FRONTIER - A general 2D and 3D geometric constraint solver Part I: architecture and implementation

Meera Sitharam[*†] Jian-Jun Oung[*] Adam Arbree[*] Naganandhini Kohareswaran[*]

May 21, 2004

Abstract

FRONTIER is a general 2D and 3D, opensource, variational constraint solver, designed and implemented to meet several requirements that correspond to inadequacies in current variational constraint solvers. These requirements include: the ability to mix constraints with feature based and other representations; generality and corresponding computational efficiency in dealing with complex, cyclic or 3D constraint systems; user-driven navigation of the set of solutions that is complete in some well-defined sense; recognizing and offering the user systematic, efficient and complete methods to detect and correct inconsistencies and ambiguities; overall transparent architecture, implementation, data flow and representation including portable modules and an extensible 3D visual user interface.

Part I of this manuscript describes FRONTIER's input-to-output functionality, architecture and implementation (specifically how it meets the last requirement above), along with essential background, motivation and comparisons with previous work. It also formulates a list of previously unsolved 6 technical and 2 implementational problems that capture the above requirements. The solutions to the implementational problems are discussed in Part I. The other problems, requiring algorithmic solutions are discussed in Part II.

FRONTIER leverages the advantages of the first author's existing geometric constraint decomposition algorithm called the Frontier Vertex Algorithm, as well as some of its previously published properties such as the systematic detection and correction of overconstraints. However, the solutions presented here - to the above list of technical and implementational problems - are described here for the first time.

Keywords: Variational geometric constraint solving, Cyclical and 3d geometric constraint systems, Decomposition of geometric constraint systems, User navigation of solution conformations, Feature-based and assembly modeling, Conceptual design, Parametric constraint solving, Underconstrained and Overconstrained systems, Degree of Freedom analysis, Constraint graphs.

Organization

[*]University of Florida, Work supported in part by NSF Grant CCR 99-02025, NSF Grant EIA 00-96104
[†]*corresponding author: sitharam@cise.ufl.edu*

1 Introduction and Motivation

Today's CAD systems largely restrict variational constraint representations to 2D cross sections. This persists despite the general consensus that advocates a judicious use of 3D variational constraints for the intuitive expression and maintanence of certain complex and cyclic relationships that often occur between features, parts or subassemblies [7] [15] [10] [40] [43] [42] [62] [64] [66].

It is generally understood that purely procedural, history based representations risk tediousness and inefficiency in the generation and maintanence even in simple cases such as the examples shown in Figures 5 and 4.

Such representations often require the designer to perform hand calculations, or explicit trial and error constructions on screen. Furthermore, intuitively local changes and updates could require long "rollbacks," at worst repeating the design process from scratch. These difficulties persist, even if the procedural representation incorporates B-rep or CSG solids, variational constraints on 2D cross sections combined with sweeps, extrusions or parametric constraints.

Scope.

We restrict our discussion here primarily to shape and geometry information at the conceptual design stage, and we will loosely use the word feature (hierarchy) to denote feature, part and subassembly (hierarchies). Within these bounds, we stay consistent with FEMEX and other standard definitions of feature hierarchy. See [8], [3] [6, 11, 12], [54, 55]. With few exceptions, we generally restrict ourselves rigid, non flexible shape elements as primitive geometric objects, and variational and assembly constraints expressible as polynomial equations, although we explain in Part II, Section 4 how we deal with certain simple inequalities. We usually rely on generic combinatorial or degree of freedom analysis during the constraint decomposition process. However, we detect exceptions, and provide methods for correctin misclassification in 3D caused by algebraic dependences. While we sketch how to facilitate the use of geometric constraints in conjunction with other geometric representations of features, parts and subassemblies in a feature hierarchy, our focus is not the conversion from one geometric representation to another: we generally assume the existence of such interfaces. We also do not deal with persistence, generic naming and related issues such as repeatability, feature matching and recognition. Other important problems and extensions that we do *not* discuss here are listed in Part II, Section 8, under future work and open problems.

Within the above scope, we identify 5 requirements below that must be satisfied by an effective constraint engine in order for 3D constraints (variational and assembly) to be used to their full potential in today's CAD systems.

New Contributions of this 2 part Manuscript

The contributions of this 2 part manuscript can be summarized as the methods using which FRONTIER - introduced in [57] and available as GNU opensource software [65] - meets the 5 requirements below.

It is necessary to distinguish the contributions of this manuscript from those of related other papers by the first author [38, 39], [35, 36, 39], in [53], [26], which led up to the development of FRONTIER. A guide to the reader: our previous results are sketched in the background Sections 2 and 4 only. Later sections - Section 6 and 7 of Part I and all of Part II - are devoted entirely to new contributions.

The new contributions are made precise as a series of 8 technical problems in Section 5. Subsequently, the first two implementational problems is devoted a section here and the remaining 6 problems, requiring algorithmic solutions are discussed in Part II of this manuscript.

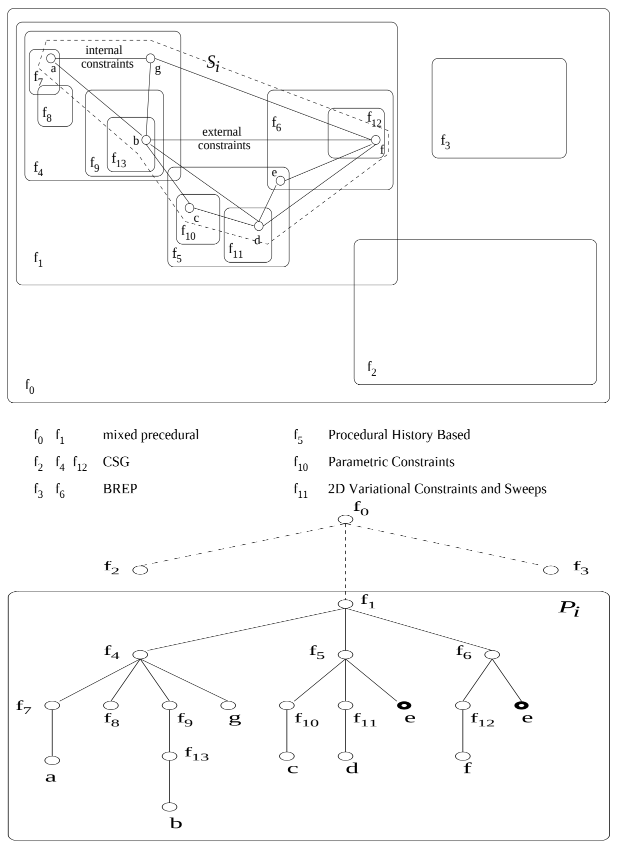


Figure 1: Constraint system S_i and underlying feature hierarchy P_i are input to FRONTIER; features $f_0, \ldots, f_{11}$ are in mixed procedural representation; a, b, c, d, e are participating feature handles; $f_0, \ldots, f_3$ are features at a higher level of hierarchy in larger CAD model

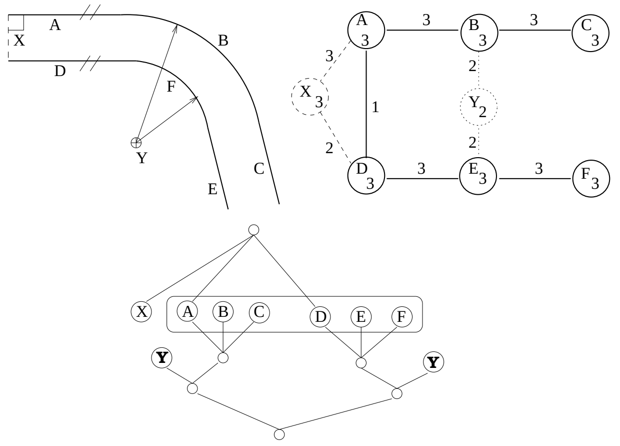


Figure 2: Multiple views (dotted and dashed) with different referencing elements x and y; numbers represent dofs; two views have intertwined feature hierarchies (above and below); box contains net shape elements common to both views

In order to motivate and formally state these problems, we first state the 5 requirements, briefly describe the progress so far in addressing the 5 requirements, compare them to FRONTIER's, provide basic background on geometric constraint solving (Section 2), give an input-output description of FRONTIER, (Section 3), as well as background on the prior work that led up to FRONTIER (Section 4).

1.1 Requirement 1: Generality combined with Efficiency

Geometric constraint engines in 2D and 3D need to be both general and efficient in order to be effective in solving complex, cyclic, spatial constraint systems. In general, efficiency and generality are strongly competing requirements. Crucial to tackling this tradeoff is the efficient generation of a close-to-optimal *decomposition and recombination (DR) plan* for such constraint systems, in order to deal with the tractability bottleneck in constraint solving: minimizing the size of simultaneous polynomial equation systems, thereby controlling the dependence on exponential time algebraic-numeric solvers. In particular, a general algebraic-numeric solver is practically crippled in dealing with even moderately sized systems. Consider the notorious constraint problem of finding a line in 3D that is tangent to 4 given spheres, or in other words, at a specified distance from 4 given points [14]. While this problem cannot be decomposed, its difficulty – relative to its size – dictates the need for decomposition of larger systems.

1.1.1 Previous work and Comparisons

FRONTIER's DR-planner is based on the recently developed Frontier vertex algorithm (FA) [38], [53], [26] which builds upon nearly a decade of work on geometric constraint solving. While this body of work implicitly relied on earlier methods for constraint system decomposition, the optimal DR-planning problem was isolated or defined in the literature for the first time only in [36].

Until then, many 2D variational constraint solvers used decomposition methods that detected only specific patterns of constraints such as triangles, thereby lacking generality. In particular, implementations of most of these methods are likely to perform poorly on at least one of the natural examples shown in Figures 5 and 4.

Others suffered from different drawbacks, such as far-from-optimal DR plans, and their inability to incorporate a desired input decomposition into features and subassemblies (see next requirement below). Most of these previous methods such as e.g.[4, 59, 60, 4], [40, 41, 18, 20], [1, 61, 51, 48], are classified and their performance is formally analyzed with respect to a number of relevant new measures in [38].

A serious difficulty in 3D variational constraint solving is the the lack of a combinatorial characterization of rigidity even for point and distance constraint systems. See [24]. In particular, the few existing 3D variational constraint solvers are not able to even deal with simple rigid subsystems in 3D that are not amenable to a degree-of-freedom analysis, but are yet crucial building blocks of most generically wellconstrained systems. As a result, practically no 3D constraint solvers existed and one of the most general 3D constraint solving problem that could be tackled by prior solvers was the pipe-routing problem of [58]. A general 3D constraint solver should both automate and provide systematic help to the user to both recognize and deal with nongeneric, special-case behavior that is not predicted by an automated degree-of-freedom analysis such as the "bananas problem" e.g., Figure 6. FRONTIER augments the Frontier vertex algorithm to deal with such commonly occuring 3D problems. This is discussed in Part II, Section 3.

1.2 Requirement 2: Mixed representations and Feature Hierarchies

Designers find it intuitive to represent many spatial features as a procedural history or an almost linear sequence of attachments, extrusions, sweeps, or CSG Boolean operations such as intersections, or parametric constraints etc. permitting B-rep and other representations of some features along the way. On the other hand, designers appreciate the expressiveness and intuitiveness of using variational constraints. It would be desirable if a procedural history could incorporate both 2D and 3D variational constraints, as well as other representations. In this paper, we denote such representations as *mixed* representations. See Figure 1. In particular, it would be desirable to use variational constraints to express the interaction of a collection of features, parts or subassemblies at some level of a feature hierarchy. Recursively, the features could themselves be represented entirely using other representations, or in a similar, mixed manner, using constraints to relate the sub-features. This is a natural representation, since regardless of the way in which the features at any given level are represented, constraints between features at that level could be specified between primitive geometric objects or *handles* belonging to the features.

To achieve this type of representation, the decomposition plan discussed under Requirement 1 has to incorporate an *input, partial decomposition* representing the underlying feature, part or subassembly hierarchy, a partial order, typically represented as a Directed Acyclic Graph or *dag*. See Figures 1 and 2. This incorporation of an input, conceptual design decomposition is crucial also in order to capture design intent, or assembly order and allow independent and local manipulation of features, parts, subassemblies or subsystems within their local coordinate systems. This is also crucial for providing the user with a feature repertoire, the ability to paste into a sketch already resolved features and constraint subsystems that are specified in another representation or allowing certain features or easier subsystems to be solved using other, simpler, methods such as triangle-based decomposition or parametric constraint solving. Sometimes the user would prefer to specify a priority at the vertices of the dag which dictates the order of resolution of the features, parts, subassemblies or subsystems. Note that parametric constraint solving can in fact be achieved as a special case, where the order is a complete, total order. More generally, this can be used in the CAD database maintanence of multiple product views as in [29], [12], for example the design view and a downstream application client's view may be somewhat different constraint systems and the two feature hierarchies may not even be refinements of one another, but intertwined. Furthermore, each view could contain different referencing shape elements that are not part of the net shape and therefore, not part of the other views. See Figure 2. This is particularly the case when these referencing shape elements are generated during the operations of a history-based procedural representation.

1.2.1 Previous Work and Comparisons

Previous work on such mixed representations can be classified into two broad types. The first type, such as [46, 47, 13], dictates a unified representation language which is an amalgamation of variational constraints with other representation languages such as CSG and Brep. The second type, such as [25] wrestles with a heterogeneous approach, using many servers, one for each representation language, so that the appropriate one can be called when required. Both approaches, while highly general in scope, have their obvious drawbacks. Our approach through FRONTIER has a narrower focus: how to freely enable a variational constraint representation of feature interactions using feature handles at any level within a mixed procedural representation of the larger feature hierarchy, permitting the features to be independently manipulated. FRONTIER represents feature handles as so-called *Frontier vertices* of a *rigid cluster* corresponding to the feature, in the corresponding constraint graph. (See Sections 2 and 4 for formal definitions). More importantly, its DR-planner incorporates an input design decomposition or feature hierarchy. This is discussed in Part II, Section 2.

It is shown in [38] that many of the previous decomposition methods listed above (or any obvious modifications of them) would inherently fail to incorporate even tree-like input design decompositions. FRONTIER's DR-planner incorporates multiple views, or intertwined input design decompositions. This is discussed in Part

II, Section 2. This is also crucial for designer-guided, conceptually meaningful navigation of the solution space, see below. This is discussed in Part II, Section 4.

1.3 Requirement 3: Conceptually meaningful navigation of the Solution Space

Constraint solvers should deal with the standard and well documented problem of classifying and steering through multiple generic solutions, realizations or conformations in a conceptually meaningful manner: specifically permitting recursive navigation of the solution space of each feature and subassembly in the input partial decomposition. Other methods of picking out solution choices should be permitted such as the appearance and relative orientation of primitives in the sketch in searching for a solution; and to help choose specific solutions, incorporating variety in the constraint repertoire to include constraints for navigation, such as functional and equational or engineering constraints and simple inequalities.

1.3.1 Previous work and Comparisons

The first approach [16, 19, 58] attempts to automatically pick a solution that is as close as possible to the appearance of the input sketch. Typically, chirality (or relative orientation) of the geometric primitives in the sketch is used as the guide. A related approach uses explicitly defined "navigation" constraints provided by the user. These could include chirality constraints, and other constraints related to chirality (by oriented matroid theory) such as intersection or nonintersection of convex hulls or subsets of the points, and reduce to specific polynomial inequalities, as well as relational or engineering constraints [20].

A second approach gives the user an indexing [5], of the solution space using a DR-plan of the constraint system as a guide. The methods of [5] provide tractable steering for simple, 2D constraint systems that are triangle decomposable, or have linear DR-plans, similar to ruler and compass constructible systems, that permit solving for one geometric primitive at a time.

In this case, each such addition provides two "bifurcation" choices that are typically two reflections of the newly solved-for primitive. However, such DR-plans exist only for special constraint systems and even then may not incorporate a user's conceptual feature decomposition, denying the user's prefered navigation path.

The third approach (FRONTIER's) is a hybrid method that both uses navigation constraints and offers two options: a *visual walkthrough* with backtracking or automatic search for a solution. FRONTIER uses its DR-plans to provide the user control over navigation by recursively navigating the features and subassemblies of the input feature hierarchy, or design decomposition. In addition, since FRONTIER is built on a clear, formal basis, its architecture and algorithms decouple the methods for optimal DR-planning (via the Frontier vertex algorithm, FA) from methods that are built upon a good DR-plan, regardless of how it is found. As a result several of the latter methods that were coupled with prior decompositions, carry over. This is discussed in Part II, Section 4.

For example: relational and engineering constraints are often used to specify desired solutions; these typically relate one distance to another without fixing either. The method for dealing with such constraints based on the decomposition given in [4, 4, 18, 20], carries over to the DR-plan given by the Frontier vertex algorithm and can be adopted by FRONTIER. Chirality constraints reduce to polynomial inequalities given by certain determinants and prior methods for dealing with these carry over. Note however that dealing with general polynomial inequalities as constraints is a difficult problem related to solving underconstrained systems. See Part II, Section 8.

1.4 Requirement 4: Dealing with Inconsistencies and Ambiguities Flexibly

Partly due to the rich declarative power and expressiveness of variational constraints (as opposed to more rigid procedural representations), constraint engines should support the management of a wide variety of ambiguities and inconsistencies of both input and output. In addition, to allow the user to deal with such ambiguities in general and complex constraint systems, constraint solvers should permit flexible user feedback and navigation at every stage of the constraint specification and solving process. This includes: (a) detecting over(under)specified (over)underconstrained systems and different constraint systems representing multiple client views; (b) facilitating efficient offline user updates of various parts of the constraint system including the input design decomposition (c) online, instantaneous resolution of partially specified systems, (d) permitting automatic as well as user-driven, step-by-step inspection for algebraic dependencies and instabilities and pre as well as post processing of the (indecomposable) algebraic systems sent to the canned algebraic/numeric solver; which is related to (e) easy editability of the constraint repertoire as well as editability of the constraint-to-(algebraic) equation parsing mechanism. In addition, providing all of this flexibility of user feedback and interaction also requires (f) a user interface that supports it.

1.4.1 Previous Work and Comparisons

Issues (a), (b), above gain from a good decomposition method that permits a partial decomposition to be input.

Detection of combinatorially overconstrained clusters is achieved by many DR-planners, including [51], [27], and a modification of [61] d escribed in [38]. See Figure 8. Most solvers ask the user to remove constraints in an over-constrained situation but give poor guidance which constraints to delete. In contrast, recent work in [26] provides a sequence of unique, well-defined, complete lists of overconstraints that can be generically removed without making the system underconstrained. This method extends directly to a solution to the problem of constraint reconciliation of multiple client views, and to a method of efficiently updating the sytem once one of these constraints is removed. This method is based on the frontier vertex algorithm, and is hence directly incorporated into FRONTIER. This is discussed in Section 4.

Underconstrained sketches arise often in practice. No satisfactory method exists for a systematic description and navigation of the infinite solution space of underconstrained systems (see Part II, Section 8), but many constraint solvers including [51] and [27] detect them. In fact, the complete set of maximal wellconstrained clusters of underconstrained systems can be read off as the "sources" or "roots" of a FRONTIER's DR-plan [53]. For underconstrained systems, FRONTIER also provides a sequence of exhaustive lists of constraints(types, not values) that can be generically added (a so-called completion) without making the system overconstrained. This was implemented in [65] (2001 version for general 2D constraint systems, 2002 version for a small class of 3D constraint systems, and by the 2003 version for most 3D constraint systems). A proof of why this method works is given in a paragraph of Part II, Section 6. This should be compared with a method for obtaining completions for a class of triangle decomposable 2D constraint systems which is the subject of the paper [45]. Obtaining the values associated with these completion constraints represents the remaining difficulty in solving underconstrained systems. However, for the CAD application, there are effective heuristics to infer these values [21, 68, 73] from the input sketch of the constraint system.

Issue (b), the question of efficient updates was mostly ignored before the DR-planning problem was formally defined and the quality and performance of DR-plan structure shown to be crucial for the entire geometric constraint solving process in [38]. Moreover, since the constraint systems are often simple and triangle decomposable, with effectively linear DR-plans whose subsystems were of small size and typically involved solving a single quadrati c equation, those constraint solvers such as [58] implement updates simply redoing the entire system from scratch. For general 3D constraint systems and more complex DR-plans, this approach is intractable. The FRONTIER geometric constraint solver leverages properties of its FA DR-plans in conjunction with the modularity and clarity of its architecture and other routines to perform efficient updates. This is discussed in Part II, Section 6.

Issue (c) namely online and interactive resolution (say, of the partial input constraint sketch so far) is related to the resolution of general underconstrained systems. This is, in general, a complex problem which has no satisfactory solution to date. See Part II, Section 8. The difficulty is however ameliorated by the fact that while the the online solving and partial output process need to be fast and inexpensive, some incorrectness and incompleteness can be tolerated, since the partial output is usually only meant to guide the user and the errors will be rectified when the input is complete and the (correct and complete) offline algorithm is run.

The best overall performance to date with respect to online and underconstrained system resolution is that of the commercial 2D variational geometric constraint solver by D-cubed amenable to a triangle based decomposition method [58]. FRONTIER's approach is to leverage its decomposition along with methods for resolving simple underconstrained systems, and parameterless constraints via its "Simplesolver." This is discussed in Part II, Section 7.

Issues (d) and (e) are crucial mainly in 3D (although they occassionally arise with the presence of angle and incidence constraints in 2D as well) and hence have not been addressed by the (primarily 2D) variational constraint solvers so far. See Figure 6. FRONTIER provides automatic as well as user-driven reduction and stability checking for algebraic dependencies (common in 3D, as in the "bananas" problem, see Figure 6) while formulating independent equations to resolve subsystems– this is a nontrivial task compounded by instabilities (during algebraic-numeric solving) caused by the presence of rotationally symmetric objects, shared objects and incidence constraints in the subsystem being resolved. This is discussed in Part II, Section 5. This is important since algebraic-numeric solvers typically need all the help they can get even on minimal or irreducible constraint systems obtained after decomposition. Interactive, flexible symbolic reduction and stability checking of of algebraic equation systems by the user are desirable. This gains from an editable and efficient constraint repertoire and interactive parsing mechanism, provided by FRONTIER. This is discussed in Part II, Section 5.

Concerning Issue (f), we first discuss user interfaces specifically of variational constraint solvers (we do not discuss parametric constraint solver, or traditional solid modeler user interfaces). These include 2D (Erep, D-cubed, I-DEAS, [5], [58]) and 3D user interfaces, such as Hoffman's 3D constraint interface built at Purdue, Bruderlin's 3D constraint interface built at TU-Ilmenau, and [58]. We also include interfaces that use some form of variational constraint solving (in a generous interpretation). The 2D user interface that is currently most successful is the commercial proprietary solver [58]. They also have a 3D constraint solver for example which solves the "pipe routing" problem. In particular, as pointed out earlier, the 2D interface is impressive at bringing to the user an online resolution of the sketch so far; and consistently and repeatably interpreting the sketch's appearance (say using relative orientation of primitive objects or chirality). The latter is used also by other constraint solvers such as [5] and a commercial solver I-DEAS. Some user interfaces also interpret gestures made by the user. Quicksketch [69] is a GUI that interprets the users strokes to construct the object rather than picking objects from the menu like most other 3D computer modeling/sketching systems. The sketches thus drawn can then be refined by defining 2D/3D constraints on them. Quicksketch is a tool for pen-based computers. Brown University's Sketch [74] uses a similar method for drawing and moving 3D objects on the screen using the mouse. Many of the commercial solid modeling and CAD systems infer geometric constraints and try to capture design intent by interpreting the users mouse movements.

It is important to note that the demands on the user interface are proportional to the number of backend facilities that the variational constraint solver offers and wants to bring to the user. The existing variational constraint solvers do not usually offer the generality and various backend facilities as discussed earlier. More significantly, they generally try to distance or hide the solving process completely from the user.

FRONTIER's user interface, on the other hand, encourages the user to interact during all aspects of the solving process, during DR-planning, building stable algebraic systems, navigating the solution space, backtracking, etc. At the same time it functions efficiently even without the user intervention and offers complete, well-defined sets of options to the user when updates need to be made to the constraint system, as discussed under Issues (a), (b) above. Hence the design of the user interface is significantly different. Extensibility is the primary issue. This is discussed in Section 7.

Some novel aspects are: FRONTIER permits 3D sketching either by editing a 3D scene graph, or by using a 2D-3D user interface which allows sketching the conceptual 3D constraint system on a 2D canvas and specifying any input decomposition requirements intuitively, but permitting the user to backtrack and navigate through the various subsystem solutions, displayed on a 3D canvas. The backtracking is facilitated by displaying the constraint system's DR-plan and allowing the user to click on subsystems in it to do the navigation. As discussed under Issues (d), (e) above FRONTIER also permits the user to create a constraint-equation parse tree that is not hardcoded, add and delete constraint types, object types, and feature types, and permits the user to massage the constraint subsystems (sent to the algebraic or numeric solver) into a stable and minimal form.

1.5 Requirement 5: Extensibility, Transparency, Portability

Finally, no constraint engine is a stand alone entity, and preferably each of its components needs to be easily and individually portable into a larger CAD system. This poses nontrivial implementation challenges particularly for constraint solvers that offer a high degree of generality and capabilities. This requires dealing with with a corresponding increase in the complexity of the individual modules, their organization and their communication.

1.5.1 Previous Work and Comparisons

Existing constraint solvers that have been well incorporated into larger CAD systems include the 2 commercial constraint solvers by D-cubed [58] and I-DEAS (both 2D). To achieve portability, FRONTIER relies on a constraint representation language and ne datastructures that are shared seamlessly by all of its modules including

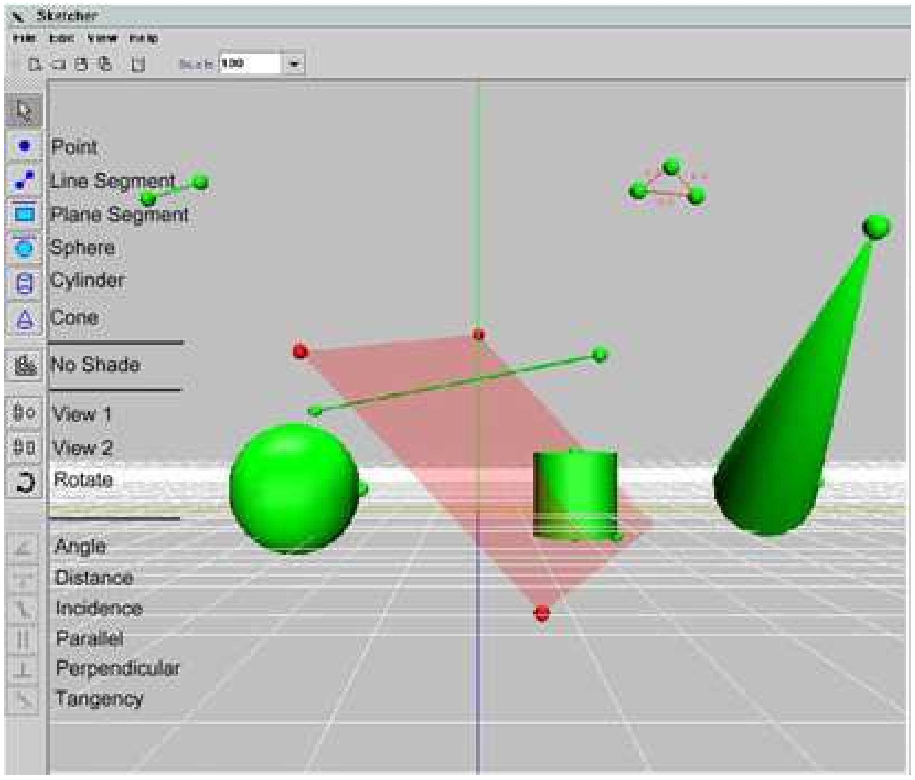


Figure 3: Typical 3D sketcher: icons show repertoire of object and constraint types

the graphical user interface. This actively facilitates almost all of FRONTIER's capabilities. The representation is based on the constraint system's DR-plan. This representation merges geometric and combinatorial information in a natural manner and is discussed in Section 6.

2 Basic Background

Geometric constraint systems also arise in many other applications besides CAD, including robotics, molecular modeling and teaching geometry [71, 56, 70, 63, 49, 51, 55, 67] [5, 16, 32, 33, 28, 35] [36, 38, 39, 37] [34, 27, 37, 22, 23, 59] [60, 7, 58, 61, 1] [26, 53, 57, 65]. For recent reviews of the extensive literature on geometric constraint solving see, e.g, [36, 48, 17]. Most of the constraint solvers so far deal with 2D constraint systems, although some of the newer approaches including [30, 31, 38, 39] [37, 7, 58] [26, 53, 57, 65], extend to 3D constraint systems.

A *geometric constraint system* consists of a finite set of geometric objects and a finite set of constraints between them. See Figure 3. The constraints can usually be written as algebraic equations and inequalities whose variables are the coordinates of the participating geometric objects. For example, a distance constraint of d between two points (x_1, y_1) and (x_2, y_2) in 2D is written as $(x_2 - x_1)^2 + (y_2 - y_1)^2 = d^2$

A *solution or realization* of a geometric constraint system is the (set of) real zero(es) of the corresponding algebraic system. In other words, the solution is a class of valid instantiations of (the position, orientation and any other parameters of) the geometric elements such that all constraints are satisfied. Here, it is understood that such a solution is in a particular geometry, for example the Euclidean plane, the sphere, or Euclidean 3 dimensional space. A constraint system can be classified as *overconstrained*, *well-constrained*, or *underconstrained*. Well-constrained systems have a finite, albeit potentially very large number of *rigid* solutions; their solution space is a zero-dimensional variety. Underconstrained systems have infinitely many solutions; their solution space is not zero-dimensional. Overconstrained systems do not have a solution unless they are *consistently overconstrained*. Well or overconstrained systems are called *rigid* systems.

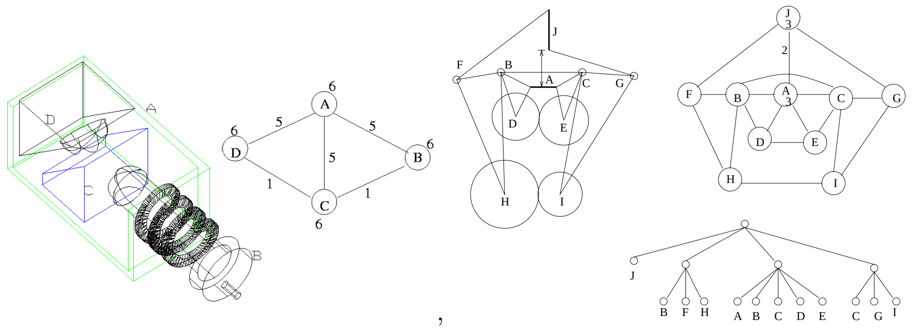


Figure 4: 3D example (underconstrained), and dof constraint graph; 2D example, constraint graph, DR-plan all unnumbered vertices and edges have dof weights 2 and 1

Technically, geometric constraint solving straddles combinatorics, algebra and geometry. Specifically, it overlaps combinatorial rigidity theory, geometric methods for kinematics, robotics and mechanisms, automated geometry theorem proving, geometric invariant theory, and computational algebraic geometry (solving specific classes of polynomial systems arising from geometric constraints).

The question of "to what extent can geometric constraint problems be approached combinatorially?" is highly non-trivial and important. It provides a framework for a systematic discussion of progress in the area and leads to a natural organization of the new contributions to be discussed in this manuscript.

2.1 Constraint Graphs and Degrees of Freedom

Recall that geometric constraint problem consists of a set of geometric elements and a set of constraints between them. A geometric constraint graph $G = (V, E, w)$ corresponding to geometric constraint problem is a weighted graph with n vertices (representing geometric objects) V and m edges (representing constraints) E; $w(v)$ is the weight of vertex v and $w(e)$ is the weight of edge e, corresponding to the number of degrees of freedom available to an object represented by v and number of degrees of freedom (dofs) removed by a constraint represented by e respectively.

For example, Figures 4, 11, 5 show 2D and 3D constraint systems and their respective dof constraint graphs. Several more 3D constraint systems whose graphs have vertices of weight 3, 5 and edges of weight 1,3 can be found in Figures 11, 12, 6, 7, 8. One more 2D constraint system whose graph has vertices of weight 3 (variable radius circles) and 2 and edges of weight 1 can be found in Part II, Figure 13.

Note that the constraint graph could be a *hypergraph*, each hyperedge involving any number of vertices. A subgraph $A \subseteq G$ that satisfies

$$\sum_{e \in A} w(e) + D \geq \sum_{v \in A} w(v) \tag{1}$$

is called *dense*, where D is a dimension-dependent constant, to be described below. Function $d(A) = \sum_{e \in A} w(e) - \sum_{v \in A} w(v)$ is called *density* of a graph A.

The constant D is typically $\binom{d+1}{2}$ where d is the dimension. The constant D captures the degrees of freedom of a rigid body in d dimensions. For 2D contexts and Euclidean geometry, we expect $D = 3$ and for spatial contexts $D = 6$, in general. If we expect the rigid body to be fixed with respect to a global coordinate system, then $D = 0$.

Next, we give some purely combinatorial properties of constraint graphs based on density. These will be later shown to be related to properties of the corresponding constraint systems.

A dense graph with density strictly greater than $-D$ is called *overconstrained.* A graph that is dense and all of whose subgraphs (including itself) have density at most $-D$ is called *wellconstrained.* A graph G is called *well-overconstrained* if it satisfies the following: G is dense, G has atleast one overconstrained subgraph, and has the property that on replacing all overconstrained subgraphs by wellconstrained subgraphs (in any manner), G remains dense. A graph that is wellconstrained or well-overconstrained is said to be *rigid* or a *cluster.* A dense graph is *minimal* if it has no dense proper subgraph. Note that all minimal dense subgraphs are clusters but

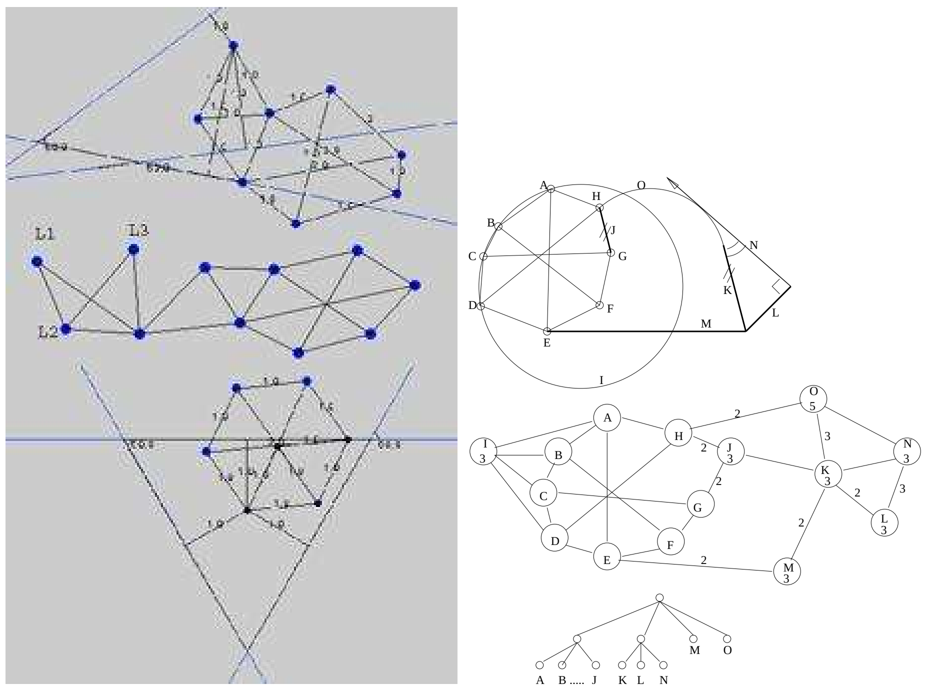


Figure 5: Two more 2D constraint systems examples; constraint graphs; DR-plans; all unnumbered vertices and edges have dof weights 2 and 1. On left: vertices L1, L2, L3 in constraint graph represent the line objects; solution shown at bottom

the converse is not the case. A graph that is not a cluster is said to be *underconstrained.* If a dense graph is not minimal, it could in fact be an underconstrained graph: the density of the graph could be the result of embedding a subgraph of density greater than $-D$.

To discuss how the graph theoretic properties based on *degree of freedom (dof) analysis* described above relate to corresponding properties of the corresponding constraint *system*, we need to introduce the notion of *genericity.* Informally, constraint system is generically rigid if it is rigid for most of choices of coefficients of system. More formally we use the notion of genericity of e.g, [9]. A property is said to hold *generically* for polynomials $f_1, \ldots, f_n$ if there is a nonzero polynomial P in the coefficients of the f_i such that this property holds for all $f_1, \ldots, f_n$ for which P does not vanish.

Thus the constraint system E is generically rigid if there is a nonzero polynomial P in the coefficients of the equations of E - or the parameters of the constraint system - such that E is solvable when P does not vanish. For example, if E consists of distance constraints, the parameters are the distances. Even if E has no overt parameters, i.e, if E is made up of constraints such as incidences or tangencies or perpendicularity or parallelism, E in fact has hidden parameters capturing the extent of incidence, tangency, etc., which we consider to be the parameters of E.

2.2 Inadequacy of a pure dof analysis

A generically rigid system always gives a cluster, but the converse is not always the case. In fact, there are well-constrained, even minimal dense clusters whose corresponding systems are not generically rigid and are in fact generically not rigid. Consider for example the Figures 7, 6, 8, 11 related to the so-called “bananas” problem in 3D, which can be detected as the root cause beneath large class of combinatorial misclassifications, although this detection is nontrivial.

A combinatorial dof analysis of the 3D constraint system in Figure 7(top) would correctly report the left and right subsystems ($P1, P2, P3, P4, P5$ and P_1, P_6, P_7, P_8, P_5 respectively) and the whole system to be well-constrained clusters. Figure 8 (bottom) has the same number of constraints, but a dof analysis would correctly report both that the left subgraph as overconstrained and the whole as underconstrained. Figure 6 (left) also

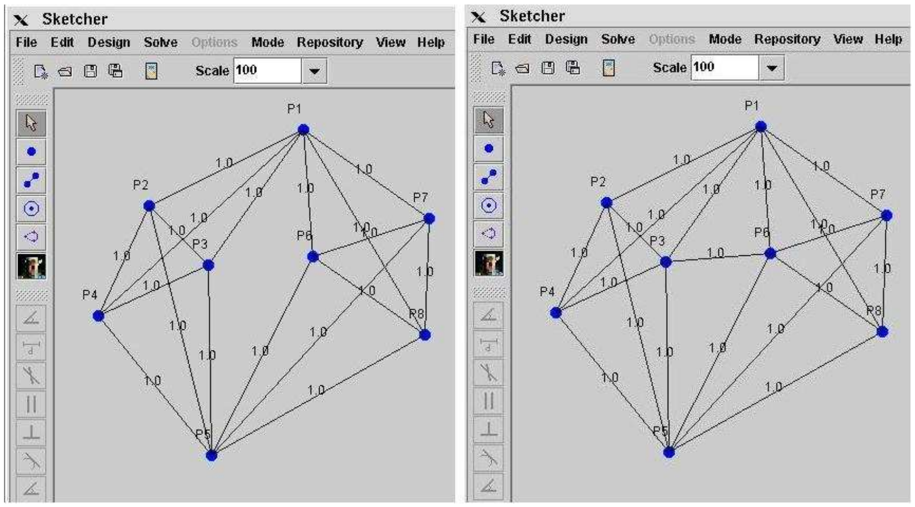


Figure 6: Algebraic overconstraint in 3D: constraint system drawn on a 2D canvas; corresponding constraint graphs have vertices of weight 3 and edges of weight 1

has the same number of constraints and appears to be a wellconstrained cluster according to a dof analysis. However, while the left and right subsystems are (in fact) wellconstrained clusters, the whole system is generically overconstrained. In a well-defined sense, this is an *algebraic* as opposed to *combinatorial* overconstraint. However, when *restricted to consistently overconstrained situations* (those choices of distances - such as in this example - that are guaranteed to admit a solution), the system in Figure 6 (left) is generically underconstrained, although the system on Figure 6(right) is generically wellconstrained. A similar classification holds for Figure 11.

In fact, constraint system is algebraically overconstained if the common overlap of *any* subset of its wellconstrained clusters is underconstrained. The above “bananas” is a special case of this. However, the dof analysis is inaccurate only in the “bananas” situation. Another standard example, in 4 dimensions the graph $K_{7,6}$ representing distances is minimal dense, and hence a cluster, but it does not represent a generically rigid system. However, it should be noted that in 2 dimensions, according to Laman's theorem [50] if all geometric objects are points and all constraints are distance constraints between these points then any minimal dense cluster represents a generically rigid system.

In fact, there is no known, tractable characterization of generic rigidity of systems for 3 or higher dimensions, based purely properties of the constraint graph. While it is possible to determine generic rigidity by examining the corresponding graph rigidity matrix of indeterminates, (based on so-called infinitesimal framework rigidity) this approach and the best known “almost-characterizations” and conjectures based on graph rigidity and matroids (in 3 dimensions) lead to algorithms that have typically exponential behavior, and no known conjectures has held up in 4 dimensions [72], [24].

In fact even in 2D, while Laman's theorem [50] combinatorially characterizes generic rigidity of point and distance systems, there are no known combinatorial characterizations of rigidity, when other constraints besides distances are involved.

For example, Figure 9 shows a case of angle constraints in 2D: 3 line segments with 3 incidence constraints form a triangle with 3*4-3*2 = 6 (resp. 3*6-3*3 = 9) degrees of freedom. It would appear that to make it wellconstrained, we can introduce 3 angle constraints (each of which removes 1 dof). But in fact, this would make it algebraically overconstrained. (This can be leveraged for building light weight, online, constraint solvers for special constraint systems. See Part II, Section 7).

NOTE: We will nevertheless rely heavily on combinatorial dof analysis of constraint graphs - carefully augmented by some checks for algebraic dependence as in Part II, Section 3) - to determine generic rigidity of constraint systems; hence from now on we will use the terms *rigid system* and *cluster* interchangeably.

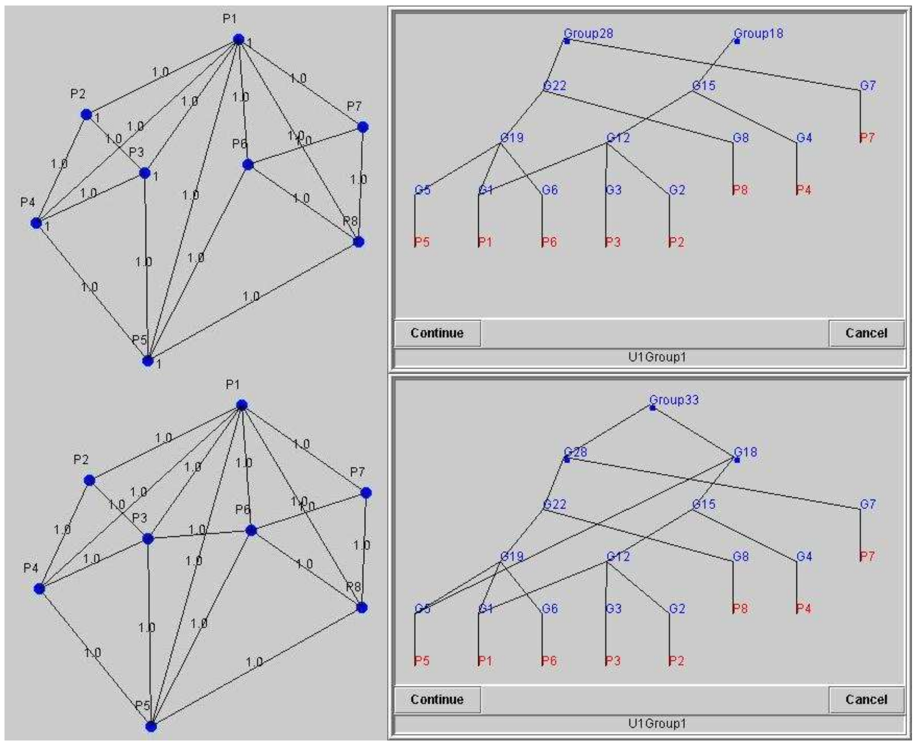


Figure 7: Modifications to 3D system in Figure 6: Combinatorially well constrained (DR-plan has single source, top) and underconstrained (DR-plan has many sources, bottom)

2.3 The need for decomposition: DR-plans and their properties

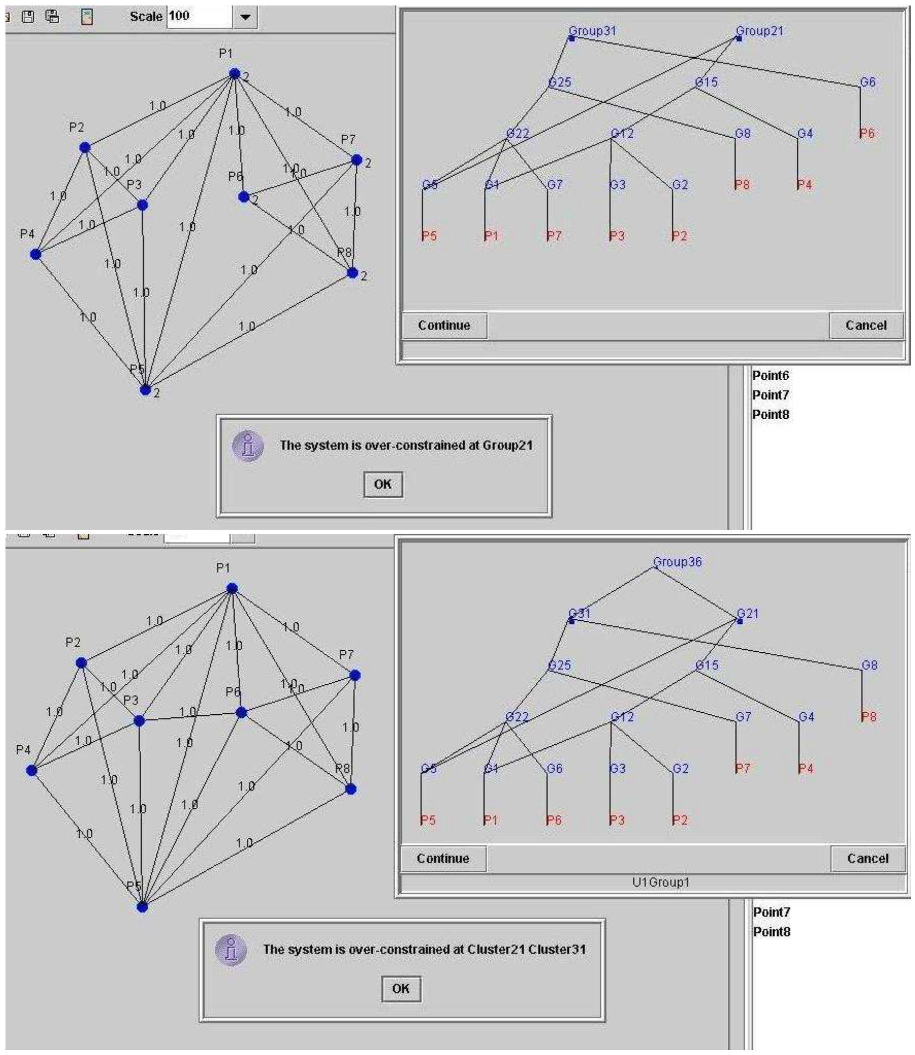
The overwhelming cost of solving a geometric constraint system is the size of the largest subsystem that is solved using a direct algebraic/numeric solver. This size dictates the practical utility of the overall constraint solver, since the time complexity of the constraint solver is at least *exponential* in the size of the largest such subsystem.

Therefore, an effective constraint solver should combinatorially develop a plan for (recursively) decomposing the constraint system into small subsystems, whose solutions (obtained from the algebraic/numeric solver) can be (recursively) recombined by solving other small subsystems. Such a recombination is straightforward, provided all the subsystems are generically rigid (have only finitely many solutions). The DR-planner is a graph algorithm that outputs a *decomposition-recombination plan (DR-plan)* of the constraint graph. In the process of combinatorially constructing the DR-plan in a bottom up manner, at stage i, it locates a wellconstrained subgraph or cluster S_i in the current constraint graph G_i, and uses an abstract *simplification* of S_i to to create a transformed constraint graph G_{i+1}.

Although recursive decompositions were used for geometric solving from the beginning, DR-plans and their properties were formally defined for the first time in [38]. See Figures 4, 5. Formally, a DR-plan of a constraint graph G is a directed acyclic graph (DAG) whose nodes represent clusters in G, and edges represent containment. The leaves or sinks of the DAG are all the vertices (primitive clusters) of G. The roots or sources are all the maximal clusters of G. There could be many DR-plans for G. See Figures 13, and 10. An *optimal* DR-plan is one that minimizes the maximum fan-in. The *size* of a cluster in a DR-plan is its fan-in (it represents the size of the corresponding subsystem, once its children are solved).

Besides solving efficiency purposes, it is straightforward that a DR-plan is required for the classification problem (the output requirement (1) above) and for the underconstrained detection and completion problem (output requirement (4) above). In addition, it will be clear from Section 4.3, and Part II, Sections 4, 6, that it is indispensable also for navigation, dealing with overconstraints and for efficient updates (output requirements (2), (3) (5) above).

Sometimes a somewhat modified definition of a DR-plan is used that incorporates another partial order



The system is over-constrained at Group21

The system is over-constrained at Cluster21 Cluster31

Figure 8: Combinatorially overconstrained clusters in well (single source in DR-plan) and underconstrained (multiple sources in DR-plan) graphs

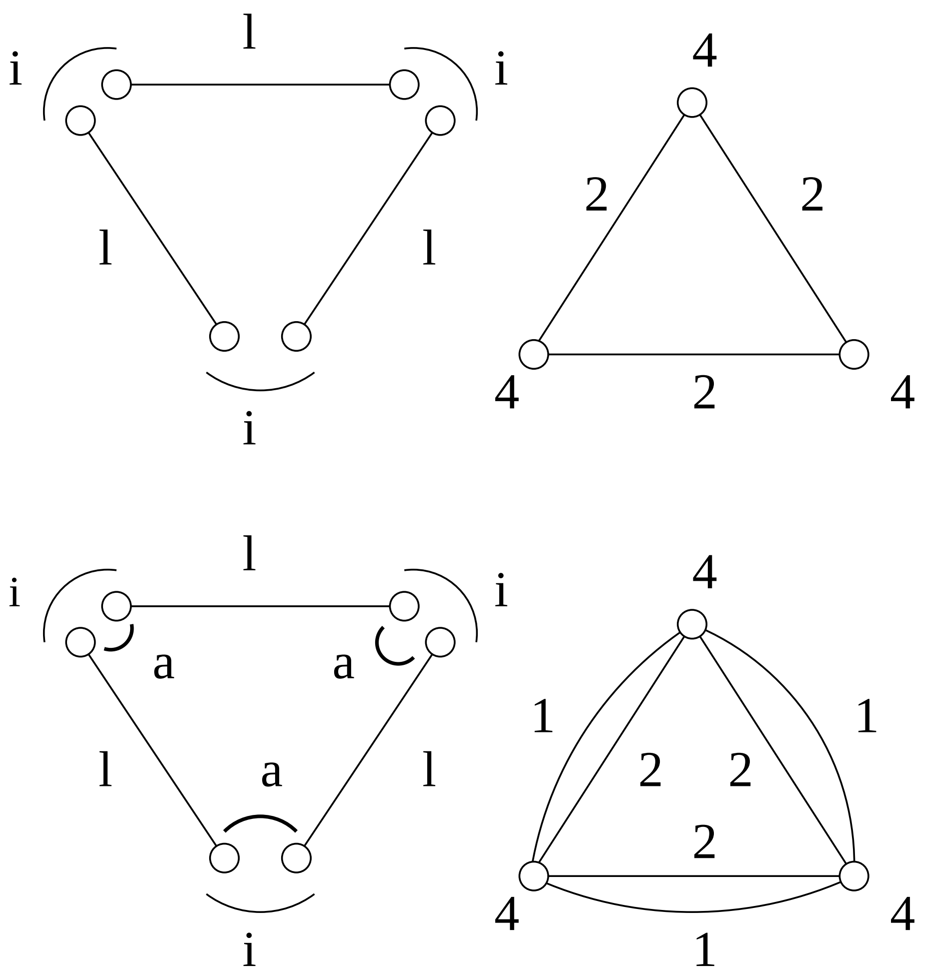


Figure 9: Algebraic overconstraint caused by 2D angle constraints.

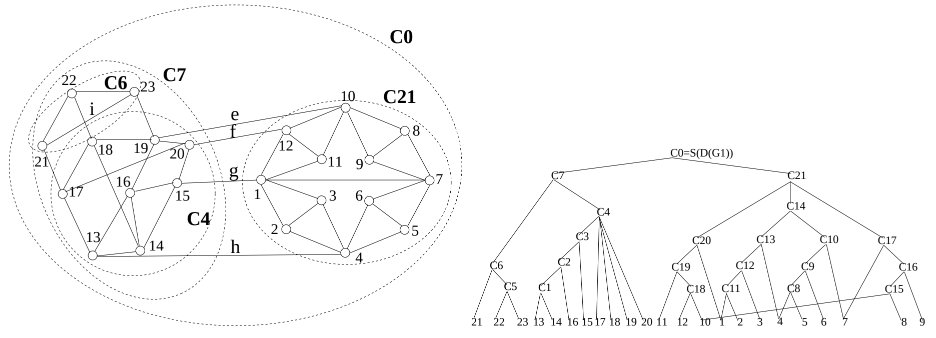


Figure 10: 2D constraint graph G1 and DR-plan; all vertices have weight 2 and edges weight 1

called the *solving priority order*, which is consistent with the DR-plan's DAG order, but could be more refined. This is particularly useful in 3D for assembly constraint systems as well as for correcting inaccurate dof analyses (as will be seen in Part II, Section 3. The intent is that clusters that appear later in the order need to be solved after the clusters that appear earlier. In fact, the nodes in such a DR-plan may not be independent clusters that appear in the original constraint graph or constraint system. They become wellconstrained clusters only in the transformed constraint system (resp. graph) after earlier clusters in the solving order are already solved (resp. simplified).

A few other properties of DR-plans are of interest. We would like the *width* i.e, *number* of clusters in the DR-plan to be small, preferably linear in the size of G: this reflects the complexity of the planning process and affects the complexity of the solving process that is based on the DR-plan. Since clusters are typically minimal dense subgraphs (see Section 2), it is desirable for DR-plans to have the *cluster minimality* property: i.e., for any node in the DR-plan, no proper subset of at least 2 of its children induces a cluster. Another desirable property is that the DR-plan *incorporate an input partial decomposition.* I.e., given an input DAG P whose nodes are subgraphs of G and whose edges represent containment, a DR-plan of every node in P should be embedded in the output DR-plan for G.

All properties defined above for DR-plans transfer as performance measures of the *DR-planners* or DR-planning algorithms. It is shown in [53], that the problem of finding the optimal DR-plan of a constraint graph is NP-hard, and approximability results are shown only in special cases. Nonapproximability results are not known. See Part II, Section 8. However, most DR-planners make adhoc choices during computation (say the order in which vertices are considered) and we can ask how well (close to optimal) the *best* computation path of such a DR-planner would perform (on the worst case input). We call this the *best-choice approximation factor* of the DR-planner.

3 Basic Input-Output capabilities of FRONTIER

Note: This is illustrated by the various screenshots of the FRONTIER's operation, throughout this 2-part manuscript.

FRONTIER is intended to be called by a CAD system at any stage where variational (2D or 3D) constraints are used to specify relationships between primitive geometries at some level of a feature or assembly hierarchy. While FRONTIER is intended to interact directly with the GUI of the calling CAD system, it also has its own graphical user interface to be discussed later in this paper and in Part II.

Assuming that its own GUI is being used for input/output, we describe below is a generic session with FRONTIER.

The generic input to FRONTIER consists of the following.

(1) A 2D or 3D geometric constraint system (see Section 2), S, i.e, a set of primitive geometric objects, each of a specified type, and constraint types relating pairs (or larger subsets) of these objects. See Figure 3. Some constraint types have a metric value specification and these could be constants or variables related to other such values (in the case of so-called *relational or engineering* constraints). In addition, *intervals* of values are permitted for some of the incidence or intersection constraints which are effectively chirality or oriented matroid constraints, declaring (non) intersections of convex hulls, and used primarily for navigation of the solution space (for example the constraint that 2 line segments should or should not interesect). The constraint system (including 3D systems) is often sketched on a 2D canvas. See Figures 11 and 12.

(2) An input partial decomposition of the constraint system P obtained from one or more feature, part or subassembly hierarchy design views. The features or subassemblies (nodes of the DAG) are intended to be manipulated independently within their own local coordinate systems, as well as moved around relative to each other in the coordinate system of a higher level feature or subassembly. There could be a priority order given to the resolution of the features (a linear order degenerates to parametric constraint solving), or subassemblies. See Figure 13.

(3) Some of the features or subassemblies in the input hierarchy may have been already solved and may be chosen from a repertoire of commonly occuring such features, parts or subassemblies. In this case, they may not be input as constraint systems; they are flagged and treated en-bloc as already solved rigid bodies.

(4) Additions, changes, updates to all of the above. See Part II, Figure 16.

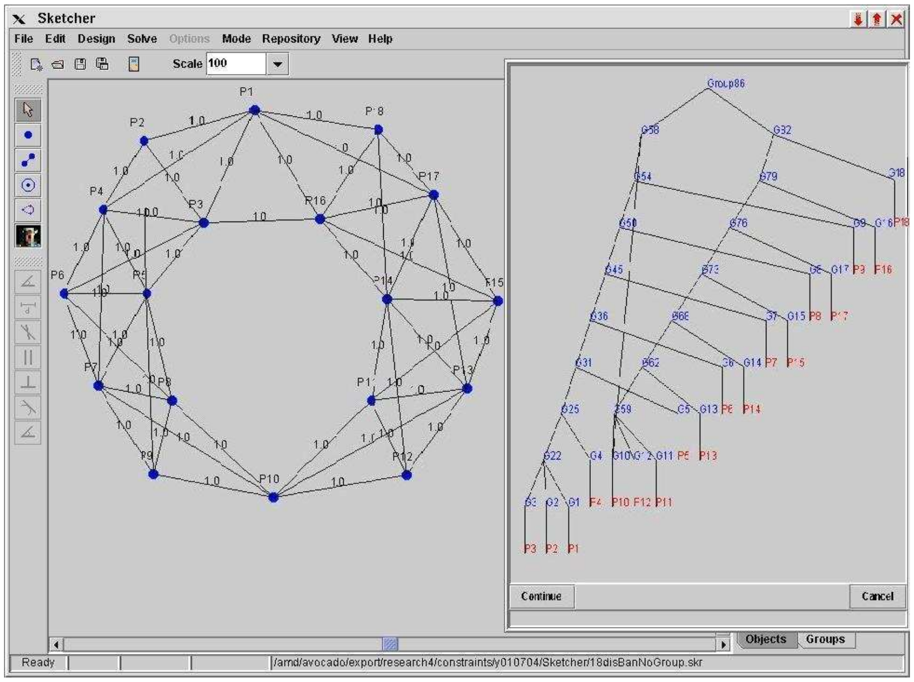

Sketcher
File Edit Design Solve Options Mode Repository View Help
Scale 100
Continue
Cancel
Objects Groups
Ready
/amd/avocado/export/research4/constraints/y010704/Sketcher/18disBanNoGroup.skr

Figure 11: 3D generically algebraically overconstrained system, sketch on a 2D canvas; for given set of distances, wellconstrained; corresponding DR-plan

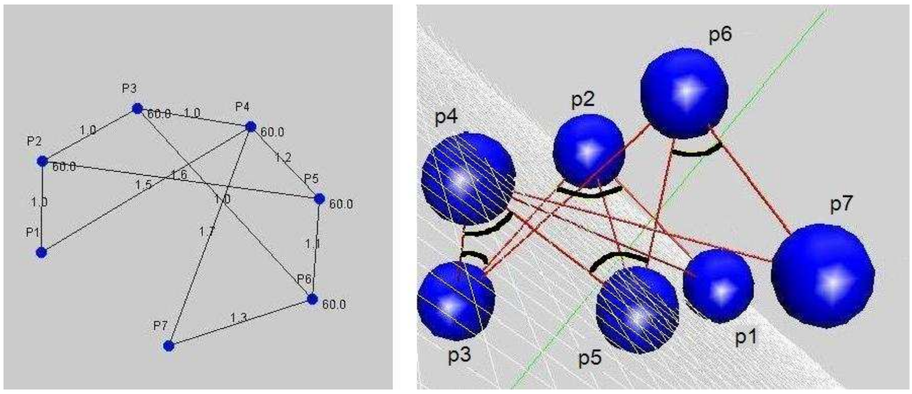


Figure 12: 3D constraint system drawn on 2D canvas with line and point objects and distance and angle constraints (graph has vertices of weight 3(points and lines); edges of weight 3 (incidences) and 1 (distances and angles). Solution (right)

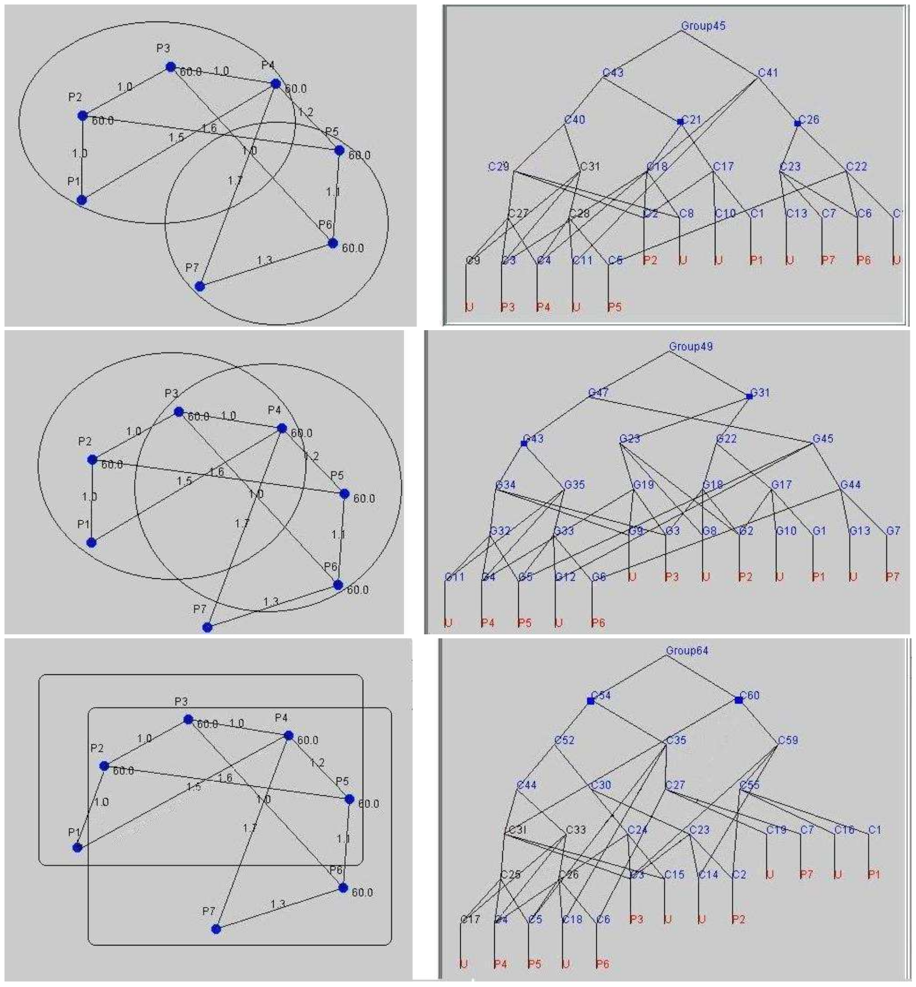


Figure 13: Left: input partial decompositions for 3D constraint system shown in Figure 12; groups are features or subassemblies. Right: 3 different DR plans incorporating corresponding input decomposition. Features appear as clusters or, if underconstrained, their complete set of maximal clusters. Features may (not) intersect on (non) trivial subgraphs.

(5) Other interactive user input requested by the geometric constraint solver during solution space navigation. See Part II, Figures 10, 11, 12, 13.

FRONTIER's outputs are the following.
(1) *Classification* of the input constraint system S as (generically) wellconstrained, overconstrained or underconstrained. Classification of each feature or subassembly in the hierarchy P as wellconstrained, overconstrained or underconstrained. See Figures 7, 8, 6.
(2) In the case the system is classified as wellconstrained, FRONTIER gives a method for systematic (with or without user intervention) *navigation* of the solution space, specifically, the various solutions, realizations or conformations of each of the wellconstrained features, parts and subassemblies of the associated input partial decomposition P, in priority order (if one is given) See Part II, Figures 10, 11, 12, 13, and Figure 12.
(3) In case combinatorially overconstrained (potentially *inconsistent*) features, parts, subassemblies or subsystems are present, see Figures 8 (for any requested collection of them), FRONTIER gives a complete, navigable representation of the wellconstrained *reductions*; for example an exhaustive list of existing constraints any one of which can be removed without generically making the entire system or any of the subsystems (in the collection) underconstrained. For each such removal, a second exhaustive list of constraints that can be removed, and so on, resulting in a sequence of lists.
(4) In case of those (sub)systems classified as underconstrained (or ambiguous), FRONTIER gives a decomposition into a complete set of maximal clusters (i.e., where all the maximal clusters are present). In addition, a complete, tractable, navigable representation of its wellconstrained *completions.* This question again has a well-defined interpretation using rigidity matroids which we omit here. One such navigable representation could be an exhaustive list of constraints (participating objects and type of constraint, but not value) that can be added without generically making the system overconstrained. For each such addition, a second exhaustive list of constraints that can be added, and so on, resulting in a sequence of lists.
(5) A method of efficiently *updating* the output when incremental changes are made to the input. For example: a contraint value is changed, a constraint is added or removed, an object is added or removed, a feature is added etc. These could be offline or online updates (see settings of a typical constraint solver below).

3.1 FRONTIER's phases, solving options, modes and settings

As discussed in Section 2 and Section 4, FRONTIER has a *planning* phase, which is purely combinatorial, mainly graph theoretic resulting in a DR-DAG; and a *solving* phase where the polynomial systems corresponding to the nodes of the DR-DAG are actually solved using an algebraic-numeric solver. In the latter phase, two solving options are available. The *autosolve* option in which FRONTIER searches and displays a solution or realization of the entire input system (in case it is wellconstrained). There could be exponentially many such solutions, the process is not steered by the user and each solution is displayed as it is found. See for example the full solution displayed in Figure 12. In the *navigate* option, the user is permitted to guide the process closely by picking one of the available solutions (displayed on a separate window) for each subsystem appearing as a node in the DR-DAG, which should include all the maximal well or well-overconstrained subsystems of the user's input conceptual feature, part or subassembly hierarchy See Part II, Section 2 and Section 4. Backtracking starting at any subsystem of the DR-DAG is fully supported, and partially solved systems should be displayed. See Figures 13 and Part II, Figures 10,11, 12, 13.

FRONTIER's *generate* mode is used when the input is new; the *update* mode is used when incremental changes are being made to any of the part of an earlier input; the underlying algorithms of the planning and solving phases permit efficient updates for the changed input. See Part II, Figure 16. One important advantage of modularizing the phases is that in the update mode (Part II, Section 6), the DR-planning phase is entirely omitted when, for example, only value changes are made to existing constraints – these do not generically change the DR-plan. FRONTIER's *offline* setting would be used if the entire input is constructed (sketched using the GUI) before the planning and solving phases are begun. The *online* setting is used when an instant resolution of each constraint is required even as it is being input, typically by sketching on the GUI screen. See Part II, Figure 17 and Part II, Section 7.

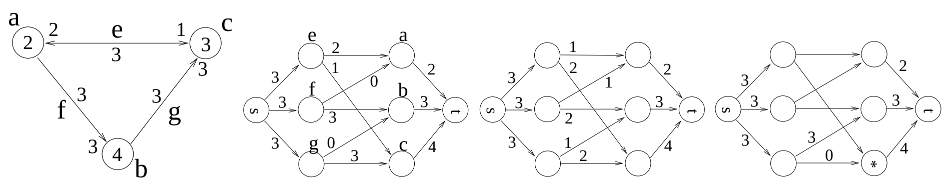


Figure 14: From Left. Constraint graph G with edge weight distribution. D is assumed to be 0 (system fixed in coordinate system); A corresponding flow in bipartite G^*. Another possible flow. Initial flow assignment that requires redistribution

4 Previous Work: The Frontier Vertex Algorithm (FA) DR-Planner

Decomposition algorithms based on constraint graphs have been proposed since the early 90's based on recognition of subgraph patterns such as triangles [21, 59, 60, 58] [51, 55]; and based on Maximum Matching [61, 1]. However, prior to [38], the DR-planning problem and appropriate performance measures for the planners were not formally defined. That paper also gives a table comparing 3 main types of DR-planners, with respect to these performance measures including those in Section 3 and Section 2.

In this this section, we sketch the properties of the Frontier vertex DR-plans and the corresponding DR-planner (FA DR-planner) [39, 37, 53, 57, 26, 65] which was designed specifically to excel simultaneously in these strongly competin performance measures. FRONTIER uses this DR-planner; in particular several new algorithms related to DR-planning introduced in Part II (Sections 2 and 3). are built upon this DR-planner.

Note. The pseudocode in Appendix of Part II incorporates the new algorithms into the existing FA-DR-planner.

4.1 The Frontier Vertex DR-plan (FA DR-plan)

Intuitively, an FA DR-plan is built by following two steps repeatedly:

1. *Isolate* a cluster-minimal dense subgraph C in the current graph G_i (which is also called the *cluster graph* or *flow graph* for reasons that will be clear below). Check for algebraic dependencies that could cause a dof misclassification.

2. *Simplify* C into $T(C)$, transforming G_i into the next cluster graph $G_{i+1} = T(G_i)$ (the recombination step).

4.1.1 Isolating Clusters

The isolation algorithm, first given in [35, 36] is a modified incremental network maximum flow algorithm. The key routine is the *distribution* of an edge (see the DR-planner pseudocode in the Appendix of Part II) in the constraint graph G. For each edge, we try to *distribute* the weight $w(e) + D + 1$ to one or both of its endpoints as *flow* without exceeding their weights, referred to as "distributing the edge e." See *DistributeEdge* in the pseudocode in Part II, Appendix. This is best illustrated on a corresponding bipartite graph G^*: vertices in one of its parts represent edges in G and vertices in the second part represent vertices in G; edges in G^* represent incidence in G. As illustrated by Figure 14, we may need to redistribute (find an augmenting path).

If we are able to distribute all edges, then the graph is not dense. If no dense subgraph exists, then the flow based algorithm will terminate in $O(n(m + n))$ steps and announce this fact. If there is a dense subgraph, then there is an edge whose weight plus $D + 1$ cannot be distributed (edges are distributed in some order, for example by considering vertices in some order and distributing all edges connecting a new vertex to all the vertices considered so far). It can be shown that the search for the augmenting path while distributing this edge marks the required dense graph. It can also be shown that if the found subgraph is not overconstrained, then it is in fact minimal. If it is overconstrained, [35, 36] give an efficient algorithm to find a minimal cluster inside it.

4.1.2 Cluster Simplification

This simplification was given in [39, 37]. The found cluster C interacts with the rest of the constraint graph through its *frontier vertices*; i.e., the vertices of the cluster that are adjacent to vertices not in the cluster. The

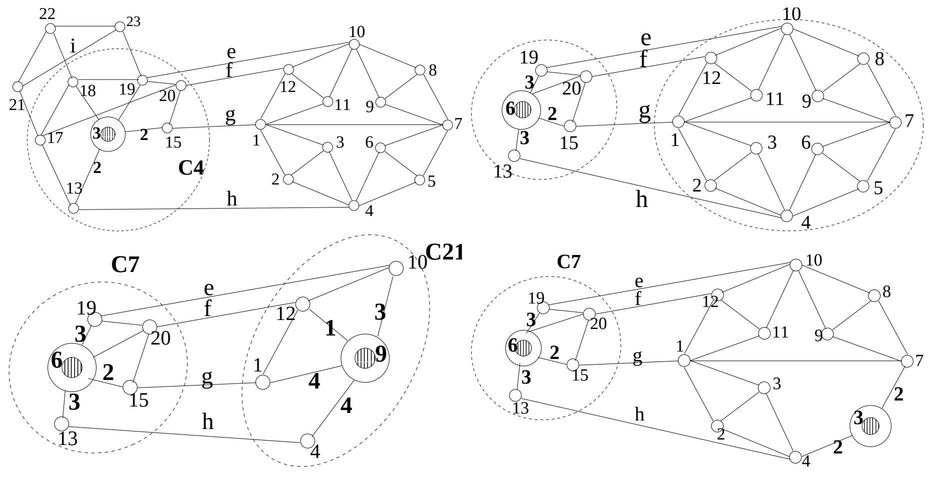


Figure 15: From left: FA's simplification of graph givin DR-plan in Figure 10; clusters are simplified in their numbered order: C4 is simplified before C7 etc.

vertices of C that are not frontier, called the *internal vertices*, are contracted into a single *core* vertex. This core is connected to each frontier vertex v of the simplified cluster $T(C)$ by an edge whose weight is the the sum of the weights of the original edges connecting internal vertices to v. Here, the weights of the frontier vertices and of the edges connecting them remain unchanged. The weight of the core vertex is chosen so that the density of the simplified cluster is $-D$, where D is the geometry-dependent constant. This is important for proving many properties of the FA DR-plan: even if C is overconstrained, $T(C)$'s overall weight is that of a wellconstrained graph, (unless C is rotationally symmetric - see Part II, Section 3, for how this important exception is treated). Technically, $T(C)$ may not be wellconstrained in the precise sense: it may contain an overconstrained subgraph consisting only of frontier vertices and edges, but its overall dof count is that of a wellconstrained graph.

Figure 15 illustrates this iterative simplification process ending in the final DR-plan of Figure 10.

4.2 The Frontier Vertex Algorithm (FA DR-planner)

The challenge met by FA is that it provably meets several competing requirements including those of Section 3 and Section 2. The graph transformation performed by the FA cluster simplification is described formally in [39, 37] that provide the vocabulary for proving certain properties of FA that follow directly from this simplification. However, other properties of FA require details of the actual DR-planner that ensures them, and are briefly sketched here.

Note: A detailed pseudocode of the FA DR-planner (the existing version, explained in [53], incorporating the new algorithms developed in Part II of this manuscript, can be found in the Appendix of Part II).

The basic FA algorithm is based on an extension of the *distribute* routine for edges (explained above) to vertices and clusters in order for the isolation algorithm to work at an arbitrary stage of the planning process, i.e, in the cluster or flow graph G_i.

First, we briefly describe this basic algorithm. Next, we emphasize the parts of the algorithm that ensure 4 crucial properties of the output DR-plan:
(a) dealing with combinatorially overconstrained subgraphs in order to obtain correct DR-plans (in a well-defined sense) for wellconstrained graphs, as well as
(b) for underconstrained graphs - which requires outputing a complete set of maximal clusters as sources of the DR-plan;
(c) controlling width of the DR-plan to ensure a polynomial time algorithm; and
(d) ensuring cluster-minimality of clusters in the DR-plan.

Three datastructures (elaborated in Section 6) are maintained. The current *flow or cluster graph*, G_i the current *DR-plan* (this information is stored entirely in the hierarchical structure of clusters at the top level of the DR-plan), and a *cluster queue*, which is the top-level clusters of the DR-plan that have not been distributed so far, in the order that they were found (see below for an explanation of how clusters are distributed). We start with the original graph (which serves as the cluster or flow graph initially, where the clusters are singleton vertices). The DR-plan consists of the leaf or sink nodes which are all the vertices. The cluster queue consists of all the vertices in an arbitrary order.

The method *DistributeVertex* (see pseudocode of Part II, Appendix) distributes all edges (calls DistributeEdge) connecting the current vertex to all the vertices considered so far. When one of the edges cannot be distributed and a minimal dense cluster C is discovered, its simplification $T(C)$ (described above) transforms the flow graph. The flows on the internal edges and the core vertex are inherited from the old flows on the internal edges and internal vertices. Notice that undistributed weights on the internal edges simply disappear. The undistributed weights on the frontier edges are distributed (within the cluster) as well as possible. However, undistributed weights on the frontier edges (edges between frontier vertices) may still remain if the frontier portion of the cluster is severely overconstrained. These have to be dealt with carefully. (See discussion on dealing with the problems caused by overconstraints below.) The new cluster is introduced into the DR-plan and the cluster queue.

Now we describe the method *DistributeCluster* (see pseudocode of Part II, Appendix). Assume all the vertices in the cluster queue have been distributed (either they were included in a higher level cluster in the DR-plan, or they failed to cause the formation of a cluster and continue to be a top level node of the DR-plan, but have disappeared from the cluster queue). Assume further that the DR-plan is not complete, i.e., its top level clusters are not maximal. The next level of clusters are found by distributing the clusters curently in the cluster queue. This is done by filling up the “holes” or the available degrees of freedom of a cluster C being distributed by D units of flow. The *PushOutSide* successively considers each edge incident on the cluster with 1 endpoint outside the cluster. It distributes any undistributed weight on these edges + 1 extra weight unit on each of these edges. It can be shown that if C is contained inside a larger cluster, then atleast one such cluster will be found by this method once all the clusters currently in the cluster queue have been distributed. The new cluster found is simplified to give a new flow graph, and gets added in the cluster queue, and the DR-plan as described above.

Eventually, when the cluster queue is empty, i.e, all found clusters have been distributed, the DR-plan's top level clusters are guaranteed to be be maximal. See [53] for formal proofs.

Note: Throughout, in the interest of formal clarity, we leave out ad hoc, but highly effective heuristics that find simple clusters by avoiding full-fledged flow. One such example is called “sequential extensions” which automatically creates a larger cluster containing a cluster C and a vertex v provided there are atleast D edges between C and v. These can easily be incorporated into the flow based algorithm, provided certain basic invariants about distributed edges is maintained (see below).

This completes the description of the backbone of the basic FA DR-planner. Next we consider some details ensuring the properties (a) – (d) above.

4.2.1 (a) Problems caused by overconstrained subgraphs and trivial intersections

In Figure 16, after C_1 and C_2 are found, when C_1 is distributed, C_1 and C_2 would be picked up as a cluster, although they do not form a cluster. The problem is that the overconstrained subgraph W intersects C_1 on a trivial cluster, and W itself has not been found. Had W been found before C_1 was distributed, W would have been simplified into a wellconstrained subgraph and this misclassification would not have occurred.

It has been shown in [53] that this misclassification can be avoided (W can be forced to be found after C_2 is found), by maintaining the following invariant: always distribute all undistributed edges connecting a new found cluster C (or the last distributed vertex that caused C to be found), to all the vertices distributed so far that are outside the cluster C. Undistributed weight on edges inside C are less crucial: if they become internal edges of the cluster, then this undistributed weight “disappears” when C is simplified into a wellconstrained cluster; there is also a simple method of treating undistributed weight on frontier edges so that they also do not cause problems - the method and proof can be found in [53]).

4.2.2 (b) Finding a complete set of maximal clusters in underconstrained graphs

While the DR-planner described so far guarantees that at termination, top level clusters of the DR-plan are maximal. It also guarantees that the original graph is underconstrained only if there is more than one top level cluster in the DR-plan. However, in order to guarantee that *all* the maximal clusters of an underconstrained

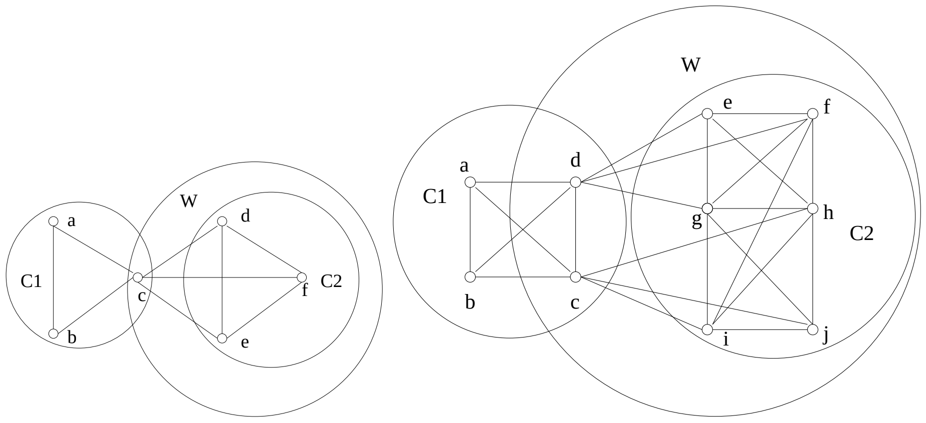


Figure 16: Finding W first will prevent misclassification: Left 2D example, Right 3D example.

graph appear as top level clusters of the DR-plan, we use the observation that any pair of such clusters intersect on a subgraph that reduces (once incidence constraints are resolved) into a trivial subgraph (a single point in 2D or a single edge in 3D). This bounds the total number of such clusters and gives a simple method for finding all of them. Once the DR-planner terminates with a set of maximal clusters, other maximal clusters are found by simply performing a *Pushoutside* of 2 units on every vertex (in 2D) or every vertex and edge (in 3D), and continuing with the original DR-planning process until it terminates with a larger set of maximal clusters. This is performed for each vertex in 2D and each edge in 3D which guarantees that all maximal clusters will be found. See [53] for proofs.

4.2.3 (c) Controlling width of the DR-plan

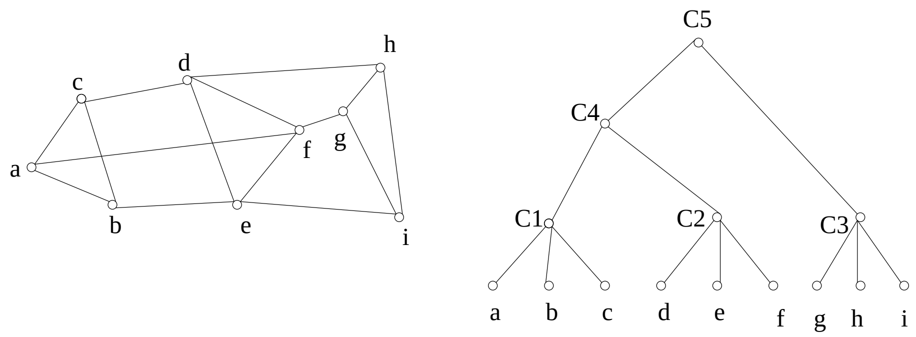


Figure 17: Prevent accumulation of clusters

FA achieves a linear bound on DR-plan width by maintaining the following invariant of the cluster or flow graph: *every pair of clusters in the flow graph (top level of the DR-plan) at any stage intersect on at most a rotationally symmetric subgraph.* FA does this by repeatedly performing 2 operations each time a new potential cluster is isolated.

The first is an *enlargement* of the found cluster. In general, a new found cluster N is enlarged by any cluster D_1 currently in the flow graph, if their nonempty intersection is *not* a rotationally symmetric or trivial subgraph. In this case, N neither enters the cluster graph nor the DR-Plan. Only $N \cup D_1$ enters the DR-plan, as a parent of both D_1 and the other children of N. It is easy to see that the *sizes* of the subsystems corresponding to both $N \cup D_1$ and N are the same, since D_1 would already be solved.

For the example in Figure 17, when the DR-plan finds the cluster C_2 after C_1, the DR-planner will find that C_1 can be enlarged by C_2 The DR-planner forms a new cluster C_4 based on C_1 and C_2 and puts C_4 into the cluster queue, instead of putting C_2 to cluster queue. (Refer to the pseudocode in PartII, Appendix : *AddChild, CanCombine.*)

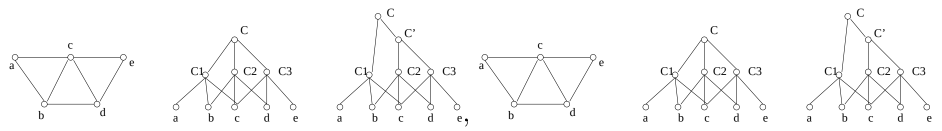


Figure 18: Ensuring Cluster Minimality: E is a set of essential clusters that must be present in any subset of the children of C that form a cluster. In this case, E itself forms a cluster. C' is a cluster made up of a proper subset of at least 2 of C's children

The second operation is to iteratively *combine* $N \cup D_1$ with any clusters $D_2, D_3, \ldots$ based on a nonempty overlap that is not rotationally symmetric or trivial. In this case, $N \cup D_1 \cup D_2$, $N \cup D_1 \cup D_2 \cup D_3$ etc. enter the DR-plan as a staircase, or chain, but only the single cluster $N \cup D_1 \cup D_2 \cup D_3 \cup \ldots$.. enters the cluster graph after removing D_1, D_2, D_3 ....

Ofcourse, both of these processes are distinct from the original flow distribution process that *locates* clusters. Refer to the DR-planner pseudocodes: *addChild, canCombine* in Part II, Appendix.

4.2.4 Cluster Minimality

Recall from Section 2 that a desirable property of DR-plans is that for any cluster in the DR-plan, no proper subset of atleast 2 of its child clusters forms a cluster. This property is crucial for correcting combinatorial overconstraints (see Section 4.3 and [26]) and dof misclassifications in pure combinatorial dof analysis, arising from algebraic overconstraints (see Part II, Section 3). FA ensures this using a generalization of the method *Minimal* of [35, 36] which finds a minimal dense subgraph inside a dense subgraph located by *DistributeVertex* and *DistributeEdge*.

Once a cluster C is located and has children $C_1, \ldots, C_k$, for $k \geq 2$, the recursive method *clusMin* (see pseudocode in Part II, Appendix) removes one cluster C_i at a time (replacing earlier removals) from C and redoes the flow inside the flow graph restricted to C, before C's simplification. If a proper subset of atleast 2 C_j's forms a cluster C', then the clusMin algorithm is repeated inside C' and thereafter in C again, replacing the set of child clusters of C that are inside C' by a single child cluster C'. If instead no such cluster is found, then the removed cluster C_i the *essential.* I.e., it belongs to every subset of C's children that forms a cluster. When the set of clusters itself forms a cluster E (using a dof count), then clusMin is called on C again with a new child cluster E replacing all of C's children inside E.

4.3 Correcting Combinatorial Overconstraints

A method for dealing with combinatorial overconstraints as per the output requirement in Section 3 - was given recently in [26]. It is based on showing that there is a unique, well-defined set of *reducible* constraints that can be deleted without making the graph underconstrained. See Figure 19. (Detecting and correcting algebraic overconstraints of Section 2 is a more difficult problem and is discussed in Part II, Section 3.)

The combinatorial overconstraint correction method is further based on key properties of the FA-DR-plan and gives an efficient (generally linear time) solution that retains full generality and moreover:
(i) Traverses the given DR-plan top down to incrementally output this unique set of constraints in reverse solving order, minimizing the need to solve again previously solved portions of the DR-plan;
(ii) Selects constraints from those parts of the constraint system that the user identifies;
(iii) Isolates information that can be routinely stored and maintained as part of the DR-plan, making the above process even more efficient; and
(iv) Automatically updates (see Figure 19) the DR plan with minimal reorganization, once one of the reducible constraints is removed, which may cause many lower level clusters of the DR plan to become underconstrained.

This method is described in [26] and is part of the FRONTIER solver [65]. This method is deceptively simple but makes sophisticated use of the FA DR-plan and hence requires an intricate proof of correctness.

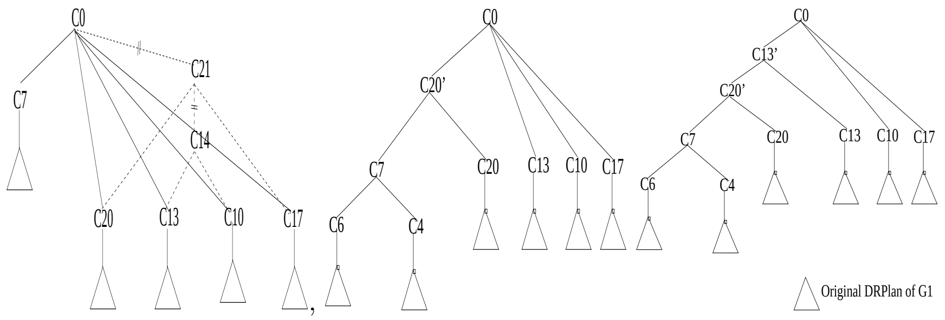


Figure 19: Left: unoptimized DR-plan of $G1$ in Figure 10, after reduction of edge $(1, 7)$; modification path is marked; Middle and Right: two optimization steps

5 Problem List

The properties of the Frontier vertex algorithm FA discussed in Section 4 are incorporated into FRONTIER. Here we present a further list of concrete technical problems whose solutions represent the *new* contributions of this 2-part manuscript. The first two are implementational problems discussed in the next 2 sections of Part I, and the remainder are problems that require algorithmic solutions and are discussed in Part II.

- Problem 1: Designing the architecture for the implementation of a full-fledged, portable geometric constraint solver based on FA. These are discussed in Part I, Section 6. Crucial data structures form the basis for a constraint representation language, and are designed to facilitate the seamless communication between the individual components of the geometric constraint solver, as well as the portability of the individual components. This directly addresses Requirement 5 of the Introduction, and plays a role in all of the other requirements as well.
- Problem 2: User interface that permits easy editability of the constraint, feature, part and subassembly repertoire, and supports the high degree of flexible and online user feedback and interaction of Problem 5. This addresses Requirement 4(f) of the Introduction and is discussed in Part I, Section 7.
- Problem 3: Augmenting FA beyond [39, 53], specifically to deal with (multiple) input feature, part or subassembly decompositions that result in a feature DAG. This automatically allows priority assignments in the feature DAG, thereby incorporating, for example, assembly order, triangle-based decomposition as well as parametric constraint solving as special cases. This automatically also allows to paste in features and already solved parts of a sketch, i.e, delineate parts of the input constraint system as already solved. This augmentation of FA addresses Requirement 2 of the Introduction and is discussed in Part II, Section 2.
- Problem 4: Dealing with the inadequacy of combinatorial dof analysis in 3D, specifically with simple algebraic dependencies between constraints that cannot be combinatorially detected by a degree of freedom analysis – one example concerns rotationally symmetric clusters and other motifs that occur commonly and are crucial for building larger clusters, but are responsible for a large class of dof analysis misclassifications, although detecting them is highly nontrivial. Again, since these modifications do not immediately preserve other crucial properties of FA, they result in further modifications to restore those properties. This addresses Requirement 1 of the Introduction and is discussed in Part II, Section 3.
- Problem 5: Conceptually meaningful navigation or steering through the solution space, recursively navigating the solution spaces of features, subassemblies in the input partial decompositions as well as other clusters in the DR-plan, with fully flexible backtracking. Incorporating resolution methods for relational, engineering and other navigation constraints specific inequality constraints and chirality checks. This addresses the Requirement 3 of the Introduction, is a key achievement of FRONTIER's Equation and Solution Manager (ESM) and is discussed Part II, Section 4.
- Problem 6: Formulating stable, independent algebraic equations from constraints for the resolution of individual clusters of the DR-plan. (These are then sent to the canned algebraic-numeric solver). The

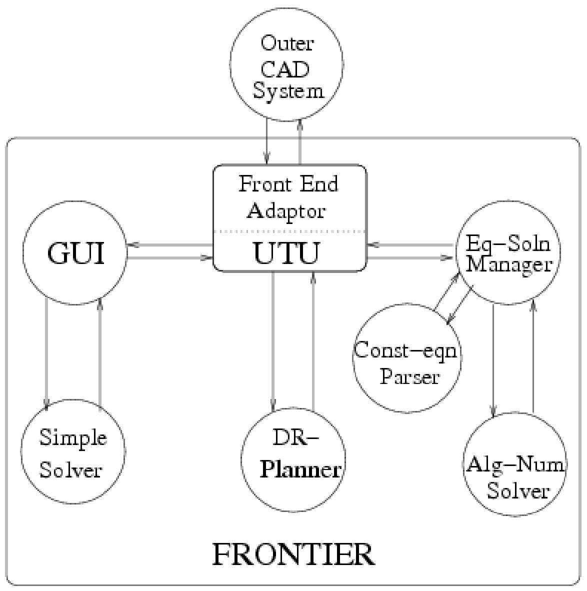


Figure 20: Organization of FRONTIER

formulation of a stable set of equations requires nontrivial checks for algebraic dependencies in 3D which is compounded by the presence of shared objects and incidence constraints between the child clusters that are to be resolved into a parent cluster. This is facilitated by maintaining an editable constraint-to-equation parse tree, which in addition permits step-by-step inspection and pre-processing as well as post-processing of the simultaneous polynomial system to be sent to the canned algebraic-numeric solver. These are other achievements of of FRONTIER's ESM, discussed in Part II, Section 5; and addresses Requirements 1 and 4(d,e) of the Introduction.

- Problem 7: Detecting and dealing with underconstraints (overconstraints were dealt with in [53] and the method is sketched in Section 4.3, and incorporated into FRONTIER). In addition, efficient user *updates* are handled by FRONTIER's update mode – discussed in Part II, Section 6. However, efficient updates are achieved through functionalities offered by FRONTIER's datastructures, its DR-planner, its Equation and Solution Manager, and its Universal Transfer Unit (UTU). This addresses Requirements 4(a,b) of the Introduction.

- Problem 8: Incorporating simple methods for (online) resolution of certain types of underconstrained sketches – this is achieved by FRONTIER's Simplesolver. This is discussed in Part II, Section 7. This addresses some aspects of Requirement 4(c) of the Introduction.

6 Solution to Problem 1: Organization and Dataflow

The FRONTIER system is composed of six modules:

- The GUI (Section 7), sometimes just refered to as Sketcher, is written in Java, Java 3D is the main driver for FRONTIER and is used for all input and output. The GUI is used to input: the constraint system and a conceptual decomposition or feature hierarchy, (these could have been partially solved and stored from an earlier session), any updates to these, and interactive user input during the constraint parsing and equation building stages and solution space navigation. The GUI is also used to modify the object, contraint and feature repertoire or constraint-to-equation parse tree. The GUI is used to output: the DR-plan of the input constraint system consistent with the input conceptual decomposition, the best possible online resolved display of it using the Simplesolver; (Part II, Section 7) the system of equations as they are being parsed in stages from the constraints, the subsystem solutions corresponding to the nodes of the DR-plan, the partially solved system as the navigation proceeds, and the final solution.
- The universal transfer unit (Part II, Section 6, written in C++) coordinates communication between the various modules and makes decisions about which modules are called for various update operations. It additionally manages the translations required by FRONTIER's modules, for example, it converts the geometric constraint system input obtained from the Sketcher or GUI to a dof graph input for the DR-planner. See description of dataflow below.
- The Simplesolver (Part II, Section 7, written in Java) that is used for a rough online resolution of the input sketch for display purposes.
- The DR-planner (Section 4 and Part II, Sections 2, 3, *pseudocode in Appendix*; written in C++) obtains/updates the DR-plan that is used to guide the solving.
- The Equation and Solution Manager (ESM) (Part II, Sections 4,5, *pseudocode in Appendix* written in C++) obtains/updates subsystems corresponding to the nodes of the DR-plan to the Algebraic-Numeric Solver to obtain all the possible solutions (given a fixed choice of solutions for the child subsystems), and manages the user's navigation of the solution space by appropriately storing the solutions, as well as traversing and backtracking on the DR-plan. (The Algebraic-Numeric Solver is off-the-shelf and not discussed in this manuscript).
- The *Constraint to Equation Parser (CEP)* (Part II, Section 5, written in C++) is a submodule of the ESM used to build the subsystems corresponding to the nodes of the DR-plan, with input from the user. It also allows the user to change the constraint-to-equation parse tree (to improve efficiency or when new constraint or object types are added).

Next, we discuss the dataflow between these modules. See Figure 20. To do so, we first explain the main datastructure, organized based on the DR-plan structure (Sections 2 and 4), that is shared seamlessly by all the modules. The datastructure is physically shared by the C++ modules, and is communicated, through the UTU, to the Java modules using a Java Native Interface (JNI arrays); however the same datastructure is conceptually mirrored in the object oriented hierarchies used by the Java modules. See Section 7.

A secondary, related datastructure organizes the constraints in a hierarchy for efficient parsing into equations. This datastructure is also conceptually mirrored by the constraint hierarchy used by the Sketcher (see Section 7). The datastructure is called the constraint-to-equation parse tree and is used by the CEP while building stable systems of equations to send to the algebraic-numeric solver.

These datastructures together address Problem 1 of the list in Section 5. The details of the secondary CEP datastructure are discussed in Part II, Section 5; however the dataflow between the CEP and the other modules is discussed here.

6.1 Primary Datastructures

These are based on the FA DR-plan described in Section 4.

6.1.1 Original Graph and Flow Graph or Cluster Graph

The graph datastructure is simply contains two lists and two parameters. The parameters, the depth of the graph and the dimension of the graph, store the depth of the largest cluster (distance to leaf or sink in the DR-plan) and the dimension of the objects in the graph represents respectively. The two lists are a list of vertices (geometric objects) and a list of edges (constraints). The structures of the vertex and edge objects are the bulk of the graph datastructure. The graph structure along with a *Cluster queue* of current undistributed clusters in the graph (See Section 4 and the top level clusters in the DR-plan (the Cluster structure and DR-plan are described below) constitute the conceptual Cluster or Flow graph.

An *edge* object represents a single constraint within the geometric system that is being processed. The fields of the edge object can be divided into two groups, the descriptive parameters and the control parameters. The descriptive parameters group stores the various parameters that describe each particular constraint. They include a constraint ID, Type, the degree of freedom weight of the constraint, and the ID's of the endpoint vertices.

The control parameters group stores parameters that are used to control the process of flow distribution. See Section 4. First there are two flags, scan and label. These flags are used to mark those edges that should be tested as part of the current augmenting path search and those objects that are part of the currently found cluster respectively. Next, there are the three fields that store information about the flows on the edge: a left flow field and a right flow field, which record the flows onto the left and right endpoints of this edge; and the possible flow field that records the current amount of flow left to distribute for this edge. Finally, there is a predecessor vertex field. This field stores the ID of the vertex that was processed to bring this edge into the augmenting path. Along with a complementary field, predecessor edge, in the vertex object. This field allows the current augmenting path to be searched as a linked list.

The *vertex* object stores information about the corresponding geometric object and like the edge object, has several internal fields that can be divided into descriptive and control groups. The descriptive group contains several simple fields, the ID of the object, the integer Type of the object, the weight (number of degrees of freedom) of the object, and a list of the edge ID's adjacent to this vertex. Additional fields include a list of flags one for each degree of freedom indicating whether each degree of freedom is free. This allows the user or the ESM to fix some of the degrees of freedom and solve for the remainder.

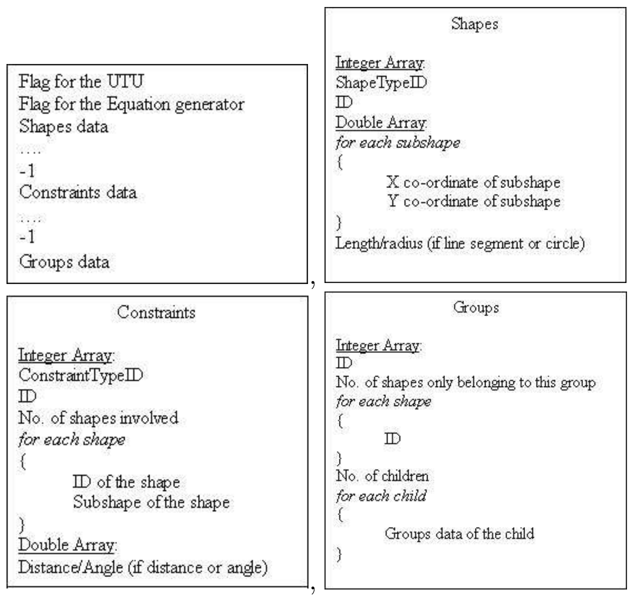
The control group is a bit more complicated for the vertex object. First it contains identical label and scan flags to indicate vertices in the augmenting path found during the edge distribution (see Section 4), and vertices in the current cluster respectively. Second it contains a frozen field. This boolean field is used in the presence of input conceptual decompositions See Part II, Section 2. When a vertex is marked as frozen it is not processed during distribution. Freezing groups of vertices is crucial to finding a DR-plan that incorporates a user defined input feature hierarchy. Finally, there are possible flow and existing flow fields. The possible flow is a count of the amount of flow that this vertex can still accomodate. I.e., the number of degrees of freedom of the vertex that have not yet been appropriated by flows on incident edges. The existing flow is the amount of flow that this vertex already contains.

6.1.2 Cluster

A cluster object represents the frontier vertex simplification of a wellconstrained or a well-overconstrained subgraph, as well as its original subgraph. See Section 4. The key to this representation is the division of the vertices within the subgraph into interior and frontier vertices. Interior vertices are vertices within the subgraph, which have no adjacent edge that is not part of the subgraph. The remainder of the vertices in the subgraph are frontier vertices. Within the cluster object, the frontier vertices are stored in a list. However, the interior vertices and all of the edges between them are converted into a single new vertex, the *core* of the cluster. This new vertex object is also stored in the cluster. These cluster objects are hierarchical and form a directed acyclic graph (DAG), i.e, their sub-DR-plan. The DAG interrelationships are stored within each cluster as a list of its immediate children.

Next, the cluster contains more descriptive fields, a group ID corresponding to a feature, if any, in the input feature hierarchy, that this cluster corresponds to (note that not every feature in the input feature hierarchy is a cluster; however, every cluster feature must appear in the output DR-plan – see Part II, Section 2). Another important descriptive field is the cluster type that indicates whether this cluster is wellconstrained, rotationally symmetric, well-overconstrained (combinatorially), algebraically overconstrained etc. (see Part II, Section 3 for the algorithmic importance of this field); a list of original vertices that constitute this cluster; and two lists of inner edges and frontier edges that store the edges between the frontier vertices and the core and the edges between the frontier vertices respectively.

Next, the cluster object has several fields that store the solved degrees of freedom of the cluster. See Part


```
Flag for the UTU
Flag for the Equation generator
Shapes data
....
-1
Constraints data
....
-1
Groups data
```
,

```
Shapes

Integer Array:
ShapeTypeID
ID
Double Array:
for each subshape
{
        X co-ordinate of subshape
        Y co-ordinate of subshape
}
Length/radius (if line segment or circle)
```

```
Constraints

Integer Array:
ConstraintTypeID
ID
No. of shapes involved
for each shape
{
        ID of the shape
        Subshape of the shape
}
Double Array:
Distance/Angle (if distance or angle)
```
,

```
Groups

Integer Array:
ID
No. of shapes only belonging to this group
for each shape
{
        ID
}
No. of children
for each child
{
        Groups data of the child
}
```

Figure 21: JNI communication of Shapes, Constraints, Feature hierarchy data.

II, Section 4 for algorithmic details. A list of strings is stored, one for each solution of the cluster returned from the algebraic-numeric solver (given fixed chosen solutons to each of the child clusters). When the user selects a desired solution, that string is parsed into a list of actual degree of freedom values for this cluster. Multiple copies of the list are permitted, one for each parent of the cluster in the DR-plan, if they exist. The final field is a flag that indicates whether this cluster has been solved or not.

The *DR-Plan* object is simply a pointer to the root node of a list of clusters as they are described above. In case the graph is underconstrained, the root node is a dummy node, whose children are the actual sources of the DR-plan, i.e, a complete set of maximal clusters. See Section 4.

6.2 Communication between the Modules

The Sketcher is used for all user input and for all output displays. It shares its Java datastructures directly with the Simplesolver, also written in Java, which performs a rough online resolution (to the extent of its capability) of the input constraint system as it is being sketched (see Part II, Section 7). The backend C++ modules are used for more detailed and accurate solution using constraint system decomposition (DR-planner) feature solution space navigation (ESM) etc. All information passed between the Sketcher and the backend modules (C++) proceeds via the UTU. It is passed through a JNI interface of three arrays, one of integers, one of doubles and one of characters. The sketcher writes information into these three arrays, and natively calls the C++ coded UTU with these three arrays as parameters. The UTU parses these arrays and from them creates (or restores, see below) the graph and DR-Plan data structures described above - that are used by both the ESM and the DR-Planner to process the user's request. The UTU then directs the both the DR-Planner and the ESM to accomplish the specific request made by the user. These requests are differentiated by the use of an integer flag that is passed as the first integer in the integer array.

While the DR-Planner never needs to initiate communication with the user, the ESM must continually interact with the user (through the UTU) to select various subsytem solutions and to direct the path towards final solution. See Part II, Section 4. The communication between the ESM and the Sketcher is accomplished through the same three arrays described above. While there are numerous types of information the ESM can pass back to the Sketcher through the UTU, their process is always the same. First, the UTU must translate the current state of the DR-Plan and graph data structures into the three communication arrays. Then all additional communication information is written to the three arrays. The Sketcher reads only this communication information, but stores the residual data structure information while it processes the communication information. The residual data structure information is passed back to the UTU, if necessary, during the next request, and using it, the UTU can rebuild the current state of the solution process and the ESM can continue where it left off. The detailed actions of the Sketcher, UTU, and the ESM in each mode of communication are outlined in the next sections.

6.2.1 Communication during the Generate Mode

The Generate mode of communication begins when the Sketcher has a new graph to send to the DR-Planner and the ESM. In this case, there is no residual data structure information to return the ESM; any such information that the Sketcher is storing is discarded. The Sketcher sets the request flag to signal a new input graph, and then the writes a list of the current objects and their parameters into the three communication arrays. Following this is a list of all the current constraints and parameters and a list of all the user-defined groups. The Sketcher then calls the UTU. The UTU parses these lists, creates the input graph, and calls the DR-Planner with them as parameters for the *planning phase*. The completed DR-Plan is returned to the UTU, which then sends it to the ESM for the *solving phase*.

The ESM takes the DR-Plan, parses it into the communication arrays and returns it to the Sketcher (following the procedure described above). The Sketcher displays the DR-Plan to the user and allows the user to approve of the DR-Plan before solution begins. If the plan is not approved, the Sketcher enters the *Update Mode*, and interactive editing of the graph is permitted (see below and Part II, Section 6 for details of the underlying algorithms). However, if the plan is approved, the Sketcher simply passes the request flag, set to the continue option, back to the UTU along with a solving option flag that indicates whether the user chooses to navigate the solution space (*Navigate option*), or whether the ESM should automatically solve the system (*Autosolve option*).

6.2.2 Communication during the Navigate option

When the UTU receives the continue flag, the UTU simply sends the reconstructed graph and DR-Plan directly to the ESM. When the ESM receives the continue flag, it begins the solution-navigation process and a *visual walk-through* or steering through the solution space (See Figures in Section 3, and Part II, Section 4 also for underlying algorithmic details). The ESM traverses the DR-plan bottom up, communicating with the Constraint-to-Equation Parser (CEP) to build a stable system of equations (See Part II, Section 5 for details – this process also permits interaction with the user described separately in a subsection below), and communicating with the algebraic-numeric solver. This continues until the ESM reaches the point where it has several subsystem solutions to offer to the user. At this point, the user must select one of several possible realizations or conformations that the ESM will use to create the final solution. This information must be passed from the ESM back to the Sketcher for display. The ESM sends back two lists of information via the UTU. First, for each solution possibility, the ESM writes out a list of the objects in the subsystem. For each object, the ESM writes the object's ID, and a list of its solved degrees of freedom. Second, the ESM writes a list of the clusters contained in the current DR-Plan and a boolean flag to indicate whether they have been solved or not. This information allows the user to consider the current position of the solution in the DR-Plan when making his/her choice of a solution. As a final step, the ESM sets a flag to indicate that this information is a new solution possibility and returns.

The Sketcher displays each solution possibility or conformation to the user in a separate window. The original sketch and a *partially solved sketch*, incorporating the subsystem solutions chosen so far into the unsolved original sketch are also displayed and allows the user to make a choice of several options described below. (See Figures in Section 3, and Part II, Section 4.

1. Select a solution and continue solution - In this case, the Sketcher simply sets the ESM communication flag to indicate a solution choice and sends the UTU the number of the chosen solution. The ESM uses this to updates the DR-Plan to contain the object positions contained in the chosen solution and continues the solution navigation process. Assuming that the user always selects a solution, this process of solving, solution selection, and DR-Plan update would continue until all the subsystems in the DR-Plan have been solved.

2. Choose to backtrack - If the user decides that none of the current solution choices are to their liking, they may chose to edit his earlier selections. For editing, the user has two options: resolve the current cluster with new choices for its children's solutions, or repeat the solving of some previous cluster after making new choices for the solutions of its children. In either case, the Sketcher sends a flag back to the UTU that indicates that backtracking is to occur and an ID of the cluster from which the users wishes to continue solution. The UTU edits the DR-Plan and resets the “solved” flags that indicate that that cluster is the next to be solved. The ESM will ignore any previous solutions if the fin flag has been unset, and when the UTU calls it with the altered “solved” list, it will continue solving from an earlier point exactly as described above. Note that the user can interactively update the graph (see below) until a final solution is returned.

3. Choose to update - This final option indicates that the user believes there is something wrong with the input graph itself, rather than the bifurcation choices that have been made earlier. Many minor modifications of the graph do not require the re-planning and solution of the system. These changes can be made interactively during solution and much of the original plans and solutions can be reused. The communications and dataflow for the update mode is described in its own section below, the algorithmic details of update operations are described in Part II, Section 6.

Of course, sometimes no solution will exist for a given cluster in the DR-Plan. In this case, the ESM will have no solution options to return to the user. In this instance, the ESM first decides what is the nature of the lack of solution, either zero solutions, infinite solutions (underconstrained), or erroneous solution (see Part II, Sections 4, 5 for descriptions of these problems caused by the presence of navigation and inequality constraints or unstable systems respectively). Then rather than a solution possibility, the ESM returns a no solution flag to the Sketcher, and an integer code to indicate the type of non-solution encountered. In this case, the user is presented with only options 2 and 3 above, which will allow him or her to edit the solution until a valid solution is found.

Assuming a solution to the current system exists and the user can select subsystem solutions to find it, all communication in the Generate Mode will end when all the root clusters in the DR-Plan have been solved. At this point, the ESM writes a flag to indicate that a final solution is to be returned. Then it returns a ordered list of all objects, their ID’s and the final positions of their degrees of freedom. The ESM ends and the solution process is completed.

6.2.3 Communication during the Autosolve option

If the user chose the autosolve as opposed to the navigate option, there is no interaction between the user and the ESM after the DR-Plan has been accepted. The process of auto-solving is the same as the navigate solution option (as described above, details in Part II, Section 4), except that when the ESM would request a bifurcation choice from the user, the ESM automatically chooses a selection. The ESM will search the entire space of bifurcations for the first set that returns a valid solution for each cluster in the DR-Plan. At the end of the auto-solution process the ESM either returns a final solution exactly as in the Generate Mode, or it returns a flag that indicates that no valid solution exists.

6.2.4 Communication during the Update Mode

The Update mode, as mentioned above, allows the user to interactively edit the input graph during the process of solution. The algorithmic details can be found in Part II, Section 6. Update mode allows five changes to be made to the graph during solution: change a constraint parameter, add a constraint, remove a constraint, add or delete an object, and add or delete a feature to the input feature hierarchy. The sketcher indicates which mode the user has selected with a flag and required data (for example, the ID of the object or constraint to be deleted) that it sends to the UTU. The UTU then updates the input graph and calls the DR-Planner and the ESM whichever is required, to correctly edit the graph and continue solution with the least amount of repeat solving. See Part II Section 6 for details on how UTU makes these decisions and the resulting operations performed by the DR-planner and ESM.

6.2.5 Communication between the Constraint-to-Equation Parser (CEP) and the GUI

The CEP (method and datastructure described in Part II, Section 5) permits the user to both set up the constraint-to-equation parse tree, and to interactively build subsystems of equations that the ESM sends to the algebraic-numeric solver. This communication also takes place via the UTU, and through a JNI array to the GUI. The GUI opens a text window for this communication.

The GUI options allow the user to set interactive/automatic mode for the CEP at any point during a session. In the interactive mode of the CEP, the user is again given the option of taking the Entry mode (where the user can input or modify the parse tree) or the Parse mode (in which case, the user is prompted during the solving phase, each time when a subsystem of equations is built by the ESM to send to the algebraic-numeric solver). In automatic mode of operation, the default choice is to shut off all user interaction with the CEP.

7 Solution to Problem 2: Design and Implementation of an Appropriate GUI

Specifications on the design of the GUI are imposed by the various functionalities of FRONTIER that are brought to the user via the GUI are described in Section 1 with figures in Section 3 (as well as in various sections of Part II). These functionalities occur during the various modes (generate and update), phases (planning and solving), solving options and settings (online and offline). Other specifications are imposed by the architecture of FRONTIER, in particular, the dataflow between the GUI (which is the main driver) and the other modules during the various modes, phases, solving options and settings - via the the constraint representation language and seamless datastructure used by the modules, are all described in Section 6.

Here, we concentrate on the internal design of the GUI that (i) permit the above specifications to be met (ii) ensure extensibility, robustness, resource efficiency and scalability. Hence we only discuss the *representation of geometric objects and constraints* for the purposes of input constraint system sketches, extending the input object and constraint repertoire, communication with back-end modules and output display.

We do not discuss other interesting user perception and sketch or gesture interpretation issues that arise during the constraint system sketching process, especially in 3D; similarly, we do not discuss other important display issues. For example, providing user friendly and intuitive, but efficient utilization of the canvas for display. This is needed while refreshing the display by a rough online resolution of the input system as it is being sketched (by the FRONTIER's Simplesolver discussed in Part II, Section 7), while displaying subsystem solutions on a separate conformation window during navigation (Part II, Section 4), while displaying a partially solved system, which incorporates solved parts into the original unsolved sketch, and while displaying the final solution. Other input/output and storage issues of the GUI are visible from FRONTIER screenshots in Sections 3 and in various sections of Part II. These are not discussed here, although their implementation poses some interesting problems such as the following. An intuitive mechanism for inputing design decompositions; output display of the DR-plan which involves a good graph drawing algorithm; modifying the object, constraint or feature repertoire; the related issue of storing partial or complete results of a session such as a sketch along with partial solution information such as the DR-plan and some of the subsytem solutions in a file (a feature repertory), that can be opened at a later session, as *part* of a new, larger constraint system; inputing and modifying constraint-to-equation parse tree to be used by the ESM; simplifying the modifications to the algebraic equations for a subsytem that is being constructed by the ESM, specifically by the constraint-to-equation parser (input); solution space navigation and backtracking by clicking various subsystems on the DR-plan or buttons to choose subsystem solutions; updating the constraint system or input decompositon (input), etc. All of these issues are however strongly influenced by the representation of the geometric objects and constraints which we discuss here.

The main features of the object and constraint representation are modularity and hierarchy within the objects and constraints. These properties are a direct effect of the clean and object-oriented nature of the code, which simplifies any future expansion and hence provides extensibility. Simultaneously, this GUI representation also reflects the internal geometric properties of these objects in a natural way, i.e., the manner in which they are represented and analyzed by the back-end modules (as subgraphs in the DR-planner, or subsystems in the ESM). This ensures a common representation and the seamless dataflow between the modules, not requiring awkward translations from one representation to another.

There are three different versions of Sketcher, the 2D Sketcher, the 2D-input 3D-output Sketcher, and the 3D Sketcher. We shall first discuss the implementation details of the 2D Sketcher. The 2D-input 3D-output Sketcher and the 3D Sketcher have a similar design, thus maintaining consistency in addition to the other features.

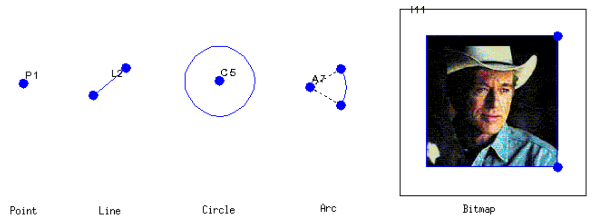


Figure 22: 2D Shapes or Objects

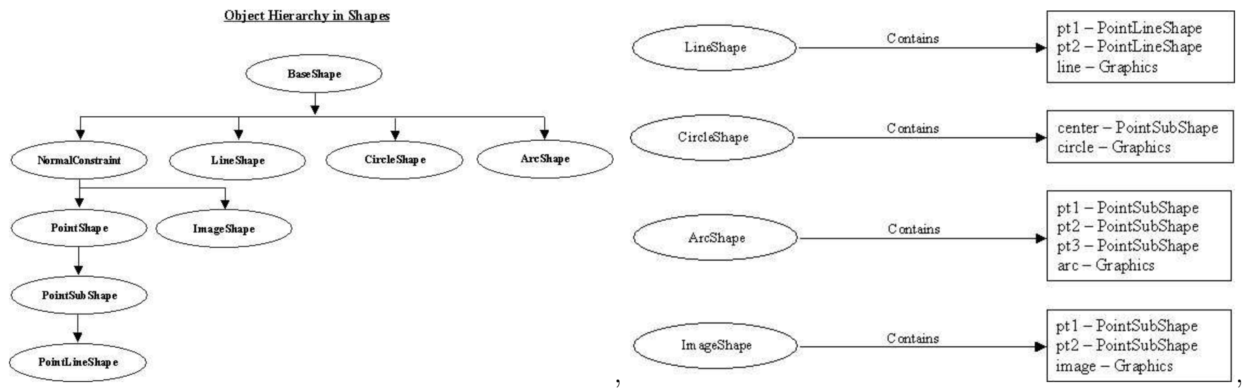


Figure 23: Hierarchy within 2D shapes or objects

7.1 2D Sketcher and Canvas

The 2D Sketcher is written in JAVA using JAVA AWT and Swing packages. The application can be divided (implementation-wise) into two major components the part that handles the display with all the associated components and the interface that communicates with the UTU. The 2D Sketcher has the following primitive geometric objects: points, lines, rays, line segments, circles and arcs and the bitmap “image” object, which represents an arbitrary rigid object that has already been solved and is perhaps represented in another representation language by the calling CAD module and could potentially just been converted to a bitmap, with understood dimensions. The constraints that can be applied to these objects are: distance, incidence, perpendicularity, parallel and tangency constraints. Each of the shapes has a unique ID, and similarly each of the constraints have a unique ID.

7.1.1 Representation of 2D Objects (Shapes) and Constraints

All the shapes descend from a baseShape class. The properties that are common to all the shapes like: Name: Name of the shape, ID: Unique ID of that particular instance, ShapeTypeID: Unique ID for that class of shape, Selected Flag that indicates whether the shape is selected or not, Color etc., are declared in this class. The Line, circle and the arc shapes are descendents of this class. The normalShape class is a descendent of the baseShape class. It also includes another property. The objects of this class have x, y coordinates defined by their position on the screen. These objects can be dragged on the screen using the mouse. The point and the “image” shapes are both descendants of this class. The following figures show the object hierarchy within the shapes and the various attributes of each shape.

Another key feature of the design is the pointSubShape class, which is a descendent of the pointShape class. All the other objects like the line, circle, arc and the image shapes have sub-shapes that are of type pointSubShape. For example the line segment has two sub-shapes that are its end points. In case of the line segments the sub-shapes are of type pointLineShape. The pointLineShape is a descendent of the pointSubShape, which can be set to infinity. The line and the ray are considered to be special cases of the line segment where both or one of the end points is set to infinity, respectively. The positions of the shapes that are not of type normalShape are defined by the positions of their sub-shapes that are of type normalShape.

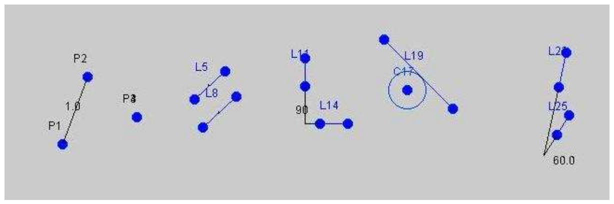


Figure 24: 2D Constraints

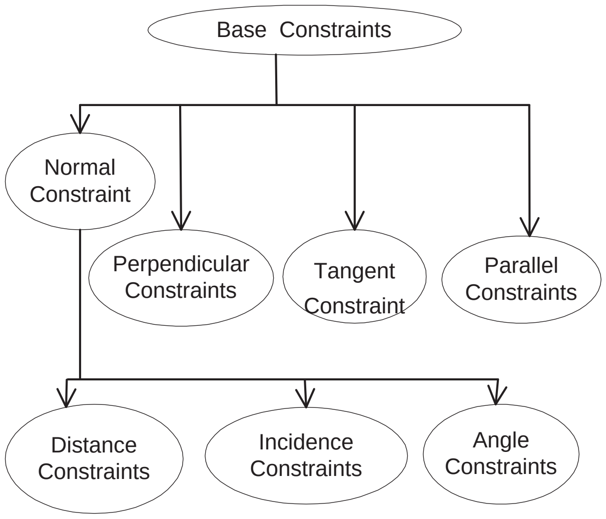


Figure 25: Constraint Hierarchy

This object-oriented design proves to be very advantageous for most of the functionalities of Sketcher (both display and communication with back-end modules). Good examples are the writeToStream/readFromStream methods that are used to save to file and read from one respectively. These methods in the baseShape class write/read the common properties and then call the writeAdditionalProps/readAdditionalProps method in the derived classes to write/read additional properties of the shapes. In each derived class (of shapes) these methods are overridden to write/read the specific properties of that particular shape and then calls the writeAdditionalProps/readAdditionalProps methods of the its sub-shapes, if any. For example in lineShape this method writes/reads the length of the line segment and then calls the writeAdditionalProps/readAdditionalProps methods of its end points. In the writeAdditionalProps/readAdditionalProps methods of the end points the coordinates of the points are written/read. Similarly in case of circle shape the radius is written/read and then the writeAdditionalProps/readAdditionalProps method of its center is called.

The constraints are also implemented in a similar manner. All constraints descend from a baseConstraint class. The metric constraints like distance and angle have a property associated with them that define their value while the logical constraints like tangency, parallelism, and perpendicularity do not. Each constraint maintains an array of shapes that are associated with that constraint.

For some of the constraints just specifying the shapes involved, is not enough. We also need to specify which part (sub-shape) of that shape is involved. For example if we specify an incidence constraint between a point and a line segment then we have to specify which end point of the line segment is incident with the point. Similar to the hierarchy in shapes there is normalConstraint class that descends from the baseConstraint class. All the descendents of the normalConstraint class have an additional property that specifies which sub-shape of each of the two shapes are involved in the constraint. Displaying the constraints using graphics, by itself is an interesting issue. The representation of the constraint needs to be intuitive enough for the user to be able to interpret it easily. The simplesolver module usually resolves the incidence, tangency constraints online for display (See Part II, Section 7), and these are not represented explicitly by any graphics. Again the object-

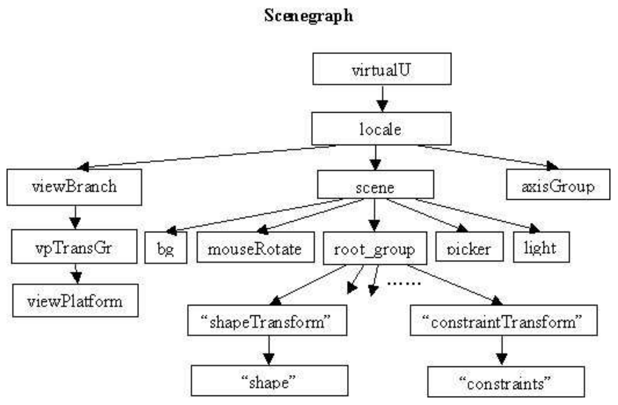


Figure 26: Partial scenegraph used to display 3D scenes in Sketcher

oriented design proves to be useful with the constraints, for the same reasons discussed earlier. For example the method drawConstraint is called every time the screen is repainted. And this method is overridden in all the constraints to display the specific graphics of that constraint.

7.2 2D-Input-3D-Output Sketcher and the 3D sketcher

The first version of Sketcher is capable of handling both 2D and 3D constraint systems. The user can choose to work in either the 2D mode or the 3D mode. In the 2D mode it functions exactly like the 2D version. In the 3D mode, the user is allowed to draw the 3D sketch on a 2D canvas and solve it in the 3D mode. However the subsystem solution choices, partially solved sketch and the final sketch are displayed on the 3D canvas. The solver does not use the input coordinates of the objects to calculate the solution (relative orientations of the points placed on the canvas may be used as chirality navigation constraints during solving, but the actual coordinate values of points on the canvas are not used). So the Sketcher simply sends a 0 as the z coordinate for all the objects. The other modules treat the sketch as a 3D problem and solve for all the 3 dimensions. When the subsystem solution choices are displayed all the objects have 3 meaningful dimensions so they are displayed on a 3D canvas. And the final output is also displayed on a 3D canvas.

The implementation of the 2D parts of this Sketcher is similar to that of the 2D Sketcher. The only difference is that every time a 2D shape or constraint is created a corresponding 3D shape or constraint is created and attached to the scenegraph, which may or may not be used in future. This is done to allow the user to switch between the two modes easily at any point. The implementation of the 3D objects (Shapes and Constraints) is the same in both the 2D-input 3D-output version and the full 3D version. These objects are implemented using JAVA3D (heavy weight) [2]. On the other hand, 2D Sketcher is written using JAVA AWT and JAVA Swing (light weight). A heavyweight component is one that is associated with its own native screen resource (commonly known as a peer). A lightweight component is one that "borrows" the screen resource of an ancestor (which means it has no native resource of its own – so it's "lighter"). In particular, this means that a canvas3D will draw on top of Swing objects no matter what order Swing thinks it should draw in. As a result, JAVA3D and JAVA Swing are not fully compatible with each other [44]. To walk around this, fixes introduced by the Swing team have to be followed.

We use a Java3D scene graph for rendering, sketching and editing purposes. The scene graph is a graph structure that contains Java3D nodes. Each node connection represents a parent-child relationship.

The 3D Sketcher is a second version of Sketcher that allows the user to interactively sketch in 3D space. There are many issues that need to be addressed in case of a 3D GUI [52] in order to provide the user the intuitive feel of sketching, moving and manipulating objects in a 3D scene, including camera, visibility, shading and texture issues.

However, we restrict our discussion primarily to the object and constraint representations. The 3D Sketcher uses the same object hierarchy as the 2D Sketcher and the 2D-input 3D-output Sketcher, but the 3D Sketcher has some more 3D Shapes.

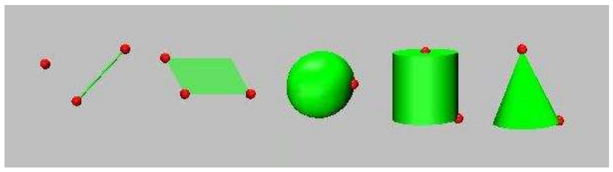


Figure 27: 3D Shapes or Objects containing the point subshape as handle

A scene graph is constructed in such a way that state information cannot be shared among sub-graphs. This enables Java3D to render scenes concurrently. The viewBranch node governs the camera position. The axisGroup node contains the nodes corresponding to X, Y and Z axes and the X-Z plane that are displayed at all times in the 3D scene to help the user visualize the 3D space. The scene node has a mouseRotate node that allows the user to rotate the whole scene. Along with the scene the axes and the plane are also rotated using an other node (not shown in the figure). The picker node under the scene node allows the user to pick any of the objects that are present in the scene. The light node is responsible for the lighting in the scene. The rootGroup node contains the all the nodes corresponding to the different shapes and constraints in the scene.

7.2.1 Representation of 3D Objects and Constraints

The point is represented by a small sphere in 3D space. The basic hierarchy among the shapes is maintained exactly as it is in 2D Sketcher. The point is the basic shape and all the other shapes have sub-shapes that are points and act as handle to manipulate the position or the dimensions of the object. The line segment has two sub-shapes that are its end points. The line segment consists of a thin cylinder connecting the two end points. The orientation and the length are calculated using the positions of the two endpoints of the line segments. The length is changed by appropriately scaling the cylinder along its axis. Then the mouse directed rotation is applied to the scaled cylinder to position it accurately.

In addition to the points and the line segments in the 2D-input 3D-output Sketcher there are some more shapes in the 3D Sketcher. See Figure 3.

The plane shape is a quadrilateral having three handles, each positioned at one of the four corners of the rectangle. The plane of the object can be changed by dragging the handles. The handle can also be used to extrude and or rotate the plane shape as required. The sphere object has only one handle, which is used to change the radius of the sphere. The position can be changed by dragging the sphere itself. The cylinder object is similar to a line segment except that it has one more handle that is used to change the radius of the cylinder. This handle is placed at on the circumference of the cylinder. The handles along the axis of the cylinder can be used to rotate the cylinder or change the height of the cylinder. The cone object is similar to the cylinder object having three handles.

Currently, the 2D-input 3D-output Sketcher has only three constraints the distance constraint, angle constraint and torsion angle constraint. For 3D examples, it allows an angle constraint to be specified between three points/sub-shapes. The three shapes involved in an angle constraint can be sub-shapes of line segments or a point objects. It considers the point that was picked second among the three points to be the vertex of the angle. The torsion angle is between 2 line segments that may not share a point. Two specified points are identified, forcing the line segments into a common plane, and the constraint then specifies the angle between them.

8 Conclusion of Part I

We introduced and motivated FRONTIER using 5 requirements that are essential for 2D and 3D variational constraint solvers in CAD applications; compared FRONTIER's approach and performance with other constraint solvers on each of these requirements; provided basic background on geometric constraint solving; precisely described the input-output capabilities of FRONTIER; and provided background results on the Frontier Vertex DR-planner on which FRONTIER is based. This set the preliminaries for listing the new contributions of this 2 part manuscript as a list of 8 technical problems. FRONTIER's architecture and implementation address 2 of these problems (Problems 1 and 2 of Section 5) and have been discussed here. The remainder of the problems, requiring algorithmic solutions, will be discussed in Part II.

References

[1] S. Ait-Aoudia, R. Jegou, and D. Michelucci. Reduction of constraint systems. In *Compugraphics*, pages 83–92, 1993.

[2] Alice. http://www.alice.org/jalice/docs/java3d/api/index.html. 2000.

[3] V. Allada and S. Anand. Feature-based modeling approaches for integrated manufacturing: state-of-the-art survey and future resear ch directions. *International Journal for Computer Integrated Manugfacturing*, 8:411–440, 1995.

[4] W. Bouma, I. Fudos, C. Hoffmann, J. Cai, and R. Paige. A geometric constraint solver. *Computer Aided Design*, 27:487–501, 1995.

[5] W. Bouma, I. Fudos, C. M. Hoffmann, J. Cai, and R. Paige. A geometric constraint solver. *CAD*, 27:487–501, 1995.

[6] W.F. Bronsvoort and F.W. Jansen. Multiview feature modeling for design and assembly. In J.J. Shah, M. Mantyla, and D.S. Nau ed.s, editors, *Advances in Feature Based Modeling*, pages 315–330. Elsevier Science, 1994.

[7] B. Bruderlin. Constructing three-dimensional geometric object defined by constraints. In *ACM SIGGRAPH*. Chapel Hill, 1986.

[8] G. Brunetti and B. Golob. A feature based approach towards an integrated product model including conceptual design information. *Computer Aided Design*, 32:877–887, 2000.

[9] D. Cox, J. Little, and D. O'Shea. *Using algebraic geometry*. Springer-Verlag, 1998.

[10] R.P. Zuffante D.C. Gossard and H. Sakurai. Representing imensions, tolerances, and features in mcae systems. *IEEE Computer Graphics and Application*, 8:51–59, 1988.

[11] K.J. de Kraker, M. Dohmen, and W.F. Bronsvoort. Multiway feature conversion to support concurrent engineering. In *ACM Symposium on Solid Modeling*, pages 105–114. ACM press, 1995.

[12] K.J. de Kraker, M. Dohmen, and W.F. Bronsvoort. Maintaining multiple views in feature modeling. In *ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 123–130. ACM press, 1997.

[13] U. Doering and B. Bruderlin. A declarative modeling system. In P. Brunet, editor, *CAD Systems Development -Tools and Methods*, page To appear. SpringerVerlag, 1999.

[14] C. Durand. *Symbolic and Numerical Techniques for Constraint Solving*. PhD thesis, Purdue University, Computer Science Dept, 1998.

[15] L. Eggli, C. Hsu, G. Elber, and B. Bruderlin. Inferring 3d models from freehand sketchers and constraints. *Computer Aided Design*, 29:101–112, 1997.

[16] I. Fudos. *Constraint solveing for computer aided design*. Ph.D. thesis, Dept. of Computer Sciences, Purdue University, August 1995.

[17] I. Fudos. *Geometric Constraint Solving*. PhD thesis, Purdue University, Dept of Computer Science, 1995.

[18] I. Fudos and C. M. Hoffmann. Correctness proof of a geometric constraint solver. *Intl. J. of Computational Geometry and Applications*, 6:405–420, 1996.

[19] I. Fudos and C. M. Hoffmann. Correctness proof of a geometric constraint solver. *J. Comp. Geometry and Applic.*, 6:405–420, 1996.

[20] I. Fudos and C. M. Hoffmann. A graph-constructive approach to solving systems of geometric constraints. *ACM Trans on Graphics*, pages 179–216, 1997.

[21] I. Fudos and C. M. Hoffmann. A graph-constructive approach to solving systems of geometric constraints. *ACM Transactions on Graphics*, 16:179–216, 1997.

[22] X. S. Gao and S. C. Chou. Solving geometric constraint systems. I. a global propagation approach. *CAD*, 30:47–54, 1998.

[23] X. S. Gao and S. C. Chou. Solving geometric constraint systems. II. a symbolic approach and decision of rc-constructibility. *CAD*, 30:115–122, 1998.

[24] Jack E. Graver, Brigitte Servatius, and Herman Servatius. *Combinatorial Rigidity.* Graduate Studies in Math., AMS, 1993.

[25] J.H. Han and A.A.G. Requicha. Modeler-independent feature recognition in a distributed environment. *Computer Aided Design*, 30:453–463, 1998.

[26] C Hoffman, M Sitharam, and B Yuan. Making constraint solvers more useable: the overconstraint problem. *to appear in CAD*, 2004.

[27] C. M. Hoffmann and C. S. Chiang. Variable-radius circles in cluster merging, Part I: translational clusters. *CAD*, 33:in press, 2001.

[28] C. M. Hoffmann and R. Joan-arinyo. Symbolic constraints in geometric constraint solving. *J. for Symbolic Computation*, 23:287–300, 1997.

[29] C. M. Hoffmann and R. Joan-Arinyo. Distributed maintanence of multiple product views. *Manuscript*, 1998.

[30] C. M. Hoffmann, A. Lomonosov, and M. Sitharam. Finding solvable subsets of constraint graphs. In Smolka G., editor, *Springer LNCS 1330*, pages 463–477, 1997.

[31] C. M. Hoffmann, A. Lomonosov, and M. Sitharam. Geometric constraint decomposition. In Bruderlin B. and Roller D., editors, *Geometric Constr Solving and Appl*, pages 170–195, 1998.

[32] C. M. Hoffmann and J. Peters. Geometric constraint for CAGD. In M. Daehlen, T. Lyche, and Schumaker L., editors, *Mathematical Methods for Curves and Surfaces*, pages 237–254, 1995.

[33] C. M. Hoffmann and R. Vermeer. Geometric constraint solving in R^2 and R^3. In Du D. Z. and Hwang F., editors, *Computing in Euclidean Geometry*, pages 266–298, 1995.

[34] C. M. Hoffmann and B. Yuan. On spatial constraint solving approaches. In *Proc. ADG 2000, ETH Zurich*, page in press, 2000.

[35] Christoph M. Hoffmann, Andrew Lomonosov, and Meera Sitharam. Finding solvable subsets of constraint graphs. In *Constraint Programming '97 Lecture Notes in Computer Science 1330, G. Smolka Ed., Springer Verlag*, Linz, Austria, 1997.

[36] Christoph M. Hoffmann, Andrew Lomonosov, and Meera Sitharam. Geometric constraint decomposition. In Bruderlin and Roller Ed.s, editors, *Geometric Constraint Solving.* Springer-Verlag, 1998.

[37] Christoph M. Hoffmann, Andrew Lomonosov, and Meera Sitharam. Planning geometric constraint decompositions via graph transformations. In *AGTIVE '99 (Graph Transformations with Industrial Relevance), Springer lecture notes, LNCS 1779, eds Nagl, Schurr, Munch*, pages 309–324, 1999.

[38] Christoph M. Hoffmann, Andrew Lomonosov, and Meera Sitharam. Decomposition of geometric constraints systems, part i: performance measures. *Journal of Symbolic Computation*, 31(4), 2001.

[39] Christoph M. Hoffmann, Andrew Lomonosov, and Meera Sitharam. Decomposition of geometric constraints systems, part ii: new algorithms. *Journal of Symbolic Computation*, 31(4), 2001.

[40] Christoph M. Hoffmann and Pamela J. Vermeer. Geometric constraint solving in R^2 and R^3. In D. Z. Du and F. Hwang, editors, *Computing in Euclidean Geometry.* World Scientific Publishing, 1994. second edition.

[41] Christoph M. Hoffmann and Pamela J. Vermeer. A spatial constraint problem. In *Workshop on Computational Kinematics*, France, 1995. INRIA Sophia-Antipolis.

[42] C. Hsu, G. Alt, Z. Huang, E. Beier, and B. Bruderlin. A constraint-based manipulator toolset for editing 3d objects. In *1997 ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications*, 1997.

[43] C. Hsu and Bruderlin B. A degree of freedom graph approach. In *Geometric Modeling: Theory and Practice*. Springer Verlag, 1997.

[44] The Java 3D Community Site (J3D). http://www.j3d.org/faq/. 1999.

[45] R. Joan-Arinyo, A. Soto-Riera, S. Vila-Marta, and J. Vilaplana-Pastó. Transforming an under-constrained geometric constraint problem into a well-constrained one. In *ACM Symposium on Solid Modeling and Applications,Proceedings of the eighth ACM symposium on Solid modeling and applications*, pages 33–44, 2003.

[46] R. Klein. Geometry and feature representation for an integration with knowledge based systems. In *Geometric modeling and CAD*. Chapman-Hall, 1996.

[47] R. Klein. The role of constraints in geometric modeling. In Bruderlin and Roller ed.s, editors, *Geometric constraint solving and applications*. Springer-Verlag, 1998.

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

[49] G. Kramer. *Solving geometric constraint systems: a case study in kinematics*. MIT Press, 1992.

[50] G. Laman. On graphs and rigidity of plane skeletal structures. *J. Engrg. Math.*, 4:331–340, 1970.

[51] R. Latham and A. Middleditch. Connectivity analysis: a tool for processing geometric constraints. *Computer Aided Design*, 28:917–928, 1996.

[52] G. Leach, G. Al-Qaimari, M. Grieve, N. Jinks, and C. McKay. http://goanna.cs.rmit.edu.au/ gl/research/hcc/interact97.html. 1997.

[53] Andrew Lomonosov and Meera Sitharam. Approximation algorithms for finding minimum dense subgraphs of constraint graphs. In *submitted*, 2004.

[54] M. Mantyla, J. Opas, and J. Puhakka. Generative process planning of prismatic parts by feature relaxation. In *Advances in Design Automation, Computer Aided and Computational Design*, pages 49–60. ASME, 1989.

[55] A. Middleditch and C. Reade. A kernel for geometric features. In *ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications*. ACM press, 1997.

[56] G. J. Nelson. A costraint-based graphics system. In *ACM SIGGRAPH*, pages 235–243, 1985.

[57] J. J. Oung, M. Sitharam, B. Moro, and A. Arbree. Frontier: fully enabling geometric constraints for feature based design and assembly. In *abstract in Proceedings of the ACM Solid Modeling conference*, 2001.

[58] J. Owen. www.d-cubed.co.uk/. In *D-cubed commercial geometric constraint solving software*.

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

[60] J. Owen. Constraints on simple geometry in two and three dimensions. In *Third SIAM Conference on Geometric Design*. SIAM, November 1993. To appear in Int J of Computational Geometry and Applications.

[61] J.A. Pabon. Modeling method for sorting dependencies among geometric entities. In *US States Patent 5,251,290*, Oct 1993.

[62] A. Rappoport and M.J.G.M. van Emmerik. User interface devices for rapid and exact number specification. *ACM Transactions on Graphics*, 12:348–354, 1993.

[63] D. Roller. An approach to computer-aided parametric design. *CAD*, 23:303–324, 1991.

[64] Jaroslaw P. Rossignac. Constraints in constructive solid geometriy. In *Workshop on Interactive 3D Graphics*, pages 23–24. Chapel Hill, 1986.

[65] Meera Sitharam. Frontier, opensource gnu geometric constraint solver: Version 1 (2001) for general 2d systems; version 2 (2002) for 2d and some 3d systems; version 3 (2003) for general 2d and 3d systems. In *http://www.cise.ufl.edu/∼sitharam, http://www.gnu.org*, 2004.

[66] W. Sohrt and B. Bruderlin. Interaction with constraints in 3d modeling. *International Journal of Computational Geometry and Application*, pages 405–425, 1991.

[67] N. Sridhar, R. Agrawal, and G. L. Kinzel. An active occurrence-matrix-based approach to design decomposition. *CAD*, 25:500–512, 1993.

[68] N. Sridhar, R. Agrawal, and G. L. Kinzel. Algorithms for the structural diagnosis and decomposition of sparse, underconstrained design systems. *CAD*, 28:237–249, 1995.

[69] Department of Com puter Science Technical University of Ilmenau, Computer Graphics Group and Automation. http://rabbit.prakinf.tu-ilmenau.de/qsketch.html. 2001.

[70] P. Todd. A k-tree generalization that characterizes consistency of dimensioned engineering drawings. *SIAM J. Discrete Mathematics*, 2:255–261, 1989.

[71] D. C. Gossard V. Lin and R. Light. Variational geometry in computer-aided design. In *ACM SIGGRAPH*, pages 171–179, 1981.

[72] W. Whiteley. Rigidity and scene analysis. In *Handbook of Discrete and Computational Geometry*, pages 893 –916. CRC Press, 1997.

[73] B. Yuan. *Research and implementation of geometric constraint solving technology.* Ph.D. thesis, Dept. of Computer Science and technology, Tsinghua University, China, November 1999.

[74] R. C. Zeleznik, K. P. Herndon, and J. F. Hughes. Sketch: An interface for sketching 3d scenes. *Proceedings of SIGGRAPH 96 (New Orleans, August 1996)*, pages 163–170, 1996.