INTERACTIVE MODIFYING THE METHOD SET OF A GEOMETRIC CONSTRAINT

David Podgorelec, Borut Žalik, Simon Kolmanič
University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova 17, SI-2000 MARIBOR, SLOVENIA
tel: ++386 62 221-112, fax: ++386 62 225-013
e-mails: david.podgorelec@uni-mb.si, zalik@uni-mb.si, simon.kolmanic@uni-mb.si

Abstract: In the paper, a way of making geometric constraints more flexible is described. This approach is employed in our new interactive 2D constraint-based drawing system. Multidirectional constraints should be very powerful to find exactly the results expected by a user when more than one solution is possible. By defining constraint method priorities, we can satisfy some designer's intents but not all of them. So we leave a possibility to reorder method lists of particular constraints to the user. Two new terms: parallel and alternative methods are also introduced. Finally, we present some basic characteristics of two our geometric constraint solving systems.

Keywords: geometric constraints, geometry, geometric modelling, CAD, constraint solving, graphs.

1 INTRODUCTION

CAD systems were intended to realise a support of a whole design process so that a designer could concentrate on a really creative part of the design, but conventional geometric modellers were/are actually used only for activities that occur near the end of the design process: detailing the geometry of artefacts, analysis of such artefacts, generating production drawings, etc. In the early phases of the design which include all the creative work, a designer still depends exclusively on himself. The system only accepts solutions, presents and analyses them. This is the most serious drawback of conventional CAD systems - they have no intelligence. Really efficient assistant of designers should be able to perform automatically all well-defined, but boring, awkward and time-consuming tasks instead of the designers. A user only specifies object's shape and size in a declarative way, and the system takes care of making the drawing in accordance to the specification. The user specifies what to draw, not how to draw it [Pabon92], [Sunde87], [Žalik95], [Žalik96a].

Recently, new approaches to offer a designer more efficient support were proposed. They include methods of artificial intelligence, hierarchical and variational modelling, object-oriented programming, and fast prototyping. Their common goal is to improve communication between a user and a modeller by giving to the latter an intelligence to conceive what the former intends. One of these concepts is introducing *geometric constraints* .

A geometric constraint is a relation among geometric objects that should be satisfied. Explicit dimensions of distances and angles, constraints of parallelism, perpendicularity, collinearity, tangency, concentricity, and prescribed radii can be included. The constraint's solving mechanism consists of methods that recalculate values of one or more object's parameters to achieve consistency of all inserted constraints. The system is not intended to have any knowledge of what is drawn or designed. It should provide a mechanism to solve or at least try to solve any configuration of geometric objects and relations among them, and the solution should be independent of the order in which the constraints are written down. Such a constraint solver is called *variational*. Unfortunately, general techniques are weak, so the solving system is application-dependent and hardly adaptable to other applications [Bouma95], [Žalik96a].

It is usually difficult for human designers to specify exactly geometric constraints needed to define an object unambiguously. A *well constrained* problem has a finite number of solutions. If there exist an infinitive number of solutions, a problem is *underconstrained*, and if there are not solutions, it is *overconstrained*. In such cases, the system should be able to show the designer the set of conflicting constraints. Specifications that are impossible to solve because the values of object parameters are contrary to some mathematical theorems should also be recognised. For example, a triangle with one side longer than the sum of the two others does not exist.

Coupled with the fact that a well constrained problem has, in general, exponentially many solutions, only one of which satisfies the user's intent, constraint solvers have to address two distinct tasks:

- Determine whether the problem can be solved and, if so, how.
- Among the possible solutions, identify the one that the user intends [Bouma95].

A wide variety of geometric constraint solvers coping more or less successfully with the first task were developed, but the second one was passed over in silence by the majority of authors. More than just giving to a user an ability to choose among several offered solutions should be done. If the system was developed to assist a designer during all the design process, then it must be at least able to suggest him one of the computed solutions. In this paper, we shall concern ourselves with this task among all. But we still do not believe it will ever be solved satisfactory without any interaction of the user. Only the user knows exactly what he needs. Some heuristics can be used by a system but they cannot always assure exactly the solution intended by the user. Especially because different designers design in different ways.

Both our constraint solving systems described in this paper belong to propagation methods discussed in the next chapter. Some properties of both systems will be compared in the third chapter. In the fourth chapter, our interactive method for adaptation of constraints to different users intents is described. We shall try to confirm the usefulness of this tool by some examples.

2 PROPAGATION METHODS

Different approaches to geometric constraint solving were proposed. Rather detailed classifications are given in [Bouma95], [Lee96], and [Žalik96a], for example, but they differ from one author to other. Both, our new system and the older one called Basic Font Feature Design (BFoFD) are representatives of the propagation methods, so we shall describe this approach a bit more precisely.

Propagation methods were a popular approach in early constraint solving systems. Both geometric entities and geometric constraints are presented by an undirected constraint graph. Each entity node contains momentary values of entity parameters (variables), and a constraint node stores a set of methods capable to solve the constraint. Each graph edge represents whether an entity is directly affected by a constraint. The constraint graph is bipartite, while there are not direct connections between two distinct constraint nodes or between two entity nodes. Known values of variables propagate through edges of the graph. When a node receives enough information, it fires, calculates one or more values and offers them to adjacent nodes. The process terminates when there is no node that can fire. The graph is undirected, so each edge can either supply a node with needed information or return the results of the calculations. This principle enables the use of *multidirectional constraints*. More than one method should be present in the method set of such a constraint. For example, a constraint *On(p, l)* involved in our systems requires that a point *p* lies on a line *l*. This can be achieved either by moving the point or by moving the line, so the constraint contains at least two methods. The choice of a method depends on the presence of the data on the edges of a node. But this holds only for systems which use *refinement model* where variables are initially unconstrained and their values are progressively refined after each edit step. In contrast, in the *perturbation model*, variables have some default values associated with them that satisfy all the constraints. After each edit operation, a value of one or more variables is perturbed and the constraint solver has to adjust values so that the constraints are again satisfied. This model enables better interactiveness and implies implementation of *incremental* constraint satisfaction algorithms able to take advantages of previous computations rather than starting over each time there is a change in the set of constraints [Sanne93]. While in such a system all the values are present at the node's arcs all the time, the presence of the data on edges cannot be used to determine the order of choosing methods. Methods are therefore used in some predefined order or some supplementary rules are introduced to decide which method to use.

The propagation techniques remain the simplest and the basic mechanism for the derivation of solutions that satisfy given constraints, but they suffer from some serious drawbacks. None of them can guarantee that an existing solution will be derived, and the majority of them fail to solve cyclic constraint situations. But these methods are still the nearest to the human way of thinking, what preserves their popularity even today, although the knowledge-based and symbolic constraint solvers are supposed to be much more powerful and efficient [Bouma95].

3 BFoFD AND OUR NEW CONSTRAINT SOLVING SYSTEM

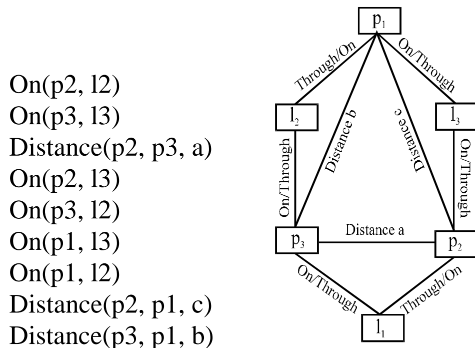
Generally, BFoFD enables definition of any 2D geometric objects, but it is first of all intended for a font design. The constraint set is adapted for this task, and the method sets of particular constraints are just sufficient, not containing too complicated, time-consuming and numerically unstable algorithms. This is supported by introducing a rule of *minimal disturbance*: the constraint solver always searches for a solution that causes minimal changes of entity parameters. Such a system is naturally based on a perturbation model. All these insure satisfactory speed of constraint solving and completely interactive input

of geometric entities and constraints. Unfortunately, the system is only partially incremental in sense it does not require the insertion of the whole sketch at once. But after each edit operation, it still checks all the parameters of entities forming a connected figure.

BFoFD uses a special data structure called the *biconnected constraint description graph* (BCDG). Graph nodes coincide with geometric entities, and constraints are presented by oriented graph arcs. Two entities connected with a multidirectional constraint require a pair of contrary oriented arcs. While a constraint can also affect only one entity, or connect more than two entities as well, the arc connecting exactly two entities cannot suffice for any presentation. An arc with a free end is used in the first case, and a structure containing three arcs is employed for a constraint connecting three entities. Constraints directly affecting more than three entities are not incorporated yet.

BFoFD cannot handle cyclic constraint configurations. The problem appears because only the information local to the observed graph node is used at each step. A node can fire only if the needed data is received from its adjacent nodes, but some of these neighbours can also expect some information from the observed node. Even in this case, some configurations can be solved successfully if particular variables are fixed by other constraints. But such a solution depends on an order of solving constraints, and therefore cannot be treated as a general approach to solving cyclic constraint situations. This fact also implies that we did not manage to create a completely variational constraint solver with BFoFD. Trying to increase a number of solvable configurations is the main goal of our recent researches. Although the statement that the propagation methods are inappropriate for handling cyclic situations has been seen in numerous papers, especially those written by the authors of methods based on some other principle, the proof is still missing, and therefore, we do not see any reason to give up our attempts. Especially because our experience with BFoFD was not bad at all.

Once a constraint graph is chosen, an order of visiting its nodes has to be determined. These two things together form the solving algorithm. Different propagation methods vary the most in the graph structure and the way of choosing graph nodes.



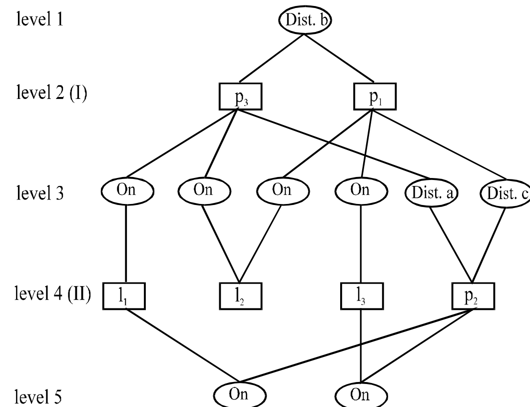


Figure 1: BCDG and the bipartite constraint graph for a triangle constrained by its side lengths

In the graph used in our new method, both entities and constraints are presented as nodes. An edge exists between an entity node and a constraint node if and only if the entity is directly affected by the constraint. This *bipartite* constraint graph was described in the section about propagation methods already. In Fig. 1, we can compare it with the BCDG. An example of constraining a triangle by its side lengths is used. Levels of the bipartite graph will be described together with the mechanism providing incrementality. They are assigned considering the sequence of adding constraints listed at the left side of the figure. What makes our method different from general propagation approach and from BFoFD?

All the actions are done in constraint nodes, and they immediately update values stored in adjacent entity nodes. In BFoFD, the principle was just the opposite. Values stored in a particular entity node were calculated by using the data describing other entities, and the constraints affecting the observed entity.

A mechanism providing some kind of incrementality has been developed. Levels corresponding to distances between the beginning node and observed nodes are used for this task. While the influence of any change decreases with a distance from the node involving this change, it is easy to prove the following sentence: *Once an entity level is reached*

where all the variable values are unchanged after an edit operation, then all the following entity levels remain unchanged, too. This holds only if actions defined by the last edit operation are grouped at the top level of the graph. Once this level is established, levels of all nodes are determined. There are five types of edit operations that activate the solving procedure:

1. *Inserting a new entity.* The new entity is not affected by any constraint yet, and therefore not connected with other parts of the graph. The solver need not be activated after this operation.

2. *Deleting the entity.* All the constraints affecting the deleted entity directly are deleted, too. All the solutions still hold, but the solution set is usually extended. A well-constrained problem can become underconstrained, and an overconstrained problem can change to well-constrained. Incrementality is preserved by grouping the constraints affecting entities that used to share some deleted constraints with the deleted entity at the top level.

3. *Adding a new constraint.* A constraint node is created and connections to affected entity nodes are established. The new node is the beginning node (the only node at level one) of the graph.

4. *Deleting the constraint.* Constraints that used to share some entities (they are not deleted) with the deleted one form the top level of the graph.

5. *Modifying numerical parameters of a particular constraint.* A constraint with modified parameters becomes the beginning node.

The graph is passed twice. During the first pass, the topology and dimensions are established. The graph is also decomposed to subgraphs presenting topologically connected entities. In the second pass, particular objects are positioned and oriented according to the inserted positioning constraints *Point, AngleValue, HLine* and *VLine*. What are the benefits of such approach?

A designer is usually interested in a shape and size of a product only, and absolute coordinates mean nothing to him. There is an option that enables him to simply exclude the second pass. The same can be achieved by disabling all the methods of the positioning constraints in our interactive tool.

Subgraphs presenting connected components can be solved independently. Once a decomposition is done, it is not necessary to observe all the graph after a single edit operation. The decomposition also facilitates the detection of overconstrained and underconstrained parts of the graph. Finally, an integration of constraint-based design and form feature modelling is facilitated.

Normally, the connected component is positioned and oriented by a point and a slope of a line. If two points were determined or two lines were oriented, the component was usually identified as overconstrained, or in the best case, a redundant constraint was detected. Obviously, the positioning constraints solved too early usually render the solver's work more difficult. If the method sets of particular constraints are not powerful enough, the problem can even become unsolvable. The following example is very persuasive. We have two parallel lines and want them to be at the requested distance from each other. There are two points also, each lying on one of the lines. If at least one of the points is not fixed by the *Point* constraint, then it suffices to move parallely the line together with this point to the required distance from the other line. But if both of the points are fixed, then both of the lines have to be rotated. This task is much harder to solve and requires much more powerful method sets of particular constraints.

Described edit operations (1-5) preserve incrementality. They define an unique order of visiting graph nodes, and form a general part of the constraint solver. Its application is not restricted to design 2D drawings, not even to the geometry or mathematics only. On the other hand, it seems that we loose this generality by distinguishing between positioning and other constraints. For this reason, the classification is done by assigning priorities to the constraints rather than by implementing two separated classes. Actually, such implementation enables us to group constraints to even more than two classes and to solve them in more than two passes. In addition to this, the user can mark pass(es) where a decomposition and/or uniting are done.

Here we start to discuss another part of the system: the constraint set. This feature is application-dependent. We managed to develop the constraint solver which is not under its influence, but a user is only interested to get the solution as good and as soon as possible, no matter if some constraint particularities are built in the solver. Constraints used in BFoFD enable a user to draw a 2D scene in a simple, natural way. Therefore, we have kept them in our new system, too. But our recent studies have confirmed that many problems thought to originate in cyclic constraint configurations, actually arise from weaknesses of particular constraints. If the constraint *Through(l, p)* is only capable to move the line l parallely to pass through the point p, and if the constraint *Distance(p1, p2, D)* only moves one of the points *p1* and *p2* along the line determined by them, then they surely do not suffice for all kinds of ruler-compass problems.

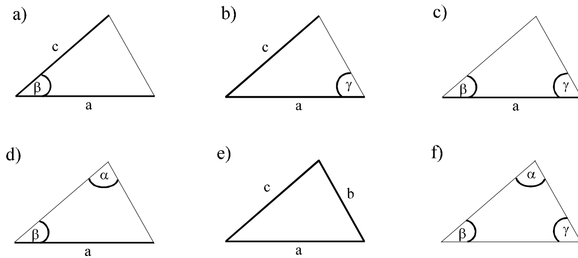


Figure 2: Different ways of constraining a triangle

In Fig. 2, six ways of defining a triangle by its sides and angles only are presented. BFoFD is only able to solve configurations a and c. Some rotations have to be performed in cases b, d and e. A triangle in the picture f is of course underconstrained because three angles are not presenting three independent parameters. The surprisingly bad result restricts some users in using their own designing style. Obviously, several constraints need more powerful method sets for designing general geometric scenes in 2D space.

3.1 Visible and Auxiliary Geometry

An important feature of BFoFD is a division of the geometry into visible and auxiliary part. The auxiliary lines, arcs and points are widely used in hand-made technical drawings, and also supported in classical drawing systems where geometric entities are placed onto several layers that can be independently displayed or turned-off, so the most of the designers are used to enjoy the benefits of this approach. In BFoFD, each entity of the visible geometry has its equivalent (or more of them) in the auxiliary part, and the opposite is not necessary. When the auxiliary geometry is constrained, the visible part is constrained, too. Therefore, it suffices to employ constraints that operate on the auxiliary geometry only. While the entities of the visible geometry could be pretty sophisticated, it would be rather difficult to constrain them efficiently by using a constraint set affecting them directly. On the other hand, components of the auxiliary geometry need not be too complex, while they were introduced only to simplify manipulation of the visible entities, and will not be presented in the final drawing. An ellipse is constrained by its axis, and a Bézier curve by its control points. Handling with a variety of complex objects is efficiently reduced to manipulation of points and lines only. By constraining only two types of entities, the required constraint set is importantly reduced, too. The implementation is much easier and quicker, although the entities of the same complexity are still available in the final drawing as the part of the visible geometry, of course.

A requirement of additional data describing the auxiliary geometry can be considered as the only disadvantage of this principle. But the reader should be informed that the majority of the auxiliary geometry is created by the system when the visible objects are inserted into the sketch. All self-understandable constraints are also added automatically [Žalik96b].

3.2 A Constraint Set for 2D Drawings

A result of introducing the auxiliary geometry is that only two types of entities (points and lines) have to be constrained. In consequence of this, the number of necessary constraint types has also been considerably reduced. According to the classification made by Aldefeld [Aldef88], predicates defining constraints in BFoFD are divided into two groups:

- *Dimensional constraints* determine positions, distances, coordinates, and angles. Their constituent parts are variables which can be considered as parameters of a geometric object.

- *Structural constraints* determine spatial relationships between geometric elements which do not change. If we require, for example, that two lines are parallel, they have to stay parallel permanently.

In Fig. 3, the constraint set employed in both our drawing systems is shown. The picture is presenting the main window of our incremental tool Method Tree 1.0 described later. Constraints *Point, Distance, AngleValue, PosXY* and *Angle* are dimensional, and all the others are structural constraints. The detailed description of actions provided by particular constraints can be found in [Žalik96a], for example.

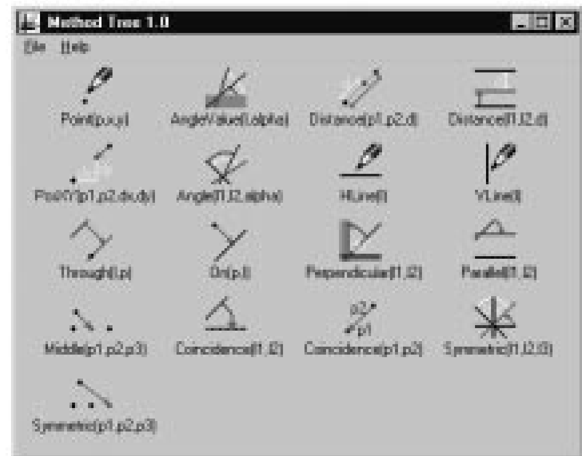


Figure 3: The constraint set of our 2D constraint-based drawing system

4 MODIFYABLE METHOD SETS OF PARTICULAR CONSTRAINTS

While different designers design in different ways, a

particular drawing system is not capable to satisfy all their requests. Nowadays, all professional software products offer to a user an ability to use and save his own settings. Not only the appearance of a user interface can be configured. Different decision ways through a program or even different algorithms can be chosen. Therefore, our idea to enable a user to interactively modify constraint method sets is not wholly innovative but rather an adaptation of this widely used approach.

Solutions found by a variational constraint solver should not depend on order of inserting constraints. But such constraints should provide very powerful method sets able to find really all the solutions what is extremely difficult to implement. In many cases, only a small subset of all possible solutions (or in the worst case, none of them) is found. The user should be aware of these drawbacks of propagation methods. If the CAD system is intended to give an intelligent support to the user, then the user is also expected to help the system by providing it the input data as good as possible. This is particularly important if the rule of minimal disturbance is used. If a line should be vertical in the final drawing then do not draw it horizontal! If the user respects such instructions, a probability that the desired solution is included in the found solution subset increases importantly. But this does not suffice to the user. If more than one solution is found then it is desired that exactly the one that user intends is displayed immediately. Just suppose that after each edit operation, five or six solutions are displayed before the right one. If the system has to protect a user from such stresses then it should receive some additional information on user's intents. Some users might be used to measure angles in the negative direction rather then in positive, or they were maybe using some constraint of the name that occurs in our constraint set, in some other drawing system already and want that the same action is provided by this constraint. Our tool enables them to configure the constraints in the way they want.

Both objects in Fig. 4 are constrained by the same set of constraints. The only difference originates from direction of angle measurements. The constraint *Angle(alpha, l1, l2)* can be solved by four methods. Two of them move the line *l1* to establish the required angle *alpha* between it and the line *l2*. Once, *alpha* is measured clockwise, and in another method counter-clockwise. The other two methods move the line *l2*. Therefore, four different solutions are obtained. Are they really different? The topology is obviously not changing, but according to the shape, the solutions can be grouped into two subsets only. It depends on the user, whether rotated or mirrored objects are thought to present the same solution or the different ones.

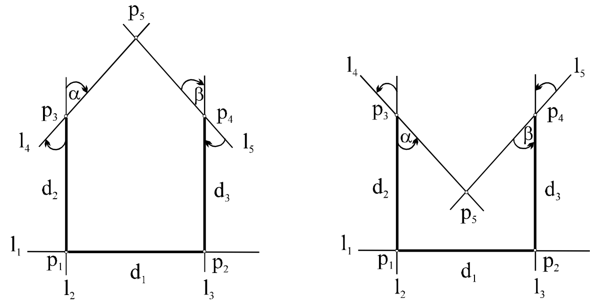


Figure 4: Change of shape as a consequence of the reverse angle orientation

4.1 Parallel and Alternative Methods

There are two reasons why a particular constraint needs more than one method:

- Constraints are multidirectional. If more than one entity is involved in a constraint equation, then theoretically, each of these entities can be moved to satisfy the constraint. In practise, this is not always possible because the degrees of freedom of some entities can be restricted by other constraints. At the moment, methods that simultaneously move more than one entity are not involved in our system yet.
- The constraint equation has multiple solutions. An entity can be moved to several positions that all satisfy the constraint requirements.

Methods originating in the fact that the constraints are multidirectional are chosen sequentially until the successful method is found or all such methods are employed. An order of these methods is of great importance while some methods will probably never or very rarely be executed. If the constraint *On* successfully moves the point, there is no need to move the line, too. Such methods therefore exclude each other, and will be named *alternative methods*. If, on the other hand, the same entity can be moved in different ways to satisfy the constraint, then all successfully obtained configurations have to be stored in a set of eventual solutions. Methods organised in this way are called *parallel*.

The majority of constraints contain both, parallel and alternative methods. Different solutions can be found and several methods are available for calculating a particular solution. Therefore, the organisation of methods of a particular constraint is two-dimensional, and can be implemented as a binary tree. The methods are presented as the tree nodes. The left son of a particular node is the next alternative method, and the right son presents one of the adjacent parallel methods. Such a structure is included in each constraint node of our bipartite constraint graph. In

Fig. 5, the procedure which provides all actions required at a particular method node during the process of constraint solving is listed. The procedure is recursive and has to be called with the parameter presenting the root node of the method tree.

```
void Method::Visit_Node(void)
{   if (!Calculate_The_Solution())
        if (alternative != NULL)
            alternative->Visit_Node();
    if (parallel != NULL)
        parallel->Visit_Node()
}
```

Figure 5: Procedure for constraint satisfaction

Only the function "*Calculate_The_Solution*" which extends the solution set with eventual new results depends on the constraint equation. Eventual conflicts with other constraints are also treated there. While the procedure *Visit_Method_Node* is the only point where the constraint solver meets application-dependent operations, and while these latter can be assigned to the corresponding objects outside this procedure, our solver is completely general and can be built-in any user interface regardless of the area of using the constraints. This shell also enables that the method tree can be reorganised in a simple and efficient way. An interactive tool was developed to support this task.

4.2 Implementation

The tool titled *Method Tree 1.0* is implemented in C++ programming language and runs on personal computers under Windows operating system. Let us first describe its behaviour from the user's point of view.

A constraint is selected from the constraint list first, and the window presenting its methods is opened then. The methods are presented by rectangular buttons. The client area is divided into three parts. Upper section contains buttons of the method tree presenting active methods. These are the methods available to the constraint solver during execution of the algorithm from Fig. 5. The method tree is a binary tree. Each node except the root one has its unique father and maximally two sons: the alternative and the parallel method. The alternative subtree is displayed immediately below its root node indented some columns to the right. The parallel son is displayed then with the same indent from the left margin as its father. The middle section of the window involves buttons presenting the list of disabled methods. The bottom part is reserved for displaying the information about the action currently executed. Actions are activated by selecting two buttons: the origin and destination method. An active method (origin) can be moved to any position in the method tree or to the top of the list of disabled methods. In the similar way, a disabled method can be reactivated. There are also two clickable labels which offer an ability to move the method into an empty method tree or disabled list. Both, an origin and destination can be selected either with the left or right mouse button while four different adjacency relations can be established between two nodes in the method tree: father - alternative son, alternative son - father, father - parallel son, and parallel son - father. In Fig. 6, an example of using our interactive tool is presented. An active method 3 is being disabled.

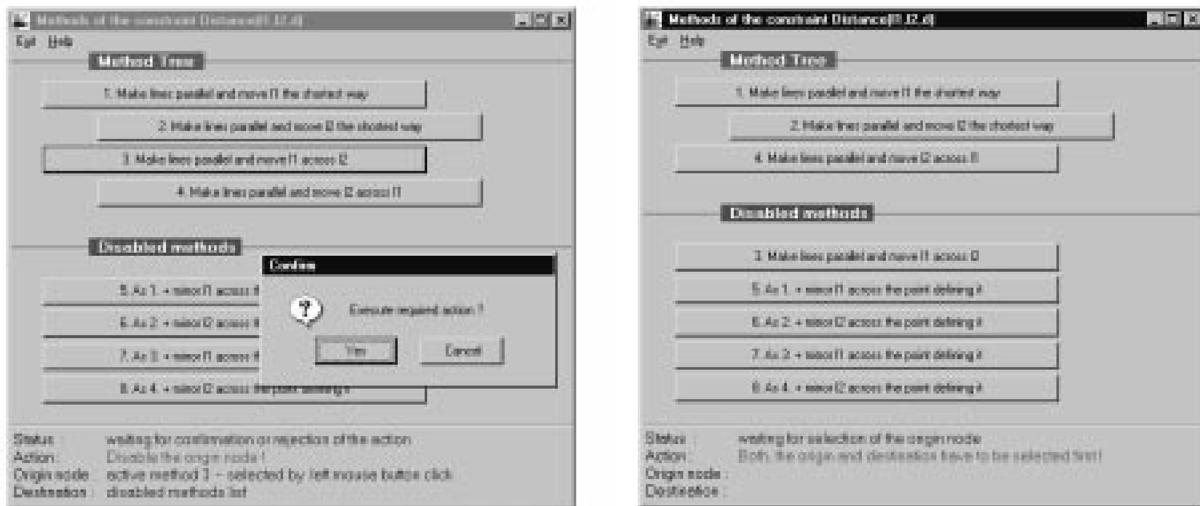


Figure 6: Example of using our interactive tool - disabling a method: a) before action, b) after it.

Of course, any of the actions causes an appropriate reorganisation of the pointers in the data structure. Saving and loading operations are also provided. The tool can be used as an independent project or as a part of the constraint solver's environment. A file that contains method trees of all the constraints is used for communication between the tool and the constraint-based drawing system. After these data are loaded to the drawing program, appropriate procedures have to be assigned to each of the methods, and the solver is ready for its work.

5 CONCLUSIONS

In the first part of the paper, the idea of our new propagation method for the geometric constraint solving is presented. The method is intended to overcome or at least facilitate some difficulties with our another constraint-based drawing system called BFoFD. These problems originate in cyclic constraint configurations and in too weak constraint method sets. We also wanted to generalise some principles. Namely, BFoFD was first of all designed for a special purpose: the font design, and the new method is completely application independent while the constraint set has no influence to the solver's behaviour. Both, BFoFD and the new method are based on the perturbation model, but BFoFD uses the rule of minimal disturbance which builds the solution by minimal changes at each step (and fails if this sequence is leading into a dead-lock). Our new method on the other hand offers to the user more than one solution, but still not all of them. It can still fail although a solution exists. This is the common problem of all propagation methods. The improvement is achieved by organising the methods of particular constraints into parallel groups always employed and not into the groups of reciprocally excluding alternative methods only. Therefore, methods of a particular constraint form a binary tree. Its data structure is initialised by the data stored in a configuration file. Positions of particular methods in the tree are therefore not fixed what enables an interactive modifying of the tree structure. An incremental tool described in the second part of the paper was developed to support this task.

The main goal of introducing our *Method Tree* tool is to enable a user to use his own designing style. Our constraint solver generally calculates more than one solution. A user-friendly system should always try to display the solution desired by the user first. Only if the user is not satisfied, other solutions are displayed. While the sequence of storing the solutions depends on order of methods in constraint method sets, the user is enabled to modify this order according to his designing style and other requests. The tool is also an indispensable assistant of us, the system developers. Although the behaviour of the solver is fully determined, it is still difficult to predict all the actions. Usually, it is almost impossible to identify the method which causes some change. The tool enables to exclude particular methods. This operation is very useful during debugging the system. We all know that the method sets are not closed yet. When some new method will be added or some existing one will be changed, the observation and analysis of effects caused by it will be facilitated by ability to isolate it from other methods of the same constraint.

6 REFERENCES

[Aldef88] Aldefeld, B., *Variation of Geometries Based on a Geometric-Reasoning Method*. Computer-Aided Design, vol. 20, no.3, pp. 117-126, 1988.

[Bouma95] Bouma, W., I. Fudos, C. Hoffmann, J. Cai, R. Paige, *Geometric Constraint Solver*. Computer-Aided Design, Vol. 27, No. 6, pp. 487-501, 1995.

[Lee96] Lee, J. Y., K. Kim, *Geometric reasoning for knowledge-based parametric design using graph representation*. Computer-Aided Design, Vol. 28, No. 10, pp. 431-841, 1996.

[Pabon92] Pabon, J., R. Young, W. Keirouz, *Integrating Parametric Geometry, Features, and Variational Modeling for Conceptual Design*. International Journal of Systems Automation: Research and Applications (SARA) 2, pp. 17-36, 1992.

[Sanne93] Sannella, M., J. Maloney, B. Freeman-Benson, A. Borning, *Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm*. Software-Practice and Experience, Vol. 23(5), 529-566, 1993.

[Sunde87] Sunde, G., *A CAD System with Declarative Specification of Shape*. Eurographics workshop on Intelligent CAD Systems. Noordwijkerhout, The Nederlands, 1987.

[Žalik95] Žalik, B., *Font Design with Incompletely Constrained Font Features*. Proceedings of the Third Pacific Conference on Computer Graphics and Applications, Pacific Graphics '95, Seoul, Korea, 1995.

[Žalik96a] Žalik, B., N. Guid, G. Clapworthy, *Constraint-based Object Modelling*, Journal of Engineering Design, Vol. 7, No,. 2, pp. 209-232, 1996.

[Žalik96b] Žalik, B., S. Kolmanič, D. Podgorelec, *A Drawing System Based on Separated Visible and Auxiliary Geometry*, Advances in Computer-Aided Design, Proceedings of CADEX'96, Hagenberg, Austria, 1996, pp. 151 - 160.