Eric Enderton

Department of Electrical Engineering and Computer Science
Computer Science Division
University of California, Berkeley

Interactive Type Synthesis of Mechanisms

March 30, 1990

RESEARCH PROJECT

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, in partial satisfaction of the requirements for the degree of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:

Committee:	[illegible]	Research Advisor
	CARLO H. SÉQUIN	Print Name
	4/1/1990	Date
	[illegible]	Second Reader
	John Canny	Print Name
	4/7/90	Date

Interactive Type Synthesis of Mechanisms

Eric Enderton*

March 30, 1990

Abstract

As a step towards CAD systems that model function as well as form, we have developed an interactive tool for qualitative mechanism design. Called the mechanism editor, it enables the user to quickly sketch abstract planar mechanisms made up of polygonal links connected by revolute and prismatic joints. The user may interactively control the parameter of a joint, and the editor will compute and display the resulting motion of the entire mechanism.

When the user requests an animation, the editor builds a plan to be executed repeatedly as the user flexes the input joint. The graph of the mechanism's topology, in which each link is represented by a node and each joint by an edge, is used for counting degrees of freedom, including the detection of overconstrained submechanisms. The graph is modified during the planning stage to represent the portion of the mechanism that remains to be solved. Each step in the plan solves a small subgraph that matches the graph of one of the available base cases. This step computes the relative positions of the links in the subgraph. This subgraph is then contracted to a single node, leaving a smaller graph to be solved. After a number of steps, the graph is reduced to one node, and the positions of every link are known relative to ground. There are some mechanisms whose graphs cannot be solved in this fashion.

A set of base cases, composed of the RRR, PRR, and RPR mechanisms, is described in detail. Simple trigonometry yields closed-form solutions for these base cases. Therefore the whole plan represents a closed-form solution. Each base case typically has either two possible solutions, in which case the plan step must choose one of them, or it has zero solutions, in which case the mechanism breaks. A number of possible policies for these situations are presented.

Several earlier systems, most of them descendants of Sutherland's Sketchpad, are described and compared to the mechanism editor. Some implementation details are presented, including control structures for the user interface and for the planner.

Submitted in partial fulfillment of the requirements for a Master's Degree at the University of California, Berkeley, Department of Computer Science, under the direction of Professor Carlo Séquin.

*This material is based upon work supported under a National Science Foundation Graduate Fellowship.

Contents

1 Introduction

1.1 Motivation

Existing computer aided design tools are useful primarily in the late stages of design, when details are being filled in and drawings are being prepared. We are interested in building tools for the early stages of design, when the designer is exploring rough design concepts and evaluating qualitatively different alternatives. For computer programs to assist with these evaluations, they must work with descriptions of the *function* of the system being designed, and not merely the form.

The domain we consider is the design of mechanical linkages. Linkage design may be loosely separated into the phases of *type synthesis* and *dimensional synthesis* [Olson 85]. Type synthesis determines the types of the joints – revolute, prismatic, spherical, etc. – and the topology of the mechanism, while dimensional synthesis determines the distances between the joints, that is, the dimensions of the links. Once a designer has fixed the type of a mechanism, the dimensional synthesis problem amounts to numerical optimization in an n-dimensional parameter space, for some fixed n. Many programs have been written for this; most of them are specific to a single mechanism type [Olson 85].

We consider programs that assist the designer's exploration of the space of qualitatively different mechanisms. We call this *type exploration*, even though it involves not only the type of the mechanism but also its rough, qualitative dimensions (such as whether one link is longer or shorter than another). Effective and rapid type exploration requires a tool with a fluent interface for entering and modifying rough mechanism designs. The tool should allow the designer to view the motion of a mechanism, immediately and interactively. The designer is concerned first with function; the function of a mechanism is its motion.

1.2 The Mechanism Editor

This report describes a tool for sketching mechanical linkages and previewing their motion. The tool is an interactive computer program dubbed the *mechanism editor*. The editor runs on a color graphics workstation. It maintains a diagram of the mechanism on the screen, and provides menu- and mouse-based commands that manipulate the mechanism. With these commands, the user may add a link (that is, a rigid body), sketching it into the diagram with the mouse. The user may also form joints between links, and delete links and joints. More interestingly, the user may flex a joint, with the editor displaying the resulting motion of the entire mechanism.

The editor is limited to planar mechanisms. Links are essentially polygons, and joints are either revolute (pin joints) or prismatic (sliders). But it is intended that the ideas and methods presented here should be applicable to problems in three dimensions and with a wide range of joint types.

This report also describes how the tool does what it does, and compares the approach taken to those taken by other authors.

We begin with a scenario of the program being used to sketch and animate a few simple mechanisms.

1.3 A Scenario

1.3.1 Building A Four-Bar Linkage

Starting with a clean slate, the user chooses the "create new link" option from the menu. She moves the mouse to each vertex of the new link and clicks the left mouse button, except for the last vertex, where she clicks the right button. In three clicks, she has a triangular link, which is

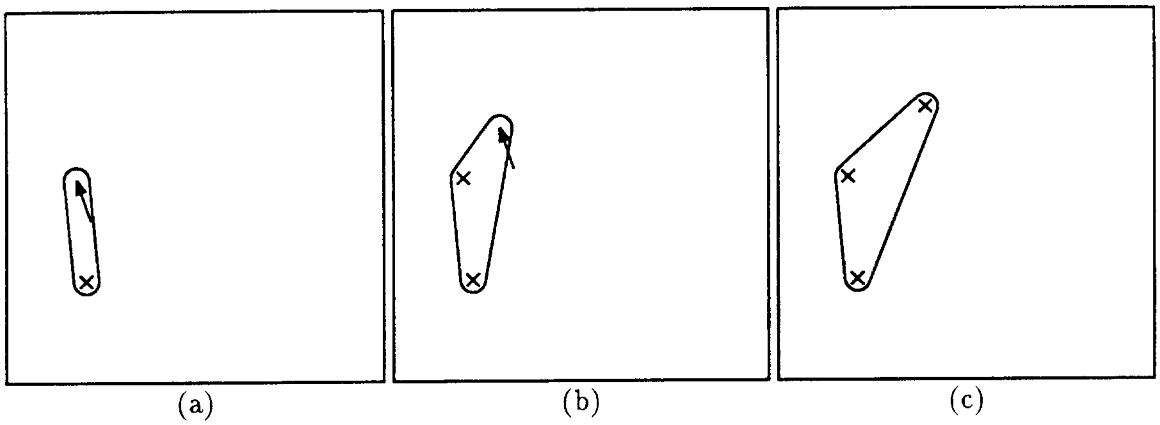


Figure 1: First, the user draws a triangular link. Mouse clicks are marked with an 'x'; current mouse position is marked with an arrow. (a) Just before the second click. (b) After the second click. The image rubber-bands as the mouse is moved. (c) After three clicks, the link is complete.

drawn on the screen as an outline of a triangle with rounded corners (Figure 1). While a new link is being created, the image "rubber-bands" as the user moves the mouse; the user can always see what shape she will get, before she clicks.

To draw the second link, the user chooses the same menu option, and again outlines a polygon with the mouse. But when the mouse gets near a vertex of the first link, the rubber-banding vertex snaps to the existing vertex, and the screen shows a circle around the two coinciding vertices, indicating that, if she clicks the mouse, the user will have formed a pin joint between the two links. Color is used to make the visual feedback easier to interpret. While a new link is being outlined, the new link is bright red and any new joints are bright orange. Existing joints are light green and each existing link has its own color selected from a palette of blues and darker greens. The user accepts the new link by clicking the right mouse button for the last vertex, and the new link and the pin joint are redrawn in the cooler colors. Figure 2 shows the two links, with the small circle that represents the pin joint between them.

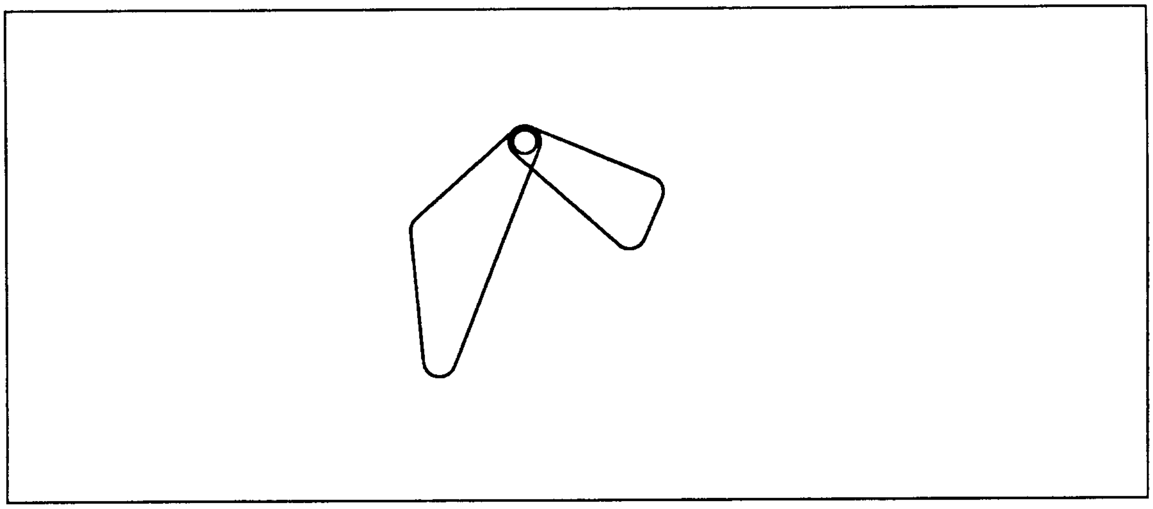


Figure 2: A second link has been drawn, connected to the first by a pin joint.

The user now draws a third link, pin-jointed to the second, in a similar manner. In this case, the new link has only two vertices. See Figure 3.

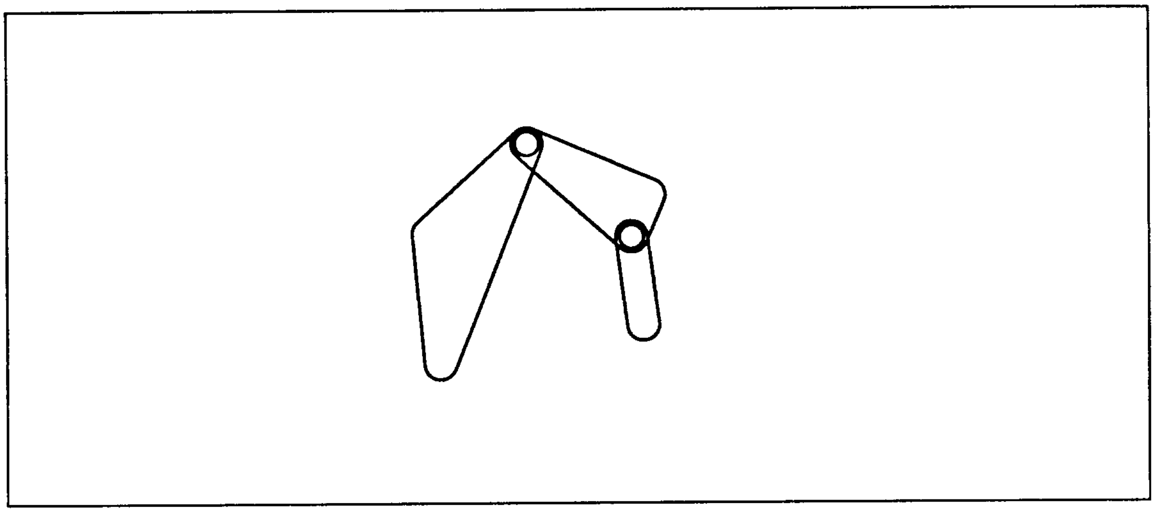


Figure 3: Three links, two joints.

As it stands, the mechanism is now floating freely in space. The next task is to root it to the ground plane. The user picks the "fixed pivot" option from the menu. Now, as the mouse gets close to a free vertex (that is, one that is not already part of a joint), the screen shows two concentric circles at that vertex, and the associated link lights up. The two circles indicate a fixed pivot, i.e., a pin joint between that link and the ground. The user clicks to accept, and the two circles go from orange to green. Figure 4 shows the mechanism after both the first and the third links have been pinned to ground.

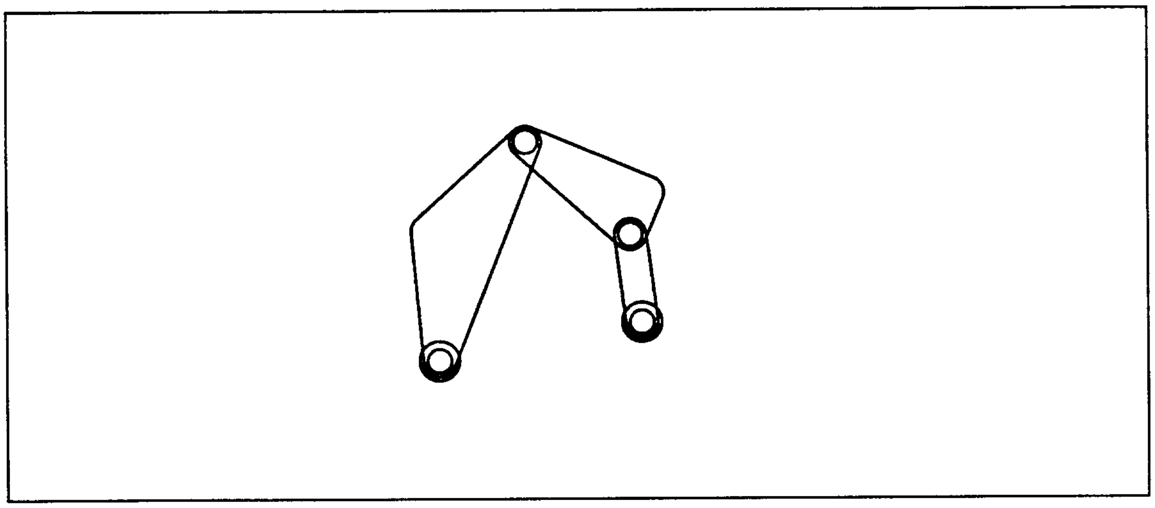


Figure 4: The four-bar linkage. Nested circles indicate fixed pivots.

This mechanism is called a *four-bar linkage* (the ground counts as one). It is a closed chain with one degree of freedom; depending on the relative distances between the joints, the links on the ends can pivot either back and forth along an arc or in a complete circle.

1.3.2 Animating the Linkage

To try this out, the user selects the "rotate fixed pivot" menu option. She moves the mouse close enough to a fixed pivot that a cross appears over it, and then clicks. Now the mechanism can be exercised, in real time. The cross is a sort of crank. By moving the mouse about the screen, the user causes the cross to rotate; the long arm always points at the mouse. The pivoted link always rotates with the cross, and the mechanism editor attempts to find a valid motion of the rest of the links as well – one that will keep all the joints joined and the fixed pivots fixed. Figure 5 shows the four-bar linkage being exercised. The four-bar linkage may be rotated from the other fixed pivot as well.

Figure 5 also shows some positions of the crank that the mechanism cannot reach (because the sum of the lengths of the second and third links is not enough to reach the end of the first link). This is discussed in Section 3.2.1.1.

1.3.3 Editing the Linkage

Once the user tires of twirling the four-bar around, and exits "motion mode" by clicking the mouse again, she may continue to edit the mechanism. Figure 6 shows a possible six-bar linkage, formed by adding two more links onto the center link of the four-bar linkage. The figure also shows the mechanism being animated.

The user now backs up to the situation where we had just two links (see Figure 2), via the "delete link" menu option. (Deleting a link also deletes any joints in which it participated.) She adds a third link as before, but using a prismatic joint, also known as a slider, instead of a revolute one. To do this, she first goes to one free vertex of the second link and clicks; this produces an orange pin joint, as before. But as she moves the mouse near the second free vertex on that same link, the pin joint turns into a slider. The slider involves two vertices on each link; it constrains all four of these vertices to be collinear. It is displayed as four parallel lines running between the two pairs of coincident vertices. (Like a pin joint, it is orange until accepted, and light green thereafter.) The user clicks the left mouse again, then clicks the right mouse on the third vertex of the polygon, accepting the link and the new joint.

The user now pins the free end of the new link to the wall with a fixed pivot, and once again selects the "rotate fixed pivot" option, in order to view the motion of the mechanism. The new mechanism and its motion are shown in Figure 7.

As the slider slides, its four parallel lines stay parallel, but the inner two move with one link while the outer two move with the other. This gives the appearance (in a rather schematic way) of a sliding member on runners.

Throughout all of this, the mechanism editor also maintains on the screen a textual status window. Something is printed in this window whenever the user interface changes state (which means every mouse click). By reading these messages, even a novice user can keep track of which mouse button does what at each point in time. The window also displays the number of links, number of joints, and number of degrees of freedom in the mechanism.

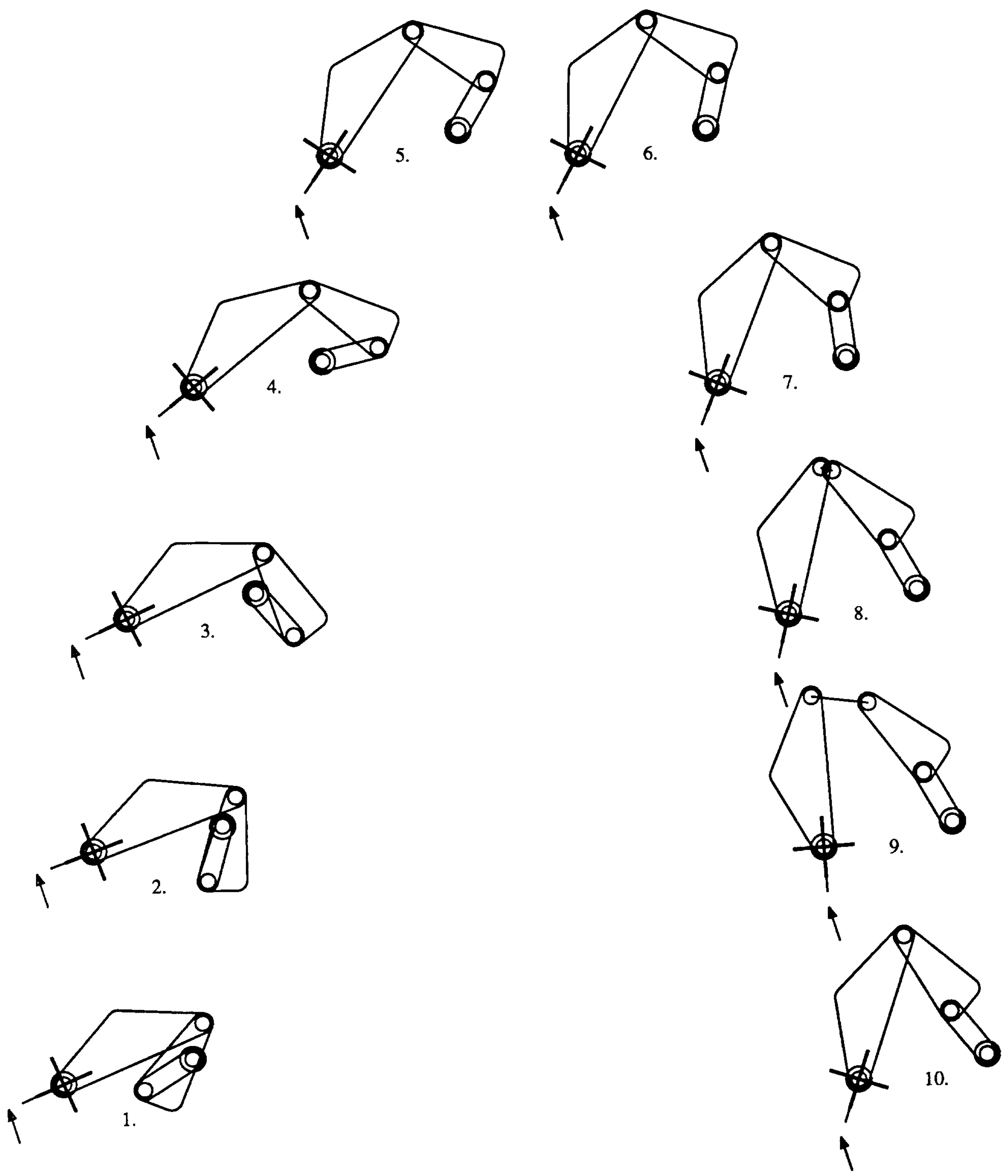


Figure 5: The four-bar linkage being moved. Also, the four-bar linkage being broken (frames 8 and 9) and then unbroken (frame 10).

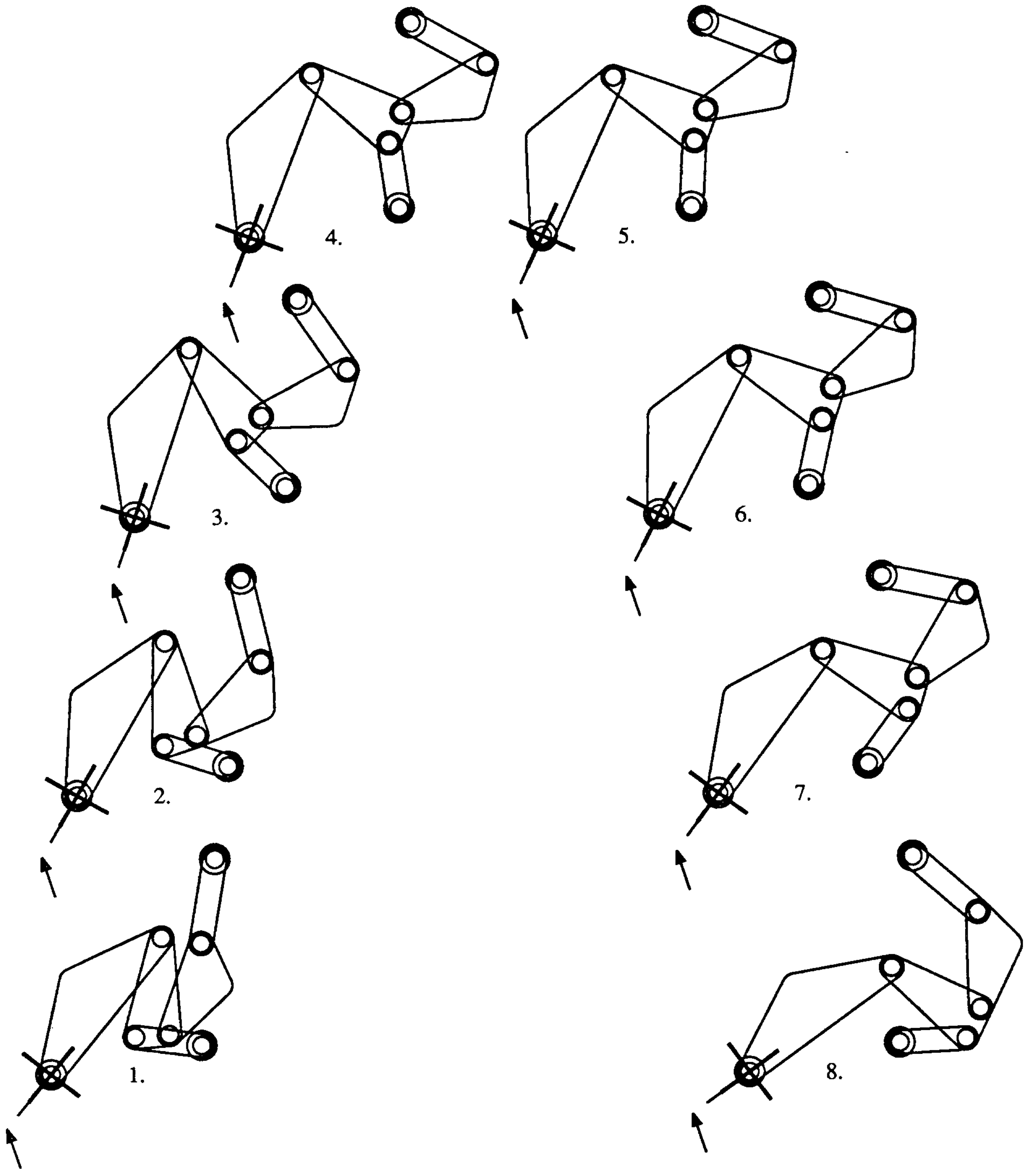


Figure 6: A six-bar linkage and its motion.

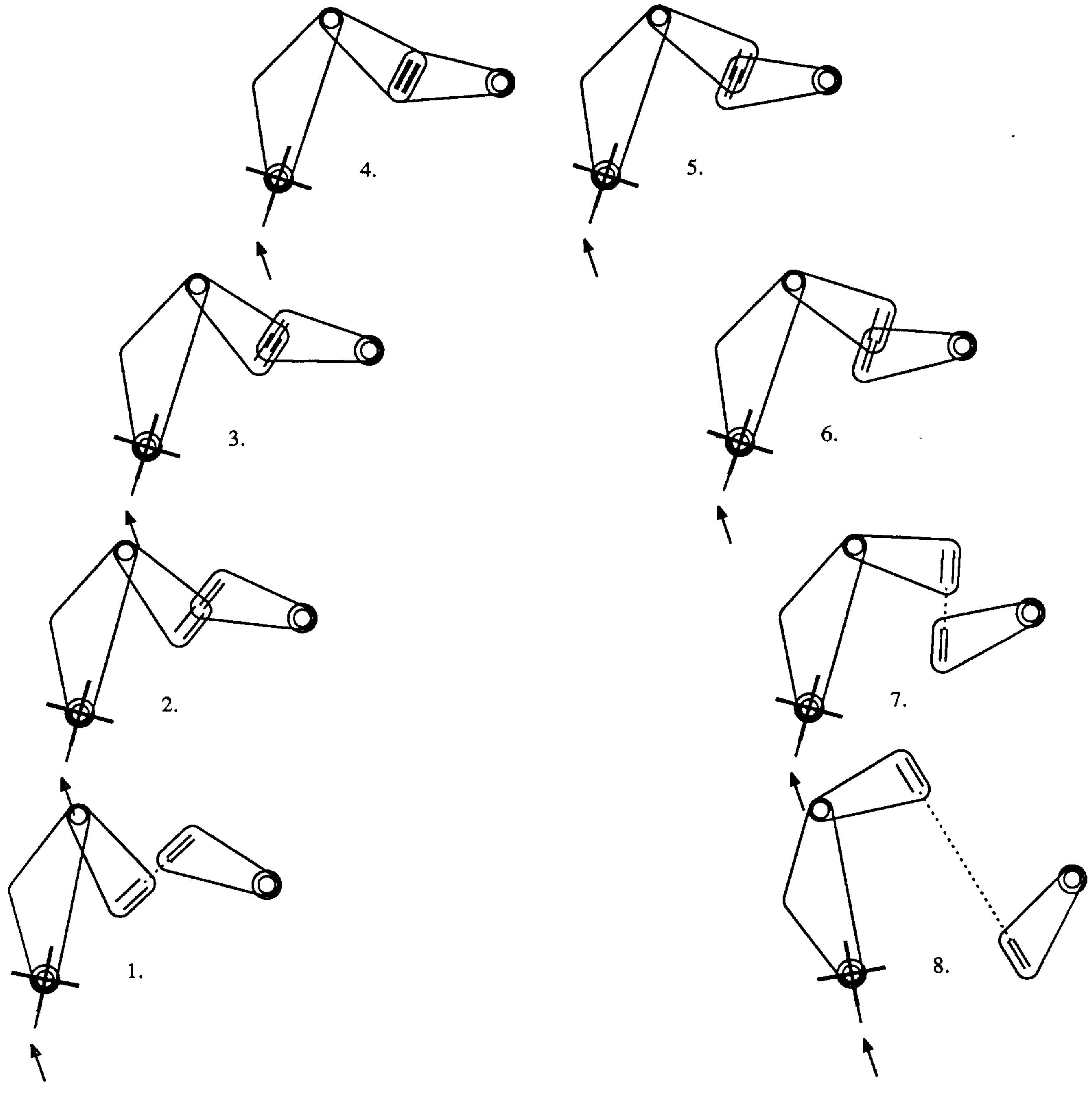


Figure 7: A sliding mechanism and its motion.

2 Mechanisms in the Abstract

2.1 Approach Definition

Convenient and rapid type exploration requires direct real-time interaction with the mechanisms under investigation. In order to provide this, the editor presents a *direct manipulation interface* to the user. The structures being edited have graphical representations on the screen, and, to a large extent, editing operations are performed by picking, dragging, and sweeping out these representations with the mouse.

In order to provide fast response of the mechanism to user input, the editor uses *closed form solutions* whenever possible. The alternative to closed-form calculations would be iterative numerical methods. A closed-form calculation evaluates a fixed expression that yields the exact answer (or rather, it would yield the exact answer if it were evaluated exactly). Iterative methods generate approximate answers, which improve as more calculations are performed; the total amount of calculation to be done is determined at run time. Since closed-form calculations are typically much faster than iterative ones, they are, at least superficially, the best suited to producing the smooth and rapid animations that we want.

The mechanism editor computes only the mouse-driven *kinematics* of the user's mechanism, not its dynamics. This means that there are no force terms nor momentum terms involved; we are simply computing the static positions of the links, given the positions of the input variables.[1]

The mechanism editor's user interface is inspired by Borning's constraint-based simulation program ThingLab (discussed in Section 6.3). Its use of closed-form kinematics solutions is modeled on Suh and Radcliffe's kinematics package LINKPAC (discussed in Section 6.1). Our work attempts to combine these ideas, to further automate closed-form solutions, and to analyze the results.

2.2 Building, Planning, and Moving

Following Borning's ThingLab, we break up the editor's work into three distinct phases. First, there is *build time*, while the user is modifying the topology of a mechanism, such as by adding joints or deleting links. The third phase is *move time*, during which the topology of the mechanism is fixed, but the user is interactively varying a continuous geometric property, such as a joint angle, and the editor is computing and displaying the resulting motion of the entire mechanism. In order to make the animations as smooth as possible, the move time computations should be as fast as possible. To speed the move time computations, there is a second phase, *plan time.* Once the user has requested an animation, and specified what parameters he will control with the mouse during the animation, the editor assembles a *plan*, to be executed repeatedly at move time. The plan is a sequence of function calls that, when made, bring the mechanism to a new, consistent state, based upon the mouse input. A plan is put together once (per animation) and then executed repeatedly, for as long as the user continues to manipulate the mechanism from the same parameters.

Plan time and move time for mechanism animations are analogous to compile time and run time for programs. Build time might be analogous to edit time, for program editors that constantly maintain a parse tree for the program being edited.

[1]The term kinematics also includes velocities and accelerations. The scheme described here could easily be extended to compute these, just as LINKPAC does; see Section 6.1.

2.3 The Mechanism Graph

In a sketch of a mechanism, particularly one made up of revolute joints, it is easy to see a sort of graph, with the joints as nodes and the links as edges (or multi-edges) connecting them. There is a dual to this graph that turns out to be more useful; it is the mechanism graph that has been used by mechanical engineers since 1964.

> The *graph* of a kinematic chain is obtained by representing each link by a *vertex* and each kinematic pair [i.e., joint] connecting two links by an *edge* connecting the corresponding vertices. [Olson 85]

(We will call the graph's vertices *nodes* in order to avoid confusion with the vertices of a link's polygonal shape.) The editor always maintains such a graph for the mechanism being edited. The graph changes only as a result of editing operations that change the mechanism's type; maintaining the graph is therefore done at build time.

Each link in a mechanism is a rigid body that may move over time. We can think of each link as having a *local coordinate system* in which the link's rigid geometry is described. To know the complete geometry of the mechanism at a certain time – and in particular to be able to draw this geometry – we need to know the transformation from each link's local coordinates to some known coordinate frame, such as world coordinates. Furthermore, a joint may be thought of as a *parameterized change of coordinates.* A pin joint, for instance, has some fixed geometry, namely its coordinates in the local coordinate systems of each of the two links that it joins. It also has a single variable parameter, namely its angle, measured as the angle between two reference directions, one on each link. Given a value of the joint parameter, the joint specifies the relationship between the links' two coordinate systems. Using the fixed geometry of the joint, a transformation from one local coordinate system to the other can be written as a matrix that depends on the joint parameter. The same is true of any type of joint: given the joint parameter(s), it determines the relation between its two links' local coordinate systems.

In this view, a node in a linkage's graph represents one link's coordinate system. (One can imagine a mechanism smoothly transmuting into its graph by imagining that everything rigidly attached to a particular coordinate system is contracting to a node.) Meanwhile, an edge of the graph represents a parameterized transformation between the two coordinate systems of the two nodes that it connects.

A joint can also be thought of as a constraint between the two coordinate systems. Thus the mechanism graph is the graph of the constraint system corresponding to the mechanism.

One coordinate system of interest is the world coordinate system, or in other words, the fixed ground frame. Accordingly, the mechanism graph always contains a node for the *ground link.* Unlike other links, the ground link does not have a corresponding polygon on the screen.

2.4 Degrees of Freedom

2.4.1 Grubler's Formula

The allowed motions of a link in a plane are rigid motions, and so are parameterized completely by three numbers: two for translation in the plane and one for rotation. In other words, placing a link in the plane has three degrees of freedom. Pin and slider joints each have one degree of freedom. Therefore, adding a pin or slider joint to a collection of links removes two degrees of freedom: the coordinate system of the second link would have required three parameters to specify, but now can be specified by using only one parameter – the joint parameter – plus the coordinate system of the first link. So, to determine the number of degrees of freedom (DOFs) in a linkage, we add three DOFs for each link, except for the ground link (which has no DOFs), and we subtract two DOFs for each one-DOF joint such as a pin joint or a slider. This gives us *Grubler's formula:*[Suh 78]

$$DOFs = 3 \cdot (links - 1) - 2 \cdot joints$$

For instance, the four-bar linkage has four links (including ground) and four joints; this give us one degree of freedom, as expected.

There are some special situations in which Grubler's formula is inaccurate. These are discussed in the next two sections.

2.4.2 Rigid Submechanisms

If a mechanism has zero DOFs, then it is rigid. What if Grubler's formula gives us a number less than zero? Then the mechanism is overconstrained. This can mean that a mechanism has been specified which cannot be assembled, or that a mechanism has been assembled that could not have been if the lengths involved had been slightly different. See Figure 8.

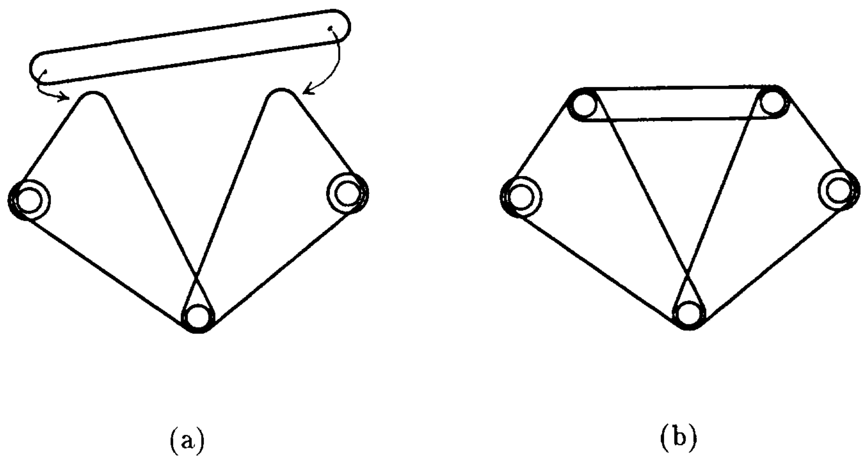


Figure 8: Minus one degrees of freedom. (a) This mechanism cannot be assembled. (b) This mechanism can be assembled, because the top bar has exactly the right length: it was constructed in place.

A remarkable advantage of the editor's direct-manipulation interface is that there is no way for a user to specify a mechanism that cannot be assembled. The user can only create *satisfied* constraint systems. (The exception to this is if a mechanism is assembled and then later broken by motion, as discussed in Section 3.2.1.1.) During initial construction, the danger of a mechanism being overconstrained poses no special difficulty to the editor.

The difficulty is that, if one of these overconstrained mechanisms is included as a submechanism in something larger, then Grubler's formula will fail. This happens because, while the overconstrained mechanism can move as a rigid body (DOFs = 0), it contributes negative DOFs to the formula. For example, consider pinning an extra link onto Figure 8; the extra link is free to move, so the mechanism does have a degree of freedom.

In order to detect and handle this situation, the mechanism editor groups any rigid set of links (DOFs $\leq$ 0) together into a super-link that we call an R-link (for "rigid link"). More precisely, an *R-link* is any maximal rigid submechanism. The editor maintains these maximal groupings along with the mechanism graph. Equivalently, we may say that it maintains two graphs, plus the mapping between them: the first is a graph of links, the second is a graph of R-links, and the mapping is a contraction (graph homomorphism) from the first onto the second. See Figure 9. It is

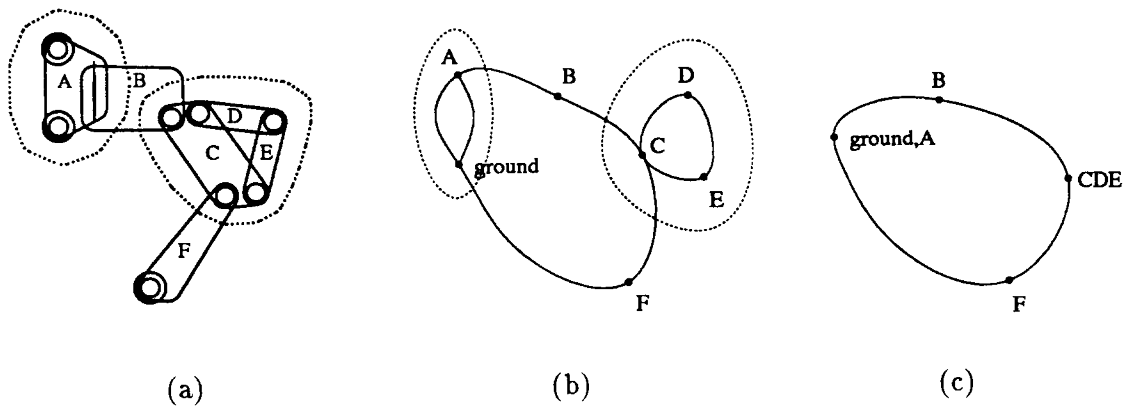


Figure 9: (a) A mechanism. (b) Its graph. (c) Its R-link graph. Rigid substructures, indicated by dotted lines, are contracted to single nodes in the R-link graph.

the second graph that is actually used for planning motions (and for printing the number of degrees of freedom on the screen). It has the property that, for any vertex-induced subgraph, Grubler's formula yields a positive number.

It is still useful to think of a node as representing a coordinate system, even in the R-link graph. Since the joints inside an R-link cannot flex, all the links within an R-link may be reduced to a single coordinate system in a manner that is constant regardless of any motion of the mechanism.

To let the user know when a rigid submechanism has been created, the mechanism editor uses colors. Links in the same R-link are drawn with the same color. Actually, this is not always visually striking, since the colors are all blues and greens and the image is only a line drawing. But at least it is a hint. One important special case is visually very clear: links that become part of the ground R-link (that is, links that become rigidly joined to the ground plane) are always drawn in grey.

The mechanism editor detects R-links by brute force. Whenever joints are added to the mechanism, the editor exhaustively tests all subsets of the set of R-links. If any subset is found to induce a subgraph with zero or fewer degrees of freedom, by Grubler's formula, then the subset is merged into a single R-link (and the subsets containing this new R-link are also checked). Of course, the running time of this algorithm is exponential in the size of the mechanism. There may be more efficient algorithms, but for the small mechanisms we usually draw (ten or fewer links), the brute-force method suffices. (One algorithm that does not work is checking the R-link graph only for three-cycles and two-cycles. This would catch the majority of rigid submechanisms that arise in practice, but would fail to detect the rigid three-legged table shown in Figure 10 below.)

2.4.3 Genericity

Even in the absence of overconstrained submechanisms, Grubler's formula will fail if certain coincidences occur. Suh and Radcliffe refer to these as "maverick mechanisms," and give the example of the three-legged table shown in Figure 10 [Suh 78].

It has six joints and five links, and so should be rigid. If the middle leg is longer or shorter than the other two, then the table is rigid. But if all three legs are the same length, then it moves with

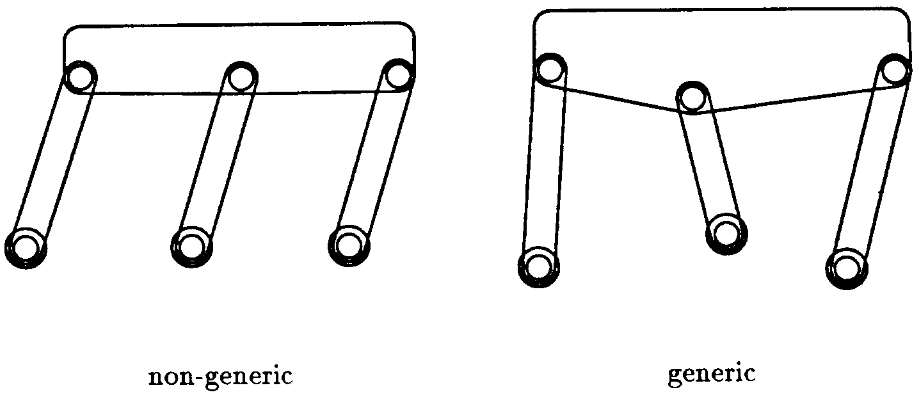


Figure 10: Three-legged table. The non-generic one can move, the generic one cannot.

one degree of freedom, just as if the middle leg were missing.

We avoid responsibility for such situations in the usual way, namely, by assuming that all our mechanisms are *generic*. That is, we assume that the mechanism we are analyzing is such that distorting its geometry by an arbitrarily small amount will not change its behavior in any discontinuous manner. The three-legged table is not generic, since distorting it slightly suddenly changes its degrees of freedom from one to zero. But the mechanism editor will not recognize this accident of geometry, and so will mistake the table for its generic cousin, the table with zero degrees of freedom. Another view of genericity is that it prevents the mechanism's constraints from being linearly dependent. Some ins and outs of this assumption are discussed in Section 5.2.4.

2.5 Contractions

With an understanding of graphs of mechanisms, we can search for closed-form solutions for mechanism motions. Inspired by Suh & Radcliffe's LINKPAC (discussed in Section 6.1), we seek to build up the solutions to larger mechanisms from the solutions to very small mechanisms. In other words, we proceed by induction.

How can we combine the solutions of small submechanisms? Solving a submechanism means computing the free parameters of its joints. With the joints' parameters fixed, all the coordinate systems of the submechanism's links are in known relation to one another. Much like a rigid submechanism, then, *a solved submechanism may be considered a single link* while the rest of the mechanism is being solved.[2]

In terms of graphs, this means that solving a submechanism lets us *contract* the corresponding subgraph to a single node. The edges in the subgraph disappear, as do all but one of the nodes; all edges going into the subgraph from the rest of the graph now go into that single node. The

[2]The difference is that, in a rigid submechanism, the relative positions of the links are constant over time, whereas the relative positions within a solved submechanism are functions of the mouse input. There are two equivalent ways to look at this. The move-time view is that, during a particular move cycle when we are given particular mouse values, the relative positions of the solved links are fixed, and thus they may all be treated as a single link. The plan-time view is that, during planning, the solved subsystem is like a single link whose lengths and angles are described by possibly complicated functions rather than by constants.

smaller, contracted graph is the graph of the mechanism that now remains to be solved. Solving a submechanism reduces the problem of solving the original graph to the problem of solving a contracted graph.

These contractions arise in two ways. Firstly, the user may fix the free parameter of a joint, e.g., by flexing the joint with the mouse. This contracts the two links being joined into a single link, eliminating the now-fixed joint from the graph to be solved. Secondly, the editor may solve for the joint parameters of a small submechanism, if that submechanism matches one of the *base cases* that the editor knows how to solve. This contracts the small submechanism into a single link.

2.6 Planning and Moving

At plan time, we construct a plan to be executed repeatedly at move time. Each cycle through the plan computes a position of the mechanism, based on the joint parameters that the user is controlling with the mouse. (Alternatively, the user could control the position and/or orientation of a link with the mouse. This is equivalent to controlling the parameters of joints between that link and the ground plane.) These joint parameters are the *input* to the move cycle. In order to display the new state of the mechanism, that is, a state consistent with the input parameters, the editor must compute the remainder of the joint parameters, or, equivalently, the transformation from each link coordinate system to the ground link system. These transformations are the *output* of the move cycle.

So, at plan time, we construct a list of expressions for computing the outputs of a move cycle from its inputs. Each expression results from applying a base case solution to a small submechanism in the graph. When we apply a base case solution, we contract the graph of the mechanism. We keep doing this until the entire graph is contracted to a single node. At this point, we have a list of expressions for reducing everything, including the ground link, to a common coordinate system. Applying this list of expressions – executing the plan – solves the mechanism.

This chapter has described a few abstractions – mechanism graph, Grubler's formula, rigid submechanism – and has described the algorithms used by the mechanism editor. These algorithms and abstractions are quite general. They are applicable to mechanisms with joint types other than just rotational and prismatic, to mechanisms that include "black box" inputs or transformers, and to mechanisms in three dimensions. (Mechanisms that these methods do not handle are discussed in Sections 5.2.)

3 Planning, Solving, Breaking, and Branching

3.1 The Planner

In order to keep our implementation as simple as possible, while still demonstrating the basic ideas, we restrict the motion animation problem to the case where there is one degree of freedom in the mechanism, one input parameter, and that parameter is the joint angle of a fixed pivot. (A fixed pivot is a rotational joint between a link and the ground.)

The motion problem may be viewed either as the problem of finding the joint parameters or as the problem of finding the link coordinate systems. Since the link coordinate systems – that is, the link-to-world coordinate transformations – are what is needed for drawing the links on the screen, we consider these transformations to be the output of executing a plan.

The planning algorithm, then, is as follows. We begin by taking the R-link graph for the mechanism, marking the ground link as fixed, and marking all other links as unfixed. We will mark each R-link *fixed* as we output a step of the plan that will update that link's link-to-world coordinate transformation, or in other words, a step that will fix the link's coordinate system relative to the ground system. This also keeps track of the current contraction of the R-link graph. At any time, the fixed R-links are exactly those that have been merged with the ground R-link. Unfixed R-links have not been merged with anything. (Merges into the ground link are the only merges that this data structure can represent. Because we are solving only a restricted motion problem, this is sufficient.)

Each step of the plan is a function call; the function call is to be made each time the plan is executed. The first step is a call to the routine for obtaining the input parameter. This routine obtains the joint angle of the input pivot from the user's mouse, and from this computes the coordinate system of the pivoted link. Since this step will correctly update the coordinate system of the pivoted link, we output this step, and mark the input link fixed.

We then examine the contracted R-link graph, looking for a subgraph that will match a base case that we know how to solve. Since our next step is to solve the subgraph completely, it must be a rigid subgraph. In fact, as will be explained in Section 3.2, the graph of each base case is a triangle (a 3-cycle made up of three links and three joints). So we may restrict our search to triangular subgraphs.

We may further restrict our search to subgraphs containing the ground node. This is because a rigid subgraph that does not contain the ground node would be made up only of unfixed R-links, and would therefore appear as a rigid subgraph in the uncontracted R-link graph. This would contradict the definition of R-link.

In sum, as we search the contracted graph for a subgraph to match to a base case, we need only look at triangular subgraphs that include the ground node. In the uncontracted graph, such a subgraph appears as two unfixed R-links that are connected to each other, and that are each connected to some fixed R-link.

We may label a triangular graph by the three joint types (revolute or prismatic) of its three edges. A triangular subgraph matches a base case when the type of each joint of the subgraph matches the type of the corresponding joint of the base case. If we find a subgraph that matches a base case, we pass it to the base case *solver*, which returns the next step in the plan. This step is a function that, when called, will solve the subgraph, using the geometry of its links and the (already updated) coordinate transformations of the fixed links to update the link-to-world coordinate transformations of the two unfixed R-links in the triangle. Having output this step, we can now mark these two R-links as fixed. This is equivalent to contracting the triangular subgraph into the ground node. (Since the triangle has zero degrees of freedom, this operation preserves the

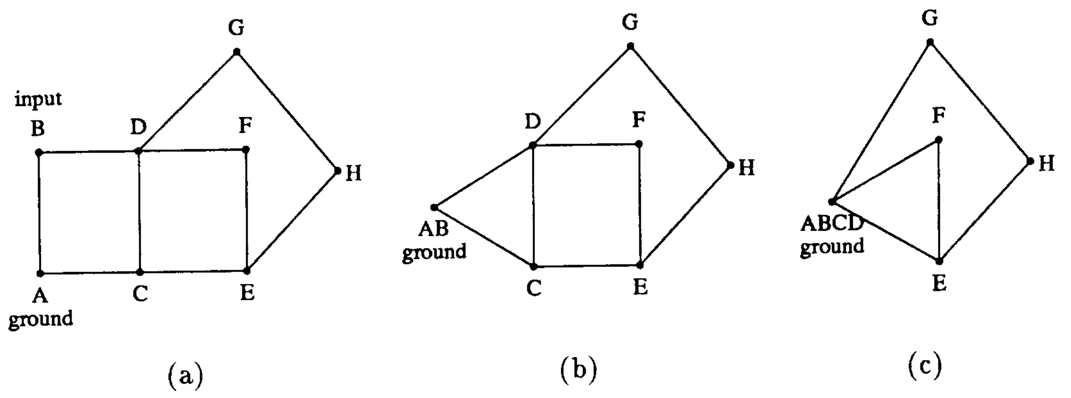


Figure 11: Solving a mechanism graph. (a) The uncontracted graph. (b) The contracted graph, after B is fixed by the input step. (c) The contracted graph, after C and D are fixed by the next step.

number of degrees of freedom in the contracted graph.)

We continue this process, at each stage identifying an application of a base case and using it to produce one more step in the plan. If all goes well, contracting the input joint creates a triangle in the graph, and each time we contract a triangle, a new triangle is created. This continues until all R-links are marked fixed, or in other words, until the whole mechanism has been resolved against the ground plane. At this point, the plan is complete; executing it will solve the mechanism. If after some steps there is no triangle, or there is no triangle that matches a base case, then we are stuck, and the plan is incomplete. These cases are analyzed in Section 5.2.

Figure 11 shows an example of this process. The uncontracted graph is shown in Figure 11a. At first, only the ground node – node A – is fixed. The first step fixes the input link, whose node is node B. Figure 11b shows the contracted graph at this stage. Nodes C and D now form a triangle with ground. (This can be seen in the uncontracted graph as well, where they are connected to each other, and they are each connected to some fixed link.) Assume we have a base case that matches the triangle AB-C-D. Then the next step will fix nodes C and D. With A, B, C, and D fixed, the contracted graph now looks like Figure 11c. Now nodes E and F form a triangle with ground. Once E and F are fixed, then G and H form a triangle with ground. Fixing G and H completes the solution of the mechanism.

We have described these techniques as if the base case mechanisms will have exactly one solution for each set of input values. But in fact they will almost always have either zero or two solutions. We term these situations *breaking* and *branching*, respectively, and discuss them in Sections 3.2.1.1 and 3.2.1.2.

Again, we have restricted the input to be a single, grounded, revolute joint only for simplicity. With only minor extensions to the above algorithm, these restrictions may be dropped. The extensions required are described in Section 5.2.1.

In the course of contracting the R-link graph, could we create an overconstrained subgraph? In particular, might we need solvers for two-link, two-joint mechanisms? A simple induction argument shows that the answer is no. Any overconstrained subgraph that could be present after a contraction would have to correspond to an overconstrained subgraph that was present before the contraction.

3.2 Base Cases

This section describes the small mechanisms that we solve directly in closed form. From these solutions, we build up plans for solving more complicated mechanisms.

Since we will solve these linkages completely, they must have zero degrees of freedom. A rearrangement of Grubler's formula shows that any mechanism with zero degrees of freedom must satisfy

$$3 \cdot (links - 1) = 2 \cdot joints.$$

Aside from the trivial mechanism of one link and zero joints, the smallest such mechanisms have three links and three joints. In order for them to have no overconstrained submechanisms, their graphs must be triangles.

We name each base case according to the joint types on its three edges, using 'R' for revolute joints (pins) and 'P' for prismatic joints (sliders). As stated in the previous section, one node of the triangle is always the ground node; we list the joint types by starting at the ground node. Considering symmetry, there are only six possible base case mechanisms: RRR, PRR, RPR, PPR, PRP, and PPP. We implement only the first three of these, on the grounds that the cases with more than one prismatic joint do not seem to arise very often. (Extra base cases may be added to the system, as described in section 3.2.4.) Figure 12 shows mechanisms for the PRR and RPR cases.

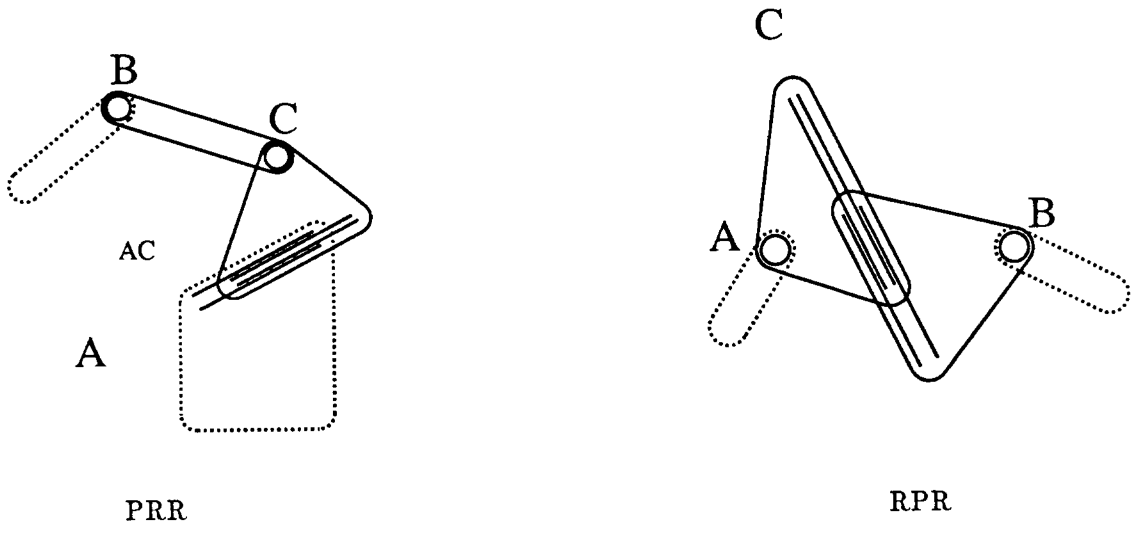


Figure 12: PRR versus RPR. The dotted links are already fixed.

The ground node is special here not so much because it is ground, but because it is the one node that may be the result of contractions, and may therefore be made up of several R-links. This changes the set of computations that may be performed at plan time, as opposed to move time. For instance, the distance between two points on the same R-link is constant throughout a motion, whereas the distance between two points on different R-links may change as the mechanism moves, even though both R-links are "fixed" by earlier steps of the plan. The former distance may be computed at plan time, whereas the latter must be computed later, at move time.

Designing solvers is a straightforward matter, involving only simple geometry and some bookkeeping. With the planning problem solved, the largest issues that remain in the design of the mechanism editor are what to do about *breaking*, which is when the geometric limits of a mechanism are exceeded, and *branching*, which is when there are two (or more) configurations of the mechanism that satisfy the input conditions and joint constraints. We discuss these issues below in the contexts of the individual solvers, where these situations are detected.

3.2.1 RRR

The prototypical situation for the RRR solver is shown in Figure 13. The summary is that, because the lengths of all three sides of the triangle are known, the law of cosines may be used to solve for the three angles. The details follow. The four-bar animation in Figure 5 uses the RRR solver.

The three joints are labeled A, B, and C, such that A and B are the joints connected to the ground node. The R-link connecting joints A and C is labeled link-AC, and similarly for link-BC.

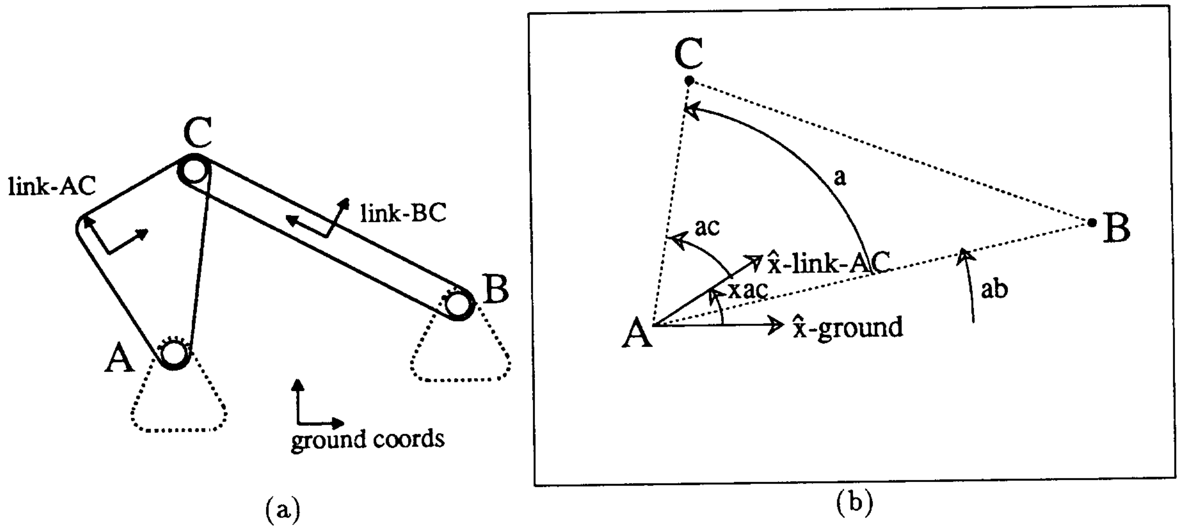


Figure 13: Notation for the RRR solver. The dotted links are already fixed.

Each of these two R-links has its coordinate system; the goal is to find the transformations from these systems to ground coordinates. The R-link geometries are constant, so that in particular we know the positions of A and C in link-AC coordinates, and the positions of B and C in link-BC coordinates. There is no R-link link-AB because joints A and B may be connected to different fixed R-links, but in any case, the positions of joints A and B in ground coordinates are known at move time (though not at plan time).

We focus first on solving for the position of R-link link-AC. Because the ground coordinates of joint A are known, it suffices to compute the rotation of the link-AC system. We solve for the angle that the link-AC x-axis forms with the ground frame x-axis, using the simple relation

$$\angle XAC = \angle AB + \angle A - \angle AC$$

where

$$\begin{aligned}
\angle XAC &= \text{angle from ground x-axis to link-AC x-axis} \\
\angle AB &= \text{angle from ground x-axis to ray } \vec{AB} \\
\angle A &= \text{angle from ray } \vec{AB} \text{ to ray } \vec{AC} \\
\angle AC &= \text{angle from link-AC x-axis to ray } \vec{AC}\ .
\end{aligned}$$

Of these, $\angle AC$ is constant and $\angle AB$ is known at move time. Angle $\angle A$ is computed from the law of cosines:

$$\cos \angle A = \frac{AB^2 + AC^2 - BC^2}{2 \cdot AB \cdot AC}$$

where

$$\begin{aligned} \mathsf{AB} &= \text{distance from A to B} \\ \mathsf{AC} &= \text{distance from A to C} \\ \mathsf{BC} &= \text{distance from B to C .} \end{aligned}$$

Length AC is a constant easily computed in link-AC coordinates, BC is a constant easily computed in link-BC coordinates, and AB is easily computed at move time in ground coordinates.

The equations for fixing the link-BC system are analogous:

$$\angle \mathsf{XBC} = \pi + \angle \mathsf{AB} + \angle \mathsf{B} - \angle \mathsf{BC}$$

$$\cos \angle \mathsf{B} = \frac{\mathsf{AB}^2 + \mathsf{BC}^2 - \mathsf{AC}^2}{2 \cdot \mathsf{AB} \cdot \mathsf{BC}}$$

where

$$\begin{aligned} \angle \mathsf{XBC} &= \text{angle from ground x-axis to link-BC x-axis} \\ \pi + \angle \mathsf{AB} &= \text{angle from ground x-axis to ray } \vec{\mathsf{BA}} \\ \angle \mathsf{B} &= \text{angle from ray } \vec{\mathsf{BA}} \text{ to ray } \vec{\mathsf{BC}} \\ \angle \mathsf{BC} &= \text{angle from link-BC x-axis to ray } \vec{\mathsf{BC}} \text{ .} \end{aligned}$$

3.2.1.1 Breaking

Since an arccosine function is used by the solver, we must ask when its argument will be out of the $[-1, 1]$ range. Algebraically, this happens when the triangle inequality

$$\mathsf{AB} + \mathsf{AC} \geq \mathsf{BC}$$

is violated. (This is for the $\angle \mathsf{A}$ computation.) Geometrically, this means that a triangle with sides of the given lengths cannot be assembled. And physically, it means that the user has turned the input crank past the limits of where the mechanism can reach. We say that the mechanism is *broken*; an example is shown in Figure 5.

The mechanism editor implements a fairly simple policy regarding breakage. The joints connecting the fixed portion of the mechanism to the as yet unfixed portion are flagged as broken, and the rest of the mechanism is not solved. That is, if some step of the plan finds that the geometry of its submechanism cannot be solved, then the rest of the plan is not executed, until, perhaps, the whole plan is executed again during the next move cycle. The unfixed portion remains stationary in its previous configuration. All links are still displayed, and broken joints are displayed with bright red lines indicating the violation of their constraints. If the user should move the input crank back into the range within which the mechanism can be assembled, then the joints will become unbroken, the red lines will vanish, and the rest of the mechanism will move again. But the user is not required to do this. The user may even end the motion with the mechanism in a broken state, and go on from there to edit the mechanism or to initiate other motions.

Several other possible policies are discussed in Section 3.2.5.

3.2.1.2 Branching

When the mechanism is not broken, the arccosine argument will normally be in the open interval $(-1, +1)$, in which case arccosine will have *two* possible values, positive and negative. These correspond to the two possible ways of placing link-AC and link-BC with joints A and B fixed:

they may be placed either above $\vec{AB}$ or below it. This problem is often called *branching*, and the alternative solutions of the mechanism are called branches, or *modes of assembly* [Suh 78].[3]

Which branch does the user want? Most often *continuity* is enough of a guide: if only one of the two possible solutions is close to the previous position of the mechanism, then that value is presumably preferable to one that would cause the animation to snap suddenly to a very different configuration. So for example, if an angle's previous value was $+45°$, and its possible next values are $\pm 50°$, then $+50°$ is the clear choice.

The trouble comes near the boundary. If the previous value is $+4°$, and the possible next values are $\pm 2°$, then it is not so clear which is desired. The canonical example is, again, the four-bar mechanism shown in Figure 5. The right crank, used as an input, may be turned through full revolutions without the mechanism breaking. Furthermore, the configurations of the left two bars is unambiguous: the mode of assembly remains constant. But the left crank can be turned only through a limited arc. If the user is to be able to put the right bar through a full revolution using the left crank as input, it must be by moving the crank to the end of its arc, reversing direction, and moving it back along the arc. Near the ends of the arc – that is, near the breaking points – the mode of assembly of the right two bars is ambiguous. (Algebraically: near $+1$ and -1, arccosine is $0°$.) In fact, the mode of assembly must change, if there are to be full revolutions.

Since the regions of ambiguous branching are adjacent to regions of breaking, the RRR solver currently implements the policy of *changing branches after breaks.* That is, if the user turns the left crank past the breaking point, and then back into the non-breaking zone, the right portion of the mechanism will "flip over" to the other mode of assembly; if the user never goes past the breaking point, then the mechanism will never change modes. The physical rationalization for this is that the device would have to go through the $0°$ point (where the bars are collinear) in order to change modes, and that if the user wants to go all the way to $0°$, he signals this intent by in fact going somewhat past this point. Thus, to make full revolutions using the left crank, the user may oscillate the crank, going slightly past the breaking point at each end on each oscillation.

In general, branching policy is determined for each base case individually, though in practice all the solvers we discuss here use analogous policies. A couple of other possible policies are mentioned in Section 3.2.6.

3.2.2 PRR

The prototypical situation for the PRR solver is shown in Figure 14a. The crank on the left is not part of what is being solved; it is there to form an example of a complete mechanism that uses the PRR solver. This mechanism shows the slider's fixed link as being grounded, but of course in general that link may move. In other words, it is fixed at move time, but not necessarily at plan time.

The solver fixes the coordinate frames of two R-links. One goes between the fixed revolute joint B and the unfixed revolute joint C. The other goes between joint C and the fixed prismatic joint A. The prismatic joint constrains some particular line on that link to be collinear with a line of known position. Joint C is thereby constrained to lie a constant distance AC away from this known line (this distance, and its sign, being determined by the link geometry). The other link constrains joint C to lie on the circle of constant radius BC centered on joint B. This places joint C at the

[3]The term "branching" presumably refers to the branches of the complex log function, and of functions (like arccosine) that can be expressed in terms of log. Strictly speaking, there are infinitely many potential values for arccosine, but these values represent only two distinct angles. In practical terms, since our ultimate use for these angles is in rotation matrices, we do not actually need the arccosine so much as the sine of the arccosine; this has only two possible values.

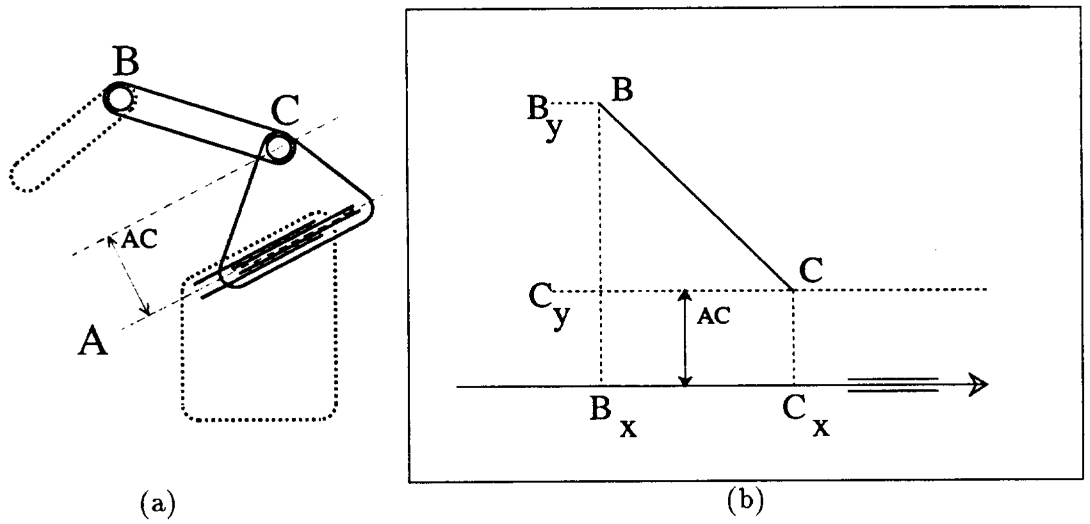


Figure 14: (a) Notation for the PRR solver. The dotted links are already fixed. (b) B and C in the slider coordinate system.

intersection of a line and a circle. This information is enough to compute the position of joint C, which in turn suffices for computing the positions of the coordinate frame of the two links that it joins.

The position of joint C is easiest to compute in a coordinate system whose x-axis lies along the line of the slider joint. In this system,

$$\begin{aligned} C_y &= \mathsf{AC} \\ C_x &= B_x \pm \sqrt{\mathsf{BC}^2 - (B_y - C_y)^2} \end{aligned}$$

where

C_x, C_y	=	coordinates of joint C in the slider system
B_x, B_y	=	coordinates of joint B in the slider system
AC	=	distance from joint C to the line of slider joint A (constant)
BC	=	distance from joint C to joint B (constant).

The equation for C_x is just the Pythagorean Theorem applied to the right triangle shown in Figure 14b.

3.2.2.1 Breaking

If joint B strays farther than AC + BC from the slider's line, then the two links together will not be able to reach the slider, and the mechanism will break. In this case (and this case only), the argument for the square root in the formula above will be less than zero. When this happens, joints A and B are flagged as broken, and the remainder of the plan is not executed.

3.2.2.2 Branching

When its argument is positive, the square root has two opposite values, corresponding to whether joint C is to the right or the left of joint B. This is very similar to the RRR case, in that the regions of ambiguous branching are adjacent to the regions of breaking. (They are separated by the two points at which the square root argument is zero; here the line BC is perpendicular to the slider.) The PRR solver is therefore built to implement the same branching policy as in the RRR case: it switches branches only after the mechanism breaks (that is, just as it is brought out of a broken state into an unbroken state). So by oscillating the crank in Figure 14, but somewhat exceeding the limit of the mechanism on each oscillation, the user may easily shuttle the A-C link back and forth, to either side of joint B.

3.2.3 RPR

In this case, neither side of the prismatic joint is yet fixed (Figure 15). Again, the two cranks are

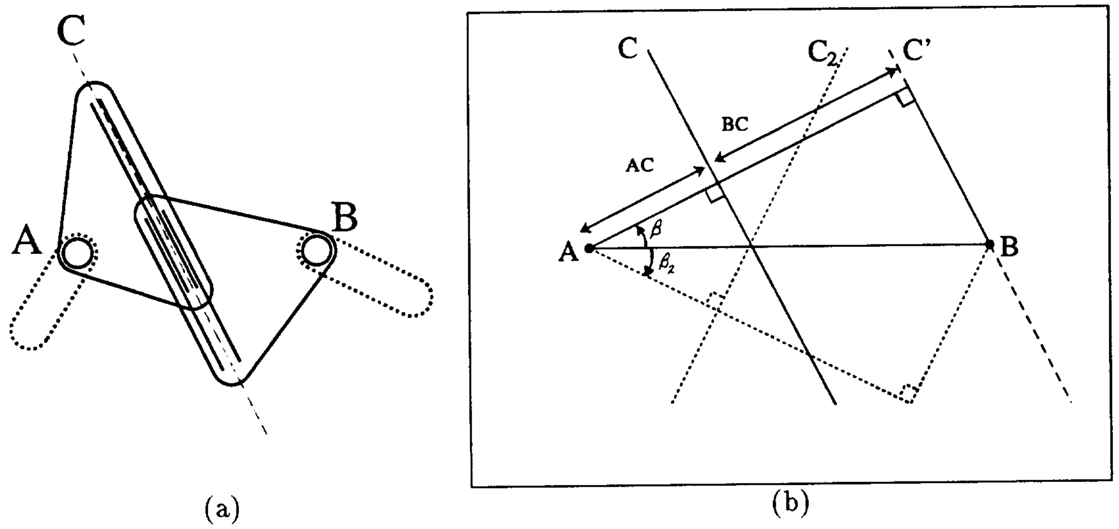


Figure 15: (a) Notation for the RPR solver. (b) Construct the line through B parallel to C.

there simply to show a possible complete mechanism, and will be, if not grounded, at least fixed before the RPR solver is called.

Line C is constrained, by the link geometries, to be at a perpendicular distance AC from joint A and at a perpendicular distance BC from joint B. AC and BC are constants. The distance AB is known (though not constant). By constructing the line C' through B parallel to C, we can form a right triangle of hypotenuse AB and side AC + BC; see Figure 15. We then have the angle

$$\beta = \arcsin \frac{AC + BC}{AB}$$

which is the angle from ray $\vec{AB}$ to the direction perpendicular to the slider. From this, we can compute the slider's orientation in the world frame; from this, and the fixed point from the revolute joint on each link, we can compute the links' orientations and positions in the world frame.

The distances AC and BC are signed; choosing an arbitrary, consistent orientation for the slider line C puts each of joints A and B consistently on either the left (+) or the right (−).

3.2.3.1 Breaking

The RPR mechanism cannot be assembled when joints A and B are too close together; specifically, the arcsine has no real value when

$$|\mathsf{AB}| < |\mathsf{AC} + \mathsf{BC}|.$$

In the limiting case, the slider line is perpendicular to the line $\overline{\mathsf{AB}}$. The mechanism can be solved with joints A and B arbitrarily far apart; the slider line approaches $\overline{\mathsf{AB}}$ in this case. (Of course, a more realistic device would have joint limits at which the slider stopped, or fell apart.)

3.2.3.2 Branching

The branching of the RPR assembly is again analogous to that of the RRR and PRR assemblies. Drawing the triangle in Figure 15(b) with its AC + BC side below $\overline{\mathsf{AB}}$, rather than above it, shows a second solution, with

$$\beta_1 = -\beta_2$$

where the angle β has value β_1 for the first solution, and β_2 for the second. These correspond to the two values of arcsine. In the limiting case, β_1 and β_2 equal 0°, so that the slider axis is perpendicular to $\overline{\mathsf{AB}}$; this is also the limiting case for breaking.

We again use the policy of switching branches after breaking. The user may rock joint A closer and farther from joint B; if A is moved close enough to B to break the assembly, then both branches may be explored.

3.2.4 Other Solvers

RRR, PRR, and RPR are the only three solvers currently implemented in the mechanism editor. These base cases have proven sufficient for a good deal of experimentation. Of course, other labeled triangles can be considered, solvers can be written for them, and mechanisms can be designed whose motion requires these solvers. This is especially true if additional joint types are added to the system, such as gears or pulleys.

Adding a base case to the mechanism editor is no more complicated than writing a solver for it. Since the planning algorithm simply calls solvers based on the labeled triangles that it finds, incorporating the new solver merely requires adding a case to one multi-case conditional statement.

3.2.5 Breaking Policy

In general, a given input joint can only be turned so far before the mechanism's geometric constraints cannot be solved, and the mechanism breaks. How should the editor respond when an input joint is turned past this limit? The goal is to provide the user with clear feedback as to how far the mechanism can go and why it can go no farther. (Of course, we want to do this while maintaining interactive speeds.) Our current policy does reasonably well, displaying each broken joint in bright red.

A significant drawback of our current policy is that it does not show the actual limit configuration of the mechanism. After the discrete motion step that crosses the limit, the broken portion of the mechanism is left immobile in its last position before the break. (This is described above in Section 3.2.1.1.) If the animation is reasonably smooth, then this position will be near the limiting one. But how near is near? In the case of a broken four-bar, for instance, it is surprising how close the two links can be to being straight without it being obvious that they cannot reach the input lever. (Cf. Figure 5.) We have displayed the fact that the linkage is broken, but because we have not displayed the limit position, the break is not visually convincing.

The current policy allows a mechanism to be moved from an unbroken state, through a series of broken states, to a new unbroken state that could not have been reached from the original position by any smooth path. This is good in that it allows more states to be explored. But it is potentially confusing for the user.

This confusion, as well as the problem of not displaying limit positions, are exacerbated if the motion is not smooth. The current implementation is quite slow (1-2 updates per second), and it requires great discipline on the part of the user not to introduce large mouse motions between updates. Of course, the worst case of this is during the all-too-frequent garbage collections. In any case, large mouse motions may jerk the mechanism wildly about, making it hard for the user to interpolate how the mechanism arrived at its new state. This is especially bad if the mechanism breaks, or if it goes from one unbroken state to another, having passed over an intermediate broken state.

Since our algorithms require so little computation, the mechanism editor could be implemented to run at video rates. (This is discussed in Section 4.5.) With scarcely any lag time between moving the mouse and seeing the effect on the mechanism, the user could more easily avoid discontinuous motions. This would eliminate the confusion in most practical cases. Theoretically, the danger of discontinuous animations would still exist, since mechanisms can be designed that amplify input motions to an arbitrarily large extent.

For systems with slow update rates, or sensitive mechanisms to animate, a limit could be imposed on the amount of motion per update. When the user moved the mouse too quickly, the mechanism would advance towards the new mouse position in steps. The size of each step might be restricted by a limit on the motion of the input joint. A more sophisticated technique would set a limit on the distance moved by any vertex in the mechanism during one step. Any motion that moved a vertex too far would not be displayed, but instead smaller step sizes would be tried, until a small enough motion was found. Restricting the step size has the undesirable effect of letting the mouse appear to be detached from the mechanism, but this may be preferable to allowing an animation to appear discontinuous.

We now consider four alternative policies for handling input motions that cause a mechanism to break: limiting, searching, stretching, and approximation.

Limiting. The simplest possible policy is to *limit* the mechanism to unbroken positions. Any motion step that would break the mechanism is rejected. The entire mechanism, including the input link, remains stationary, until the mouse is moved back into a range that moves the input joint without breaking anything. This policy has an elegant simplicity, but little else to recommend it. It does not display the breaking point of the mechanism, and it does not allow any control of the mechanism beyond the breaking point.

Searching. Instead of simply rejecting a motion that is out of range, we could *search* for the breaking point. Under this policy, an input motion that would break the mechanism would not be accepted, but instead a motion half as big would be tried. If this motion succeeded – that is, nothing broke – then this half motion would be accepted and displayed, and a motion three quarters the size of the original would be tried. In this way, a binary search is conducted to locate the mechanism's limit position. With only the unbroken states being displayed, the user, having just made a large mouse motion, would see the mechanism ooze monotonically into its limiting configuration. The search would quickly converge to within the tolerance of the graphical display.

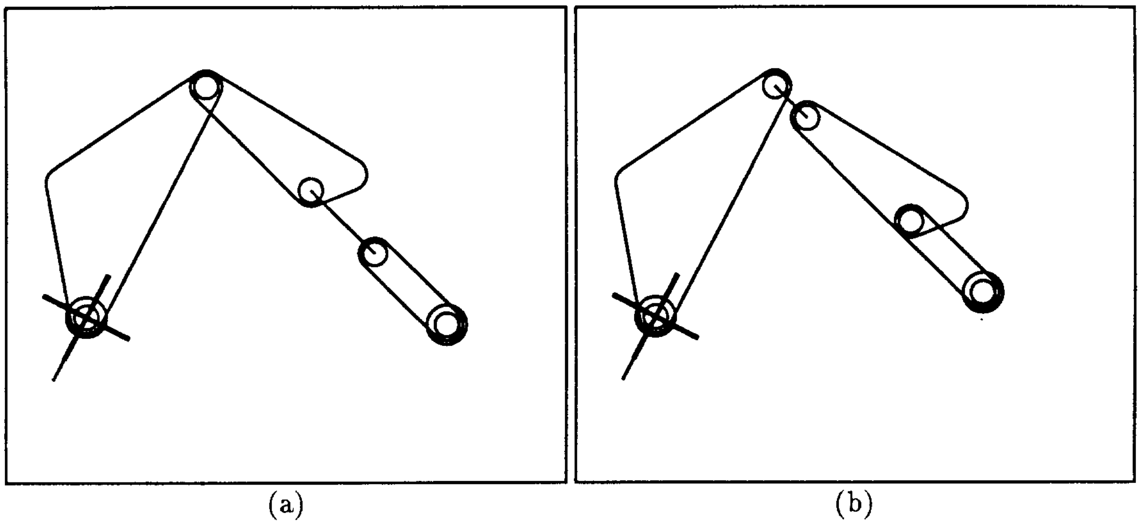


Figure 16: Two possible variants of the stretch policy for breakage. (a) Break only the inner joint, and minimize the distance. (b) Break the outer two joints, and minimize the sum of the squares of the distances. In either case, as the input crank moves, the links look as though they are being pulled by their broken joints.

Stretching. Another possibility would be to let the unsolvable joints break, as our current policy does, but to move the links involved to their "best" broken position (according to some metric, e.g., one that simulates rubber bands in the broken joints), and based on these positions to go ahead and execute the rest of the plan. Figure 16 shows examples of two possible variations on this *stretch* policy. These policies have the disadvantage of not displaying the limit position, and they do involve displaying broken joints, which may complicate both the user's life and the programmer's. But they would display the breaks in a more visually convincing manner than our current policy does. A further advantage is that, no matter how the mouse was moved, the whole mechanism would be animated. Even when a constraint was violated early in the plan, the late portions of the plan would still be executed, and the user would be able to manipulate the rest of the mechanism somewhat.

Approximation. If our current policy could sample the mouse position infinitely often, it would leave the later portions of the mechanism (i.e., the links fixed by later steps in the plan) in their limit positions, while still allowing the earlier portions to move beyond the limit point. We could create this effect by using the searching policy to find the limit point, and then letting the early portions go beyond this. Can we create a similar effect with less computation? During the discrete motion that first breaks the mechanism, the base case solver has available the coordinates of the vertices on the fixed links, in both their unbroken and their broken positions. Unfortunately, the solver does not know the full trajectories of these vertices, so it can only estimate where they would be in the limit position, just as the mechanism breaks. A policy of linear *approximation* would work well, but only for small motions.

Of these, the searching and stretching policies sound the best. But none of these policies is completely satisfactory. The only one that reliably displays limit points is the searching policy, and it does this by numerical methods. This is not in the spirit of closed-form solutions, and it is not in the spirit of making editors that are more powerful because they better understand the object being edited. It would be an important step forward to find an algorithm capable of computing in closed form the exact ranges of an input joint parameter for which a mechanism can remain unbroken.

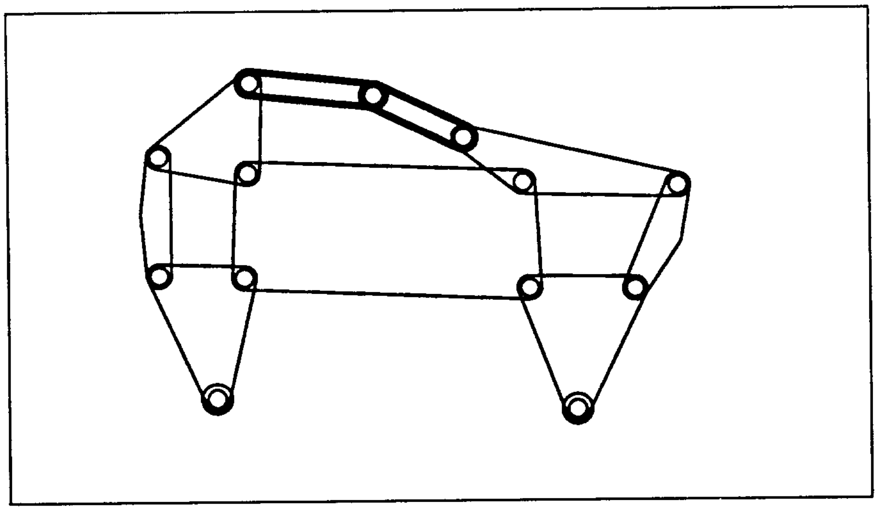


Figure 17: The difficulties of computing joint limits a priori. The two bold links constrain the motion of the whole mechanism.

Figure 17 is an example of the difficulties inherent in this problem. In the plan for solving this animation, the final solver step is the one that fixes the two center links in the top row. If this final solver is to succeed, an inequality constraint must be satisfied, namely that the distance between the endpoints of the two links be no greater than the sum of the links' lengths (a constant). (Other types of solvers introduce similar inequality constraints, such as a constant limit on the distance between a point and a line.) We would like to be able to propagate this constraint backwards through the plan, eventually arriving at limits on the input joint. When the input joint was kept within these limits, the final solver step would be guaranteed not to break. But it is not at all clear how this back-propagation could be done. The two endpoints are fixed at two different times, by separate solver steps within the plan. It does not seem that the distance constraint between the two endpoints could be meaningful to a solver step that is responsible for fixing the position of only one endpoint.

As a side issue, note that our branching policy depends on detecting that a submechanism has gone from broken to unbroken; this approach may still be used with any of the breaking policies we have described.

3.2.6 Branching Policy

There are several other policies that solvers could use that also respect continuity. One would be to branch *randomly*, either when the mechanism breaks or whenever the mechanism is near a branching point. This would tend to make the user feel powerless over the mechanism, which, unless somehow pedagogically useful, is undesirable. Another approach would be to use a *momentum term* that tries to maintain continuity of the direction of motion of the links. Our policy is equivalent to one version of this. Overall, the currently implemented policy seems the easiest for the user to understand and control; it seems best to adhere to the Principle of Least Astonishment mentioned by Borning et al. [Borning 87]

Even so, branching is not easy to control in larger mechanisms. When there are N solvers that each have two possible modes of assembly, there are 2^N modes for the whole device. Our experience has been that it is difficult to navigate in this space. To flip a particular submechanism over into its other mode, the user must cause that submechanism to break – and in order to do that, the user must first avoid breaking anything that comes earlier in the plan. Our experience has been that, once a few things have been flipped over, it can take some time to find a path back to the original mode. The airplane engine in Figure 22 is an example of this.

Better would be to give the user a way to interactively push the system into the desired mode. In other words, to obtain branching information, we should provide an extra channel of communication between the user and the program, rather than using an automatic policy. We have not yet studied the question of how to provide this channel so that it is pleasant to use. One possibility would be to let the user double-click on a joint in order to switch it to its other branch.

4 Implementation

The mechanism editor has been implemented in Common Lisp, and runs on a color graphics workstation under the X window system. Specifically, we use a Hewlett-Packard 9000/350 SRX computer, running HP Common Lisp II, which is based on Lucid's Common Lisp. HP's Starbase graphics library is used for graphics output, while the X windows library is used to handle windows, menus, and mouse and keyboard events.

In this section, we describe some highlights of the structure of the mechanism editor program. We then discuss our use of Lisp versus other language alternatives, including object-oriented programming.

4.1 User Interface Control Structure

The editor's user interface code is responsible for receiving in-coming events from the window system and for interpreting these events, calling functional routines as appropriate. A user interface is often thought of as a finite state machine. In each state (also called a mode), there is a mapping of events, such as a click of the right mouse button, to functionality, such as deleting a link or changing the interface state.

An important decision in the design of a user interface is the choice of a call tree for the control structure. One possibility is to have a top-level routine that gets events, examines them, and calls down to the other routines accordingly. The inverse possibility is to get events at the bottom of the call tree. In this case, there are various points within subroutines and subsubroutines where an event is waited for; after each of these points, the program branches based on the event received. The state of the user interface is determined by what part of the program is executing. In other words, the state is stored in the program counter. In the first case, there is only one point in the program at which events are read; therefore the user interface state must be stored in variables.

The mechanism editor uses a hybrid of these techniques. The resulting state machine is much like an augmented transition diagram. A table of *handlers*, subroutines that each handle a specific event type, is maintained. Subroutines entered in this table handle events that do not affect the state of the user interface, such as window expose or resize events. The main execution thread changes the interface state by changing its program counter, such as by entering the subroutine `ui-fixed-pivot` when the user selects the "create fixed pivot" command from a menu. The main thread calls the routine `next-unhandled-event` to read the next user action. This routine pulls events off of the system's event queue and looks them up in the handlers table. If the event has a handler subroutine, `next-unhandled-event` calls this routine, and then goes on to the next event. Once it finds an event that does not have a handler, it returns this event to the caller. In this way, for example, while the main thread waits for the user to enter a keyboard command or make a menu selection, window resize events are still properly handled.

The main execution thread can modify the handlers table, so that different sets of handlers are active in different states. This is primarily used to implement *mouse tracking*, that is, continuous graphical feedback as the mouse is being moved. The most complex mouse tracking takes place while the user is positioning a vertex in the polygon of a new link. Whenever the mouse moves, we redraw the whole link using the new mouse position. We also draw (potential) new joints, whenever the mouse position is near a free vertex in an existing link; these potential joints are created and destroyed as the mouse moves in and out of range of the free vertex. The routine that does all this must be called repeatedly as the mouse moves. To achieve this, we place the routine in the handlers table, as the handler for mouse motion events. It remains in the table only until the tracking operation is complete (which is usually until the next mouse click). The handler must keep track of its state in variables, since a separate call is made to it for each small mouse motion.

Once this control structure was selected, and once it was understood exactly when state could be stored in the program counter and when it had to be stored in variables, the user interface became easy to implement. The control structure cleanly separates the state-independent code from the state-dependent code. It does have the limitation of providing only one program counter, namely that of the main execution thread. It would be an improvement if handlers, such as the mouse tracking handler described above, could keep state in a program counter as well.

4.2 Implementation Layers

We now list the layers of the implementation, from low-level to high-level. This is primarily for the benefit of anyone who wishes to read the source code.

- At a low level, there are the X Windows and Starbase graphics packages. These packages are provided as libraries of routines callable from the C programming language.
- For these routines to be accessible from Lisp, they must be imported via HP Common Lisp's foreign function interface. This layer required some Lisp programming and some C programming. As an interface to the X Windows routines, we used U. C. Berkeley's XCL package, after porting it to the HP environment.

The remaining levels are implemented purely in Lisp.

- This layer draws on the color graphics screen. Graphics output is accomplished by calls to (the Lisp interfaces to) the Starbase routines for drawing lines, circular arcs, etc. Drawable objects – links, joints, and a few other things – are declared as subtypes of the type `drobj`. The variable `*display-list*` contains a list of all objects on the screen. The routine `gwin-redraw` redraws the graphics window (`gwin`) by first clearing the screen and then calling `draw-drobj` with each item in `*display-list*`. There is also a routine `gwin-resize` for changing the size, location, or world-to-screen coordinate transformation of the graphics window.
- This layer puts an X window over the graphics window. Calling the Starbase package directly is the only way to make efficient use of the graphics hardware in our HP workstation. On the other hand, X Windows, while unreliable, inelegant, and obscenely difficult to configure, does permit the mechanism editor to run within a popular development environment. For this reason, we decided to use the two packages in combination. Our X Windows implementation uses only the overlay planes, where it has a special "see-through" color that shows the full-color graphics beneath. We fill the interior of the X window with this transparent color. We can then see the Starbase graphics underneath, while still being able to see the menus, window titles, etc., drawn into the overlay planes by X Windows. We must be careful to draw our graphics to coincide exactly with the inside of the X window; coordinating dynamic window moves and resizes is tricky but possible.
- The next layer handles window system input. It consists of the `next-unhandled-event` routine described in the previous section, plus some handlers, handler tables, and auxiliary routines.
- Mouse tracking is implemented as a layer above `next-unhandled-event`. The main routine is `next-event-track`. This routine creates a handler for mouse motions, then calls

next-unhandled-event. The handler is created from two routines passed to next-event-track as arguments, track-on and track-off. Track-on is passed each new mouse position; its function is to draw any tracking information on the screen. Track-off's function is to undraw this information, so that the screen does not become a cluttered jumble as track-on is called repeatedly.

Next-event-track returns the last value returned by track-on. This value is normally related to picking. For instance, during a "delete link" operation, the track-on routine might return a pointer to the link that the mouse is currently closest to. The final value of this pointer is returned by next-event-track, and the link is deleted.

- Finally, there is the main execution thread. Its top level is ui-main-loop, which, based on menu selection events, calls the routines for creating a link, fixing a pivot, animating, writing a file, etc.

4.3 Implementation of Plan Time Versus Move Time

A plan is a list of executable functions – closures, in Lisp terminology – that, when executed, solve the mechanism. Each closure in the plan is the result of calling a solver on a subgraph of the mechanism. The solver is passed three joints, A, B, and C, such that A and B each have fixed links on one side; it returns a closure that, when executed, will compute the positions and orientations of the R-links AC and BC. A Lisp closure contains not only a piece of code, but also the environment in which the code should be executed; in particular, this includes the values of any local variables whose scope includes the code. This provides us with an especially simple and convenient mechanism for separating the plan-time calculations from the move-time calculations. We compute as much as we can at the time that the solver is called (plan time). We return a closure that does the rest (at move time). The results of the plan time computations are left in local variables whose scope includes the code we return. Thus, the code has access to these values when it is executed at move time. The Lisp closure mechanism is what makes this work even if the same solver is called several times: each closure returned has its own copy of the solver's local variables. The code ends up looking something like that in Figure 18.[4]

Once the plan is complete, next-event-track (described in the previous section) is called with the list of closures as part of its track-on argument. This causes the closures to be executed repeatedly as the mouse moves.

4.4 The Input Step Solver

The user controls the animation by controlling a grounded revolute joint with the mouse. Since the graph edge corresponding to this joint is the first to be contracted during planning, the first step of the plan must fix the coordinate frame of the R-link attached to this joint. The closure that does this is returned by a special solver, control-gnded-rjoynt. This closure implements the transfer function from mouse motion to joint rotation, which we call the control *strategy*.

[4]The author apologizes for any inconvenience caused by his somewhat fanciful spellings in the source code. Vertices of links are of type poynt, which is separate from, though derived from, the type point. (This is because, at first, it was felt that these vertices might have extra properties or values attached to them, in addition to simple coordinates. This proved not to be the case, and a poynt is, in practice, just a point.) The word lynk is used to mean "link" in the mechanical engineering sense, to distinguish this from "link" in the computer programming sense. The word joynt is used for consystemcy.

Also, what are herein called contractions are termed "collaptions" in the source code.

```
(defun solve-rrr (ja jb jc)
  ;; JA JB JC are the three joints in the 3-cycle being solved.
  ;;
  (let* (;; PLAN-TIME COMPUTATIONS:
         ;; Vector from JA to JC, in RLYNK-AC coords:
         (vac (point-diff (rjoynt-poynt jc jc-rlynk-ac)
                          (rjoynt-poynt ja ja-rlynk-ac)))
         (lac (point-length vac))
         ...  )
    (labels ((t-on ()
               ;; MOVE-TIME COMPUTATIONS:
               ;; start with:  Vector from JA to JB, in world coords.
               ...
               (set-point-diff vab jb-world ja-world)
               (let* ((lab (point-length vab))
                      ;; Solve the triangle (law of cosines):
                      (cos-a (/ (+ (- (* lac lac) (* lbc lbc))
                                   (* lab lab))
                                (* (* 2.0 lac)
                                   lab)))
                      ...)
                 ;; Based on these values, fix RLYNK-AC and RLYNK-BC:
                 (set-xform (rlynk-to-world rlynk-ac) ...)
                 (set-xform (rlynk-to-world rlynk-bc) ...))))

      ;; The following is also done at plan time:
      (setf (rlynk-fixed-p rlynk-ac) t)
      (setf (rlynk-fixed-p rlynk-bc) t)
      ;; Return the T-ON closure defined above:
      (list #'t-on))))
```

Figure 18: The code for a solver.

Our control strategy, described briefly in Section 1.3.2, is simple. We choose an x-axis on the ungrounded link, and always turn the link so that this x-axis points from the joint towards the mouse. This rotates the link smoothly while the mouse is far away, but becomes increasingly unstable as the mouse approaches the joint. To avoid the worst of this, we do not move the joint at all unless the mouse is at least a certain distance away from the joint. Another flaw in this control strategy is that the x-axis we choose may not be the one that the user would have guessed. Our heuristic is to use the direction described by the first two points that the user entered while drawing the link. This is easy, and it does the right thing for two-vertex links – half the time, at least. Certainly, better control strategies could be found. Even more interesting would be to investigate control strategies for controlling two (or more!) joints at once, with a single mouse.

4.5 Programming Languages

The project of implementing the mechanism editor made us acutely aware of the some of the trade-offs among programming languages. At the start of the project, we needed to choose from among the programming languages – more precisely, from among the program development systems – available to us. The C language had the advantages of efficiency and a stable development environment, including source-level debugging tools. We also considered C++ for its additional expressiveness, though its development environment was still immature. Ultimately, we chose Lisp, because it seemed the language best matched to a program that would dynamically build and execute code. It was felt that this advantage would be worth the increased costs in processing time and memory usage.

We often found Lisp's functional programming paradigm to be both elegant and practical. As explained above (Section 4.3), Lisp's notion of closures provides a clean implementation for the generation of solver steps. We make frequent use of functions as arguments (e.g. to **next-event-track**) and as elements in tables (e.g. the event handlers table); the ability to define new functions locally is handy in these situations. Even ordinary procedural code benefits; for example, the clause "if all the R-links in the list of R-links are fixed" can be written

```
(if (every #'rlynk-fixed-p *rlynks*) ...  ).
```

It is difficult to express the same idea so neatly and orthogonally in C.

On the other hand, several important parts of the program were more difficult to write, and to read, in Lisp than they would have been in C. In particular, arithmetic expressions in Lisp look more like a jumble of parentheses than like arithmetic, making errors harder to spot. There were also the difficulties of writing foreign function interfaces (discussed in Section 4.2) and of dynamically loading the foreign packages into the Lisp interpreter's memory image; these procedures often failed mysteriously.

But the biggest problem with Lisp is its appalling performance on hardware not specifically designed to support it. Our program produces painfully slow animations (1-2 updates per second). In addition, there is garbage collection (and garbage production, which is also expensive). While animating, it is not uncommon to have to wait every twenty updates or so for several seconds of garbage collection. Our algorithms perform very little computation; using a language system capable of efficient implementations, we would expect them to run at video rates (perhaps 30-60 updates per second). In a language that allowed explicit freeing of memory structures, garbage collection could be avoided altogether.

In retrospect, while it was very helpful to *think* of the program in terms of Lisp, the use of Lisp for the actual implementation appears to have been a mistake. Many aspects of development would have been facilitated by using C instead, while only a few sections of code would have been noticeably encumbered.

One such section is the generation of solver steps; this portion of the algorithm was one of the major motivations for using Lisp in the first place. Section 4.3 shows the implementation of this using closures; closures effectively save the values of local variables for future use. The same effect can be achieved in C by storing the local variables into the fields of a structure, and passing this structure to the plan step when it is executed. This does have the unfortunate effect of separating the code that *sets* the local variables (plan time) from the code that *uses* these values (move time). But the program is still wieldly.

When this code is couched in terms of C++, the structure becomes an object, and the plan step itself becomes a method (probably implemented as a virtual function). The code is still separated, as for C, but the notation is more concise. C++ also does quite well with the "if every R-link is fixed" example mentioned above. Assuming that `every` is a member function of the `list` class, we can write:

```
    if (Rlynks.every( rlynk:fixed_p )) ...    .
```

If both `every` and `rlynk:fixed_p` are implemented as in-line functions, the code will be quite efficient.

Our conclusion is this moral: Think in Lisp, write in C++.

The moral presumes that there is a fully developed implementation of C++ at hand; if all that is available is a translator from C++ to C, it is probably not worth it. In particular, we do not feel that use of an object-oriented language system is prerequisite to producing a well-designed program. We resisted the temptation to use an object-oriented programming package, such as the Portable Common Loops implementation of the Common Lisp Object System (CLOS). We were thus able to avoid the expenses (in time, memory, and administrative effort) of adding on yet another large software package. But we used an object-oriented style at times, declaring structure types and subtypes, and naming our routines appropriately. The most extreme example is the drawing routines, `draw-lynk`, `draw-rjoynt`, `draw-pjoynt`, etc., where `lynk`, `rjoynt`, and `pjoynt` are among the subtypes of the "drawable object" structure type. We believe we were able to reap most of the structural and extensibility benefits of the object-oriented approach, without actually using an object-oriented language.

5 Results

In this section, we discuss some examples of mechanisms that the mechanism editor can animate, and some classes of mechanisms that the program cannot animate successfully.

The mechanism editor can compute and display the motion of any planar mechanism of polygonal links, revolute joints, and prismatic joints that meets the following criteria:

- There is one control parameter for the animation, and that parameter is the joint angle of a fixed pivot.
- The mechanism has one degree of freedom. (Rigid submechanisms are allowed, and properly detected.)
- The R-link graph of the mechanism may be reduced to a single node by a series of contractions, the first of which contracts only the edge representing the input pivot, and the remainder of which each contract a triangular subgraph. This criterion may also be stated in the reverse direction: the mechanism's R-link graph may be constructed by starting with the two-node graph of its input pivot (and associated R-links), and adding two nodes in each step, with one joint between the two nodes and one joint from each new node to an already existing node.[5]
- The geometry of the mechanism is generic.

In the above characterization, only the last criterion is geometric, while the first three are strictly topological. Below, we will discuss in turn the prospect of relaxing each of these restrictions. But first, we show some examples of mechanisms that meet all the criteria.

5.1 Examples

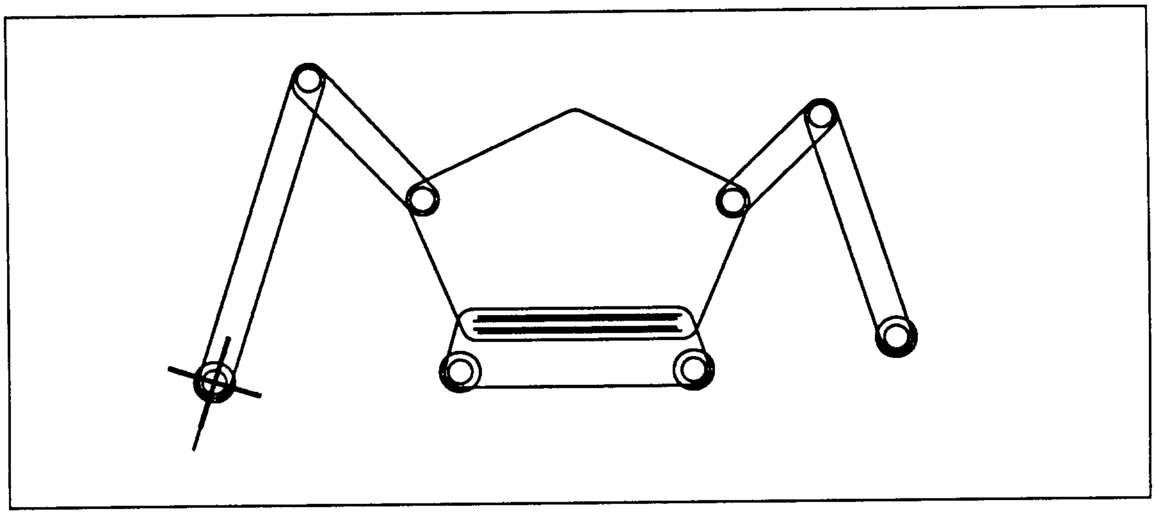


Figure 19: The crab moves sideways.

The Crab, Figure 19, has six links (including ground) and seven joints. Either "foot" may be pivoted, causing the "body" to slide back and forth horizontally.

[5]There is also the restriction resulting from the fact that the solvers for base cases containing two or more prismatic joints have not been implemented. But this is incidental.

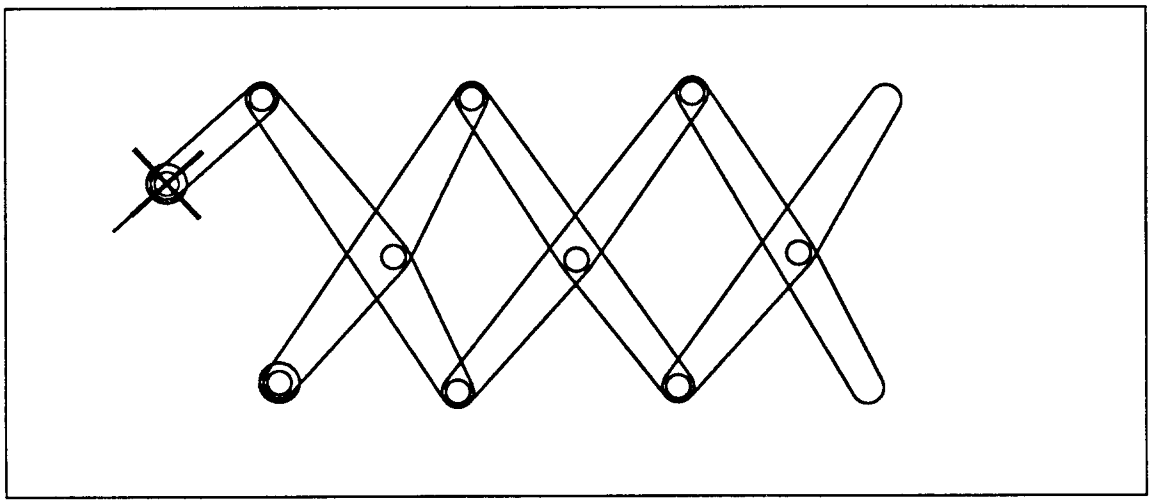


Figure 20: The wine rack.

The Wine Rack is shown in Figure 20. Cranking the upper left pivot causes it to extend and contract (though not perfectly horizontally). The plan for this motion is a series of RRR steps that solves the wine rack from left to right. In the next section, we show a variant of the wine rack that cannot be solved by our algorithms.

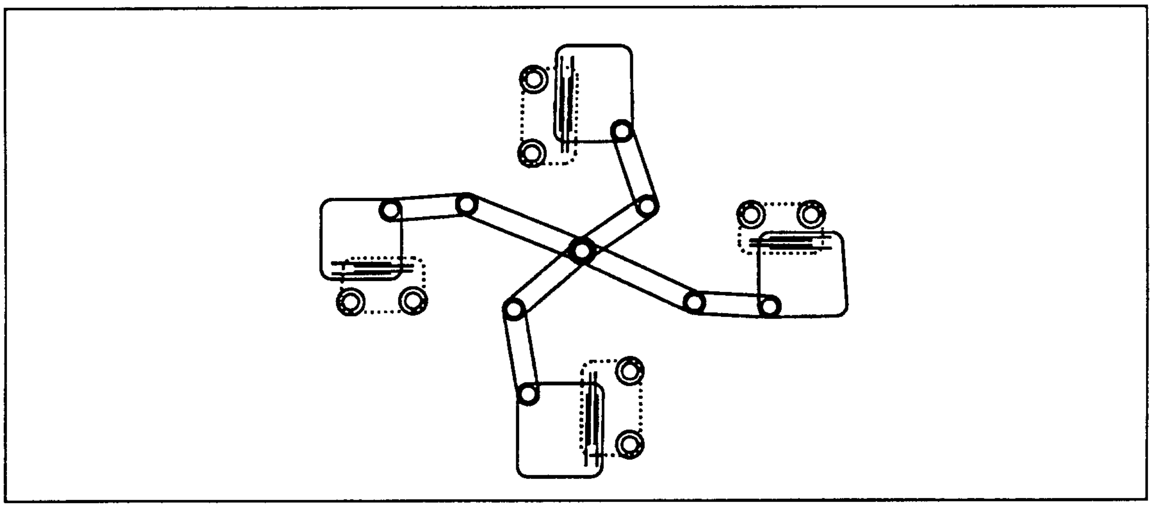


Figure 21: The radial four-piston combustion engine.

The Airplane Engine is a radial arrangement of four pistons, each sliding against a grounded "cylinder." All four pistons are connected to a single link in the center, which may be pivoted against ground to drive the pistons in and out.

If the piston rods are long enough, the mechanism should never break. But if not, pistons may individually break and then flip over into other modes of assembly. Even if the rods are long enough, sudden large motions of the driving link may cause a new mode to appear, since, in the presence of large motions, the new mode may momentarily appear more continuous than the old mode. A little of this puts the engine in an unnatural state (Figure 22). There are many more bad states than good ones, and once a bad one is achieved, finding a path back to a good one can

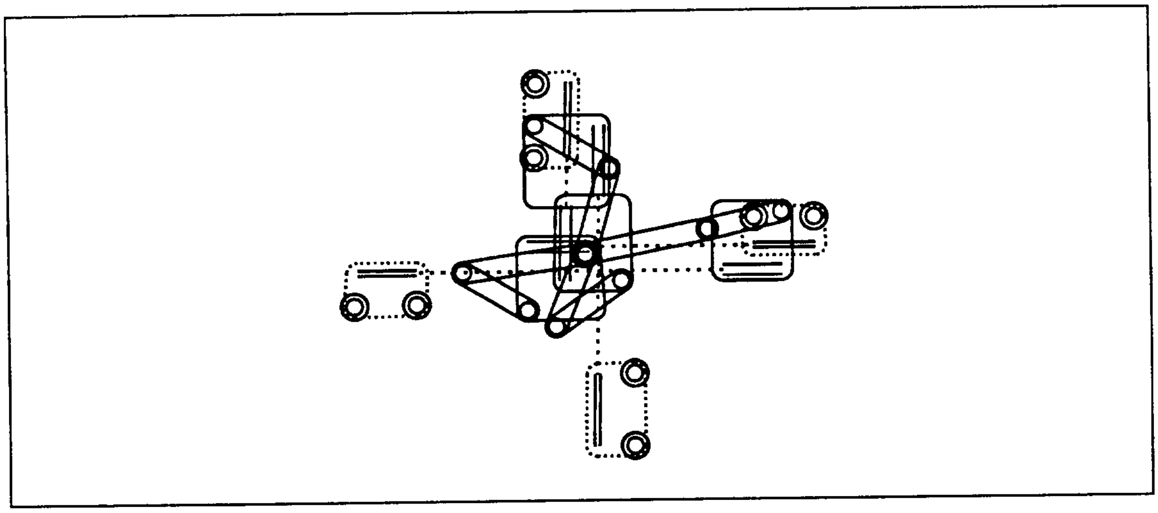


Figure 22: The same engine, a few minutes later.

be difficult. This sort of behavior may be avoided by cautious and continuous mouse usage, plus proper engine design.

5.2 Counterexamples

We now discuss mechanism animations that our editor does not know how to solve. These are animations that violate one of the four criteria stated above.

When the geometric criterion of genericity is violated, the editor does not notice. It simply treats a non-generic mechanism as if it had had been perturbed slightly to become the nearby generic mechanism. This produces consistent results, and is only in error when the non-generic mechanism has fewer degrees of freedom than the generic one would have had.

When a topological criterion is violated, one of two things happens. Either there is no way to ask the editor to perform the desired animation – for instance, if the animation has two separate control parameters – or else the planning phase fails to construct a plan that completely solves the mechanism. In the latter case, the planner returns the partial plan, one that fixes some links but not others. This plan will cause motion that violates mechanism constraints only if there are joints between links that are fixed by the plan and links that are not. The mechanism editor counts up the number of these joints, prints a warning message in its text window, and marks these joints as broken. So if there is no warning message, then the user knows that the plan will work, and the mechanism will animate. Otherwise, the user will see how much of the mechanism the editor was able to solve; the rest will be left stationary, connected to the moving part by broken joints.

We now consider each of the four criteria in turn.

5.2.1 Input Parameters

The restriction that the input parameter be a revolute joint, rather than a prismatic one, is trivial. The prismatic case can be handled by code entirely similar to that used in the revolute case (including a control solver analogous to `control-gnded-rjoynt`; see Section 4.3).

The restriction that the input joint be grounded is not as trivial, but is still quite easy to relax. Suppose neither side of the input joint is grounded. If we pick one side and temporarily pretend that it is ground, then we can solve the mechanism. Afterwards, we transform our answers into the true ground frame by multiplying them by the inverse of the transformation we found for the ground R-link. (The only complication arises if our planning algorithm gets stuck before it fixes the ground R-link. Then we don't know how to transform our answers back to world coordinates. One reasonable answer is to transform everything such that our temporary ground link stays still on the screen.) This extension, plus some additional user interface code to make use of it, can handle ungrounded input joints.

How might we solve animations that are controlled by more than one input parameter? Assuming that there are exactly as many input parameters as degrees of freedom in the total mechanism, the fundamental planning scheme can remain the same: we contract the graph edges that correspond to input joints, and turn the resulting graph over to the planner, to be contracted a triangle at a time. This works. The complication is that, at times, the planner must keep track of several contractions, each independently contracting some portion of the mechanism graph. When two contracted regions are contracted together, their coordinate systems may be resolved against each other. Eventually, all regions are resolved against ground, and the results may be displayed.

Of course, it is necessary to check that the input parameters that the user has chosen are independent. Otherwise, fixing them will overconstrain the mechanism. This condition is easy to detect in the mechanism graph.

We have implemented one case of multiple input parameters: the user can select a vertex of a link, and then drive the mechanism by moving this vertex in two dimensions (the vertex follows the mouse). The link is free to turn around the controlled vertex, as if it were connected to the mouse by a revolute joint. We found this to be a very pleasant and intuitive means of controlling an animation.

Our implementation of this differs from the approach described above in that the mouse motion is not converted to joint parameters; indeed, there are no input joints. It would have been possible to build the plan by temporarily adding some imaginary joints and links to the mechanism, and letting the mouse control these joints. For instance, the controlled vertex could be pivoted to one imaginary link, which could slide horizontally against a second link, which could slide vertically against ground. The positions of the horizontal and vertical sliders would be computed from the x and y mouse coordinates, as part of the input step. This would fix the coordinate systems of the imaginary links, and in the contracted graph the controlled vertex would be connected to ground by a revolute joint.

We use a short cut. We temporarily add a revolute joint between the controlled vertex and ground, and we use the mouse position to vary the coordinates of the grounded side of that joint. The mouse does not (directly) control any joints, and the input step does not contract any edges of the graph. (Adding the revolute joint already reduces the number of degrees of freedom by two.) It is as though a fixed pivot were added to the mechanism, but at each time step, the fixed pivot is moved, and the mechanism is re-solved. Thus, rather than build imaginary extensions to the mechanism's graph and then contract these extensions into ground as part of the input step, we work with a "pre-contracted" graph, in which the controlled side of the pivot is already part of the ground frame.

5.2.2 Underconstraint

We now consider increasing the number of degrees of freedom of the mechanism, but without increasing the number of control parameters. It is natural to build such underconstrained mechanisms when using the editor, and it is frustrating to have them not move, not because there is no consistent motion for them, but because there are too many possible consistent motions. Figure 23 shows three of the simplest cases. The first consists of two separate mechanisms (only one of which

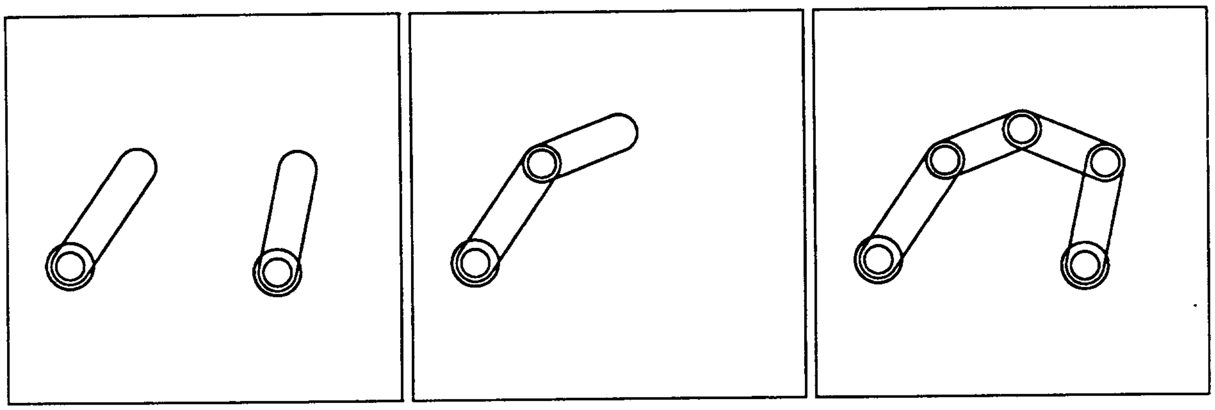


Figure 23: Underconstrained mechanisms.

is being animated); the second is an open two-link chain; and the third is a closed five-bar chain.

Given any underconstrained animation to solve, the planner will solve as much of the mechanism as becomes rigid when the control parameter is fixed.[6] The graph that is left after these contractions has no rigid submechanisms, so the current planner stops here. But the graph does have degrees of freedom. To continue solving the graph, the planner could fix the joint parameters of any remaining edge.[7] Call this edge a *free* edge. This would eliminate one degree of freedom, and create some (possibly trivial) rigid submechanism. By solving the now-rigid portions, and continuing in this way for each remaining degree of freedom, the planner could solve the entire underconstrained mechanism.

The solver step that fixes a free edge must, at run time, choose a joint value. Because it is a free edge, this choice is somewhat arbitrary. Here are some possible policies for choosing the value.

Frictive joints. Keep the old joint value.

Frictive links. Try to move the joint's links as little as possible.

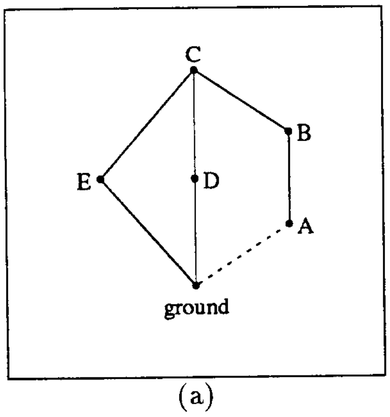
Jiggle. Add a small random perturbation to the old joint value.

This last policy would cause the free portions of a mechanism to jiggle and wander about during an animation. This might be disconcerting, but it might help in making clear to the user where the extra degrees of freedom are. The first two policies would give reasonable results on each of the examples in Figure 23, though the first one might make linkages too stiff, and serve to hide the extra degrees of freedom. But the trouble with any of these policies is that a joint value chosen by one step of the plan may cause things to break later in the plan; as discussed in Section 3.2.1.1, there doesn't seem to be any way to predict this in advance. These issues deserve further thought and experimentation.

Before the geometric decision of what joint value to choose, there is the topological decision of which edge to fix. The simplest thing is to choose an edge which connects a solved portion of the graph with an unsolved portion; otherwise a new, independent contraction will be created when the edge is contracted. In the following section, we describe mechanisms that are rigid and yet

[6]Barring the occurrence of the uncontractable mechanisms, described in the next section.

[7]This assumes that the graph is connected, which it is, unless there are portions of the mechanism that are entirely unconnected to ground.



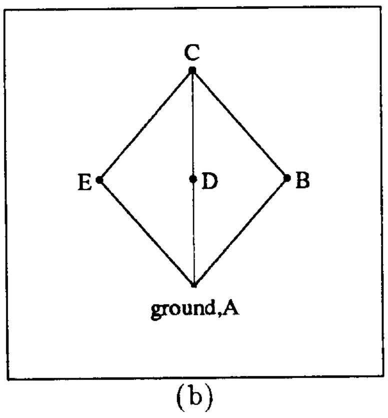


Figure 24: (a) Graph of the six-bar linkage from Figure 6. The dotted edge corresponds to the upper fixed pivot. (b) The same graph after the upper pivot's edge has been contracted. This graph contains no triangles. It is therefore uncontractable.

cannot be solved by the planner; this suggests choosing an edge based on what percentage of the rest of the graph will be solvable once that edge is fixed.

5.2.3 Closed Form Solutions

We come now to the most fundamental restriction of our technique, namely, that after the input joints' edges are contracted, the mechanism's graph must be reducable to a point by the successive contraction of triangular subgraphs. Let us call the graphs and mechanisms that can be so contracted *contractable*.[8] For these mechanisms, we have solutions in closed form. But there are uncontractable mechanisms. There are even simple examples of mechanisms that do not appear to have closed-form solutions at all. For example, consider the six-bar linkage in Figure 6. If the control joint is either of the two lower pivots, the graph is contractable, and we have a closed form solution. But trying to solve for the motion as a function of the angle of the upper pivot results in an unpleasant non-linear system in two variables. This system has no known solution in closed form.[9] (Visually speaking, we can't pull the rabbit by its ear in closed form.) The graph for this case is shown in Figure 24.

The technique we have presented is thus incomplete: it cannot successfully animate all mechanisms (not even all those with the right number of degrees of freedom). Exactly how incomplete is an important question, one to which we do not yet have the full answer. We can offer two pieces of evidence, however. The first is anecdotal: while our system was being used, nearly every one-degree-of-freedom mechanism that users built was contractable. (Admittedly, these users were computer scientists, not mechanical engineers.) The second and less encouraging piece of evidence is mathematical: there is an infinite number of distinct mechanisms that are uncontractable. Figure 25a shows the construction of an infinite family of such mechanisms. Each graph is a row of n squares, plus an input edge that is attached to each end of the row. Assume that the input edge has been contracted. Then the total graph has $2n + 3$ nodes and $3n + 3$ edges, yielding $3(2n + 3 - 1) - 2(3n + 3) = 0$ degrees of freedom, yet no proper subgraph has less than one degree

[8]We have left the term a little bit ambiguous, since it may refer either to a graph (or mechanism) after the input joints have been fixed, or to a graph (or mechanism) whose input joints have been *chosen* but not yet fixed.

[9]C. W. Radcliffe, personal communication, September 1988. We are not aware of any proof that no such solution exists.

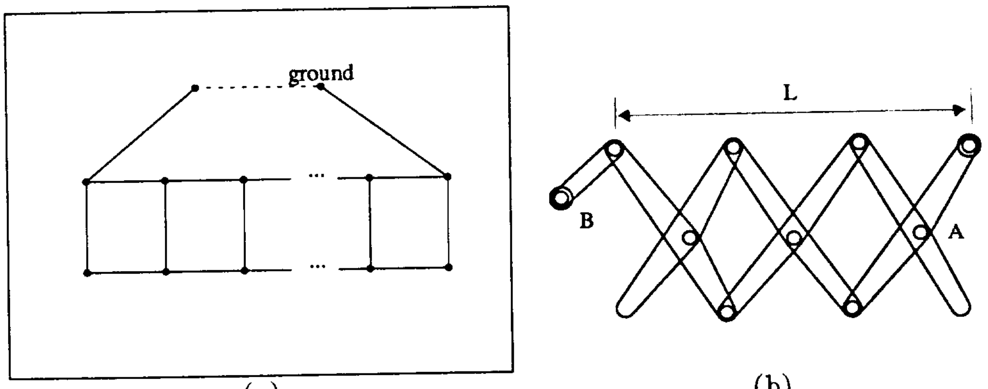


Figure 25: An infinite family of uncontractable mechanisms. (a) The graphs. The dotted edge represents the controlled joint. (b) The mechanism for $n = 2$.

of freedom. Therefore, these mechanisms cannot be solved by breaking them down into triangles, or indeed any smaller pieces – any solver than can solve one of these mechanisms must solve it all at once. A mechanism that instantiates one of these graphs is shown in Figure 25b; it is identical to the Wine Rack, save that its fixed pivot is on the side opposite from its handle.

Could the mechanism editor be extended to handle all planar mechanisms, while still maintaining our inductive approach? Perhaps new "base cases" could be implemented to solve each of the uncontractable graphs. The observation that there are infinitely many such graphs is discouraging but not unanswerable; special solvers could handle whole families, and besides, in practical terms we are only interested in handling mechanisms up to some finite size. A valuable next step for research would be to characterize the uncontractable graphs or, more fundamentally, the set of rigid graphs containing no rigid subgraphs.

Of course, the simplest approach would be to fall back on a relaxation technique, or other numerical method, to solve these cases. The planner would formulate closed-form solutions for as much of the graph as possible, recognize the remaining portions of the graph as uncontractable, and then create a numerical solver step for this remainder. Borning's ThingLab uses this approach, and we discuss it further in Section 6.3. For the example in Figure 25b, there is a solution that is an interesting combination of closed-form and numerical methods. The length L may be computed in closed form from the joint angle at A. The inverse of this closed formula is enough to solve the mechanism, since the position of the input crank at B determines what value of L is required. Numerically inverting the closed formula is more efficient than using a relaxation technique on the entire mechanism. To further speed up the computation, values of the formula could be pre-computed into a lookup table at plan time.

5.2.4 Genericity Again

The mechanism in Figure 26 gives us a fairly pathological example of non-generic behavior. As the slider slides, the positions of the all the other links may be determined from the position of the slider; the mechanism exhibits one degree of freedom. But the slider may be moved so that its pivots coincide with the fixed pivots. In this configuration, each "hinge" may rotate independently about its coinciding pivots, giving the mechanism two degrees of freedom. Not only can the number of degrees of freedom not be computed from the mechanism's graph, it is not even constant!

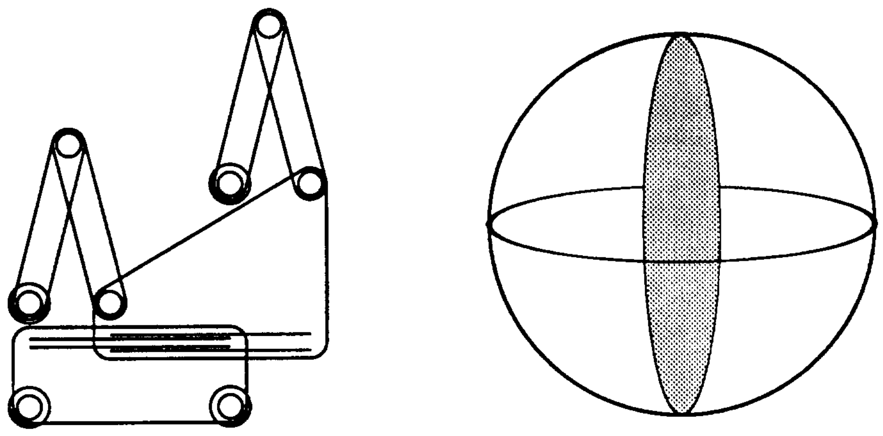


Figure 26: (a) A very non-generic mechanism. (Lengths which appear to be equal are exactly equal; lines which appear to be horizontal are exactly horizontal.) (b) The topology of its reachable configurations. The horizontal axis represents the slider position. When the slider is off-center, there are four possible states, given by the two branches of each of the two fixed pivots. When the slider is centered, the two fixed pivots may have any angle, yielding a two-dimensional region of possible configurations.

Correctly animating non-generic mechanisms would require the detection of dependent constraints. The above example shows that these can arise dynamically.

Outlawing generic mechanisms saves us a lot of trouble. How much does this limitation impact the usefulness of the editor? Here again, the nature of our quick-sketch, direct-manipulation user interface comes to our aid in avoiding unpleasant special cases. It is very difficult (though not impossible) for a user to enter a non-generic mechanism, since this usually requires creating parallel lines or other perfectly aligned constructions. Nonetheless, this will often be the user's intent. Mathematically, genericity rules out only a measure zero subset of the universe of mechanisms. But this subset has measure much greater than zero in the user's mind. For instance, the three-legged table from Section 2.4.3 undoubtedly has structural properties preferable to those of any two-legged version.

For better or worse, the mechanism editor does not capture structural properties. It models only abstract mechanisms.

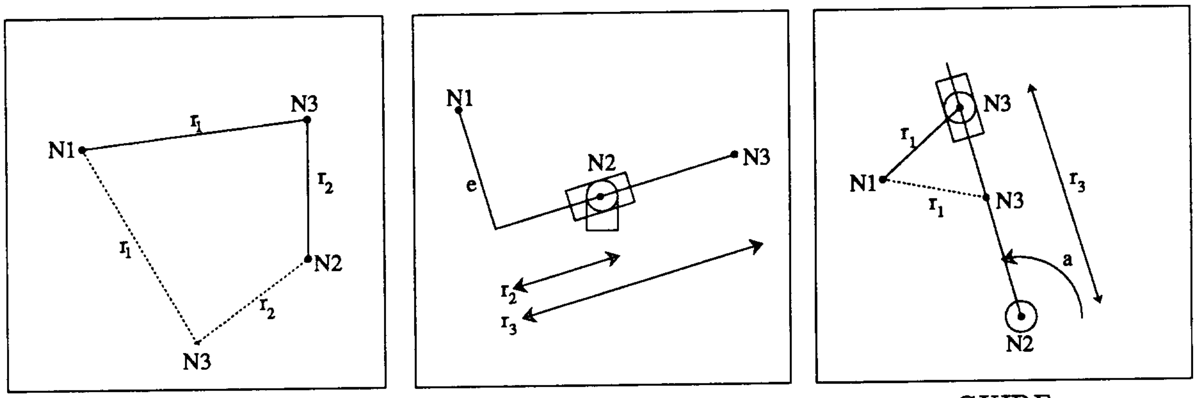
6 Past Work

Our mechanism editor is largely a combination of ideas from Suh and Radcliffe's LINKPAC and Borning's ThingLab; ThingLab is described by its author as a combination of ideas from Sutherland's Sketchpad and the object-oriented language Smalltalk. In the following sections, we discuss LINKPAC, Sketchpad, and ThingLab, plus more recent work done by Todd and Cherry. We attempt to analyze each system's capabilities for mechanism animation, and to discover what they do and do not have in common, both with each other and with our mechanism editor.

We also discuss the relative merits of physically-based modeling, a competing and (at least within the world of computer graphics) more popular approach to this sort of animation.

6.1 LINKPAC

In their undergraduate text [Suh 78], Suh and Radcliffe discuss, and present full listings of, a FORTRAN package for kinematic analysis in closed form. The package, LINKPAC, contains routines to solve for the positions, velocities, and accelerations of mechanisms built up from three basic building blocks, shown in Figure 27: the two-link dyad, the oscillating slider, and the rotating guide. Each of these routines takes some vertices as input, together with some geometric parameters, and



DYAD (RRR)
Inputs: N1, N2, r_1, r_2, mode
Outputs: N3

OSC (RPR)
Inputs: N1, N2, e, r_3
Outputs: N3, r_2

GUIDE (PRR)
Inputs: N1, N2, a, r_1, mode
Outputs: N3, r_3

Figure 27: LINKPAC's basic mechanisms.

produces one or more vertices as output. The user defines a mechanism with a list of calls to these routines, plus some other routines for describing rigid links. The list is ordered such that each vertex appears as output before it is used as input. Thus, the list both defines the mechanism and specifies how to solve its kinematics.

LINKPAC was a major inspiration for our mechanism editor's closed form solutions, and the two programs have many of the same strengths and weaknesses. Our solvers correspond exactly with their basic mechanisms: the dyad with RRR, the oscillating slider with RPR, and the rotating guide with PRR. A philosophical advantage of our mechanism editor is that, by concentrating on mechanism graphs, and by storing the shape of each rigid link in a local coordinate frame, we separate geometry from topology. More important are the practical advantages of the graphical interface and the automatic planner.

The mechanism editor uses interactive graphics and direct manipulation, and supports rapid, imprecise sketching of mechanisms. LINKPAC programs are batch mode and text based (though there is some work underway to provide more graphics). This is mostly because LINKPAC was written in 1978 (or earlier); was written within the FORTRAN culture, which discourages both interaction and flexible programming styles; and was written for, and is still used within, the constraints of undergraduate teaching resources. But it is also because mechanical engineers are interested in being able to use precise numerical measurements as inputs, and to obtain them as outputs, whereas we have exclusively emphasized rough sketching.

The other obvious distinction between the two programs is that our mechanism editor includes an automatic planner that computes closed-form solutions given only the mechanism type, while LINKPAC users express their mechanisms as solutions made up from the available solvers. The disadvantage for us is that the user can now express uncontractable mechanisms (cf. Section 5.2.3), but the advantage is that the user need not work out solutions while working out a design. The mechanism editor allows a designer to think in units other than the base cases for closed-form solutions.

In this sense, the mechanism editor tries to provide a "high level language" for mechanisms. Making the computer do more work allows the user to express the input at a higher level of abstraction. For instance, two mechanisms that differ only in the choice of input joint must be expressed with two possibly quite different "programs" for LINKPAC, but may be expressed very similarly for the mechanism editor. Also, branching decisions are explicit in LINKPAC – each basic mechanism routine takes a parameter specifying which branch to compute – but automatic in the mechanism editor. (Of course, in relieving users of this responsibility, we also relieve them of precise control over branching.)

Suh and Radcliffe have implemented a number of interesting features. LINKPAC has routines for computing velocities and accelerations, as well as trajectories. (The mechanism editor could be extended to do this; the largest question would be how to meaningfully display the data.) They have worked out many closed form solutions for spatial mechanisms (three dimensions). They present a Newton-Raphson iterative method, which can be used when there is no closed form solution. And, while no dynamic simulation code is provided, there are routines that solve for the instantaneous forces on a mechanism in the presence of springs and dampers.

6.2 Sketchpad

The first constraint-based graphical editor, Sketchpad, was written in 1962 by Ivan Sutherland as part of his Ph.D. research. With a calligraphic display and a light pen, the Sketchpad user could build planar diagrams hierarchically out of points, line segments, circular arcs, and instances of smaller diagrams. The user also entered constraints on the diagram, which Sketchpad enforced; for example, two line segments might be constrained to have the same length, or to be parallel. There was also some facility for constraining text, such as by requiring that it display the length of some line in the diagram. A constraint between objects was (optionally) visible on the screen, as a letter, indicating the type of constraint, and a circle around the letter connected to each of the objects being constrained.

This work was foundational in the field of interactive computer graphics. Sketchpad implemented many of the key ideas commonly used in graphics modeling systems today, including definition-instance hierarchies and the inexact picking of objects with a light pen (or mouse). Its programming style included recursive functions, abstract data types, and generic functions capable of handling objects of many types, in a fashion that very nearly comprises object-oriented programming, which it significantly predates.[10] Of all the work that has been done in this area to date, it is amazing how large a part of it was done at the outset by Sutherland.

In Sketchpad, each constraint was defined by a subroutine that examined the constrained variables and returned a scalar error value, which was zero if and only if the constraint was satisfied. Two methods were used to solve systems of constraints. The first, "the reliable but slow method of relaxation," was used only on systems that could not be solved by the second, "the one pass method." The one pass method, which Borning later called "propagation of degrees of freedom" [Borning 79], examines the constraint graph to find "an order in which the variables of a drawing may be re-evaluated to completely satisfy all the conditions on them in just one pass," as follows. Suppose there is a variable with enough degrees of freedom and few enough constraints that its value may be changed so as to satisfy all of its constraints. This variable may be re-evaluated last, and the constraints on it will be satisfied. So we may eliminate this variable and its constraints from the graph, and look for an ordering on the remaining variables in the same way. If the graph can be completely dismantled in this fashion, then the entire system may be solved one variable at a time, in the reverse of the order in which they were eliminated from the graph. (A small example is given below.)

Since Sketchpad's variables are essentially always points (in the plane), and its constraints always remove one degree of freedom, finding an eliminable variable means finding a point with no more than two constraints on it. Solving for that point entails solving a two degree of freedom system. Typically this system will be a simple geometric intersection; for instance, the unknown point may be constrained to be horizontal with one known point and a fixed distance from another. These systems Sketchpad could solve as tiny relaxation problems. (Relaxation may be slower than closed-form solutions would be here, but it has the elegant property that the same solving routine may be used regardless of what particular constraints are involved.)

Though they are far from being the only application that Sutherland suggests, he does discuss the use of Sketchpad for drawing and animating mechanical linkages. One of his examples is the four-bar linkage, one possible construction of which is shown in Figure 28. (Here he is seen to

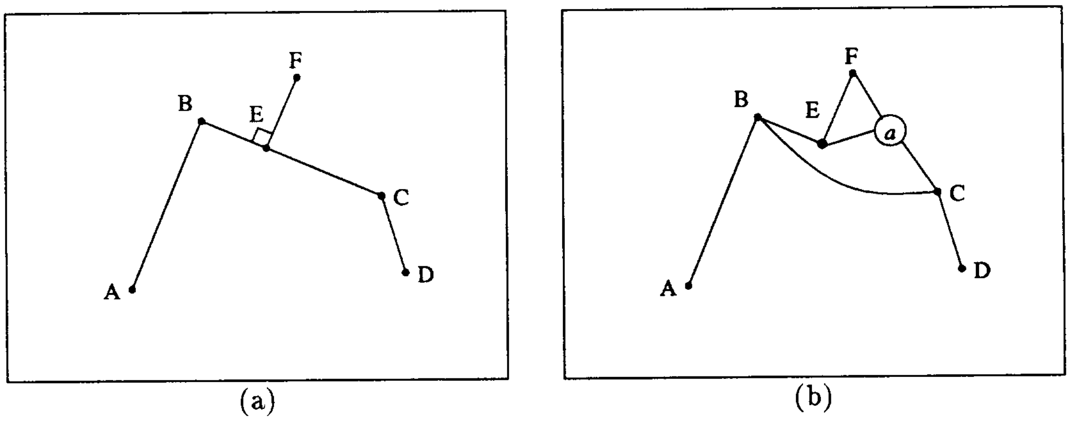


Figure 28: (a) A four-bar linkage as it might be drawn in Sketchpad. (b) The corresponding constraint graph. The angle constraint (marked 'a') is a multi-edge, connecting three points; the remaining edges are distance constraints.

share the spirit of rapid and exploratory user interface with which we have tried to approach the

[10]All this was done at a time when doubly-linked circular lists were enough of a novelty that Sutherland feels compelled to explain their use.

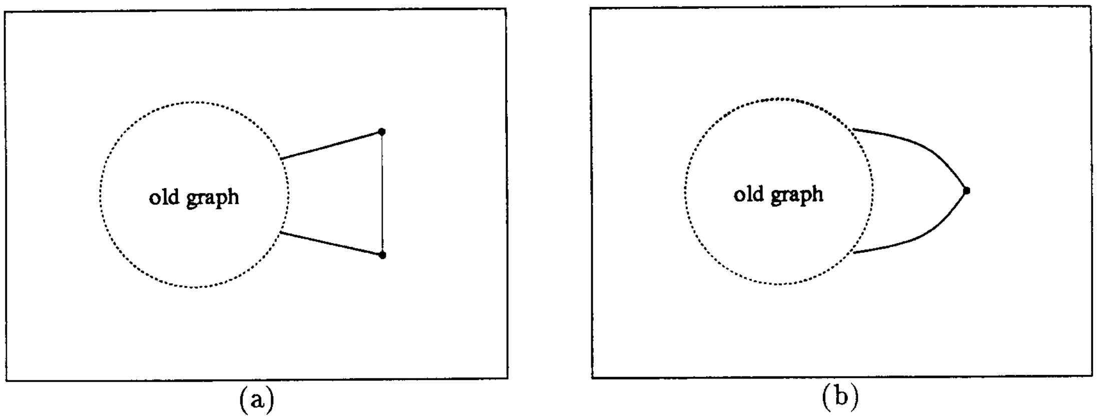


Figure 29: The basic solvable units. Each step of a solution adds this much to the solved mechanism graph. (a) Our mechanism editor. (b) Sketchpad.

mechanism editor: "Total time to construct the linkage was less than 5 minutes, but over an hour was spent playing with it.") The linkage is built of line segments whose end points have been constrained to lie a fixed distance apart; call these *sticks.* A revolute joint is merely two sticks that share an end point. Thus, coincidence constraints are something of a special case in Sketchpad (and in ThingLab). Instead of there being two separate points constrained to be equal, the two points are *merged* into a single variable. This can simplify the constraint graph considerably.

The four-bar linkage can be solved using the method of propagation of degrees of freedom, by eliminating the points in the order F, E, B. (Points A and D are grounded, while C is fixed by the input motion.) This order lets each point be eliminated while it has only two constraints on it; this guarantees that the diagram can be solved in the reverse order B, E, F. The B step fixes the links AB and BC, and is analogous to our RRR solver, with the distinction that it solves for points rather than for link coordinate systems. One could say that it solves for the joints instead of the links.

This method can also successfully find orderings for other pin-joint mechanisms, such as the six-bar mechanism in Figure 6 or the wine rack in Figure 20. In fact, this ordering process functions exactly as the planning phase does in our editor, but in reverse order.

Our planner could be made to work in reverse. Consider the two links that are fixed last by a plan; their graph will always look like the "open triangle" shown in Figure 29a. By repeatedly eliminating such open triangles from the mechanism graph, we could build up our plans in reverse order. This backwards planner would completely solve exactly those graphs that our forwards planner completely solves. The only reason to prefer forwards to backwards is that, when the planner gets stuck, at least some of the mechanism still moves. [11]

In short, this directional difference between the mechanism editor's planner and Sketchpad's one-pass planner is largely superficial. The fundamental distinction between the two planners is that in Sketchpad, the variables are points, while in the mechanism editor, the variables are coordinate systems. Our basic solvable unit – the graph that we can reduce to a point – is the open triangle in Figure 29a, containing two nodes with three degrees of freedom each, plus three constraint edges, each constraining two degrees. Figure 29b is Sketchpad's basic unit, containing one node with two degrees of freedom, plus two edges each constraining one degree. Sketchpad's basic unit has the advantage of being simpler than ours. In particular, the method of propagation of degrees of

[11] As will be discussed below, it is best to be able to solve both forwards and backwards; see Section 6.3.

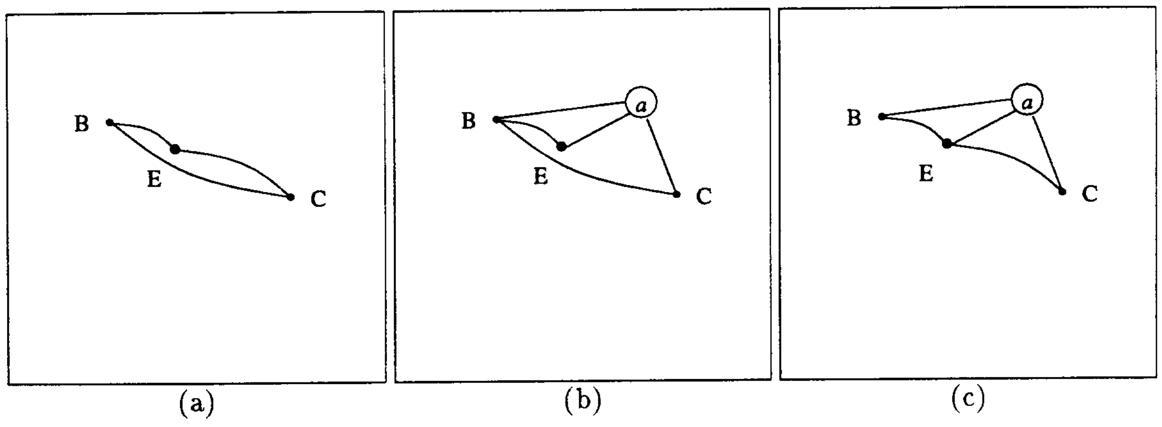


Figure 30: Three ways of expressing the line segment. Only the first two can be solved by the one-pass method.

freedom does not apply straight-forwardly to our graphs: neither node can be solved first, without reference to the constraints on the other node. Our advantage is that we have a built-in notion of rigid links, while in Sketchpad, a rigid shape must be expressed by sufficiently constraining some subdiagram. Thus, our graphs will typically have fewer edges, particularly if link shapes with many vertices are involved.

Sketchpad could be extended to detect rigid subdiagrams. As it stands, it must re-solve the shapes of the links each time the mechanism moves. Whether or not it can do this with the one-pass method may depend on exactly how the link shape is expressed. This danger can be seen even in the four-bar example above. Consider the rigid line segment containing points B, E, and C. Figure 30 shows three possible constraint graphs that express this rigidity (with neither underconstraint nor overconstraint). The first, which imposes three distance constraints, and the second, which imposes collinearity and two distances from point B, both have one-pass orderings. But the third, which also uses collinearity but takes both its distances from point E, does not have a one-pass ordering; to solve it, Sketchpad would have to resort to a global relaxation.

In all, the method of propagation of degrees of freedom is surprisingly effective in solving mechanisms made up of revolute joints. Since it can do the the work of our RRR solver, and it can do the ordering that our planner does, it can solve anything that our mechanism editor can – as long as it can solve for the rigid link shapes when necessary. For sticks and triangles, it is not difficult to find cooperative ways to express link shapes. For more complex links, there are more opportunities to build something that cannot be ordered.

Prismatic joints are more problematic. They can be expressed by means of collinearity constraints, and Sketchpad can solve certain cases. But, for instance, solving an RPR configuration (see Figure 15) requires solving for the line of the slider first; Sketchpad can only solve for points. A solution to this is to introduce lines as variables, rather than as just connections between two points. For instance, a line could be represented by its slope and offset, or by the coefficients of its line equation. This numerical representation can then be solved for. Todd and Cherry make this extension; we discuss their system below, in Section 6.4.

To summarize, we gain in efficiency, at move-time at least, because we do not have to resolve the shapes of rigid links for every cycle, and because we solve the base-case geometric intersection problems in closed form, rather than by using mini-relaxations. (Of course, we use trigonometric functions and square roots, which are computed by convergent series. Relaxation might actually have been faster on the type of machine Sutherland was using.) Our graphs are simpler if the links have fancy shapes. Sutherland's graphs are simpler if the links are just line segments. In all, our primitives are better suited to our problem – just as his are better suited to geometric constructions (like the conic section example he gives), circuit diagrams, etc.

Sketchpad falls back on relaxation when it cannot solve the system one variable at a time; our mechanism editor would definitely benefit from such an ability. This also handles overconstrained systems, by letting the constraints fail. (Sutherland uses this to advantage in his bridge-load example.) Sketchpad's backwards planner easily recognizes and solves underconstrained systems, by finding variables with only one remaining constraint edge rather than two.

6.3 ThingLab

Alan Borning brought Sketchpad up into a modern, object-oriented programming language (namely Smalltalk), and made several interesting extensions and variations. The result was ThingLab [Borning 79]. In ThingLab, as in Sketchpad, constraints typically can return a numerical value that is zero only if they are satisfied. But in addition, attached to each ThingLab constraint is a set of executable *methods*, each of which cause the constraint to be satisfied (by recomputing the values of the constrained variables). From these methods, ThingLab composes its plans. These methods are faster than the mini-relaxations used by Sketchpad.

A ThingLab method can only take into account one constraint (i.e. the constraint to which it is attached). This means that, in order for ThingLab to solve the sorts of geometric constructions we are discussing, more complex constraints must be used. For instance, consider a point that is constrained to be a fixed distance from one point, and collinear with another two. Sketchpad expresses this with two separate constraints – let us call these *atomic* constraints – and uses a mini-relaxation to solve them. In ThingLab, a method can be implemented to solve for this point in closed form, but the method must be attached to a single *bundled* constraint; this constraint will reference all four of the points involved.

In ThingLab, a mechanism can be expressed in terms of atomic constraints, just as it would be in Sketchpad, but ThingLab cannot solve such mechanisms without resorting to global relaxation. Fortunately, as Borning explains under the heading "Using Multiple Views to Avoid Relaxation" ([Borning 79] page 69), ThingLab can simultaneously represent both atomic constraints and bundled constraints on the same mechanism. In ThingLab, a constraint type can be created for each of our base cases, and our solvers may be implemented as methods for these constraints.[12] If a mechanism is built out of these bundled constraints (either initially or after being built from atomic constraints), then ThingLab will find the same closed-form solutions as the mechanism editor. The result is a system remarkably like LINKPAC: once the user has built the mechanism out of solvable submechanisms, the program can solve it. Actually the ThingLab planner is a little smarter, in that it does not require the user to re-express the mechanism when she selects a new input joint.

For solving constraint graphs, ThingLab uses both the one-pass method that Sketchpad uses and a new method that Borning calls "propagation of known states." Here, the program looks for a variable whose value can be completely determined from a single constraint. If all the other variables involved in the constraint have already been fixed by earlier portions of the plan, then this variable may be fixed by the next step of the plan. If this technique is transferred to the setting of Sketchpad, where multiple constraints may be solved simultaneously, then it becomes more general: a variable can be fixed when its value is determined by a *set* of constraints, all

[12]The PRR and RPR cases should be implemented as two methods on a single constraint type. This constraint can then be used on any triangle with two revolute joints and one prismatic joint.

of whose other variables are already fixed. The technique then becomes a forwards planner to complement Sketchpad's backwards planner. As we have mentioned, the set of graphs that may be solved completely is the same for forwards planning as for backwards planning. But when there is a portion of the graph that cannot be solved, having both forwards and backwards techniques available lets the system solve as much of the graph as possible.

Like the one-pass method, the method of propagation of known state cannot be directly applied to the mechanism editor's graphs. Since our variables (nodes) have three degrees of freedom, and our constraints only take away two, we never have enough constraints to completely determine the values at any single node (except in the case of overconstraint, which we don't need to solve at all).

When both techniques fail, ThingLab falls back on the slow but effective technique of global relaxation. But first it uses both the forwards and backwards techniques to determine which variables may be directly computed; only the remaining, unsolved, variables participate in the relaxation. ThingLab cleverly reduces the set of participating variables even further, by looking for a small subset such that, once the subset has been solved for, the others may be computed directly. We have mentioned that the mechanism editor would greatly benefit from having a global relaxation technique to fall back on; ThingLab provides an excellent model for such a technique.

ThingLab is very general and elegant in concept – a software laboratory for simulation, with a direct-manipulation interface. In designing the mechanism editor, we attempted to emulate ThingLab, but to specialize it for the editing of mechanisms. By specializing, we were able to provide a data structure that avoids the repeated resolving of rigid geometry, a planner that recognizes (and solves) certain bundled constraints automatically, and a user interface that is streamlined for rapid assembly of mechanisms.

6.4 Todd and Cherry

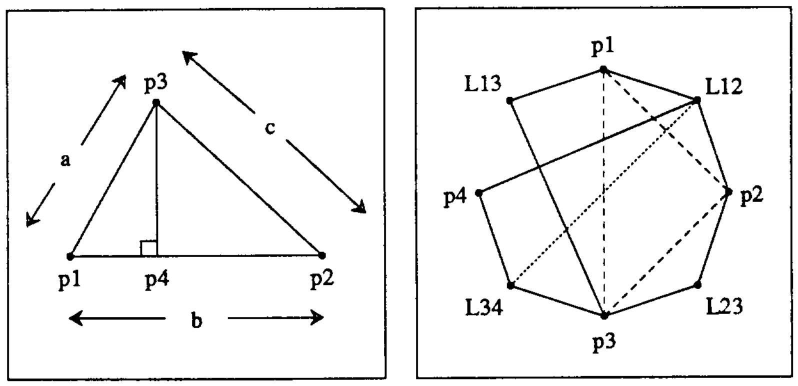
Recently, Todd and Cherry at Tektronix Laboratories have developed a system for calculating properties of dimensioned, planar drawings. [Todd 88] [Todd 89] The input is a drawing made up of points and straight lines, with some distances between points given, and some angles between lines given; Figure 31a shows an example. From this drawing the system builds a constraint graph like Sketchpad's, except that in addition to a node for each point in the diagram, there is also a node for each line in the diagram. Whenever a point is on a line, there is an "incidence" constraint between the two nodes; see Figure 31b.

They solve the constraint graph by iteratively eliminating nodes with exactly two constraints on them; this much is exactly like Sketchpad. But instead of using mini-relaxation, they use a closed-form *construction* to solve each two-constraint system. For instance, the program includes a construction for the point that is incident on each of two known lines, and a construction for the point incident on one known line and a fixed distance away from a known point. (Constructions would be called solvers in the mechanism editor, or methods in ThingLab.) A node may only be eliminated if its two constraints match an available construction.

As explained in Section 6.2, Sketchpad can already animate any revolute mechanism that our editor can animate. With both points and lines as primitives, Todd and Cherry's system can solve any mechanism our editor can solve, revolute or prismatic.[13] The caveat is the same as for Sketchpad, namely, that the system must be able to solve the rigid link shapes.

Interestingly, their system uses only symbolic computation. A length is typically an expression like "a" or "r/2", rather than a numeric value. This allows them to prove theorems by means of

[13]A constraint type for the distance between a point and a line would have to be added to their system, with some new constructions as well. This would be a trivial extension.



(a) (b)

$$p_2 = (0,0)$$
$$p_1 = (0,b)$$
$$p_3 = (\frac{-a^2+b^2+c^2}{2b},\ldots)$$
$$p_4 = (\frac{-a^2+b^2+c^2}{2b},0)$$

(c)

Figure 31: An example from Todd and Cherry. (a) The input drawing. (b) Its constraint graph. The dotted edge is an angle constraint, the dashed edges are distance constraints, and the other edges are incidence constraints. (c) The output is in the form of symbolic equations involving the variables in the drawing. [Todd 88]

diagrams, such as by showing that the distance between two particular points is identically zero. It would be interesting to compare the capabilities of their system with those of a general-purpose symbolic equation solver.

The use of symbolic computation rules out any sort of relaxation method, and places renewed emphasis on finding solutions in closed form. Unfortunately, their terminology does not distinguish between properly constrained graphs that happen to be uncontractable[14] and graphs that are actually overconstrained. This risks confusing the notions of "well defined" and "computable by this algorithm." Also, they do not discuss non-genericity, though each of their theorems is a clever example of it.

6.5 Physically-based modeling

Physically-based modeling has recently been a very active area of computer graphics research. This work presents a radically different approach to the animation of constrained, articulated models. In this approach, objects are assigned masses and moments, and interact through forces and torques. The system is governed by differential equations adapted from classical dynamics; these equations are solved numerically to find the motions of the objects. Various forms of this approach have been demonstrated with animations of chains [Isaacs 87], flags [Platt 87], cubes of gelatin [Platt 88], desk lamps [Witkin 88], and even mechanisms [Witkin 87].

Computing the motions is largely a matter of using the equation $\mathbf{F} = \mathbf{ma}$ to find the accelerations of the system's state variables and then numerically integrating the accelerations over time. Forces may arise from object interaction (e.g. collisions), from the internal stresses of an object (e.g. the stretched or compressed springs of a finite-element model), from outside forces (e.g. gravity, wind), or from user interaction (allowing the user to push or pull parts of the system into place).

A constraint may be represented by an error function, just as in Sketchpad; the closer the constraint is to being satisfied, the closer the function's value is to zero. *Constraint forces* may be introduced that push the error values towards zero. The magnitude of a constraint force will depend on the state of the system, on the non-constraint forces, and on the other constraint forces.

[14]Unconstructible, in their terminology.

For example, to keep a certain point on an object nailed to the wall, the constraint force should exactly counteract the acceleration of that point (relative to the wall) due to the other forces in the system.[15] The other forces include forces arising from other constraints; in general, finding the constraint forces requires solving a system of simultaneous equations. [Barzel 88]

Using this approach requires careful attention to the intricacies of numerical methods. The overall flavor is quite different from that of our mechanism editor, with its abstract graphs and closed-form solutions! One would expect the numerical methods to be computationally expensive, but surprisingly nimble systems have been demonstrated. In particular, Witkin has demonstrated editors that with a dozen or so links are still comfortably interactive on a Silicon Graphics IRIS 4D/70 graphics workstation.[Witkin 89] (These interactive systems often have a loose or flabby feel; for example, when the user positions an object with the mouse, the object's position may exponentially decay towards the mouse position over a half second or so. Such exponential decays help to prevent the differential equations from becoming *stiff*, which would cause the numerical methods to become either unstable or excessively time-consuming.)

Simulating equations from physics in this way is evidently an extremely powerful and general technique. The simulation provides dynamics, not just kinematics. Mechanisms need not be contractable (cf. Section 5.2.3) to be animated. The issue of branching is automatically handled by the momenta of the masses involved. (Ideally, the user is provided with interactively controlled forces strong enough to kick the model out of any undesired mode.) Since the system includes forces that tend to minimize the amount by which constraints are violated, both broken joints and overconstrained systems behave in a reasonable manner.[16] The behavior of broken mechanisms resembles that of the "stretch" policy illustrated in Figure 16.

Adding a new joint type to the mechanism editor is not prohibitively difficult, but does require deriving and implementing a suite of new solvers for the various situations in which the joint would be used. In contrast, adding a new joint to a physically-based system might only require incorporating a new constraint force equation; the actual solution machinery would likely not need to be changed at all. Extensions such as friction, gravity, and load computations - even objects bending under a load - are difficult to contemplate in a closed-form system, yet can be easily incorporated into a physical simulation.

Of particular relevance to the design of mechanisms is Witkin, Fleischer, and Barr's demonstration [Witkin 87] of a system that allows the parameters of a mechanism - lever lengths, pivot positions, etc. - to be smoothly varied, even while the mechanism is being animated! Since the system already uses numerical optimization in its simulations, parameters of a mechanism could be optimized against an objective function. This raises the exciting possibility of an interactive, visual system that combines type synthesis with dimensional synthesis.

6.6 Discussion

The basic distinction between our mechanism editor and the Sketchpad-like systems that we have discussed (Sketchpad, ThingLab, and Todd and Cherry's diagram solver) is that we describe rigid links in their own local coordinate systems, and we use these coordinate systems as primitive variables in our constraint graphs. This more cleanly separates the geometry of the mechanism

[15]However, if the point on the object starts out away from the nail - i.e. if the constraint is violated - then one component of the constraint force should be a *restoring force* that will push the point towards the nail.

[16]To handle overconstraint properly, the implementation must be able to solve non-square systems of equations. The same is true of underconstraint (the equations are non-square in the other direction). Most of the systems we have cited are capable of solving non-square systems.

from its topology. It also frees the user from having to build rigid links out of simple constraints, and it frees our program from having to solve for these shapes.

Each of these Sketchpad-like systems employs a different flavor of constraint solving. Sketchpad uses the same iterative numerical procedure for all base cases (where by base case, we mean a small, solvable configuration of constraints); ThingLab uses closed-form numerical procedures attached to bundled constraints; and Todd and Cherry use closed-form symbolic arithmetic procedures attached to each base case type. Our editor uses yet another variant: closed-form numerical procedures attached to each base case type. Closed-form procedures are more efficient than iterative ones, and numerical procedures are more efficient than symbolic ones. We have combined this with a user interface tuned for our specific application (namely type exploration), and the result is a useful and usable design tool.

But we have relied too much on our closed-form solvers. As a result, there are properly constrained mechanisms our editor cannot solve at all. (These are the uncontractable mechanisms discussed in Section 5.2.3.) A relaxation method, or other numerical method, could handle these cases; this would be an important extension of our system. We would do well to borrow the techniques used in ThingLab to make the set of variables that must be numerically solved as small as possible.

The mechanism editor's handling of broken joints is less than graceful, and underconstrained mechanisms it simply refuses to animate at all. The editor should be extended to incorporate reasonable rules for making the arbitrary choices in placing underconstrained or broken mechanisms. Sketchpad and ThingLab handle underconstraint reasonably well, simply by leaving extra degrees of freedom unchanged. (Their authors do not discuss breaking.)

Physically-based modeling is a very powerful and flexible tool. Given a sufficiently powerful set of numerical methods, this tool subsumes underconstraint, overconstraint, breaking, branching, and all base cases under a single metaphor. This same metaphor allows for the simulation of dynamics and for numerical optimization, such as optimization for dimensional synthesis. These capabilities could prove to be tremendous advantages for a physically-based system for interactive mechanism design. This approach should certainly be investigated.

In trying to make a practical physically-based mechanism design system, the main difficulty we expect to encounter is speed. Whether animations can be made interactive will depend on the speed of the available computer hardware. It will also depend on the size of the systems being animated. The time required to execute one of our closed-form plans is linear in the number of links in the mechanism. (Assuming a constant bound on the number of joints on any one link, our planning algorithms are linear too, except for the detection of rigid submechanisms.) For a physically-based method, the equations may be of linear size (assuming the same constant bound, and taking sparseness into account), but typically quadratic or even cubic time is used to solve them. [Barzel 88] Closed-form methods may be the only way to animate, say, forty or a hundred links in real time. Rates of convergence are also important to consider; especially if we insist that the animation appear rigid, rather than decaying towards valid positions over multiple updates, these rates are going to have a significant impact on speed.

As we have said, the closed-form methods presented here need to be augmented by a numerical method in order to be complete. The physically-based approach provides one such method; numerical methods that only solve geometry, rather than providing realistic dynamics, may be faster. In any case, a numerical method that can solve the uncontractable mechanisms will be able to solve the contractable ones as well. Why use closed-form methods at all? The principal justification is speed. In other words, while the closed-form approach needs to be augmented by a numerical method in order to be complete, we expect that numerical systems will need to be augmented by closed-form methods in order to be interactive. The example at the end of Section 5.2.3 indicates that there may be a rich variety of techniques that combine numerical and closed-form methods, techniques that benefit from the strengths of both.

Through this analysis of our mechanism editor and its ancestors, we have learned how best to tune our basic approach to make an effective and efficient editor for mechanism type synthesis. The next step is to fuse this approach with numerical methods. This will require close study of how physically-based modeling performs for mechanisms, how our closed-form methods may be refined, and how these techniques may be most effectively combined.

7 Future Work

Earlier sections have discussed several possible areas for further research:

- The nature (and relative size) of the set of uncontractable graphs. Solution techniques for uncontractable mechanisms. The existence or non-existence of closed form solutions. (Section 5.2.3.)
- Analyzing the capabilities and performance of physically-based modeling, and other numerical techniques, for mechanism animation. (Section 6.6.)
- Hybrid techniques for using both closed-form and numerical methods to animate mechanisms with no closed-form solution. Minimizing the number of variables solved numerically. (Implementing a backwards solver in addition to the forwards solver would be a step in this direction.) (Sections 5.2.3, 6.3 and 6.6.)
- Detecting and solving non-generic mechanisms. (Section 5.2.4.)
- Efficient algorithms for detecting rigid subgraphs. (Section 2.4.2.)
- Animating underconstrained mechanisms: policies for choosing which free joint to fix, and at what value to fix it. (Section 5.2.2.)
- Policies for mechanisms that break. Displaying the breaking point, clearly and efficiently. (Section 3.2.5.)
- Interfaces to let the user handle branching more naturally. (Section 3.2.6.)
- Interfaces to allow the user to control several input parameters at once. (Sections 5.2.1 and 4.4.)
- Displaying velocities, accelerations, forces, or stresses in a clear and useful way. (Section 6.1.)

One feature that would make the editor much more useful would be the ability for the user to *adjust* the geometry of a mechanism in a continuous manner. As it stands, any change in a link's shape must be made by deleting the link (along with all its joints) and re-entering it from scratch. Also, there is no way to reposition a link except by animating the whole mechanism. At a minimum, there should be an editing function that allows the user to reposition a link vertex with the mouse; if there is a joint at this vertex, this operation should move the relevant vertex of the other link as well. Further thought should be given to the set of editing operations provided, with the goal of allowing smoother and more convenient exploration of mechanism geometries.

Finally, an obvious possibility is to investigate similar type exploration systems for three-dimensional mechanisms. The same method for finding closed-form solutions – i.e. examining the graph of a mechanism's topology, finding subgraphs that match base cases, and contracting each subgraph as we solve it – may be used in three dimensons. Here links have six degrees of freedom; some joints have one (revolute, prismatic, helical) while others have two (cylindrical) or three (spherical, planar). Suh and Radcliffe present solvers for the RSSR, RRSS, and RCCC base cases [Suh 78]; likely, more are possible. The question of completeness comes up again: for what portion of the three-dimensional mechanisms that users would actually want to animate can we find solutions in closed form? Then there is also the question of user interfaces for interaction with simulated three-dimensional worlds, which is an entire field of research by itself.

8 Conclusions

The graph of a mechanism is an abstraction that removes the mechanism's geometry, leaving only its topology. There are several equivalent ways of viewing the mechanism graph. Each node of the graph represents a link, or (equivalently) a coordinate system, or a set of points whose relative positions are known. Each edge represents a joint, or a parameterized transformation between the coordinate systems of its two nodes. The mechanism graph is also the constraint graph. With this data structure, it is simple to analyze a mechanism for overconstraint, underconstraint, and degrees of freedom.

The mechanism graph also provides an orderly setting for building plans that solve mechanisms. We form our plans inductively, using the three-link, three-joint rigid mechanisms as base cases. Each step in the plan – each application of a base case – fixes the parameters of a set of joints. During animation, the user's mouse fixes the parameters of the input joints. Fixing a joint defines the relative positions of its two links. We represent this by contracting the corresponding edge in the mechanism graph; when the graph has been contracted to a single node, we have solved the positions of all the links in the mechanism.

Since we can solve our base cases in closed form, our plans are always closed-form solutions. This works for many mechanisms, though there are simple mechanisms – as simple as six bars – that cannot be solved by this technique of contractions, and in fact do not appear to have closed form solutions at all.

During animation, while the plan is being executed repeatedly, the mechanism's geometry comes back into play. When an input joint is turned too far, the mechanism breaks. It is important to give the user clear feedback when this happens; we have discussed the trade-offs of several ways of doing this. We have also discussed branching, i.e., the fact that there are typically two solutions to each base case. Usually the desire for continuity of motion is enough to determine which branch is appropriate. But near the breaking point, the two branches are close together. Ideally, the user should have a convenient means of controlling which branch is taken.

In non-generic mechanisms, coincidences of geometry conspire to change the mechanism's effective topology, possibly changing the number of degrees of freedom. This situation can arise dynamically during animation. Our system does not notice these coincidences.

We have discussed our system alongside several other Sketchpad-like systems. They are more alike than they are different. The mechanism editor uses coordinate systems, rather than individual points, as the variables being constrained; this is better suited to our specialized task of animating mechanisms. Our planner is automatic, in that it does not require the user to express his mechanism in terms of our base case mechanisms. (The same is true of Sketchpad, but not of ThingLab.) Our planner uses a forward solution technique, one that finds the steps of a plan in the same order that the steps will be executed. There is an equivalent backward technique. Ideally, the mechanism editor would be able to use both forwards and backwards methods, in order to solve as much as possible of an uncontractable mechanism. Relaxation or other numerical methods should then be used to solve the uncontractable core.

We found the Lisp programming language to be helpful in thinking about the program, and for implementing some aspects – in particular, its closures provided a clean and convenient way to separate plan time from move time – but we found it slow and somewhat unwieldly for general program development. Think in Lisp, write in C++.

The editor benefits from having both its user interface and its constraint-solving technique tuned for the task of mechanism type exploration. Closed-form solutions are efficient, but do not always exist. For a given application – with given hardware limitations, typical size of mechanisms, and other requirements – a hybrid technique should be developed that represents an appropriate trade-off among the universality of numerical methods in general, the power of physically-based modeling in particular, and the performance of closed-form solutions.

Acknowledgements

This work was done under the guidance of Carlo Sèquin, whose advice, support, and inexhaustible talent for generating valuable ideas played an enormous role throughout the process. Early discussions with C. W. Radcliffe were inspirational. Late discussions with John Canny provided a mathematical viewpoint, and helped to clarify several important areas. Eric Bier, Andrew Witkin, Michael Gleicher, Robert Duisberg, and Alan Borning generously took time to educate me. I am also grateful to my colleagues at Berkeley, particularly Seth Teller, Henry Moreton, Nina Amenta, Ziv Gigus, Mark Segal, and Bob Boothe, for providing discussion, inspiration, assistance, moral support, and T-shirts.

Many thanks to Luis Miguel, Hervé Da Costa, Ken Lutz, Joseph Hohl, Brian Shiratsuki, and Keith Sklower for their help in assembling and maintaining the computing environment. Many thanks also to Kathryn Crabtree, who shields us from virtually all administrative concerns.

For the remarkable patience and understanding shown by my friends, family, and co-workers, all of whom put up with me while I was doing this work, I am deeply indebted. Nancy Botkin and Ernie Rideout each came to my rescue with advice and active support on multiple, crucial, occasions. Judit Mandi was a marvelous study partner; I would like to thank her and to congratulate her on her medical license. Doug Kay and the computer graphics department of Industrial Light and Magic have been very generous. I would also like to particularly thank Peter Goldberg, David Anderson, Rob Culverhouse, Alex Melnick, Bert Enderton, Mom, and Dad (who won't like the split infinitive).

References

[Barr 88]	Alan H. Barr, "Teleological Modeling," *Developments in Physically-Based Modeling* course notes, SIGGRAPH '88, ACM, August 1988.
[Barris 88]	Wesley C. Barris, Sridhar Kota, Donald R. Riley, and Arthur G. Erdman, "Mechanism Synthesis Using the Workstation Environment," *IEEE Computer Graphics & Applications*, pp. 39-50, March 1988.
[Barzel 88]	Ronen Barzel and Alan H. Barr, "Controlling Rigid Bodies with Dynamic Constraints," *Developments in Physically-Based Modeling* course notes, SIGGRAPH '88, ACM, August 1988.
[Bier 86]	Eric Allan Bier and Maureen C. Stone, "Snap-Dragging," SIGGRAPH '86, ACM, August 1986.
[Borning 79]	Alan Borning, *ThingLab – A Constraint-Oriented Simulation Laboratory*, Xerox Report SSL-79-3, Xerox PARC, July 1979.
[Borning 87]	Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf, "Constraint Hierarchies," *OOPSLA '87 Proceedings*, pp. 48-60, October 1987.
[Crawford 85]	R. H. Crawford, W. W. Charlesworth, and M. J. Bailey, "The Design, Analysis and Display of Three-Dimensional Mechanisms Using a CAD Executive," *Mechanism and Machine Theory*, Vol. 20, No. 4, pp. 251-256, 1985.
[Guillemin 74]	Victor Guillemin and Alan Pollack, *Differential Topology*, Prentice-Hall, Englewood Cliffs, N.J., 1974.
[Isaacs 87]	Paul M. Isaacs and Michael F. Cohen, "Controlling Dynamic Simulation with Kinematic Constraints, Behavior Functions and Inverse Dynamics," Proc. SIGGRAPH '87, pp. 215-224, ACM, July 1987.
[Luck 85]	Kurt Luck, Karl-Heinz Modler, and Jörg Reber, "Computer-Aided Design in Mechanisms," *Mechanism and Machine Theory*, Vol. 20, No. 4, pp. 297-302, 1985.
[Mittelstadt 85]	William A. Mittelstadt, Donald R. Riley, and Arthur G. Erdman, "Integrated CAD of Mechanisms," *Mechanism and Machine Theory*, Vol. 20, No. 4, pp. 303-311, 1985.
[Myklebust 88]	Arvid Myklebust, "Mechanical Computer-Aided Engineering," *IEEE Computer Graphics & Applications*, pp. 24-25, March 1988.
[Olson 85]	Daniel G. Olson, Arthur G. Erdman, and Donald R. Riley, "A Systematic Procedure for Type Synthesis of Mechanisms with Literature Review", *Mechanism and Machine Theory*, Vol. 20, No.4, pp. 285-295, 1985.
[Platt 87]	John Platt, Demetri Terzopoulos, Kurt Fleischer, and Alan Barr, "Elastically Deformable Models," *Topics in Physically-Based Modeling* course notes, SIGGRAPH '87, ACM, July 1987.

[Platt 88] John C. Platt and Alan H. Barr, "Constraint Methods for Flexible Models: the Tutorial Notes," *Developments in Physically-Based Modeling* course notes, SIGGRAPH '88, ACM, August 1988.

[Ryan 81] Daniel L. Ryan, *Computer-Aided Kinetics for Machine Design*, Marcel Dekker, New York, 1981.

[Suh 78] C. H. Suh and C. W. Radcliffe, *Kinematics and Mechanisms Design*, Krieger, Malabar, Florida, 1978.

[Sutherland 63] Ivan Sutherland, "Sketchpad: A Man-Machine Graphical Communication System," *Proceedings of the Spring Joint Computer Conference*, pp. 329-345, AFIPS, January 1963.

[Thatch 88] B. R. Thatch and Arvid Myklebust, "A PHIGS-Based Graphics Input Interface for Spatial-Mechanism Design," *IEEE Computer Graphics & Applications*, pp. 26-38, March 1988.

[Todd 88] P. H. Todd and G. W. Cherry, "Symbolic Analysis of Planar Drawings," Tektronix Laboratories Technical Report No. CR-88-03, February 1, 1988.

[Todd 89] Philip Todd, "A k-Tree Generalization That Characterizes Consistency of Dimensioned Engineering Drawings," *SIAM Journal on Discrete Mathematics*, Vol. 2, No. 2, May 1989.

[Witkin 87] Andrew Witkin, Kurt Fleischer, and Alan Barr, "Energy Constraints On Parameterized Models," Proc. SIGGRAPH '87, pp. 225-232, ACM, July 1987.

[Witkin 88] Andrew Witkin and Michael Kass, "Spacetime Constraints," SIGGRAPH '88, pp. 159-168, ACM, August 1988.

[Witkin 89] Andrew Witkin and Michael Gleicher, "forcefields," computer program, 1989.

[Witkin 90] Andrew Witkin, Michael Gleicher, and William Welch, "Interactive Dynamics," *Computer Graphics*, Vol. 24, No. 2, ACM, 1990.