A Geometric Constraint Solver

William Bouma* Ioannis Fudos† Christoph Hoffmann*
Department of Computer Science, Purdue University
West Lafayette, IN 47907-1398

Jiazhen Cai‡ Robert Paige‡
Department of Computer Science, Courant Institute
251 Mercer Str., New York, NY 10012

Report CSD-TR-93-054, August 1993§

Abstract

We report on the development of a two-dimensional geometric constraint solver. The solver is a major component of a new generation of CAD systems that we are developing based on a high-level geometry representation. The solver uses a graph-reduction directed algebraic approach, and achieves interactive speed. We describe the architecture of the solver and its basic capabilities. Then, we discuss in detail how to extend the scope of the solver, with special emphasis placed on the theoretical and human factors involved in finding a solution — in an exponentially large search space — so that the solution is appropriate to the application and the way of finding it is intuitive to an untrained user.

*Supported in part by ONR contract N00014-90-J-1599, by NSF Grant CDA 92-23502, and by NSF Grant ECD 88-03017.

†Supported by a David Ross fellowship.

‡Supported in part by ONR contract N00014-90-J-1890, by AFOSR grant 91-0308, and by NSF grant MIP 93-00210.

§This report and others are available via anonymous ftp to arthur.cs.purdue.edu, in directory pub/cmh and subsidiaries

1 Introduction

Solving a system of geometric constraints is a problem that has been considered by several communities, and using different approaches. For example, the symbolic computation community has considered the general problem, in the context of automatically deriving and proving theorems from analytic geometry, and applying these techniques to vision problems [9, 14, 26, 27]. The geometric modeling community has considered the problem for the purpose of developing sketching systems in which a rough sketch, annotated with dimension and constraints, is instantiated to satisfy all constraints. This work will be reviewed in the next section. The applications of this approach are in mechanical engineering, and, especially, in manufacturing.

With this work, we have mainly manufacturing applications in mind. Our purposes and goals are as follows:

1. We develop a constraint solver in which the information flow between the user interface and the underlying solver has been formalized by a high-level representation that is neither committed to the particular capabilities or technical characteristics of the solver, nor is dependent on the interface. Such a representation becomes the basis for archiving sketches in a neutral format, with the ability to retrieve the archived sketch and edit it — possibly in a different system with a different solver [24, 23]. Our solution is also a building block for a larger project of developing a new generation of CAD systems based on a neutral, high-level geometry representation that expresses design intent and preserves the ability to redesign.

2. We explore the utility of several different general-purpose and interoperating rapid prototyping languages and systems for developing specific tools for experimenting conveniently with a variety of ideas and approaches to constraint solving. Aside from well-known special purpose tools such as LEX and Yacc [25], our constraint solver also makes use of the high level language SETL2 [45] to specify complex combinatorial algorithms and the transformational system APTS [12] to perform syntactic analysis and symbolic manipulation of geometrical constraint specifications.

3. We study a number of neglected aspects of constraint solving, in particular the process of redirecting the solver to a different solution of a well-constrained sketch, and to devise generic techniques for extending the capabilities of the solver while preserving interactive speed.

This paper reports substantial progress in all three problem dimensions, and identifies a number of open issues that remain.

2 Approaches to Geometric Constraint Solving

We consider only well-constrained, two-dimensional sketches formed from points, lines, circles, segments and arcs. Constraints are explicit dimensions of distances and angles, as well as constraints of parallelism, incidence, perpendicularity, tangency, concentricity, collinearity, and prescribed radii. We exclude relations on dimension variables and inequality constraints. In particular, 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 only has to be topologically correct. The constraint solver determines from the sketch the geometric elements that are to be found, and processes the constraints to determine each geometric element such that the constraints are satisfied.

Our constraint solver is *variational.* That is, the solver is not obliged to process the constraints in a predetermined sequence, and the constraints specified by the user are not parametric in the sense that they must be determined serially, each as an explicit function of the previous constraints. This is analogous to writing the constraints in a *declarative* language, where the solution is independent of the order in which the constraints are written down. This greatly increases the generality of the constraint solving problem, and demands solvers that are based on advanced mathematical concepts.

While the users of geometric constraint solving systems think geometrically and express themselves with visual gestures, the underlying constraint solvers typically work with a different internal representation. Most users will be quite unaware of the nature of the underlying representation, and of the internal workings of the constraint solver. Coupled with the fact that a well-constrained geometric constraint problem has, in general, exponentially many solutions, only one of which satisfies the user's intent, constraint solvers therefore have to address two distinct tasks:

1. Determine whether the problem can be solved and if so, how.

2. Among the possible solutions, identify the one the user has intended.

Most of the literature assumes tacitly that the second task is easy to discharge. In Section 5, we question this assumption and show why Task 2 is difficult for applications.

Before describing our approach to Task 1, it is useful to characterize other approaches in the literature.

2.1 Numerical Constraint Solvers

In general numerical constraint solvers, the constraints are translated into a system of algebraic equations and are solved using an iterative method. When based on Newton iteration, such solvers require good initial values, which implies that the initial sketch must almost satisfy all constraints already. The solvers are quite general, and are capable of dealing with overconstrained, consistent constraint problems. Many constraint solvers switch to iterative methods in situations where the given configuration is not solvable by the native method.

Nonlinear systems have an exponential number of solutions, but Newton iteration will find only one. Numerical solvers based on Newton iteration are therefore inappropriate when the initial sketch is only topologically correct, or when the solver locks into a solution that is unsuited to the application and has no method with which to find more suitable alternatives.

Sketchpad [51] was the first system to use the method of numerical relaxation. Relaxation is slow but quite general. Many systems like ThingLab [3] and Magritte [22] can do relaxation as an alternative to some other method. In [2] a projection method is presented for finding a new solution that minimizes the Euclidean distance between the old and the new solution.

Newton-Raphson iteration has been used in a number of systems, and is faster than relaxation, but it may converge to the wrong solution. Unfortunately, when this happens, the user has no recourse to instruct the solver to find alternatives. Juno [37] uses the original sketch as initial state. The CPSM system of Solano and Brunet [47] also uses a numerical solver that first deals with sequential constraints and then solves circularly interdependent constraints.

A modification of Newton-Raphson was developed in [35], where an improved way for finding the inverse Jacobi matrix is presented. Furthermore, the paper proposes dividing the constraint matrix into submatrices, with the potential of providing the user with information about the constraint structure of the sketch. Although this information is usually quantitative and not very specific, it may help the user make modifications if the solver fails. A method that represents constraints by an energy function and then searches for a local minimum using the energy gradient is presented in [56].

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, or using another, less classical repertoire of construction steps. In these methods, the constraints are satisfied constructively, by placing geometric elements in some order. This is more natural for the user and makes the approach suitable for interactively debugging a sketch that cannot be solved or has been solved unsatisfactorily, from an application point of view.

2.2.1 Rule-Constructive Solvers

One version of the constructive approach uses rewrite rules to discover and execute the construction steps. We call this approach *rule-constructive solving*. Although a Logic Programming style of programming is a good approach for prototyping and experimentation, the extensive computations searching and matching rewrite rules constitute a liability.

Bruderlin and Sohrt [6, 46] solve constraints in this way and incorporate the Knuth-Bendix critical-pairs algorithm [29]. They show that their method is correct and solves all problems that can be constructed using ruler and compass. The method can also be proved to confirm geometric theorems that are provable in their system of axioms. Bruderlin and Sohrt have implemented an experimental constraint solving system in Prolog. They do not address how to devise rules for determining automatically which of the possible solutions is the one the user intended.

Aldefeld [1] uses a forward chaining inference mechanism. He assumes that lines are directed, and formulates additional rules that restrict the number of possible solutions. A similar method is presented in [52], where handling of overconstrained and underconstrained cases is given special consideration. Sunde in [50] also uses a rule-constructive method but has different rules for representing directed distance and undirected distance, thus adding flexibility for dealing with the root identification problem discussed in Section 5. In [58] the problem of nonunique solutions is handled by imposing an order on triples of geometric elements. A detailed description of a complete set of rules for 2D design can be found in [55], where the scope of the rules is also characterized. Finally, a technique called Meta-level Inference is introduced in [10]. The paper claims that this technique, combined with multiple sets of rules and their selective application, reduces the search space. The method has been applied in PRESS [10], a program for algebraic manipulation.

2.2.2 Graph-Constructive Solvers

Another version of the 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, the construction steps are carried out to derive the solution. We call this approach *graph-constructive solving*. It is fast, more methodical than the rule-constructive approach, and is proved to be sound. However, as the repertoire of possible constraints increases, the graph-analysis algorithm has to be modified.

Fitzgerald [20] follows the approach of dimensioned trees by Requicha [43]. Only horizontal and vertical distances are allowed in this method and so the applicability of the method is limited. Todd in [53] generalized the dimension trees of Requicha. Owen in [38] presents an extension of this principle to include circularly dimensioned sketches, and DCM [19] is a commercial constraint solver using this method.

Since our basic algorithm is based on many of the ideas of [38], we describe Owen's solvers in more detail. The constraint solver described in [38] is a graph-constructive solver in which the constraint graph is analyzed for triconnected components. Each triconnected component is reduced to a number of elements that interact with other components, and a determination is made how the various geometric elements whose nodes are in each graph component fit together. Thereafter, each component can be separately determined. This procedure is recursive in that once components have been reduced, they in turn can become members of triconnected components in the reduced graph. A key aspect of the solver is that only constraint configurations are considered that can be solved using ruler-and-compass construction steps. Algebraically, this is equivalent to solving only quadratic equations, so that the specific coordinate computations do not require sophisticated mathematical computations. In [38] a proof is given that the solver is complete for ruler-and-compass constructible point configurations with prescribed distances that are algebraically independent.

DCM [19] shares with the algorithm of [38] the characteristic that it begins by determining the interaction of geometric element groupings before filling in the individual elements in each group. In addition, we infer that the commercial version has a significant number of additional rules and transformations that can be applied to the constraint graph in order to extend the scope of the basic algorithm. In many cases, the graph reduction requires linear time only.

Kramer [32] uses a similar approach. However, instead of determining the equations of the geometric elements at each construction step, Kramer determines coordinate transformations that successively place points and associated coordinate frames relative to each other subject to constraints. Kramer's constraint solver is for 3D and deals with constraints that arise in kinematics and characterize basic joint types. Thus, a revolute constraint matches the points and aligns a pair of coordinate axes, allowing a single degree of freedom, a rotation about the aligned axes. Complex geometric elements are placed implicitly by choosing a suitable number of points on them whose coordinate frames are relatively fixed, and then placing each point.

2.3 Propagation Methods

Constraint propagation was a popular approach in early constraint solving systems. The constraints are first translated into a system of equations involving variables and constants, and an undirected graph is created whose nodes are the equations, variables and constants, and whose edges represent whether a variable or constant occurs in an equation. The method then attempts to direct the graph edges so that every equation can be solved in turn, initially only from the constants. To succeed, various propagation techniques have been tried, but none of them is guaranteed to derive a solution when one exists. For a review see [33, 46].

Sketchpad [51] uses propagation of degrees of freedom and propagation of known values. Pro/ENGINEER [5, 40] uses propagation of known values. Propagation of known values is the inverse process of the propagation of degrees of freedom. Propagation of degrees of freedom is a more abstract method that essentially does a graph reduction. In the propagation of known values, we can account for special values and therefore make the method slightly more powerful than pure propagation of degrees of freedom. Both methods are global, unstable, and do not work for cyclically dimensioned sketches.

CONSTRAINTS [48] uses retraction, which is a localized version of propagation of known values that stores information about each variable's interdependencies. A similar technique is used in [34]: First, known values are propagated locally. Then, the remaining simultaneous constraints are solved if they form a linear system of equations. In general, retraction is faster but less powerful than propagation of known values.

Graph transformation is sometimes used in conjunction with some propagation method. In pure graph transformation, some subgraphs of the constraint graph are identified and are replaced by simpler subgraphs. Bertrand, described in [33], is a general-purpose constraint specification language, and is implemented using a propagation method in conjunction with an inference mechanism. Leler calls this technique *augmented term rewriting*. In essence, augmented term rewriting is a graph transformation mechanism using term rewriting rules. Additionally, assignments are supported, as is variable typing, and these additions make augmented term rewriting more expressive than the term rewriting mechanism of pure PROLOG.

ThingLab uses the *Blue* and *Delta Blue* algorithms described in [3, 21], that are based on a local propagation of degrees of freedom within the constraint graph. Magritte [22] employs propagation to transform the undirected constraint graph, and then uses breadth-first search to derive all solutions.

2.4 Symbolic Constraint Solvers

The constraints are transformed into a system of algebraic equations. The system is solved with symbolic algebraic methods, such as Gröbner's bases, e.g., [9], or the Wu-Ritt method [57, 14]. Both methods can solve general nonlinear systems of algebraic equations. The methods have also been used in mechanical geometry theorem proving [16, 17, 15, 26].

In [30, 31], Kondo considers the addition and deletion of constraints by using the Buchberger's Algorithm [7, 8] to derive a polynomial that gives the

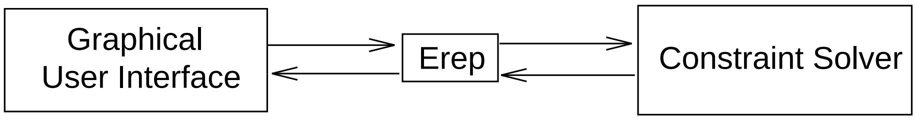

Figure 1: Architecture of the Constraint Solver

relationship between the deleted and added constraints.

2.5 Hybrid Solvers

Often, constraint solving systems use a combination of the above methods. One method is attempted, and if it does not succeed, another one is tried. The main difficulty is that some of the methods may require exponential time before giving a negative response.

3 The Constraint Solving System

3.1 Information Flow and Rationale

The overall architecture of the constraint solver is as shown in Figure 1. The user draws a sketch and annotates it with geometric constraints. The allowed constraints include relations such as tangency, perpendicularity, etc, and explicit dimensioning of angles and distances. Excluded for now are relations between dimension variables. Additional capabilities include interacting with the solver to identify a different, valid solution. The geometric elements available at this time are segments, points, and circular arcs. Auxiliary lines, points and circles may also be defined.

The user interface translates the specification into a textual language that is a faithful record of the problem. Although the user could edit this textual problem specification, this is unnecessary, because the specification is edited and updated automatically from the visual gestures by the user interface. The language has been designed to achieve the objectives of [24] — a neutral problem specification that makes no assumptions about the architecture of the underlying constraint solving algorithm. Thus, it is quite easy to federate Owen's solver [38], or any other constraint solver capable of handling the geometric configurations we consider.

The textual problem specification is handed to the constraint solver engine which translates the constraints into a graph, and, as described later, solves them by graph reductions that govern the workings of an algebraic, variational constraint solver. The solver capabilities are the consequence of specific construction steps that have been implemented. If a particular constraint problem can be solved using the known construction steps, then our solver will find a solution. Where the construction steps involve ruler-and-compass constructions, only quadratic equations need to be solved. But some construction steps are permitted that are not ruler-and-compass, and in those situations the roots of a univariate polynomial are found numerically. In those situations, the polynomial has been precomputed except for the coefficients which are functions of the specific constraint values. The solver architecture is optimized for speed subject to the strict requirement that the information flow between user interface and solver does not depend on the internals of either component.

In the worst case, a well-constrained geometric problem has exponentially many solutions in the number of constraints. This is because the solutions correspond to the algebraic set of a zero-dimensional ideal whose generating polynomials are nonlinear. Our solver can determine all possible solutions. But doing so every time would waste time and overwhelm the user. So, certain heuristics, described later, narrow down the solutions to a final configuration that corresponds to the intended solution with high probability. It would be nice if methods could be devised that identify the solution the user intended every time. But even in very simple situations, additional information that would help doing so would lead to provably intractable problems. This would be incompatible with our goal of interactive speed. Instead, we have developed a paradigm for finding the right solution by using the solver interactively when its automatic heuristics are insufficient.

Our system will be a component of a constraint-driven variational CAD system based on a high-level, declarative, editable geometry representation (Erep) as discussed in [24, 23]. Such an overall architecture poses several challenges. One of them is efficient variational constraint solving, and we address this problem here. Another, key challenge is to formulate the language in a neutral way, committing it neither to the particulars of the user interface nor of the solver algorithms. This is a more subtle challenge because the way in which dimensions are displayed in the sketch has to make some assumptions about the capabilities of the user interface. Likewise, interacting with the solver to find alternative solutions requires conceptualizing the solution process in a way that makes no assumptions about how they are found. Here, we assume only that the solver is capable of undoing the last placement operation, and can look for a different placement of a geometric element. The textual protocol for communicating these matters is encapsulated.

3.2 System Implementation

The two-dimensional geometrical design system has two main components, a graphical interface and a constraint solver engine. The graphical interface is a C++ program [49] that interacts with X Windows in order to allow the user to sketch a drawing using labeled points, lines, circles, etc. The user is also expected to supply initial constraints between these geometric elements. This initial design is turned into an Erep specification and is passed as text to the constraint solver.

The solver is written using two novel software tools — the APTS transformational programming system [12] and the high-level language SETL2 [45] — each having special features that the solver exploits. The front-end to the constraint solver engine is an APTS program that reads the Erep program and type checks it. For example, we check that only lines participate in angle constraints. If there are no obvious type errors, the Erep program is transformed into an equivalent Erep specification in a normal form in which only distance and angle constraints are allowed. For example, incidence constraints are translated to zero-distance constraints. The specification of the orientation of lines in angle constraints is also regularized. Relations representing a constraint graph are then extracted from the Erep program and are exported via a foreign interface to a SETL2 program that implements the main algorithmic part of the solver.

The SETL2 program implements a new and extensible algorithm described later that analyzes the constraint graph to determine whether the Erep program is well constrained. If it is, then a particular solution (i.e., a specific placement of the geometries) is computed as a set of relations that are imported into the APTS program. Finally, the APTS program incorporates the solution into the Erep program, and passes it back as text to the graphical interface for display.

The use of such novel systems as APTS and SETL2 is motivated by the special needs of our project. A major component of our research involves the discovery and implementation of complex nonnumerical algorithms. Our goal of high performance based on a new algebraic approach to constraint solving entails deep graph-theoretic analysis of implicit dependencies between constraints, and complex graph traversals based on such analysis. A wide variety of heuristics seem available to us, but a proper evaluation requires extensive labor-intensive computational experiments.

The ease with which complex combinatorial algorithms can be implemented and modified in the SETL language [44] is well known. Snyder's new SETL2 language [45] significantly improves SETL in regard to its convenience in algorithm specification, its compile- and run-time reliability and performance, and its portability. The SETL2 language allows the physical organization and even the performance of data structures and algorithms to be modeled abstractly using mathematical data types that are algebraically formed from conventional data by constructors for tuples, sets, and maps. These data types can be manipulated by a rich repertoire of set-theoretic dictions such as arbitrary choice, nondeterministic search, set comprehension, and quantification. Using SETL2 has allowed us to implement our algorithms with surprising speed. In the future we also hope to make use of a promising new technology, just now being reported, for mechanically transforming prototype SETL2 programs into high performance C code [13].

Another major part of our research develops a logical framework for specifying and solving 2-dimensional geometric constraints. The Erep language provides a formal syntax and semantics essential to problem specification and problem solving. We seek a rich language of geometries and constraints for conveniently describing two-dimensional drawings. The language should also support mathematical analysis and transformation by either manual or mechanical means. Within the Erep language, we seek mathematical and syntactic characterizations of classes of specifications that are *correct*; i.e., free from surface errors, *valid*; i.e., mathematical well constrained, and *practical*; i.e., efficiently solvable and able to express the concepts needed in applications.

The special syntactic, semantic, and transformational capabilities of APTS [12] are well suited to a flexible, experimental development of a logical framework with an evolving Erep language and corresponding solver. Like systems such as Centaur [4] the Synthesizer Generator [42], and Refine [41], APTS has a single uniform formalism for lexical analysis, syntactic analysis, and pretty-printing. However, the semantic formalism in APTS has several advantages over the more conventional attribute grammar approach [28] that is used in the Synthesizer Generator. APTS uses a logic-based approach to semantics in which semantic rules that define relations are written in a Datalog-like language [54, 39] but with the full expressive power of Prolog [18]. These rules are written independently of the individual grammar productions and without reference to the parse tree structure. They define relations over a rich assortment of primitive and constructed domains, and have the brevity and convenience of unrestricted circular attribute grammars. We are not aware of any implementation that allows a comparable unrestricted circularity.

The semantic formalism in APTS is also integrated with a conditional rewriting component that is lacking in both the Synthesizer Generator [42] and Centaur [4], and is more abstract and user/friendly than Refine [41]. Although only a prototype implementation of APTS is currently available, the inference and rewriting engines used to compute and maintain semantic relations involve the use of such highly efficient algorithms that the observed performance is reasonable [11]. In contrast to Refine, implemented in Common Lisp, APTS is portable to a wide variety of machines and operating systems and, in particular, to any UNIX platform.

4 Solver Algorithmics and Extensibility

First, we discuss our basic method for solving geometric constraints. It is based on Owen's method, but differs in some details. While Owen's solver is top-down, determining first the interaction between clusters of geometric elements, ours is bottom-up. We begin in the basic algorithm by placing geometric elements until a cluster has been determined. The construction steps needed are described later. Once a cluster cannot be extended, another cluster is constructed in the same way. Several clusters sharing geometric elements are then coalesced based on some simple rules also described, by a rigid motion of one with respect to the other. Coalesced clusters are again treated as clusters, so the recursive nature of Owen's algorithm is also manifest in our approach. In the basic algorithm, only quadratic equations are solved. Thus, the basic algorithm is restricted to ruler-and-compass constructible configurations.

For the larger class of geometric elements consisting of points, lines and circles, our basic algorithm and Owen's methods do not solve all ruler-and-compass constructible configurations. For example, for Subcase 1 of Table 1 below, our basic solver must be extended. DCM can solve the configuration sometimes, depending on the way the problem is posed. We suspect that a complete ruler-and-compass constructible solver for the larger class of geometric elements requires graph rewriting rules that are equivalent to the Knuth-Bendix algorithm [29].

We also discuss a general method for extending the solver to configurations that cannot be done with the basic algorithm. Our strategy places two clusters related by three constraints. The extension goes beyond ruler-and-compass constructions, and requires a root finder for univariate polynomials. Conceptually, the extension corresponds to adding new geometric construction steps. The solver could be extended arbitrarily further, in an analogous manner, but at some point the number of construction steps becomes too large, and selecting which one to apply begins to interfere with the speed of the solver.

4.1 Solving with Graph Reduction

As sketched in [38], we first translate the constraint problem into a constraint graph. 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.

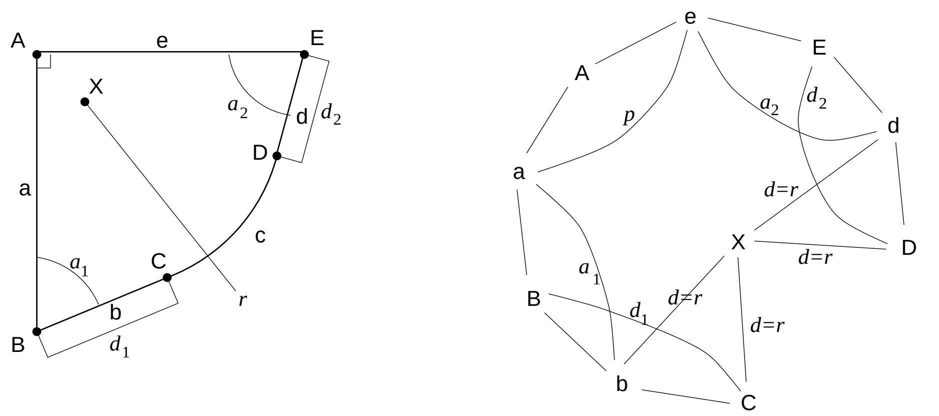

Figure 2: Example Configuration and Corresponding Constraint Graph. Unlabeled edges represent incidence.

4.1.1 Cluster Formation

The user sketch, annotated with constraints, is translated into a graph whose vertices correspond to geometric elements — points, lines and circles — and whose edges are constraints between them. In particular, a segment is translated into a line and two points, and an arc into a circle, two arc end points, and the center of the circle. For example, the sketch of Figure 2 (left) is translated into the graph of Figure 2 (right). In the graph, d represents a distance constraint, a an angle constraint, and p perpendicularity. Tangency has been expressed by a distance constraint between the center of the circle and the line tangent to the circle. All other graph edges represent incidence. Circles of fixed radius can be determined by placing the center, so there is no vertex corresponding to arc c in the constraint graph. 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.

Note that for subsequent cluster formation one of the two initial geometric elements may already belong to another cluster.

For example, in the graph of Figure 2, we may begin with elements a and B, effectively drawing a line a and placing on it the point B anywhere. We can now place in sequence b, C, and X. At this point, no additional elements can be placed and the cluster is complete, as shown in Figure 3. Note that we neither

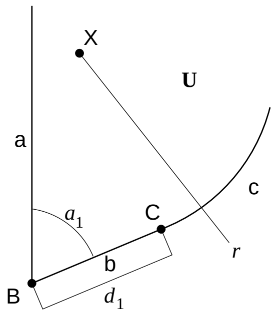

Figure 3: Cluster **U** of Figure 2

know where A is situated, nor how far the arc c extends. Starting again, two other clusters are determined. One consists of X, D, d, E, and e. The other cluster consists of a, A, and e. Note that the same geometric element may occur in more than one cluster. Both clusters are shown side-by-side in Figure 4.

4.1.2 Recursion

Two clusters with one geometric elements in common and one constraint between them can, in general, be placed with respect to each other. To do so, the shared geometric element is identified, and the remaining degree of freedom eliminated from the additional constraint. One way to implement this is as follows:

1. Let u be the shared element, v and w the constrained elements in the two clusters, as shown in Figure 5 to the left.
2. Measuring the relative position of u and v in one cluster, and the relative position of u and w in the other cluster, three constraints between u,

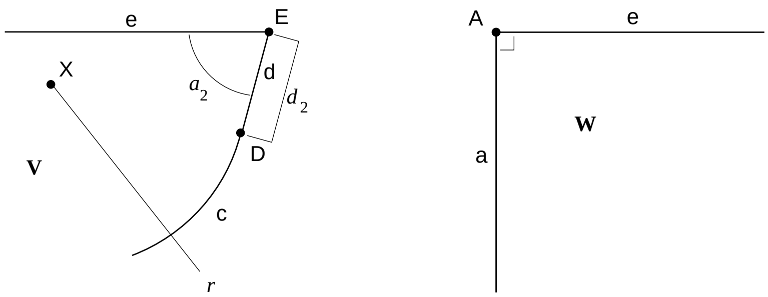

Figure 4: Clusters **V** and **W** of Figure 2

.

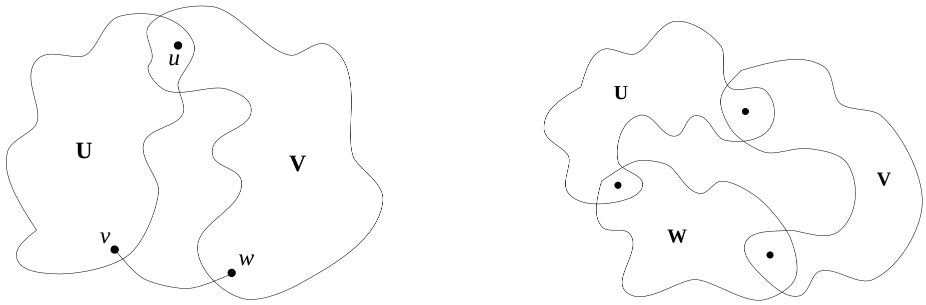

Figure 5: Recursive Cluster Placement

v and w can be formulated whose solution, using the basic clustering computations, fixes the three elements relative to each other.

3. The clusters are now placed by the three elements.

Three clusters, each sharing a geometric element with one of the others, can also be placed with respect to each other. By designating the constrained elements v and w in the two-cluster case as a third cluster, the three-cluster placement operation formally subsumes the two-cluster case. Figure 5 shows both cases. Exceptions to the cluster rules concern specific degeneracies. For example, if clusters **U** and **V** have two lines in common, then they can be placed with respect to each other. However, if the two shared lines are parallel, then the position of the two clusters cannot be completely determined.

In the example of Figure 2, there are three clusters sharing the elements a, e, and X. To place them, we compute the distance of X from a in cluster **U**, and the distance of X from e in cluster **V**. The angle between a and e 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 rules conceptually build a larger cluster from two or three smaller ones. Additional clusters sharing two elements with this new "super cluster" can be 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.

4.1.3 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

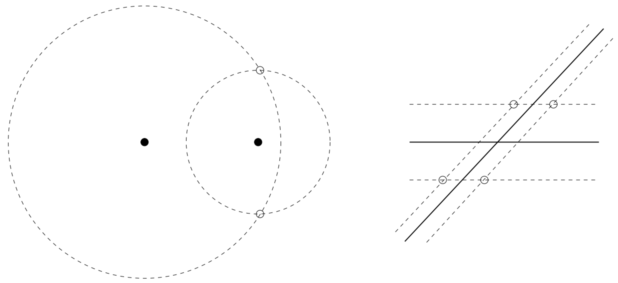

Figure 6: Point Placement Rules: Left, by Distance from Two Points; Right, by Distance from Two Lines.

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 6.

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:

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

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

Figure 7: Indidence of A with a and C with c and Resulting Constraint Graph.

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 from which to the unknowns are easier to find. Preprocessing can be done using Gröbner bases; e.g., [8].

4.1.4 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 7 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.

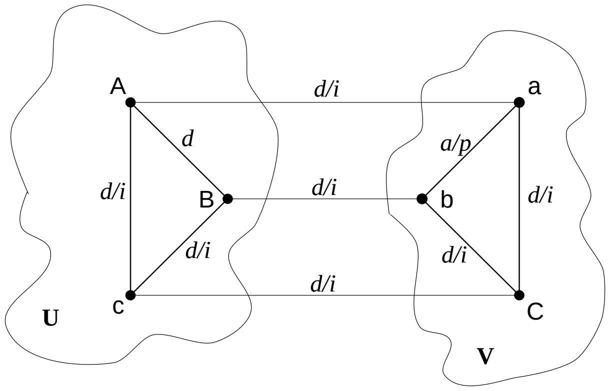

Figure 8: Case $(p, p, l) \equiv (l, l, p)$

4.2 Solver Extensions

The basic algorithm for solving constraints given before can be extended to handle more complex geometric situations. The strategy discussed here generalizes the placement of two clusters with respect to each other, when three constraints between them are given.

4.2.1 Placing Two Clusters

Consider the extension necessary to handle the situation shown in Figure 8. A cluster **U** and a cluster **V** have been solved separately, and three distinct elements have been identified in each that are constrained such that the two clusters can be placed with respect to each other. Since six distinct elements are involved, the basic algorithm cannot solve this problem.

If we examine every possible configuration of two clusters so related by three constraints and add graph reductions that place such clusters with respect to each other, then the solver has been extended to a much larger class of constraint problems. We analyze one constraint configuration. In this particular configuration, as well as in a number of other cases, the solver's competence is extended beyond ruler-and-compass constructible configurations.

Assume that in cluster **U** the three elements are the points A and B and the line c, and in cluster **V** the elements are two lines a and b and a point C. The constraint possibilities are d for distance, i for incidence, a for angle, and p for parallel. Depending on the combination of these constraint types, a construction sequence can be determined that fixes one cluster with respect to the other.

In the configuration considered, it is advantageous to fix cluster **V** and move cluster **U** relative to it such that all constraints are satisfied. Conceptually, we

Subcase Number	Properties of $\mathbf{U}$	Properties of $\mathbf{V}$	Constraints Combination	Method of Solution
1.	$\mathsf{A}\ i\ \mathsf{c}$ $\mathsf{B}\ d/i\ \mathsf{c}$ $\mathsf{A}\ d\ \mathsf{B}$	$\mathsf{C}\ i\ \mathsf{a}$ $\mathsf{C}\ d/i\ \mathsf{b}$ $\mathsf{a}\ a/p\ \mathsf{b}$	$\mathsf{A}\ i\ \mathsf{a}$ $\mathsf{B}\ d/i\ \mathsf{b}$ $\mathsf{c}\ i\ \mathsf{C}$	*(A)*
2.	$\mathsf{A}\ i\ \mathsf{c}$ $\mathsf{B}\ d/i\ \mathsf{c}$ $\mathsf{A}\ d\ \mathsf{B}$	$\mathsf{C}\ d\ \mathsf{a}$ $\mathsf{C}\ d\ \mathsf{b}$ $\mathsf{a}\ a/p\ \mathsf{b}$	$\mathsf{A}\ i\ \mathsf{a}$ $\mathsf{B}\ d/i\ \mathsf{b}$ $\mathsf{c}\ i\ \mathsf{C}$	*(B)*
3.	$\mathsf{A}\ d\ \mathsf{c}$ $\mathsf{B}\ d\ \mathsf{c}$ $\mathsf{A}\ d\ \mathsf{B}$	$\mathsf{C}\ i\ \mathsf{a}$ $\mathsf{C}\ d\ \mathsf{b}$ $\mathsf{a}\ a/p\ \mathsf{b}$	$\mathsf{A}\ i\ \mathsf{a}$ $\mathsf{B}\ d/i\ \mathsf{b}$ $\mathsf{c}\ i\ \mathsf{C}$	*(B)*

Table 1: Essential Combinations of $(p, p, l) \equiv (l, l, p)$. Constraint symbols are i = incident, d = nonzero distance, p = parallel, a = nonzero angle.

solve the cases in one of two ways:

(A) For some combinations, a sequence of ordinary construction steps places the second cluster, possibly with the introduction of auxiliary construction points, lines and/or circles.

(B) For some combinations, we consider two of the three constraints and precompute the locus of the geometric element whose constraint has been ignored for the moment. If this element is a point, the locus is an implicit algebraic curve whose coefficients are expressions in the given constraints. Then, the precomputed locus is intersected with a construction line or circle and the intersections identify those positions for the third geometric element for which all constraints are satisfied.

Note that the second method is not necessarily equivalent to a ruler-and-compass construction.

Table 1 summarizes the essential combinations and identifies which approach is to be used. The other combinations can be mapped to those of the table by replacing some of the lines in **U** and **V** with suitably positioned parallel lines.

Consider Subcase 1, assuming that B is not incident to c, and that B should be at distance e from b. The two clusters are shown in Figure 9, with r the distance of A from B and t the distance of B from c. Two families of solutions exist: Either the lines a and c coincide, possibly in opposite orientation, or the points A and C coincide. In the first situation, the locus of B is a pair of

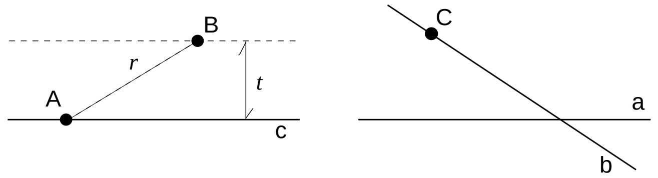

Figure 9: Subcase 1 Configuration

lines parallel to c, at distance t. Four intersections with lines parallel to b at distance e are four possible locations for B, and each of them determines the relative position of $\mathbf{U}$ with respect to $\mathbf{V}$. In the second situation, the locus of B is a circle around A, and the up to four intersections with lines parallel to b at distance e are the possible locations for B.

Now consider Subcase 2 in Table 1. We determine the curve that is the locus of B, assuming that B has coordinates (x, y). By a coordinate transformation, moreover, we can assume that C has coordinates $(0, 1)$ and that A, constrained to be on a, has coordinates $(a, 0)$. In the simplest case, A and B are both on c, distance d_1 apart. In Figure 12 this corresponds to $d_2 = 0$.

The cotangent of the angle θ between lines c and a is then $-a$, so that we can express $\sin\theta = 1/u$ and $\cos\theta = -a/u$, where $u = \sqrt{1+a^2}$. The locus of B can therefore be described by three equations:

$$\begin{aligned} xu - au + d_1 a &= 0 \\ yu - d_1 &= 0 \\ u^2 - a^2 &= 1 \end{aligned}$$

Eliminating a with a Gröbner basis computation establishes that the locus of B is the degree 4 curve

$$y^4 - 2\,y^3 + x^2y^2 + y^2(1 - d_1^2) + 2\,d_1^2 y - d_1^2 = 0$$

If $d_2 \neq 0$; i.e., if B does not lie on C, then the equations describing the locus of

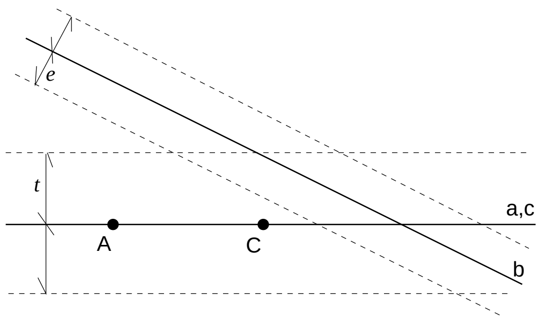

Figure 10: Subcase 1, First Solution

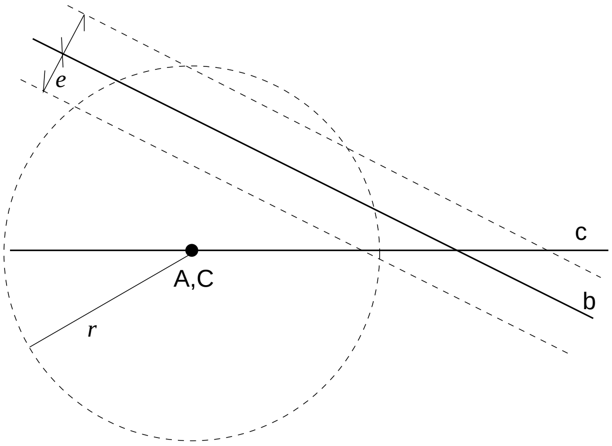

Figure 11: Subcase 1, Second Solution

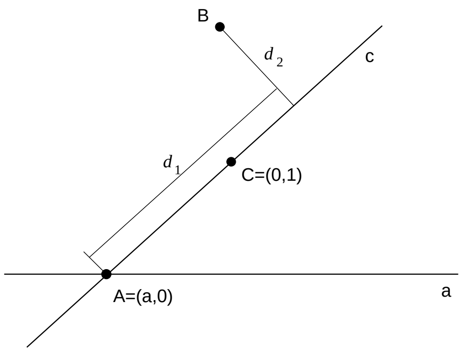

Figure 12: Case B d c, A i c

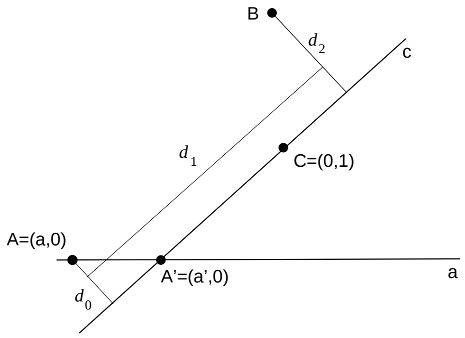

Figure 13: Case B d c, A d c

B are only slightly more complicated, and are

$$\begin{aligned} xu - au + d_1 a + d_2 &= 0 \\ yu - d_1 + d_2 a &= 0 \\ u^2 - a^2 &= 1 \end{aligned}$$

Again, B lies on a degree 4 curve whose coefficients are polynomial in d_1 and d_2 of degree 4.

The most general situation occurs when A is not on c, as shown in Figure 13. Other cases can be reduced to this situation by replacing the lines with parallel lines at a suitable distance. Referring to Figure 13, the equations describing the locus of B are

$$\begin{aligned} xu - au + d_1 a' + d_0 - d_2 &= 0 \\ yu - d_1 + d_0 a' - d_2 a' &= 0 \\ a - a' + d_0 u &= 0 \\ u^2 - {a'}^2 &= 1 \end{aligned}$$

By a Gröbner basis computation one determines that the locus of B is also a curve of degree 4. The coefficients are polynomials in the d_k of degree up to 4.

Many other combinations of three constraints between two clusters must be considered when so extending the solver. In each case, we conceptually satisfy two constraints and examine the locus of the geometric item whose constraint we ignored under the remaining degree of freedom. Thus, it suffices to examine pairs of constraints between two clusters and derive, for each arising case, the locus of the element in question. Since we have the choice of which two constraints to satisfy, the number of different cases can be reduced significantly. It is worth noting that an angle constraint between two clusters implies that the remaining relative motion is purely translational, simplifying the treatment of the ensuing cases.

If the element whose locus we determine is not a point, then we need equations for the determining quantities. In the case of lines, those are the coefficients

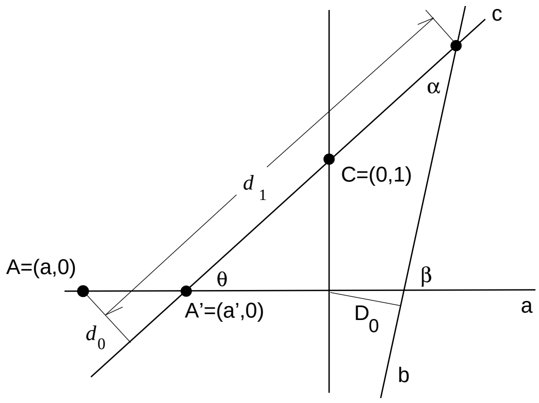

Figure 14: Configuration for Satisfying Line Constraints

of the line equation, or, equivalently, the direction angle and the distance from the origin. For example, consider the case $(p, l, l) \equiv (l, p, l)$. Here we may want to approach the situation as we did in the case $(p, l, l) \equiv (l, p, p)$, and precompute how the line equation varies with the remaining mobility when satisfying the first two constraints. In the subcase 3, the most general situation, we have to determine the distance of the moving line from the origin and the components of the normal vector after norming it to length 1; see also Figure 14. Let α be the fixed angle between the lines b and c, θ the angle determining a particular position of the moving configuration. Let γ be the direction angle of a line d with which b is to form an angle δ. Then we must solve $\alpha \pm \theta = \gamma \pm \delta$, accounting for the different positions of the moving configuration in which the constraints can be satisfied. Once θ is known, the resulting configuration is easily computed. In more complicated situations, a system of equations is formulated as for the point locus, expressing line distance from the origin and direction angle as function of the parameters and an additional quantity, such as θ or the coordinates of a moving point, and a nonlinear equation is solved that is precomputed from the system using Gröbner bases.

Ultimately, many of the cases and subcases we have to consider reduce to a few generic situations that are characterized by the selection of which two of the three constraints govern the relative motion between the two clusters. Particularly in the case of point loci, classical curves are obtained that are described in the literature; see, e.g., [36], or the literature on plane kinematics. The curve of subcase 2 above, with $d_2 = 0$, is a *conchoid* of a line. In Figure 15 a segment end is constrained to a circle and the segment incident to a fixed perimeter point C. This case is solved by the *limacon of Pascal*, a conchoid of the circle.

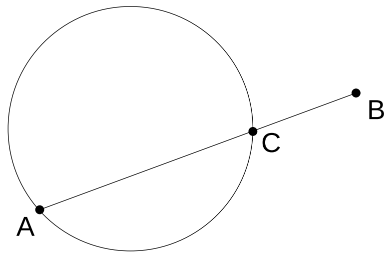

Figure 15: Locus of Segment Through a Fixed Perimeter Point is the Limacon of Pascal

5 User Interaction

In general, a well-constrained geometric constraint problem has an exponential number of solutions. For example, consider drawing n points, along with $2n - 3$ distance constraints between them, and assume that the distance constraints are such that we can place the points serially, each time determining the next point by two distances from two known points. In general, each new point can be placed in two different locations: Let p_0 and p_1 be known points from which the new point q is to be put at distance d_0 and d_1, respectively. Draw two circles, one about p_0 with radius d_0, the other about p_1, with radius d_1. The intersection of the two circles are the possible locations of q. For n points, therefore, we could have up to 2^{n-2} solutions. Which of these solutions is the intended one would depend on the application that created the constraint problem in the first place. We discuss how one might select the "right" solution. We call this the *root identification problem*, because on a technical level it corresponds to selecting one among a number of different roots of a system of nonlinear algebraic equations.

Although some solutions of a well constrained problem are merely symmetric arrangements of the same shape, others may differ structurally a great deal. Figure 16 shows several possibilities to illustrate the possible range. But an

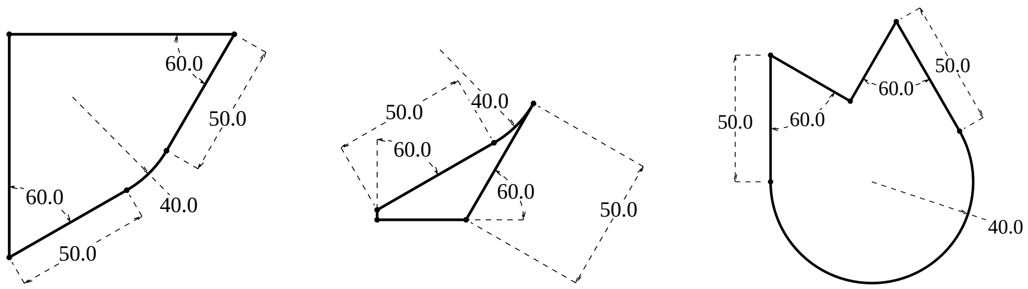

Figure 16: Several Structurally Distinct Solutions of the Same Constraint Problem

application will usually require one specific solution. To identify the intended solution is not always a trivial undertaking. Moreover, the wide range of possible solutions has severe consequences on the problem of communicating a generic design based on well-constrained sketches. Since a sketch with a constraint schema would not necessarily identify which solution is the intended one, more needs to be communicated.

In this section, we consider three approaches: selectively moving geometric elements, adding more constraints to narrow down the number of possible solutions, and, finally, a dialogue with the constraint solver that identifies interactively the intended solution. These are approaches that have to contend with some difficult technical problems. We also consider the possibility of structuring the constraint problem hierarchically. Doing so would increase knowledge of the design intent, and would diminish some of the more obvious technical problems.

5.1 Moving Selected Geometric Elements

All constraint solvers known to us adopt a set of rules by which to select the solution that is ultimately presented to the user. Whether stated explicitly, as we will later, or incorporated implicitly into the code of the solver, these rules ultimately infer which solution would be meant by observing topological and/or coordinate relationships of the initial sketch with which the user specified the constraint problem. When the solution is presented graphically to the user, it seems natural that the user, again graphically, select certain geometric elements of the final sketch that are considered misplaced. The user could then show the solver where the selected element(s) should be placed in relation to other elements by moving them with the mouse.

This very simple idea ultimately may be effective, but there are a number of conceptual difficulties that need to be overcome. For example, picking a geometric element is ambiguous. Because of the recursive nature of the solver, picking could refer to the individual element, or to the cluster or super clusters of which it became part. More importantly, the required restructuring might entail more complex operations than merely moving a single group of geometric elements. Furthermore, since the length of segments and arcs often implicitly depends on the final placement, it is not clear whether the user can reasonably be expected to understand the effect of moving geometries.

In DCM [19, 38], a *move* instruction relocates a geometric element. Thereupon, the solution can be recomputed, and other elements can be moved. it appears that the solver uses the new position coordinates when applying the normal placement heuristics selecting a solution. We found the move instruction difficult. Some of the time, the effect was as intended, but many times it was unexpected. However, with more research, a useful paradigm for identifying intended solutions of geometric constraint problems may well emerge.

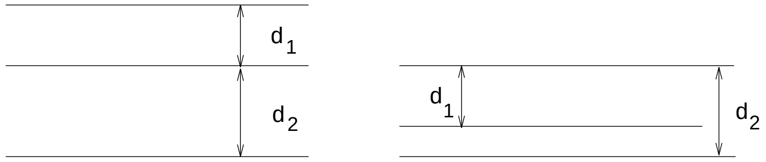

Figure 17: Two Solutions for Three Parallel Lines

5.2 Adding More Constraints

Consider once more the problem of placing n points with prescribed distances. We could narrow down which solution is meant in one of two ways: We may add domain knowledge from the application, or we may give additional geometric constraints that actually overconstrain the problem. Unfortunately, both ideas result in NP-complete problems.

For instance, 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 cross section is a simple polygon; that is, it should form a polygon that is not self-intersecting. This may be communicated by giving, in addition, a cyclical ordering of the points; i.e., the sequence of vertices of the cross section. This very simple additional requirement makes the problem NP-complete:

> **Theorem (Capoyleas)**
> Given n points in the plane that are well-constrained by $2n - 3$ point-to-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, i.e., is not self-intersecting, is NP-complete.

Consequently, there is little hope for adding domain-specific knowledge about the application with the expectation of obtaining an efficient constraint solver that finds the intended solution in all cases.

Instead of adding application-specific rules, for instance to derive simple polygons, we could add more geometric constraints. For example, consider specifying three parallel lines along with distances between two pairs of them. As shown in Figure 17, there are two distinct solutions of this well-constrained problem. By adding a required distance between the third pair of parallel lines we can eliminate one or the other case, and make the solution unique.

Overconstrained geometric problems have been carefully avoided by the field because the set of constraints might be contradictory. However, blue prints are usually overdimensioned, although not for reasons of eliminating unwanted solutions, but for limiting errors through redundancy. Again, it is unfortunate that even for the simple case of placing parallel lines the overconstrained problem is NP-complete.

Since adding constraints even in such simple situations results in NP-complete problems, it seems to us that the attractive idea of adding more constraints to narrow the range of possible solutions will not work very well in practice. It is plausible that a heuristic approach succeeds in solving this problem in a range of cases that are of practical interest, but always with the possibility that for specific instances the solver would bog down. Again, further research is needed to better understand the potentialities of the approach.

5.3 Dialogue with the Solver

The considerations above seem to suggest that no automatic approach to root identification will succeed in delivering an efficient constraint solver that gets the intended solution every time. Consequently, we feel that a promising alternative is to devise a few simple heuristics that succeed in many cases and are easy to understand. Beyond that, we rely on interaction with the user in those cases in which the heuristics fail to deliver an acceptable solution. Note that placement rules are used very widely, but are rarely discussed.

5.3.1 Placement Heuristics

All solvers known to us derive from the initial geometric sketch information that is used to select a specific solution. This is reasonable, since one can expect that a sketch is at least topologically accurate, so that observing on which side of an oriented line a specific point lies in the sketch is often reliably indicating where it should be in the final solution. However, when generic designs are archived and and later edited, one should no longer expect such simple correspondences between the sketch and the ultimate solution, because as dimension values change, so may the side of a line on which a point is situated.

In our system, we use very few but highly effective rules. We keep the number of rules to a minimum because we do not believe that root identification has a satisfactory and completely automated solution. Where the rules fail, we rely on user interaction to amend them as the situation might require. Note that our rules are fully supported by the Erep approach in that the different situations can be characterized and recorded faithfully.

Three Points: Consider placing three points, p_1, p_2 and p_3, relative to each other. The points have been drawn in the initial sketch in some position, and therefore have an order that is determined as follows. Determine where p_3 lies with respect to the line $\overline{(p_1, p_2)}$ oriented from p_1 to p_2. If p_3 is on the line, then

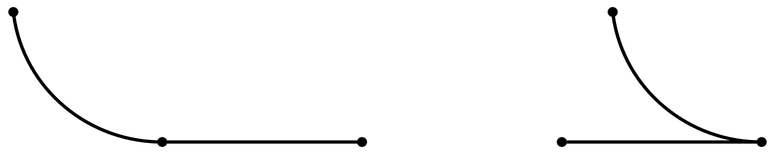

Figure 18: The Two Types of Tangency between an Arc and a Segment

determine whether it lies between p_1 and p_2, preceding p_1 or following p_2. The solver will preserve this orientation if possible.

Two Lines and One Point: When placing a point relative to two lines, one of four possible locations is selected based on the quadrant of the oriented lines in which the point lies in the original sketch. Note that the line orientation permits an unambiguous specification of the angle between the lines.

One Line and Two Points: The line is oriented, and the points, p_1 and p_2, are kept on the same side(s) of the line as they were in the original sketch. Furthermore, we preserve the orientation of the vector $\overline{p_1, p_2}$ with respect to the line orientation by preserving the sign of the inner product with the line tangent vector.

Tangent Arc: An arc tangent to two line segments will be centered such that the arc subtended preserves the type of tangency. The two types of tangency are illustrated in Figure 18. Moreover, the center will be placed such that the smaller of the two arcs possible is chosen, ties broken by placing the center on the same side of the two segments as in the input sketch. As specific *degeneracy heuristics*, an arc of length $0°$ is suppressed.

All rules except the tangency rule are mutually exclusive. They are therefore applicable without interference. The tangency rule could contradict the other rules, because dimensioned arcs and circles are determined by placing the center. In such cases, the tangency rule takes precedence. In our experiments with these rules, we found that most situations are solved as the user would expect. The rules are easy to implement, and are easy to understand for the user.

5.3.2 Selecting Alternative Solutions

A useful paradigm for user-solver interaction has to be intuitive and must account for the fact that most application users will not (and should not) be intimately knowledgeable about the technical workings of the solver. So, we need a simple but effective communication paradigm by which the user can interact with the solver and direct it to a different solution, or even browse through a subset of solutions in case the one that was found is not "right."

Conceptually, all possible solutions of a constraint problem can be arranged in a tree whose leaves are the different solutions, and whose internal nodes correspond to stages in the placement of elements or clusters. The different branches from a particular node are the different choices for placing the element or cluster. The tree has depth proportional to the number of elements and clusters. Browsing through all possible solutions would be exponential in the number of elements and would be inappropriate, but stepping from one solution to another one is proportional to the tree depth only.

We have added to our solver an incremental mode in which the user can browse through the construction tree and be visually informed which elements have been placed at a particular moment. With a button, the user steps forward or backwards in the construction sequence, thus traversing the tree path backwards, towards the root, or forward, towards a leaf. At each level, the geometric element(s) placed at that point are highlighted, and a panel displays the number of possible positions. The user can then select which one of the possible choices should be used.

For example, consider the constraint example of Figure 2. The role of the arc is clearly to round the corner that would be formed otherwise by the adjacent segments. When drawn as indicated in the figure, and with angle values larger than 45°, the solver finds the leftmost solution in Figure 16. However, when the angles are changed subsequently to 30°, the solver heuristics will select the solution shown in Figure 19, because the center of the arc remains on the same side of the adjacent segments. The user now relocates the center by changing

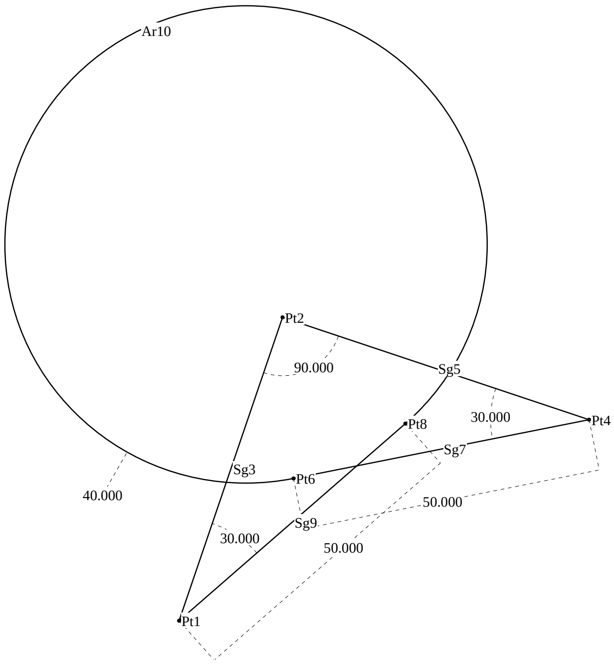

Figure 19: Default Solution After Changing Angles

the placement of Ar_{10} with respect to Sg_7 and Sg_9. By pressing the `level`

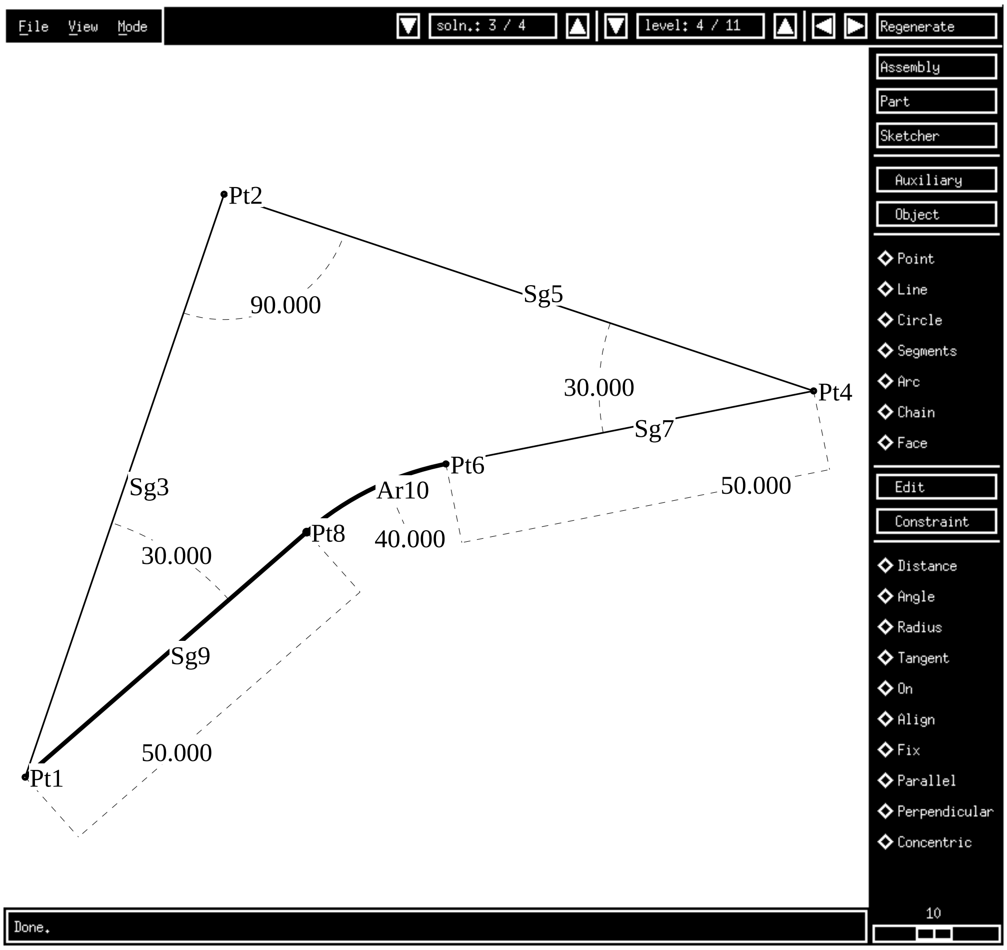

Figure 20: Interactively Changing Solutions: Elements highlighted are placed with respect to each other at this level in the solution tree.

buttons, the user returns to level 7. Here, Ar_{10} and Sg_7 are highlighted. The user now changes the solution by pressing the `soln.` buttons. This changes the arc center with respect to Sg_7 only. Continuing with the `level` buttons, on level 4 Ar_{10} and Sg_9 are highlighted. Again, a different solution is selected on that level, changing the arc with respect to Sg_9. Now the solver will construct the solution shown in Figure 20. We have found this simple interaction technique highly useful in exploring alternative solutions, and most users become effective in directing the solution process in a very short time.

In the Erep specifications of the interaction process, the solver is instructed serially to perform a back-up or to seek the next way to place an element or a cluster. This convention excludes solvers that find a solution by a numerical, iterative computation which places all geometric elements at once, unless the iteration is based on homotopy continuation techniques that can find all solutions of a nonlinear system of equations numerically.

Note that different solvers may cluster geometric elements differently and place the elements and clusters in a different sequence. Therefore, the same interaction sequence with the solver would have different effects with different solving algorithms. This cannot be avoided: To arrange the tree of solutions in canonical order, we either prescribe a canonical sequence *a-priori* in which the geometric elements have to be computed, or else we compute a canonical basis for the ideal generated by the constraint equations that describe the geometric problem, and then enumerate the associated variety in a canonical way; e.g., [8]. In the first case, we would prescribe the solver algorithm to belong to a certain family. In the second case, the ideal basis computation is equivalent to solving the constraint problem and thus constitutes committing to a canonical solver.[¶] Both ways compromise devising a neutral format of archiving. Consequently, we can neutrally archive a constraint problem (solved or unsolved), but not the manner in which to solve or seek an alternative solution. This is an intrinsic problem when solving geometric constraints.

5.4 Design Paradigm Approach

Consider solving the constraint problem of Figure 21. The role of the arc is clearly to round the adjacent segments, and thus it is most likely that the solution shown in Figure 22 on the left is the one the user meant rather than the one on the right, when changing the angles to 30°. The solver would be unaware of the intended meaning of the arc, and thus needs a technical heuristic, such as the tangent arc rule, to avoid the solution on the right. It would be much

[¶]Because such a canonical solver would be completely general, it could not be very fast in many situations, since the efficiency of constraint solvers rests on restricting the generality of the solver.

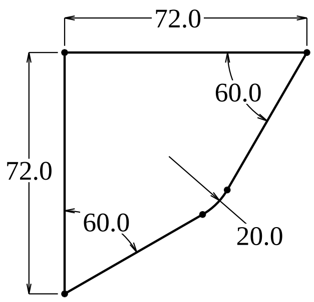

Figure 21: A Constraint Problem

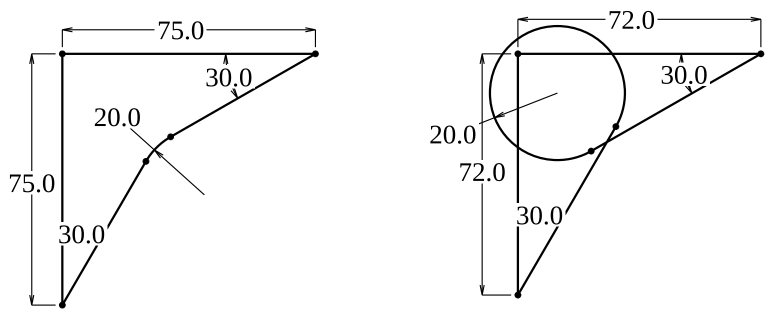

Figure 22: Two Solutions of the Constraint Problem of Figure 21 after Changing Angles

simpler if the user would sketch in such a way that the design intent of the arc is evident.

The difficulty for the geometric constraint solver is that sketches are usually *flat*; i.e., the geometric elements are not grouped into "features." It would be better to make sketches hierarchically: First, a basic dimension-driven sketch would be given. Then, subsequent steps, also dimension-driven, would modify the basic sketch and add complexity. In our example, the basic sketch could be a quadrilateral. There would be one subsequent modification adding a two-dimensional *round* with a required radius. This is analogous to feature-based design as implemented in current CAD systems.

The hierarchical approach to sketching has other important benefits. Since the sketch is structured, later modifications can be driven from constraints used in earlier steps, so that simple functional dependencies and relations between dimension variables of previously defined sketch features can be defined and implemented with trivial extensions of our basic constraint solver.

6 Summary and Future Work

Research on constraint solving should develop natural paradigms for narrowing down the number of possible solutions of a well-constrained geometric problem and devising solver interaction paradigms that allow the user to correct solutions that were not intended. With increasing penetration of constraint-based design interfaces, this problem is becoming increasingly more pressing.

Which solution is the intended one is also an issue when considering design archival in a neutral format. So far, neutral archiving formats have been restricted to detailed design without a formal record of design intent, constraint schema, editing handles, and so on. Where editable design has been archived, it has been done in a proprietary format native to the particular CAD system, and is typically a record of the internal data structures of the CAD system. In [24] we have presented alternatives. Current trends in data exchange standards indicate a growing interest in archiving constraint-based designs in which this additional information has been formalized without commitment to a particular CAD system.

In constraint-based, feature-based design, it is common to have available a variational constraint solver for 2D constraint problems, but not for 3D geometric constraints. This is particularly apparent in the *persistent id problem* discussed in [23]. A well-conceived 3D constraint solver conceivably can avoid these problems and assist in devising graphical techniques for generic design.

In manufacturing applications one is interested in functional relationships between dimension variables, because such relationships can express design intent very flexibly. Some parametric relationships can be implemented easily by structuring the sketcher as advocated in Section 5.4. Moreover, simple functional relationships are the content of certain geometry theorems, such as the theorems of proportionality, and many other classical results. Such theorems can be added to the constraint solver in a manner analogous to the extensions we have discussed before. But in general, functional relationships between dimension variables necessitate additional mathematical techniques. Geometric theorem proving has developed many general techniques that are applicable, but suitable restrictions are still needed to achieve higher solver speeds.

Geometric coverage refers to the range of shapes the constraint solver understands. In this work, we have restricted the geometric coverage to points. lines and circles. Conic sections would be easy to add, as would be splines such as Bézier curves, when translating the constraints to equivalent ones on control points. There is a rich repertoire of literature in CAGD that provides convenient tools for doing so. Yet it is far from clear whether control point manipulations are a universal tool for expressing constraints that the user finds natural, and we miss studies that analyze how to design with splines from an application's point of view.

Even with the resticted geometric coverage discussed here, some theoretical problems remain open. Although no precise analysis has been made, neither Owen's nor our constraint solving algorithm seems to run in worst case time linear in the number of graph edges. We conjecture that both algorithms run in quadratic time due to repeated traversals over regions of the graph. It would be worthwhile to analyze the worst case running times of these algorithms precisely, and study how to improve it. It is also worthwhile to consider how to minimize the arithmetic operations involved in the construction steps, and to analyze construction sequences for numerical stability.

Acknowledgement

We had several insightful discussions with John Owen from D-Cubed, Ltd.

References

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

[2] 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.

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

[4] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. Technical Report Rapports de Recherche 777, INRIA, 1987.

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

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

[7] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* PhD thesis, University of Innsbruck, Austria, 1965.

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

[9] B. Buchberger, G. Collins, and B. Kutzler. Algebraic methods for geometric reasoning. *Annual Reviews in Computer Science*, 3:85–120, 1988.

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

[11] J. Cai. A language for semantic analysis. Technical Report 635, New York University, Dept. of Comp. Science, 1993.

[12] J. Cai, P. Facon, F. Henglein, R. Paige, and E. Schonberg. Type transformation and data structure choice. In B. Moeller, editor, *Constructing Programs From Specifications*, pages 126–124. North-Holland, 1991.

[13] J. Cai and R. Paige. Towards increased productivity of algorithm implementation. ACM SIGSOFT, to appear, 1993.

[14] C.-S. Chou. *Mechanical Theorem Proving.* D. Reidel Publishing, Dordrecht, 1987.

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

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

[17] C.-S. Chou and W. Schelter. Proving Geometry Theorems with Rewrite Rules. *Journal of Automated Reasoning*, 2:253–273, 1986.

[18] W. Clocksin and C. Mellish. *Programming in Prolog.* Springer Verlag, 1981.

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

[20] W. Fitzgerald. Using Axial Dimensions to Determine the Proportions of Line Drawings in Computer Graphics. *Computer Aided Design*, 13(6):377–382, November 1981.

[21] B. Freeman-Benson, J. Maloney, and A. Borning. An Incremental Constraint Solver. *CACM*, 33(1):54–63, 1990.

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

[23] C. M. Hoffmann. On the semantics of generative geometry representations. In *19th ASME Design Automation Conference*, 1993.

[24] C. M. Hoffmann and R. Juan. Erep, a 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.

[25] S. Johnson. Yacc - yet another compiler compiler. Technical Report Computer Science Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[26] D. Kapur. A refutational approach to geometry theorem proving. In D. Kapur and J. Mundy, editors, *Geometric Reasoning*, pages 61–93. M.I.T. Press, 1989.

[27] D. Kapur and J. Mundy. Wu's method and its applications to perspective viewing. In D. Kapur and J. Mundy, editors, *Geometric Reasoning*, pages 15–36. M.I.T. Press, 1988.

[28] D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, 1968.

[29] D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergammon Press, Oxford, 1970.

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

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

[32] G. Kramer. *Solving Geometric Constraint Systems.* MIT Press, 1992.

[33] W. Leler. *Constraint Programming Languages: Their Specification and Generation.* Addison Wesley, 1988.

[34] J. Li. Using algebraic constraints in interactive text and graphics editing. In D. A. Duce and P. Jancene, editors, *Eurographics '88*, pages 197–205. Elsevier North-Holland, 1988.

[35] R. Light and D. Gossard. Modification of geometric models through variational geometry. *Computer Aided Design*, 14:209–214, July 1982.

[36] E. H. Lockwood. *A Book of Curves.* Cambridge University Press, 1961.

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

[38] J. Owen. Algebraic solution for geometry from dimensional constraints. In *ACM Symp. Found. of Solid Modeling*, pages 397–407, Austin, Tex, 1991.

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

[40] Pro/ENGINEER. *Modeling Users Guide: 2D Sketcher.* Parametric Technologies. Release 8.0.

[41] Reasoning Systems. *Refine User's Guide*, 1990. Version 3.0.

[42] T. Reps and T. Teitelbaum. *The Synthsizer Generator.* Springer Verlag, 1988.

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

[44] J. Schwartz, R. Dewar, D. Dubinsky, and E. Schonberg. *Programming with Sets: An introduction to SETL.* Springer Verlag, 1986.

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

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

[47] L. Solano and P. Brunet. A system for constructive constraint-based modeling. In B. Falcidieno and T. Kunii, editors, *Modeling in Computer Graphics.* Springer Verlag, 1993.

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

[49] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1991.

[50] G. 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.

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

[52] H. Suzuki, H. Ando, and F. Kimura. Variation of geometries based on a geometric-reasoning method. *Comput. & Graphics*, 14(2):211–224, 1990.

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

[54] J. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1988.

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

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

[57] Wu Wen-Tsün. Basic principles of mechanical theorem proving in geometries. *J. of Systems Sciences and Mathematical Sciences*, 4:207–235, 1986.

[58] Y. Yamaguchi and F. 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 North Holland, 1990.