distances, or we could treat them as a special case in the core of the constraint solver. To keep the the solver simple we choose the first solution. For the same reason we identify classes of points that lie on each other (have distance practically equal to 0), and we replace each such class by one point.

Finally we reformulate our initial constraint problem :

Given a set V of n points and lines and a set E of pairwise constraints among them, find an intuitive solution that satisfies the given constraints. A pairwise constraint may be : non-zero point-point distance, point-line distance and line-line angle. More formally $E : V \times V \rightarrow \Re$, We will often call the mapping E, configuration of the sketch. We call constraint graph of the problem the ordered pair $G = (V, E)$.

The constraint graph of a problem is an undirected graph with the nodes representing the geometries and the edges representing the constraints. The edges have labels that represent the values of the distances or angles specified.

Example: In figure 3.1 we can see a sketch involving 4 lines and 6 points. We have 8 implicit point-line distances that are 0, 2 explicit point-line distances, 3 angles and 4 point-point distances.

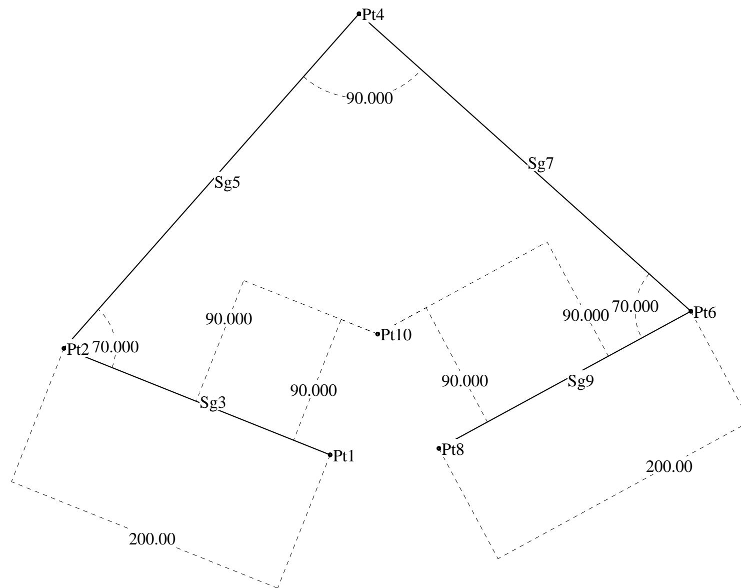


Figure 3.1 A sketch with 10 geometries constrained by 17 constraints

In figure 3.2 we see the constraint graph of the corresponding constraint problem.

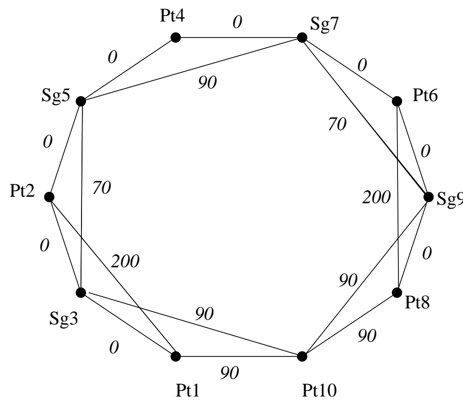


Figure 3.2 The constraint graph of the previous sketch

3.1.1 The algebraic equations describing distances and angles

Let p_1 and p_2 be two points and $l_1(n_1, r_1)$ and $l_2(n_2, r_2)$ be two lines, where n_1, n_2 are the normal vectors with $||n_1|| = ||n_2|| = 1$).

1. a distance d, $d \geq 0$ between p_1 and p_2 means :

   $||p_1 - p_2|| = d$ (two solutions)

2. a signed distance d ($d \epsilon \Re$), between p_1 and l_1 means :

   $p_1 \cdot n_1 = r_1 + d$ (one solution)

3. a distance d, $d \geq 0$ between two lines l_1, l_2 means :

   $n_1 = n_2$ and $|r_1 - r_2| = d$ (two solutions) or

   $n_1 = -n_2$ and $|r_1 + r_2| = d$ (two solutions)

4. an signed angle α between a pair of oriented lines (l_1, l_2) means:

   $n_1 = (n_x, n_y)$,

   $n_2 = (n_x \cos\alpha - n_y \sin\alpha, n_y \cos\alpha + n_x \sin\alpha)$,

3.1.2 Some more definitions

Each line or point has two degrees of freedom. Each distance or angle corresponds to one equation. If we have no fixed geometries then we expect :

$|E| = 2 * |V| - 3$, where $|V| = n$

The minus three is introduced because the resulting sketch has to be a rigid body. (this accomplished by having 3 degrees of freedom). The reason for this is that since the constraints determine only the relative position of the geometries we expect our sketch to be subject to rotation (1 degree of freedom) and translation (2 degrees of freedom).

Intuitively a dimensioned sketch is considered to be well constrained if it has a finite number of solutions for non degenerate configurations. Similarly a dimensioned sketch is considered to be under constrained if it has an infinite number of solutions for non degenerate configurations. Finally a dimensioned sketch is considered to be over constrained if it has no solutions for non degenerate configurations.

For the purposes of our problem we can give the following definitions :

A sketch is structurally over-constrained if its constraint graph contains a subgraph (not necesserily proper) with m vertices and more than $2m - 3$ edges. Note that a structurally over-constrained sketch is always overconstrained.

A sketch is structurally under-constrained if it is not structurally over-constrained, and the number of constraints is strictly less than $2 * n - 3$.

A sketch is well-constrained if it is not over-constrained, and the number of constraints is equal to $2 * n - 3$.

An erroneous configuration, of a well-constrained or under-constrained sketch is one under which there is no feasible solution.

A degenerate configuration, of a well-constrained sketch is one under which the number of solutions is infinite.

A degenerate configuration, of an over-constrained sketch is one under which there are one or more solutions (finite or infinite).

3.2 The constraint solving method

We first create the constraint graph that corresponds to our constraint problem. Specific graph reduction steps are applied that correspond to geometric construction steps with ruler and compass, and derive clusters of geometric elements that are correctly placed with respect to each other. By a recursive extension, each cluster is then considered as a virtual geometric element, and the solver places the clusters with respect to each other. The recursion can go to arbitrary depth. It is easy to prove that for well constrained sketches the constraint graph is well constrained. So by doing a depth first search we can easily determine the clusters, ie parts of the sketch that are sequentially constructible.

3.2.1 Finding the sequentially constructible subgraphs

The basic idea of the solver algorithm is now as follows:

1. Pick two geometric elements (graph vertices) that are related by a constraint (connected by an edge) and place them with respect to each other. The two elements are now known, and all other geometries are unknown.

2. Repeat the following: If there is an unknown geometric element with two constraints relating to known geometric elements, then place the unknown element with respect to the known ones by a construction step. The geometric element so placed is now also known.

For example, in the graph of Figure 3.2, we may begin with elements Sg_5 and Pt_2, by drawing a line Sg_5 and placing anywhere on it the point Pt_2 anywhere. We can now place in sequence Sg_3, Pt_1, and Pt_{10}. At this point, no additional elements can be placed and the cluster is complete, we name this cluster U (figure 3.3).

Note that we neither know where Pt_4 is situated, nor how far the point Pt_8 is located. Starting again, two other clusters are determined. One consists of Pt_{10}, Pt_8,

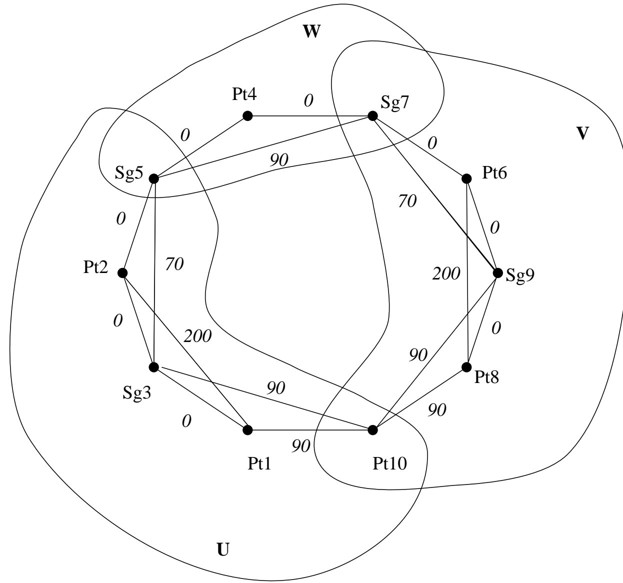


Figure 3.3 Finding the clusters of the graph

Sg_9, Pt_6, and Sg_7 (cluster V). The other cluster consists of Sg_5, Pt_4, and Sg_7 (cluster W). Note that the same geometric element may occur in more than one cluster.

All clusters are shown in Figure 3.3.

3.2.2 Recursion

Three clusters, each sharing a geometric element with one of the others, can be placed with respect to each other by identifying the shared elements.

In the example of Figure 3.3, there are three clusters sharing the elements Sg_5, Sg_7, and Pt_{10}. To place them, we compute the distance of Pt_{10} from Sg_5 in cluster U, and the distance of Pt_{10} from Sg_7 in cluster V. The angle between Sg_5 and Sg_7 in cluster W is already known. These three shared elements can now be placed, thereby fixing the relative position and orientation of the three clusters.

The cluster placement rule conceptually build a larger cluster from three smaller ones. Additional clusters sharing two elements with this new "super cluster" can be

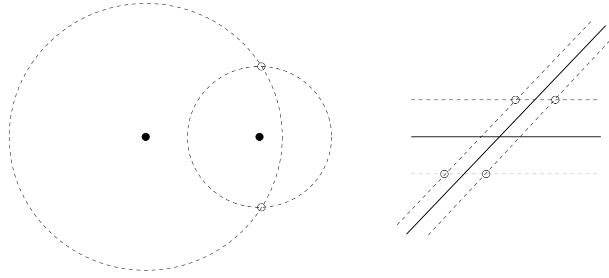


Figure 3.4 Point placement rules

added in the same way, thereby growing larger clusters from smaller ones. Recursively, super clusters can be placed with respect to each other in the same way.

3.2.3 Construction steps

The reduction steps correspond to standardized geometric construction steps, and also to solving standardized, small systems of algebraic equations. The construction steps include the following:

Basis Steps: The basis steps place two geometric elements related by a graph edge. They include placing a point on a line, placing two lines at a given angle, placing two points at a given distance, and so on. Note that in general there are several ways to place the geometric element.

Point Placements: These rules place a point using two constraints. They include placing a point at prescribed distance from two given points, or at prescribed distances from given lines, and so on. See also Figure 3.4.

Line Placements: These rules place a line with respect to two given geometric elements. They include placing a line tangent to a circle through a given point, at given distance from two points, etc.

Circle Placement: These rules place a circle of fixed or variable radius. Fixed-radius circles require only two constraints and determining them can be reduced to placing the center point. Variable-radius circles require three constraints and reduce

in many cases to the Apollonius problem — finding a circle that is tangent to three given ones.

Cluster Placement: Clusters are placed by placing shared geometries. If necessary, the relationship between the shared geometric elements is computed within each cluster, whereupon the two or three shared elements can be placed with respect to each other.

Algebraic Formulation: Geometric elements are represented as follows: Points are represented by Cartesian coordinates. A line is determined from its implicit equation in which the coefficients have been normalized:

$$\mathtt{a}: \; m\,x + n\,y + p = 0 \qquad n^2 + m^2 = 1$$

It is well-known that in this formulation p is the distance of the origin from the line. Because of the normalization, lines are determined only by two numerical quantities, the (signed) distance p of the origin from the line, and the direction angle $\cos\alpha = n$. Therefore, two constraints determine a line. Lines are oriented by choosing $(-n, m)$ as the direction of the line. Circles are represented by the Cartesian coordinates of the center and the radius, an unsigned number.

In many cases it is quite obvious how to determine the coordinates of the next geometric element from the constraints relating it to known geometric elements. By restricting to simple construction steps, the basic algorithm solves at most quadratic equations. In some cases, up to three simultaneous quadratic equations must be solved. For example, given three circles of fixed radius, finding a circle tangent to all three requires solving the following system,

$$\begin{aligned}
(x - x_1)^2 + (y - y_1)^2 &= (r \pm r_1)^2 \\
(x - x_2)^2 + (y - y_2)^2 &= (r \pm r_2)^2 \\
(x - x_3)^2 + (y - y_3)^2 &= (r \pm r_3)^2
\end{aligned}$$

where the choice of the sign on the right-hand sides determines which of up to eight possible solutions is determined. Here, (x_k, y_k) is the center of circle k, and r_k is its radius. Such constructions are done by precomputing a normal form of the system

Figure 3.5 Incidence of A with a and C with c and resulting constraint graph.

from which to the unknowns are easier to find. Preprocessing can be done using Gröbner bases; for example see, [Buc85].

3.2.4 Graph transformations

The scope of the basic solver can be extended by certain graph transformations. For example, when two angle constraints α and β are given between three lines, then a third angle constraint can be added requiring an angle of $180° - \alpha - \beta$. Similar transformation rules can be introduced for simple geometric relationships.

Graph transformations are a simple and effective technique to extend the scope of the solver. However, one should avoid transformations that restrict the generality of the solution. For example, consider the configuration shown in Figure 3.5 in which the point A is constrained to be on line a, and point C on line c. The situation implies that either lines a and c are incident, or that points A and C are incident. As discussed further below, the two possibilities lead to different solutions. If we were to apply a transformation to the constraint graph that added one of the incidences as new graph edge, then we would have excluded the other possibility, and with it some solutions. If we added both incidence edges, then we would have introduced the unwarranted assumption that both the points and the lines coincide. In each case we can exhibit examples in which a solvable constraint problem becomes unsolvable.

4. ROOT IDENTIFICATION

For a well constrained sketch we usually have a finite number of different solutions, only one of these solutions is acceptable by the user. The identification of this solution is called the root identification problem.

The distances, angles and other constraints that the user imposes upon the geometries, to get a well constrained sketch, give more than one consistent solutions. This holds even for trivial sketches. For example assume that the constraint solver is presented with the well constrained sketch shown in figure 4.1. The sketch consists of three points and three lines, on which the user has imposed the following constraints:

Pt_4 on Ln_1, Pt_4 on Ln_2, Pt_6 on Ln_2, Pt_6 on Ln_3, Pt_5 on Ln_3, Pt_5 on Ln_1, the distance between Pt_4 and Pt_5 is 80, the distance between Pt_4 and Pt_6 is 60, the angle between Ln_1 and Ln_2 is $40°$;

These constraints make the sketch well constrained.

The sketch shown in 4.2 is consistent with the above constraints but it is obviously not what the user wanted. (this particular sketch will become unambiguous only if we apply the projection rule, explained later).

The number of solutions of a well constrained sketch can be (and in general is) exponential on the number of geometries.

We can verify this even if we restrict our geometries to points only. Let's assume that we have n points constrained by $2 * n - 3$ distances, and that the constraint graph is serially $(2, C)$-constructible. We shall first place arbitrarily two points that are constrained by some distance, and then construct one point at a time from two already known points. At each one of the last $n-2$ construction steps, we have 1, 2 or 0 choices depending on the relative position of the two circles (tangent, intersected,

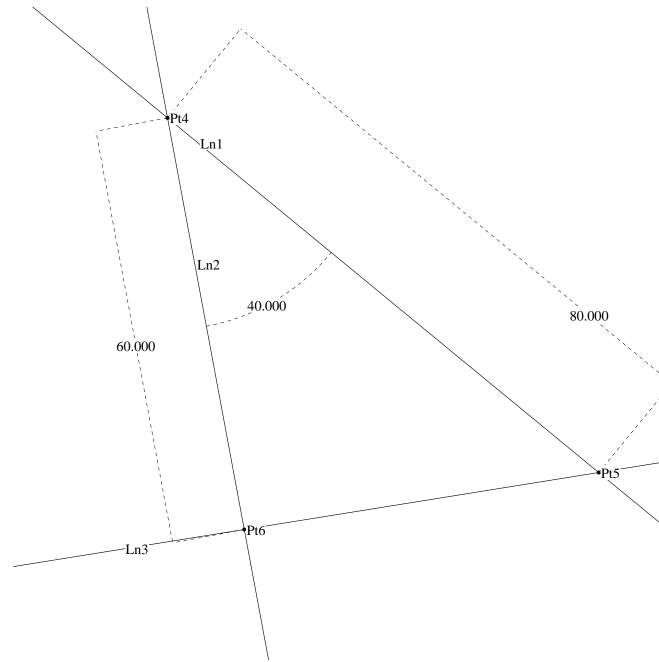


Figure 4.1 A well constrained sketch with 3 lines and 3 points

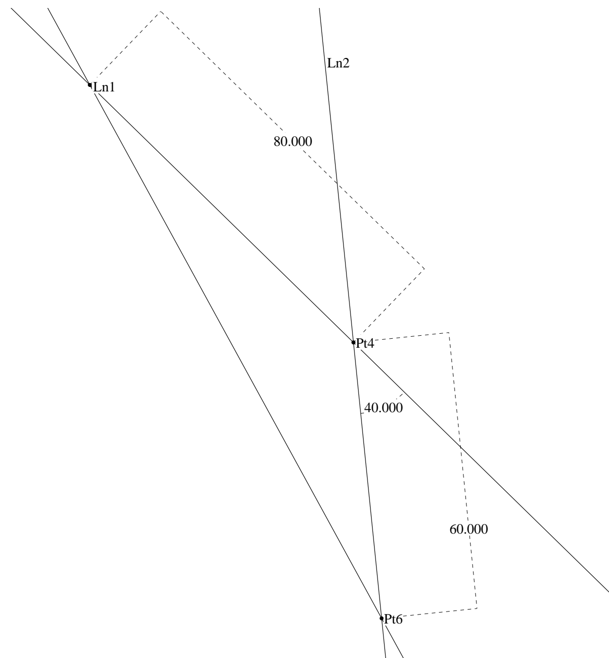


Figure 4.2 This solution is consistent but not intuitive

and disjoint respectively), so we may have 2^{n-2} solutions. These solutions are in general distinct.

The example shown in figure 4.3 is an instance of the above problem where we have exactly 2^{n-2} distinct solutions, The particular dimensioning scheme can be solved easily; we can first place the points Pt_1, Pt_2 at distance 50, then construct one point at a time from Pt_1 and Pt_2. It is obvious that we can take exactly 2^{n-2} distinct solutions by placing the points Pt_3 through Pt_n on either side of the segment $\vec{Pt_1Pt_2}$. A similar more complicated example with 2^{n-3} discrete solutions is presented in [Owe91].

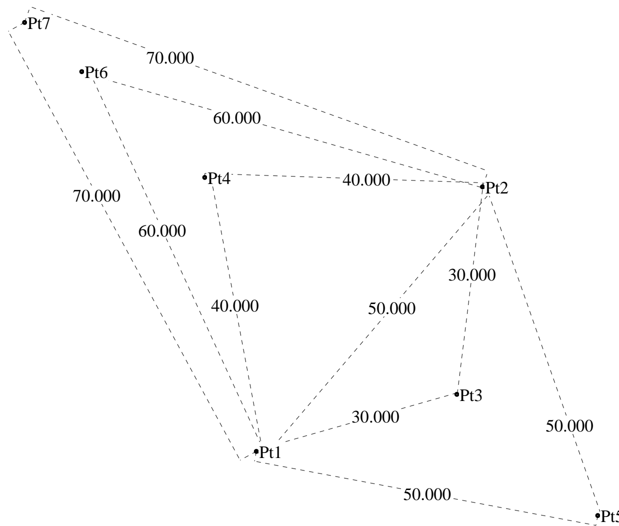


Figure 4.3 A well constrained sketch with 2^{n-2} solutions

In this chapter we will review some commonly used methods for root identification and we will pinpoint their deficiencies, then we will present the method and the interactive framework provided by Erep. Here won't address the case of underconstrained sketches, this is a different problem that is discussed in chapter 3.

4.1 Root identification for iterative methods

Most numerical (iterative) constraint solvers, instead of dealing with the root identification problem, claim that the solver will converge to the closest (numerically) solution. Since the number of solutions is exponential the problem is rather ill conditioned, for numerical methods. Until now, no research has been made on iterative methods that could be navigated towards the appropriate solution by using some extra information.

The user is usually forced to make the sketch as perfect as possible, so that the numerical values of the geometries in the draft, are close to those of the desired solution. Further more, changes in the dimension values should be kept as small as possible for the same reason. Changes in the dimension values are usually not reversible in such methods, so the system must preserve a design history.

The major disadvantage, of these methods is that once one gets a solution that he didn't wanted there is no standard way for debugging it. The user would have to experiment until he derives what he wanted. In some cases it may be even impossible to derive the desirable solution under this dimensioning scheme but the user has no way to know about this.

4.2 Specifying the correct solution by imposing general rules

The central idea in these methods is to prefer solutions with certain properties, usually related to the application. For example we may favor solutions that contain only closed, convex or non-self-intersecting objects.

There are two major problems in using such methods :

- The general rule that we impose may not comply with the intentions of the user, that usually wants to construct a sketch incrementally. This implies that in some phases of the construction the objects may be open or self-intersecting.

- Even for very restricted cases these problems are computationally hard. For example, assume that the set of points is the set of vertices of a polygonal cross section. In that case, application-specific information might require that the resulting sketch is a simple polygon; that is, it should form a polygon that is non self-intersecting. In addition we may give a cyclic ordering specifying how to connect the points to obtain the polygon. This additional requirement makes the problem NP-complete. More specifically the following holds :

  Theorem [Cap93]: Given n points in the plane that are well constrained by $2n-3$ point-point distances, and a cyclical ordering specifying how to connect the points to obtain a polygon. Then identifying a solution for which the resulting polygon is simple (non self-intersecting), is NP-complete.

4.3 Specifying the correct solution by over-constraining

This is a natural method that engineers commonly use in their drawings. In figure 4.4 we see a typical case in which this method can be applied to get a unique solution. In this figure we have three (non oriented) parallel lines, if we just determine the two distances d_1 and d_2 we get the two solutions (up to rigid motion), shown in figure 4.4. Note that since the lines are not oriented we have 2 choices for the interpretation of each distance constraint (except the first one), so we have in general 2^{n-2} solutions for n lines and $n-1$ distances that are correctly constrained. If we impose the third distance d_3, in the our example we get a unique solution.

Although this method seems appealing, it is impossible to find an efficient algorithm even if we restrict the problem in simple cases, like the generalization of the above example:

"Given n parallel lines and n parallel distances among them, is there any solution that satisfies them?"

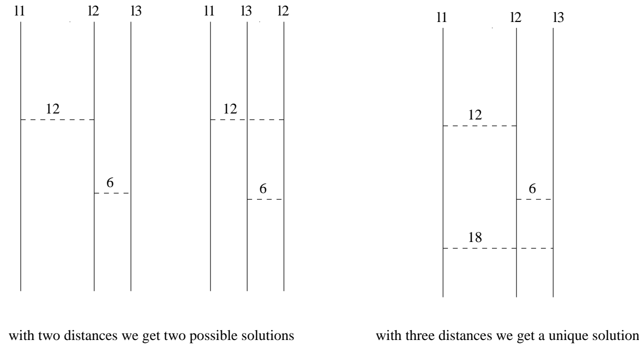


Figure 4.4 Constraining three parallel lines with distances

We will prove that this problem is NP-complete by reducing the integer partition problem [Kar72] to it (since that existential problem is NP-complete the problem of finding the correct solution is obviously at least as hard).

Proof : The integer partition problem can be formulated as follows :

"Given a set of n positive integers A, is there a partitioning of A into two disjoint sets A_1 and A_2, such that the sum of the integers in A_1 equals the sum of the integers in A_2?

Let's assume that we have the following instance of the partition problem :

$A = \{s_1, s_2, ..., s_n\}$,

for that instance consider the instance of our problem that appears in figure 4.5,

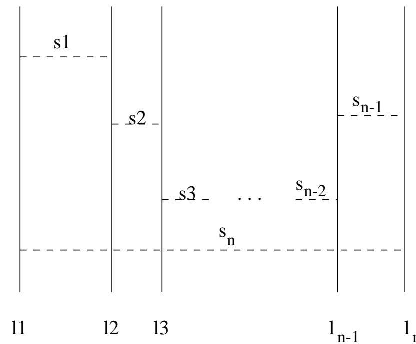


Figure 4.5 An instance of the n parallel lines problem

If the partition problem has solution, then there is a partitioning of the integers in two sets, A_1 and A_2 such that the sum of the integers in the two sets is the same. Then we can place the lines l_1 through l_n, so that all the n distances are satisfied. We place l_1 on the y-axis, then if s_1 is in A_1, we place l_2, at distance s_1 at the right of l_1, else if s_1 is in A_2, we place l_2, at distance s_1 at the left of l_1. Since the sum of the integers in the two sets is the same the last distance s_n, we will get us back to line l_1 (zero on x-axis).

If our problem has a solution there is the obvious splitting : if we choose a left placement to satisfy a distance the corresponding integer is in A_1, if we choose the right placement to satisfy a distance the corresponding integer is in A_2. Then the sum of the integers in the two sets is the same.

The problem is also in NP, since we can arbitrarily choose one of the finite number of solutions (exponentially many of course) and check whether it is consistent or not in polynomial time. So our problem is NP-complete. □

Although the partition problem is pseudo-polynomial ([GJ79]), the fact that the distances can be real numbers, means that the only bound for the corresponding integers of the partition problem is the machine precision (we may divide all integers of the partition problem instance by the larger integer, this will give us rationals in the range [0.0,1.0]).

4.4 The rules for root identification in EREP

The main philosophy behind the rules used in Erep for determining the desired solution is that the user should determine the solution by placing just the constraints needed.

We classify the rules for root identification in two categories namely natural and heuristic rules.

Whenever the user sets a distance between a line and a point, the point will have to be on the same side of the (oriented) line before and after the constraint solving, this is called the side rule. On the other hand, an angle determines completely the

relative positioning of two (oriented) lines, this is called the angle rule. Similarly the parallel distance between two lines determines completely the relative positioning of the two lines, this is called the parallel distance rule.

The side, angle, and parallel distance rules are almost natural and come from the concepts of signed distance and signed angle, that's why we call them natural rules. The usage of oriented curves, signed distances and similar information for determining unambiguously a sketch have been studied in [Ver90]. Unfortunately, as we have already seen, these rules are not sufficient. For this reason we will introduce two more rules which we shall call heuristic rules, these rules will handle the positioning of three points with respect to each other (three point rule), and the positioning of two points and one line with respect to each other (projection rule). Between two geometries that the user has not related by constraints (in a way that we will define later) we won't enforce any rules.

First we shall discuss the parallel distance and angle rules. In Erep the parallel distance and angle rules are built in, this is to say that the user has no control on them.

The parallel distance rule, is supported by our Erep grammar and has the following meaning; whenever a parallel distance between two oriented lines is defined the relative position of the two lines is completely defined. In figure 4.6 we see the four different cases for two lines constrained to lie at distance 5 from each other. Each case has different representation in the Erep grammar.

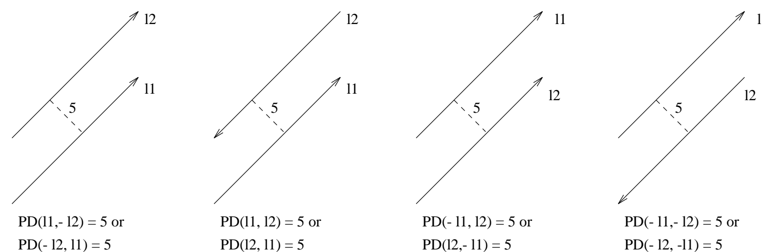


Figure 4.6 Two lines constrained to lie at distance 5

The angle rule, is supported by our Erep grammar and has the following meaning; whenever an angle between two oriented lines is defined the relative position of the two lines is completely defined. In figure 4.7 we see the four different cases for two lines constrained to form an angle of 45°, along with two of the plausible Erep representations.

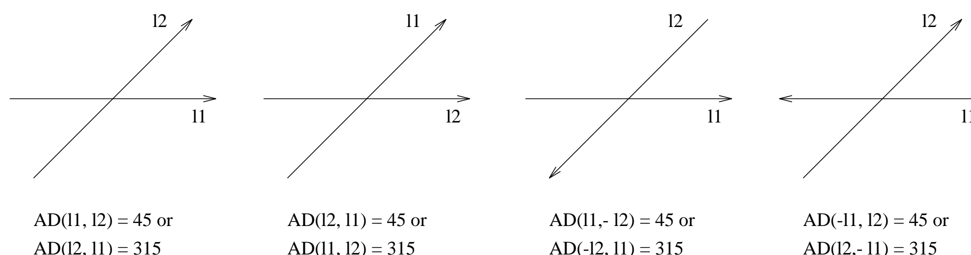


Figure 4.7 Two lines at 45° angle

Now we will explain the rest of the rules by describing their usage in our construction algorithm. For the time being we shall restrict ourselves in lines and points only. So a geometry can be either a line or a point. As we discussed earlier, the algorithm places the geometries in construction steps. At each step we construct one geometry from two already known. To reduce the number of cases we can imagine that we place 3 geometries with respect to each other and then by rotating and translating we adjust them to the coordinates of the two geometries that we already know. In this way the number of cases reduces to three. The approach that we adopted in Erep can be described by considering each case separately:

1. Three points: In this case we have three points with their pairwise distances determined. We arbitrarily place the two points p_1 and p_2 with respect to each other, so we have 0, 1 or 2 solutions for the third point p_3. Our solution is given by intersecting two circles, the circles can be disjoint (no solutions because the triangular enequality does not hold), tangent (1 solution), or they can intersect giving 2 distinct solutions. By applying the three point rule we choose the one

in which p_3 is on the same side of the oriented segment $\vec{p_1p_2}$, as it was initially. In figure 4.8 we can see the two alternatives for such a case, and the choice that Erep makes.

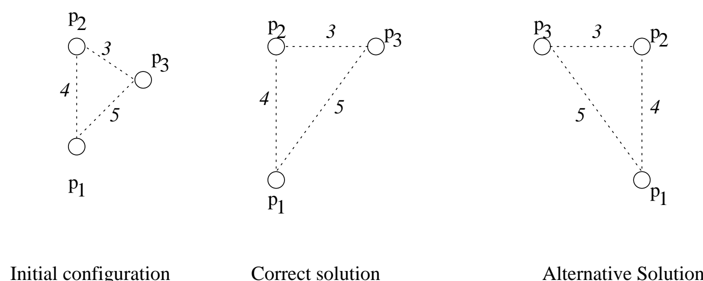


Figure 4.8 Root identification in placing three points

2. One point and two lines: In this case we have the angle between the two lines and the two point-line distances determined. We place the two lines l_1 and l_2 with respect to each other using the angle rule, then we have four choices (in general) for placing the point p with respect to the two lines. The four solutions are shown in figure 4.9. Since every line has an associated orientation, by applying the side rule we enforce the point p to lie on the same side of the two lines, as it was initially.

3. One line and two points: Here we have all three pairwise distances determined. In the general case we have 8 consistent, discrete solutions. We first place one of the points p_1 and the line with respect to each other, to do this uniquely we enforce the point to lie on the same side of the line as before (side rule). Now we have four solutions in the general case which are shown in figure 4.10, by enforcing the second point p_2 to lie on the same side of the line as it did before (side rule again) we restrict ourselves in 2 solutions, to get a unique solution we use the projection rule: from the two solutions, we select the one in which the orientation of the projection of the directed segment $\vec{p_1p_2}$ on the

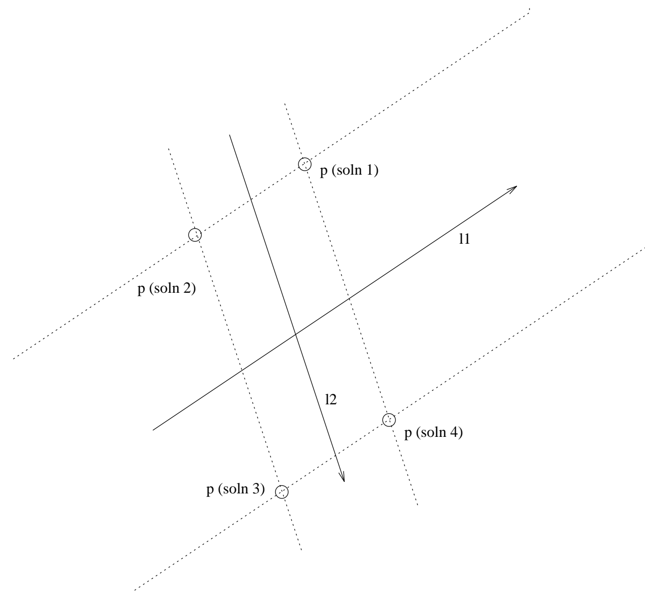


Figure 4.9 Root identification in placing a point and two lines

line preserves its relation with the orientation of the line. The initial relation can be derived by considering the inner product $\vec{t} \cdot \vec{p_1 p_2}$, where $\vec{t}$ is the tangent vector of line l, if this product is positive we conclude agreement, otherwise we conclude disagreement.

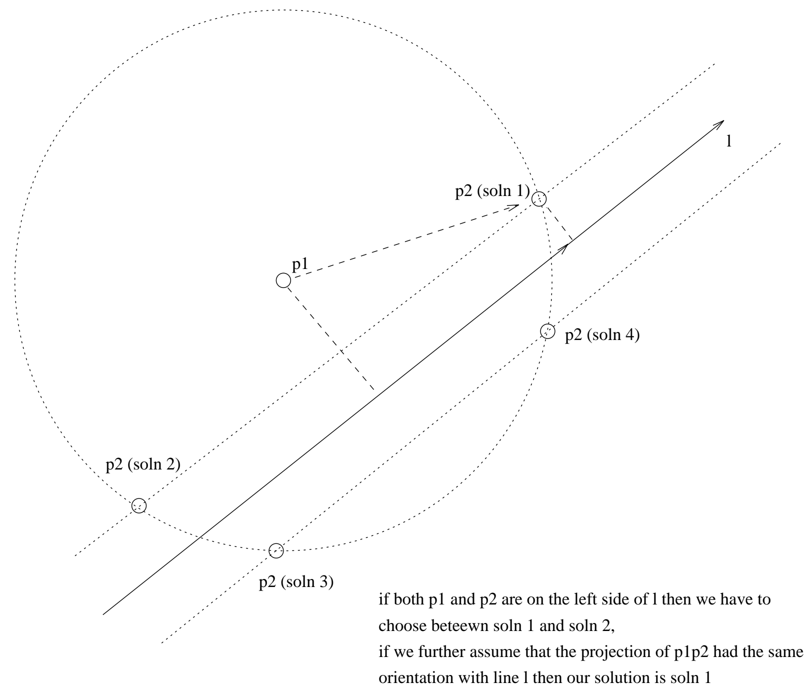


Figure 4.10 Root identification in placing two points and one line

4.5 Advantages and disadvantages of the approach adopted in EREP

The method described above is efficient, natural and well conditioned :

- It is natural for the user, because it tries to maintain the relations among the geometries according to the draft made by the user.

- In contrast to numerical root selection, it does not depend on the numerical values of the geometries in the draft made by the user. It depends only on logical relations between two or at most three geometries.

- It is easy to implement in any constructive constraint solving system.

- Changes in the dimension values are inversible. This means that if the user changes the value of one dimension from α to β, regenerate the sketch and then changes the value of the same dimension from β to α, he will take the original sketch. This holds even if the user changes simultaneously many dimensions and does not depend on the amount of change. This can be proved if we recall that the logical relations on which the constraint solver bases its choices are kept the same before and after the constraint solving.

- It is fast, so it does not add any overhead to the constraint solving algorithm, which remains almost linear.

The method used in Erep has some drawbacks, some of them can be overcome by the interactive framework for root selection, which is discussed in section 4.6

- It is rather inappropriate for any numerical (iterative) constraint solving system. This makes it inappropriate for the generalization discussed in section section for extensions, where we have to use another method.

- This method does not help us in determining a unique solution in degenerate sketches. Erep treats such sketches as underconstrained. In figure 4.11 we see a sketch that is, well constrained and serially $(2, C)$-constructible, a construction sequence may be:

  place Pt_1 and Sg_9 with respect to each other,

  construct Sg_3 from Sg_9 and Pt_1,

  construct Pt_2 from Pt_1 and Sg_3,

  construct Sg_5 from Pt_2 and Sg_3,

  construct Pt_4 from Sg_5 and Pt_2,

  construct Sg_7 from Pt_4 and Sg_5,

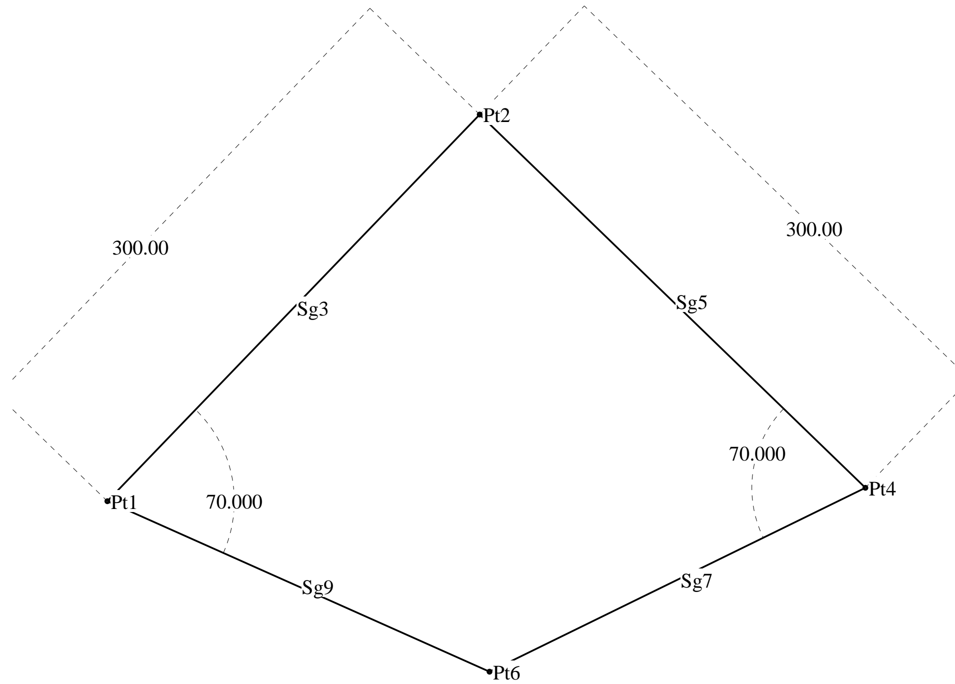


Figure 4.11 A well constrained, serially constructible sketch

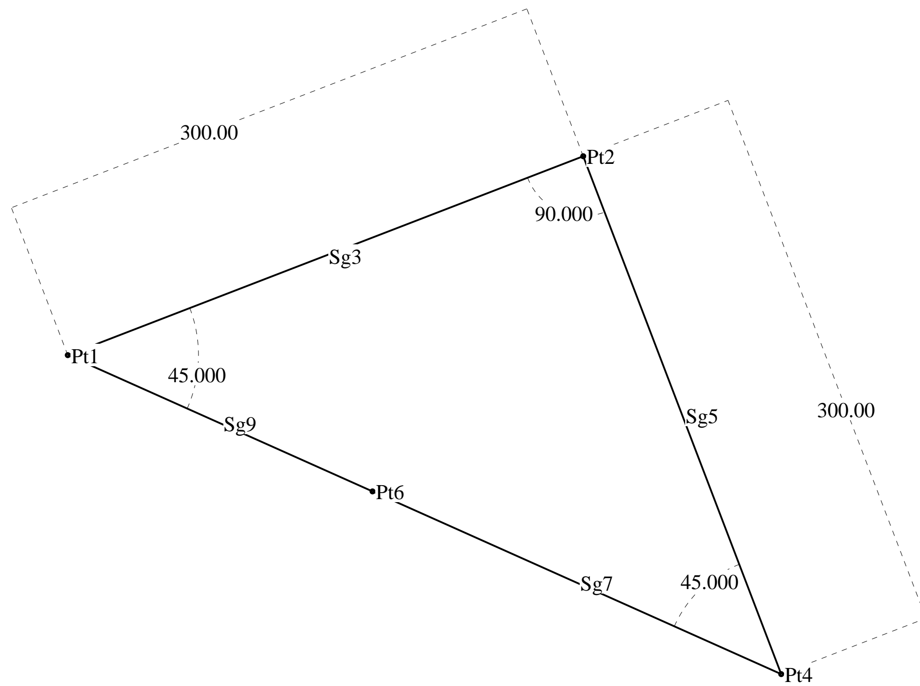


Figure 4.12 A degenerate configuration

construct Pt_6 from Sg_9 and Sg_7.

So far so good, now the user changes the 70° angles to 45°. This will cause a problem in the last step of the construction, finding the intersection point of two parallel lines may result to no solution or to an infinite number of solutions. So point Pt_6, could be placed anywhere along the lines Sg_7 and Sg_9, giving an under constrained sketch (see figure 4.12).

If we examine this particular case, and set the parameters of the problem as in figure 4.13 (the two sides have the same length, the origin corresponds to Pt_2), then we can easily verify that when $\alpha \neq \beta$, or $\alpha \neq \pi/4$, we have one unique solution for the intersection point P. When $\alpha \approx \beta = \pi/4$, then on the limit the solution is the midpoint $P = (c/2, c/2)$:

$l_1 : y = tan(\pi/2 + \alpha)x + c$

$l_2 : y = tan(\pi/2 + \alpha)(x - c)$, so

$p_x = c\frac{1+tan(\pi-\alpha)}{tan(\pi-\alpha)-tan(\pi/2+\alpha)} \Rightarrow$

$p_x = cos(\pi - \alpha)cos(\pi/2 + \alpha)c\frac{1+tan(\pi-\alpha)}{\pi/2-2\alpha} \Rightarrow$

$\lim_{\alpha\rightarrow\pi/4} p_x = c/2$

So the most intuitive and correct solution would be the midpoint, but this requires knowledge of the detailed situation that forces the two lines to become parallel. This exhibits an inherent disadvantage of the method used in Erep as compared with a numerical constraint solver that would had probably converged to the midpoint in the first place. We can overcome this drawback if we treat the sketch as underconstrained and introduce some appropriate implied constraints.

- Though the notion of signed distance is natural there are cases that the resulting sketch contradicts the intuition of the user. This is especially true for tangency constraints which are translated into distance constraints, there the usage of signed distance results in unintuitive solutions. For example, see the example depicted in figures 4.14 and 4.15.

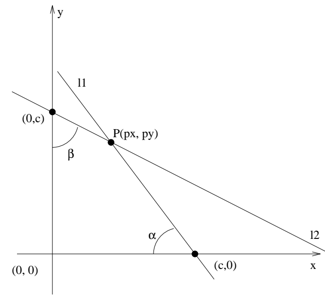


Figure 4.13 Setting the parameters for an instance of the problem

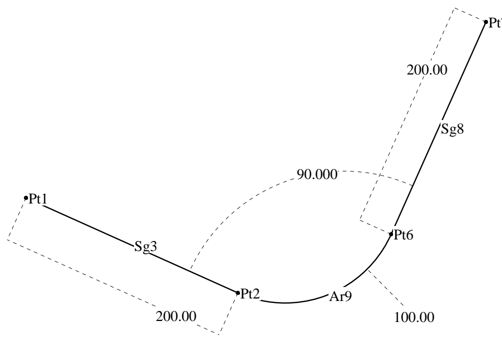


Figure 4.14 The user changes the angle from 90° to 190°

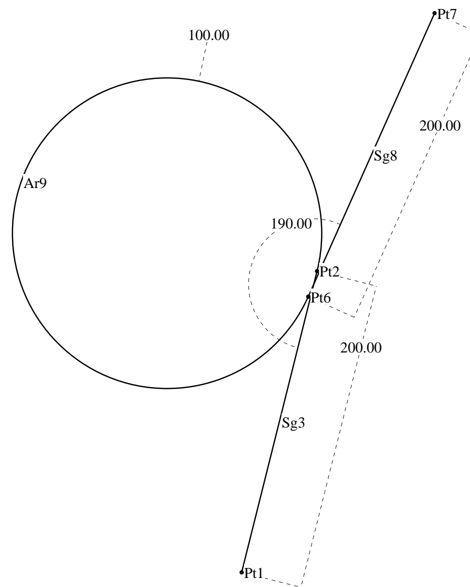


Figure 4.15 The user gets back a rather unintuitive solution

The arc is of fixed radius and is constrained to be tangent to two segments of fixed length. The user would expect the arc to bend when one makes the angle larger that 180°. This is not what we get since the constraint solver tries to keep the center of the circle (actually a point) on the same side of the two lines as it initially was (recall that circles with dimensioned radius are treated as points).

- The usage of heuristic rules may lead to situations that even an experienced user had never anticipated, the reason for this is that the three point rule and the projection rule may involve three geometries that the user had never directly related with each other. This situation comes up when we have 3 rigid bodies (clusters) that have already been solved and we want to place them with respect to each other (see section on constraint solving algorithm). Then we will place the three geometries that connect them with respect to each other, but these three geometries have never been related with each other directly by the user, this means that there is no constraint between any two of them. This problem does not come up when we are applying the natural rules since these are pairwise rules, so we may derive the information from the cluster that contains both geometries.

4.6 An interactive framework for root identification

As indicated from the above, no matter what the set of rules is, the user may get a solution that he did not expected.

There are the following approaches in the literature for dealing with this problem:

- Most variational systems won't provide the user with any alternative except redesigning the corresponding part of the sketch, by imposing a different set of constraints or by paying attention to the initial placement of the objects. Usually no hint is given to the user for what went wrong.

- Some of the systems based on rule-constructive methods, return the whole set of possible solutions, that can of course result in an exponential number of

solutions. For example, if we have 12 geometries, we will have 1024 solutions in average.

- DCM [D-C93], has introduced a move button , that has the purpose of moving a part of the design relative to the rest of it. This button can be used for implicitly affecting some the solution choices, by changing the relative position of the geometries. Since the user has no hint of how the sketch was constructed by the solver, and is not aware of the rules used, in many cases end up with an unintuitive solution. We believe that further research is needed in this direction.

In the 2D sketcher of EREP we have developed an interactive framework that makes it it possible for the user to select the appropriate solution with minimum effort.

The user can step through the construction sequence and be informed of which geometries have been placed at each construction step. This is done using the level selection arrow buttons (see figure 4.16), each level corresponds to one construction step, at each level the geometries involved in the construction are highlighted in purple, so that the user can visualize the construction sequence. In figure 4.16 we can see a well constrained sketch that is constructed by the solver within eleven steps. This information becomes available to the user after regenerating any well constrained sketch. A sketch that is not solved (regenerated) will blank the level display.

In addition, we provide the user with the capability to select an alternative solution for the particular construction step (level). After selecting a solution for a specific level, with the solution selection arrow buttons the user regenerates and gets back the new solution. This process is well suited to the intuition of the human, since each step corresponds to an elementary ruler and compass construction. As explained earlier the number of choices at each level, ranges from 1 to 8. This makes it easy for the user to browse all possible solutions at a particular level and choose the one he wants. In figure 4.16 level eleven has only one solution, so there are no alternatives for the user to select, in figure 4.17 level seven (out of eleven) has four alternative solutions

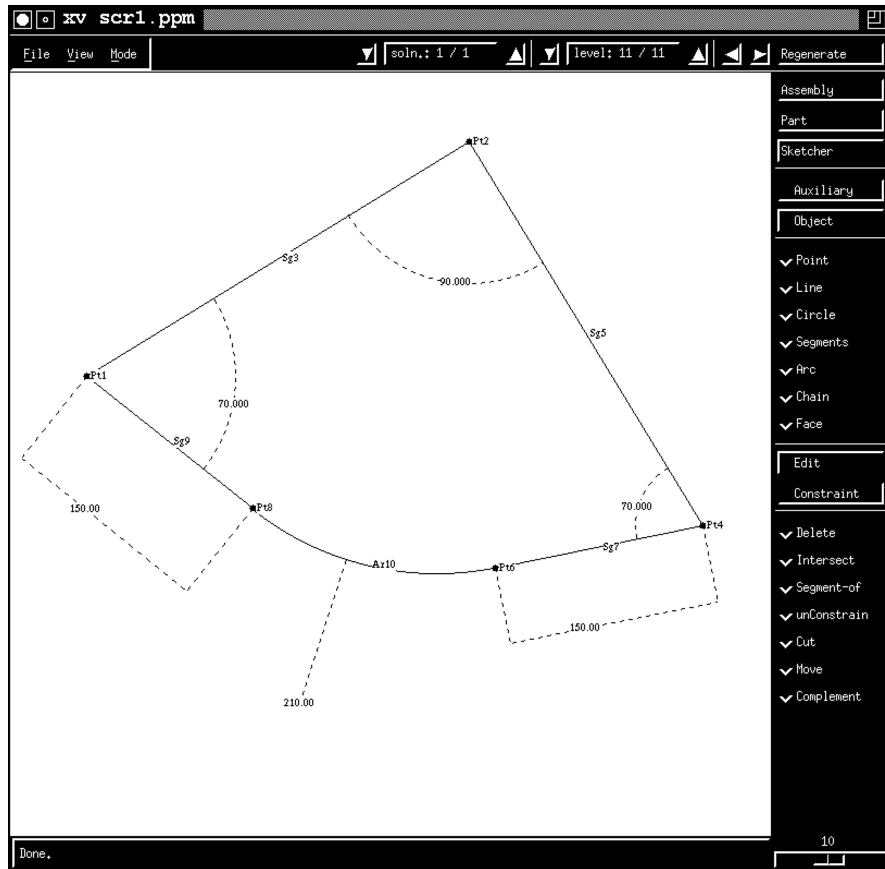


Figure 4.16 The initial sketch with two 70° angles

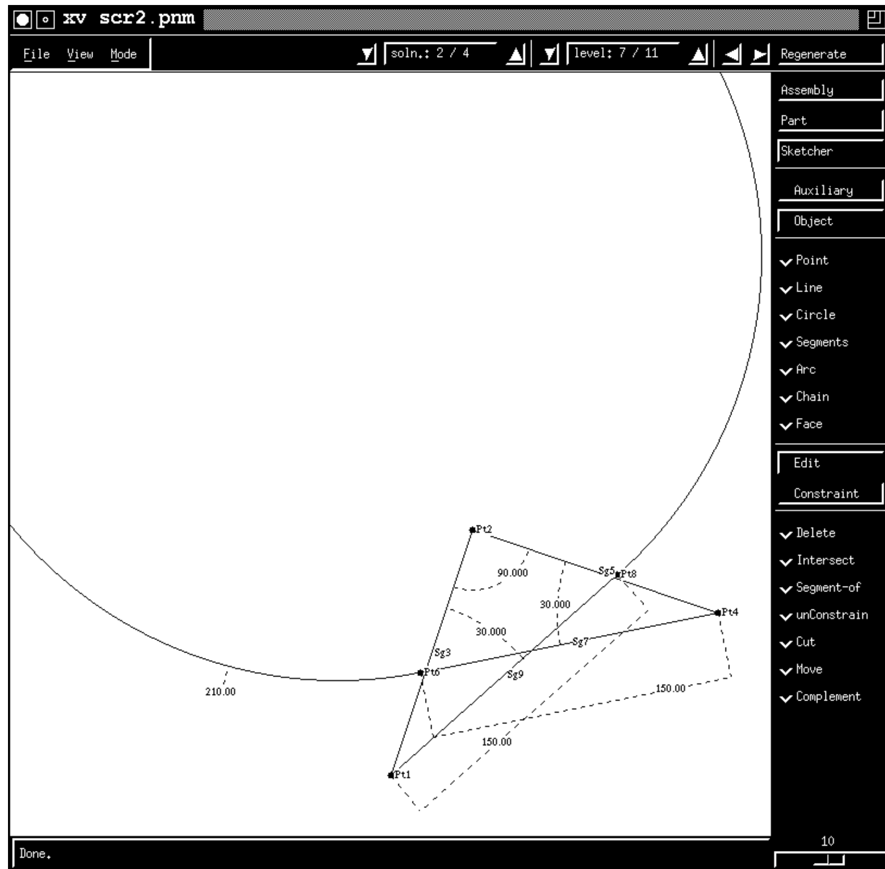


Figure 4.17 After making the two angles 30°

among which the second has been selected. The solution display is available only when the sketch is solved (regenerated), this was enforced to accomplish consistency and compatibility with the future versions of the constraint solving algorithm.

The user is allowed to affect the solution choice only at one level before each regenerate, all previous changes are disregarded except from the most recent. After regenerating the user is free to change any other level. If the user could change the choice in more than one level, the solution derived could be totally unpredictable and unintuitive.

For keeping track of this process we have provided the user with two horizontal buttons, with which he can review the sequence of sketches that his choices have caused.

Finally we have introduced a new button called complement (see figure 4.16), using this button the user can change an arc to its complimentary arc. Especially for the arc complement, instead of treating it as one more choice, we concluded that since it involves only the arc and no other geometries, the user could handle it easier as a special case. Otherwise, we would have to double the number choices for some of the steps.

Let's illustrate how this method works by considering the example depicted in figures 4.16, 4.17, 4.18, 4.19 and 4.20. The user initially sketches and regenerates, the well constrained sketch depicted in figure 4.16, in this sketch the role of the arc is clearly to round the adjacent segments Sg_9 and Sg_7.

When the user changes the two angles from 70° to 30°, the solver returns the solution shown in 4.17, instead of the desired one in figure 4.20. By using the interactive framework the user changes the solution choice at two related steps: in level 7 Ar_{10}, Sg_7 and Pt_6 are highlighted and the user selects solution 4 instead of the default 2 (figures 4.17 and 4.18), in level 4 Ar_{10}, Sg_9 and Pt_8 are highlighted and the user selects solution 1 instead of the default 3 (figure 4.19). Now he got the correct solution with the complimentary arc, so by using the complement button he eventually gets the desired solution (figure 4.20). Using the two horizontal buttons the user

may at any time review any of the previous sketches, in order to compare or correct something.

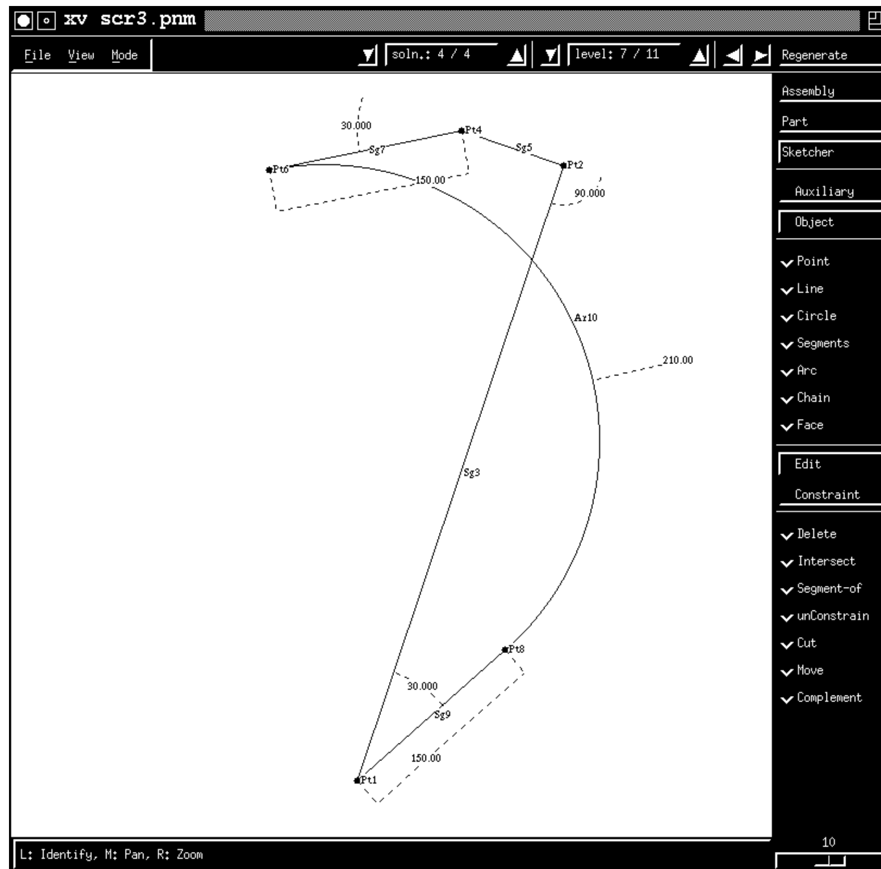


Figure 4.18 Selecting solution 4 (instead of 2) from level 7

Summarizing, the interactive root selection framework provided by the 2D sketcher of EREP, offers the following:

- The user is visually informed of the construction sequence.
- Based on the construction sequence the user can navigate the system to the correct solution.

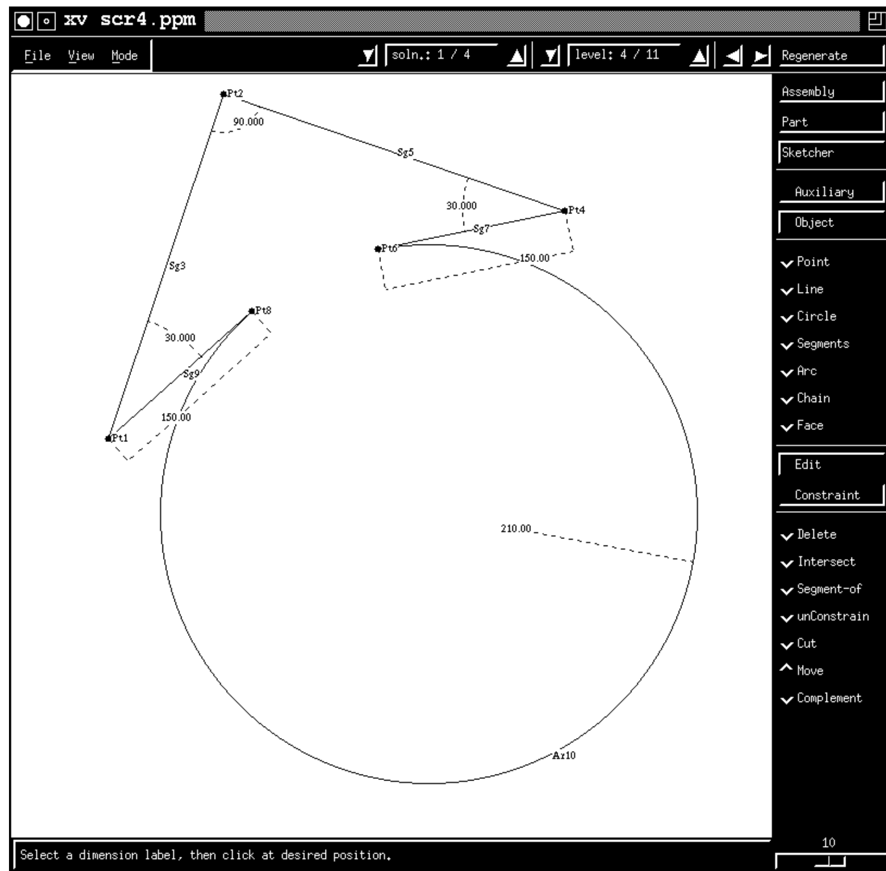


Figure 4.19 Selecting solution 1 (instead of 3) from level 4

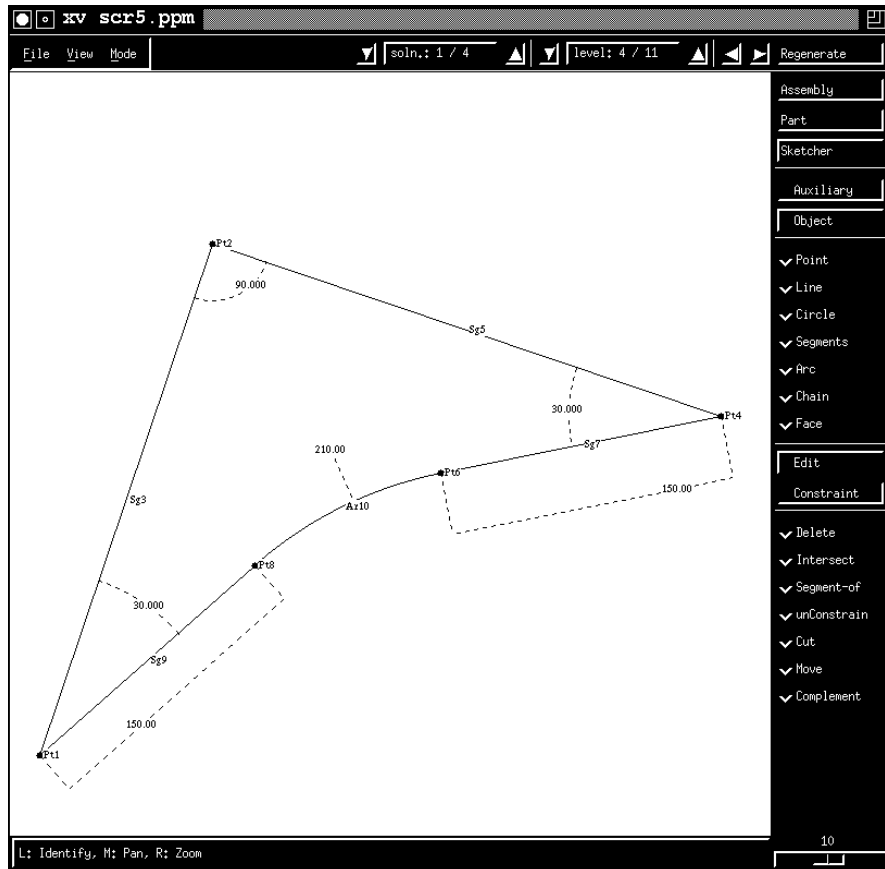


Figure 4.20 Complementing the arc Ar_{10}

- The user can review the navigation sequence, compare and change any step of it, if this is required.
- The interactive framework prohibits the user of affecting non solved sketches to accomplish consistency and compatibility.
- The user has the capability of complementing an arc by a special button.

5. CONCLUSIONS AND FUTURE WORK

We designed a representation that is capable of capturing in a concise high-level manner all the information needed for defining uniquely a dimensioned sketch. The user can even edit the textual representation directly, but there is no need to do this since every action at the level of the User Interface corresponds to a modification of the textual representation and thus is done automatically.

The representation introduced is constraint based and is supported by a fast graph-constructive constraint solver that can solve a large subset of the class of configurations that involve points, lines and circles with fixed radius and are ruler and compass constructible. The construction sequence is determined by the algorithm, so our system can be characterized as purely declarative.

The development of the constraint solver was made using a high level tool called APTS which was used to transform the input, and the high level programming language SETL2 which was used to implement the core of the constraint solver (decomposition and calculations).

Finally, we dealt systematically with the problem of root identification which, though more or less neglected by most researchers, is a source of ambiguity and confusion for users. Except for a small set of simple rules that work in most of the cases, the user is provided with a simple interactive framework for root selection.

We are currently working in developing a theoretical framework for proving the completeness and uniqueness of the reduction algorithm. Furthermore, we believe that we can reduce the worst case time complexity of the algorithm down to $O(n^{\alpha})$, where $1.5 \leq \alpha \leq 2$. As far as the average time complexity is concerned, we suspect that it can be proved to be almost linear. In future, we will need to extend the constraint solver to solve a larger class of configurations (including non ruler and

compass constructible) and to deal with circles with variable radii. This means that we will have to identify the appropriate graph reduction rules and extend the graph decomposition algorithm to handle them efficiently. The framework for root selection will also need to be extended to deal with the reduction steps of the extended algorithm. Finally, one of the major challenges is to extend this method to solving 3D constraints.

BIBLIOGRAPHY

BIBLIOGRAPHY

[Ald88] B. Aldefeld. Variation of geometries based on a geometric-reasoning method. *Computer Aided Design*, 20(3):117–126, April 1988.

[BA91] Inc Brown Associates. Conceptual Design: Tradeoffs in Performance and Flexibility. Notes on the design of Pro/ENGINEER, 1991.

[Bar87] L. A. Barford. *A Graphical, Language-Based Editor for Generic Solid Models Represented by Constraints.* PhD thesis, Dept of Computer Science, Cornell University, March 1987. TR 87-813.

[BMMW89] A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint Hierarchies and Logic Programming. In *Proc. of the 6th International Logig Programming Conference*, pages 149–164, 1989.

[Bor81] A. H. Borning. The programming language aspects of ThingLab, a constraint oriented simulation laboratory. *ACM TOPLAS*, 3(4):353–387, 1981.

[Bru86] Beat Bruderlin. Constructing Three-Dimensional Geometric Objects Defined by Constraints. In *Workshop on Interactive 3D Graphics*, pages 111–129. ACM, October 23-24 1986.

[Buc85] B. Buchberger. Grobner Bases : An Algorithmic Method in Polynomial Ideal Theory. In N. K. Bose, editor, *Multidimensional Systems Theory*, pages 184–232. D. Reidel Publishing Company, 1985.

[BW81] Alan Bundy and Bob Welham. Using Meta-level Inference for Selective Application of Multiple Rewrite Rule Sets in Algebraic Manipulation. *Artificial Intelligence*, 16:189–212, 1981.

[Cap93] Vasilis S. Capoyleas. Personal Communication. NYU, May 1993.

[CFV88] U. Cugini, F. Folini, and I. Vicini. A procedural system for the definition and storage of technical drawings in parametric form. In D. A. Duce and P. Jancene, editors, *Eurographics '88*, pages 183–196. Eurographics Association, Elsevier Science Publishers B.V. (North Holland), 1988.

[Cho87] Shang-Ching Chou. A Method for the Mechanical Derivation of Formulas in Elementary Geometry. *Journal of Automated Reasoning*, 3:291–299, 1987.

[Cho88] Shang-Ching Chou. An Introduction to Wu's Method for Mechanical Theorem Proving in Geometry. *Journal of Automated Reasoning*, 4:237–267, 1988.

[Coh90] Jacques Cohen. Constraint Logic Programming Languages. *Communications of the ACM*, 33(7):52–68, 1990.

[CS86] Shang-Ching Chou and William Schelter. Proving Geometry Theorems with Rewrite Rules. *Journal of Automated Reasoning*, 2:253–273, 1986.

[D-C93] D-Cubed Ltd, 68 Castle Street, Cambridge, CB3 0AJ, England. *The Dimensional Constraint Manager*, May 1993. Version 2.5.

[FBMB90] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An Incremental Constraint Solver. *CACM*, 33(1):54–63, 1990.

[Fit81] W. Fitzerland. Using Axial Dimensions to Determine the Proportions of Line Drawings in Computer Graphics. *Computer Aided Design*, 13(6), November 1981.

[Fuq87] T. W. Fuqua. Constraint Kernels : Constraints and Dependencies in a Geometric Modeling System. Master's thesis, The University of Utah, August 1987.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Co., 1979.

[Gos83] J. Gosling. Algebraic Constraints. Technical Report CMU-CS-83-132, CMU, 1983.

[GZS88] D. C. Gossard, R. P. Zuffante, and H. Sakurai. Representing dimensions, tolerances, and features in MCAE systems. *IEEE Computer Graphics & and Applications*, 8(2):51–59, 1988.

[HB78] R. C. Hillyard and I. C. Braid. Analysis of Dimensions and Tolerances in Computer-aided Mechanical Design. *Computer Aided Design*, 10(3):161–166, 1978.

[HJ93] C. M. Hoffmann and R. Juan. EREP : An Editable High-Level Representation for Geometric Design and Analysis. In P. Wilson, M. Wozny, and M. Pratt, editors, *Geometric and Product Modeling.* North Holland, 1993.

[Hof82] P. Hoffmann. Analysis of Tolerances and Process Inaccuracies in Discrete Part Manufacturing. *Computer Aided Design*, 14(2):83–88, 1982.

[Hof92] C. M. Hoffmann. On the Semantics of Generative Geometry Representations. working paper, 1992.

[Jus92] N. P. Juster. Modelling and representation of dimensions and tolerances : a survey. *Computer Aided Design*, 24(1):3–17, January 1992.

[Kap88] Deepak Kapur. A Refutational Approach to Geometry Theorem Proving. *Artificial Intelligence*, 37:61–93, 1988.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, NY, 1972.

[Kon90] K. Kondo. PIGMOD: parametric and interactive geometric modeller for mechanical design. *Computer Aided Design*, 22(10):633–644, December 1990.

[Kon92] K. Kondo. Algebraic method for manipulation of dimensional relationships in geometric models. *Computer Aided Design*, 24(3):141–147, March 1992.

[Lel88] Wm Leler. *Constraint Programming Languages: Their Specification and Generation*. Addison Wesley, 1988. ISBN 0201062437.

[LG82] Robert Light and David Gossard. Modification of geometric models through variational geometry. *Computer Aided Design*, 14(4):209–214, July 1982.

[Li88] Jiarong Li. Using algebraic constraints in interactive text and graphics editing. In D. A. Duce and P. Jancene, editors, *Eurographics '88*, pages 197–205. Eurographics Association, Elsevier Science Publishers B.V. (North Holland), 1988.

[Mac77] Alan Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–119, 1977.

[Nel85] G. Nelson. Juno, a costraint-based graphics system. In *SIGGRAPH*, pages 235–243, San Francisco, July 22-26 1985. ACM.

[Owe91] J. C. Owen. Algebraic Solution for Geometry from Dimensional Constraints. In *ACM Symp. Found. of Solid Modeling, Austin, TX*, pages 397–407. ACM, 1991.

[Pai93] R. Paige. Apts external specification manual. internal documentation, 1993.

[Pro] Pro/ENGINEER. *Modeling Users Guide: 2D Sketcher.* Brown Associates, Inc. Release 8.0.

[RBN88] J. R. Rossignac, P. Borrel, and L. R. Nackman. Interactive Design with Sequences of Parameterized Transformations. Technical Report RC 13740 (#61565), IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, 1988.

[Req77] A. Requicha. Dimensionining and tolerancing. Technical report, Production Automation Project, University of Rochester, May 1977. PADL TM-19.

[Req83] A. A. G. Requicha. Toward a theory of geometric tolerancing. *Int. J. Robot. Res.*, 2(4):45–60, 1983.

[Rol89a] Deiter Roller. Design by Features: An Approach to High Level Shape Manipulations. *Computers in Industry*, 12:185–191, 1989.

[Rol89b] Dieter Roller. Dimension-Driven geometry in CAD : a Survey. In *Theory and Practice of Geometric Modeling*, pages 509–523. Springer Verlag, 1989.

[Rol90] Dieter Roller. A System for Interactive Variation Design. In M. J. Wozny, J. U. Turner, and K. Preiss, editors, *Geometric Modeling for Product Engineering*, pages 207–219. Elsevier Science Publishers B.V. (North Holland), 1990.

[Rol91] Dieter Roller. An approach to computer-aided parametric design. *Computer Aided Design*, 23(5):385–391, 1991.

[Ros86] J. R. Rossignac. Constraints in constructive solid geometry. In *Workshop on Interactive 3D Graphics*, pages 93–110, Chapel Hill, NC, October 23-24 1986. ACM.

[SAK90] Hirimasa Suzuki, Hidetoshi Ando, and Fumihiko Kimura. Variation of geometries based on a geometric-reasoning method. *Computer & Graphics*, 14(2):211–224, 1990.

[SKW90] Mark S. Shephard, Elaine V. Korngold, and Rolf Wentorf. Design systems supposrting engineering idealizations. In M. J. Wozny, J. U. Turner, and K. Preiss, editors, *Geometric Modeling for Product Engineering*, pages 279–300. Elsevier Science Publishers B.V. (North Holland), 1990.

[Sny90] K. Snyder. The SETL2 programming language. Technical report, New York University, Computer Science, Courant Institute, 1990.

[Soh91] Wolfang Sohrt. Interaction with Constraints in three-dimensional Modeling. Master's thesis, Dept of Computer Science, The University of Utah, March 1991.

[SS80] G. L. Steele and G. L. Sussman. CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions. *Artificial Intelligence*, pages 1–39, January 1980.

[Sun88] Geir Sunde. Specification of shape by dimensions and other geometric constraints. In M. J. Wozny, H. W. McLaughlin, and J. L. Encarnacao, editors, *Geometric modeling for CAD applications*, pages 199–213. North Holland, IFIP, 1988.

[Sut63a] I. Sutherland. Sketchpad, a man-machine graphical communication system. In *Proc. of the spring Joint Comp. Conference*, pages 329–345. IFIPS, 1963.

[Sut63b] I. Sutherland. *Sketchpad, a man-machine graphical communication system*. PhD thesis, MIT, January 1963.

[Tod89] Philip Todd. A k-tree generalization that characterizes consistency of dimensioned engineering drawings. *SIAM J. DISC. MATH.*, 2(2):255–261, 1989.

[Tur88] Joshua U. Turner. A Mathematical Theory of Tolerances. In M. J. Wozny, H. W. McLaughlin, and J. L. Encarnacao, editors, *Geometric modeling for CAD applications*, pages 163–187. North Holland, IFIP, 1988.

[Ver90] A. Verroust. *Etude de problemes lies a la definition, la visualisation et l'animation d'objets complexes en informatique graphique*. PhD thesis, Universite Paris-Sud, France, 1990.

[VSR92] A. Verroust, F. Schonek, and D. Roller. Rule-oriented method for parameterized computer-aided design. *Computer Aided Design*, 24(3):531–540, October 1992.

[Wen86] Wu Wen-Tsun. Basic Principles of Mechanical Theorem Proving in Elementary Geometries. *Journal of Automated Reasoning*, 2:221–252, 1986.

[WFB87] A. Witkin, K. Fleischer, and A. Barr. Energy Constraints on Parameterized models. *Computer Graphics*, 21:225–232, 1987.

[Wil91] Michael R. Wilk. Equate: An Object-Oriented Constraint Solver. In *OOPSLA*, pages 286–298. ACM, 1991.

[Woo88] J. R. Woodwark. Some speculations on feature recognition. *Computer Aided Design*, 20(4), May 1988.

[Woo90] Robert F. Woodbury. Variations in Solids : A Declarative Treatment. *Computers & Graphics*, 14(2):173–188, 1990.

[Wyk82] Christopher J. Van Wyk. A High-Level Language for Specifying Pictures. *ACM Transactions on Graphics*, 1(2):163–182, 1982.

[YK90] Yasushi Yamaguchi and Fumihiko Kimura. A constraint modeling system for variational geometry. In M. J. Wozny, J. U. Turner, and K. Preiss, editors, *Geometric Modeling for Product Engineering*, pages 221–233. Elsevier Science Publishers B.V. (North Holland), 1990.