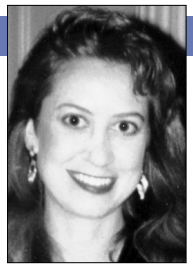
Sketching intelligent systems



By Marti A. Hearst
University of California, Berkeley
hearst@sims.berkeley.edu

Those who have been reading this feature on a regular basis no doubt have noticed my penchant for topics on human-computer interaction. An important general trend in intelligent systems is that of acknowledging the smarts of the users of our systems, thus complementing efforts to put smarts into the systems themselves. The growing importance of HCI is underscored by the naming of Douglas Engelbart as last year's recipient of the prestigious ACM Turing Award. Engelbart was also showered with honors at this year's ACM conference on human computer interaction (SIGCHI) in Los Angeles (*http://www.acm.org/sigchi* and *http://www.acm.org/pubs/contents/proceedings/chi/*). A trend that surfaced at this meeting was the move toward more natural, human-centered forms of interaction. Many presentations explored alternatives to the traditional keyboard, mouse, and monitor, replacing these with tangible physical artifacts. Examples included embedding of interfaces into an easy chair, a ping-pong table, name tags, children's plastic beads, and triangular shapes that store audio clips, which can be used to create stories based on which sides of the triangles are snapped together.

In this installment of Trends and Controversies we look at another aspect of this trend toward more natural forms of interaction: the use of sketch-based interfaces for the design of intelligent systems.

Our authors contend that current computer interfaces are too formal and precise for creative tasks such as design. When working out ideas and brainstorming, people often sketch their thoughts informally on paper and whiteboards. Sketches, by their very informalness, invite collaboration and modification. But after sketches are done on paper they must be transferred to the computer for further processing.

Why not use a computer for the sketching process itself? As Mark Gross points out below, there is an inherent contradiction in today's sketching programs—users select graphical primitives from a palette, resulting in clean, precise-looking diagrams despite the fact that only a sketch was intended. This leads to a feeling of commitment to the sketch as originally formulated, as opposed to the invitation to adjust and change that is normally associated with sketches.

By contrast, this month's authors (Mark Gross, James Landay, and Tom Stahovich) describe sketch-based user interfaces that honor the informal nature of the sketch and allow users to switch easily and naturally between sketchy, informal prototypes, and precise, formal reifications of designs. In these pages, sketches are used for the design of user interfaces (Landay), mechanical devices (Stahovich), and web pages (Gross and Landay), to retrieve images from a large collection (Gross), and to specify parameters for simulations (Gross).

Stahovich points out that a sketch is useful as a reasoning tool because it presents a particular example of a problem to think about as a starting point, when the general case is too abstract and elusive. In Gross's image-retrieval system, the sketch is an example of the kinds of images of interest. In Landay's user interface design system, the sketch is an example of the functionality and basic interaction styles of the interface to be built. In Stahovich's mechanical-device-design system, the sketch is just one example from which the system infers an entire family of designs.

Although a sketch produces a concrete example of a family of designs, its informal nature somewhat paradoxically also allows details to be left unspecified. (This is why a sketch is different from other ways of specifying examples.) Landay points out the utility of the ambiguity of a sketch, which makes it possible for a designer to delay decisions about details until an appropriate stage in the design process.

The systems described in the essays use AI techniques to convert sketches to their more precise computerized counterparts. Gross's Electronic Napkin uses symbol- and configuration-recognition techniques to define a visual language grammar. Landay's Silk uses a statistically-based gesture recognizer and a rule system to combine primitive components into more complex ones, and dynamic storyboarding, which allows users to define user intervals that contain both standard widgets and novel components. Stahovich's SketchIT uses qualitative-reasoning techniques to identify mechanical interactions between the parts of the sketch of a device and a state-transition diagram of its desired behavior.

All three authors emphasize the importance of using the most appropriate input modality for the task at hand. How to automatically determine what tools to offer when is an open question. Nevertheless, the work presented here represents real progress towards the the goal of natural, humanistic interaction with computer systems.

—Marti Hearst

The proverbial back of an envelope

Mark D. Gross, Department of Planning and Design, University of Colorado

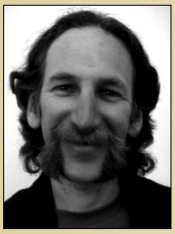
People draw diagrams and sketches to think, argue, and communicate about problems in domains ranging from mechanical engineering to music. To understand a problem better, mathematician George Polya advised, "Draw a figure ... even if your problem is not a problem of geometry."[1] We can debate whether making a figure really helps people solve problems (and if so, how); nevertheless, we draw diagrams all the time. Almost every office, laboratory, and classroom offers a whiteboard where people draw informal diagrams to illustrate their ideas, and the napkin sketch and the back of

an envelope are proverbial design tools. Especially for design applications, it makes sense to try to construct interactive systems that can accept, parse, and recognize this common mode of human communication as an input modality.[2]

Despite the prevalence of "graphical" user interfaces, we interact with computer tools for the most part by choosing from menus, pressing buttons, and entering text in forms and files. The first interactive graphics systems, notably Sketchpad,[3] employed light pens that let users point and mark directly on a cathode-ray tube image. But with raster graphics in the mid-1970s came a widespread adoption of the mouse, relegating the stylus to relative obscurity. This persisted until the early 1990s when—with the advent of pen-based computers and personal digital assistants—the computer industry suddenly rediscovered the stylus. Even today, most PDA and pen-based applications focus on input of text, not graphics.

To be useful, the machine needn't be able to make sense of Picasso's sketches: most whiteboard drawings are highly stylized or diagrammatic. They are made up of graphical symbols selected from a fairly small universe, arranged in a fairly small number of spatial relationships and often augmented by text labels. But if most drawings are diagrams, why not make them using a structured draw program interface, selecting graphical primitives from a palette and assembling them into a diagram instead of drawing them? Indeed, many "sketch" programs require the user to do exactly that, and the resulting diagrams are clean and precise. Two arguments spring to mind against this approach. First is direct manipulation: why use a menu when you can just draw what you want? Second is the objective: in design, you might prefer to work with a crude or even ambiguous drawing.

Most systems that recognize hand-drawn graphics boast a "beautify" feature—the machine converts sloppily drawn figures to perfect ones. Yet the purpose of the informal sketch argues for leaving it exactly as drawn. Beautifying just elevates its precision and apparent commitment to a level that the drawer probably did not intend. A crudely drawn sketch implicitly indicates that the scale cannot be trusted and details have been left out. But if it is beautified, a reader might be tempted to assume that it is drawn to scale. Also, especially in design, it might be preferable to leave parts of the drawing unresolved, to be determined later. For these reasons, imprecision and even crudeness are valuable in early design drawings.



Mark D. Gross is an associate professor in the University of Colorado's Department of Planning and Design and also a member of its Institute for Cognitive Science. His research interests include computer support for design processes, coordination of team design work, and human-computer interfaces. He received a BS in architectural design and a PhD in design theory and methods, both from MIT. He is a member of the IEEE, ACM, AAAI, and the Association for Computer Aided Design in Architecture. Contact him at the Sundance Laboratory, Univ. of Colorado, Boulder CO, 80309-0314; mdg@spot.colorado.edu; http://spot.colorado.edu/~mdg/.

James A. Landay is an assistant professor in the Computer Science Division of the EECS Department at the University of California at Berkeley. His research interests include human-computer interaction, user interface design tools, pen-based user interfaces, end-user programming, mobile computing, and other novel uses of computer technology. He earned a BS in electrical engineering and computer science from Berkeley and an MS and PhD in computer science from Carnegie Mellon University. He is a member of the ACM, SigChi, and SigGraph. Contact him at the Computer Science Div., Univ. of California at Berkeley, Berkeley, CA, 94720-1776; landay@cs.berkeley.edu; http://www.cs.berkeley.edu/~landay.

Thomas F. Stahovich is an assistant professor in the Department of Mechanical Engineering at Carnegie Mellon University. His research interests focus on designing and building intelligent software tools to support mechancial engineering design. He received a BS from the University of California at Berkeley and an SM and PhD from MIT, all in mechanical engineering. He is a member of the AAAI and the ASME. Contact him at the Dept. of Mechanical Engineering, Carnegie Mellon Univ., Pittsburgh, PA 15213; stahov@andrew.cmu.edu; http://www.me. cmu.edu/faculty1/stahovich/stahovich.html.

Electronic Cocktail Napkin

To explore these ideas, my students and I have been working on a project we call the Electronic Cocktail Napkin (and its successor, the Back of an Envelope). We aim to develop a freehand drawing environment that can serve as an interface to a variety of applications that support designers, such as case-based advisors, critics, simulations, libraries, and (for architects and engineers) drafting and modeling software.

We have observed that many architects, even those fluent with computer-aided design software, choose to develop their initial ideas using paper and pencil, and only translate their sketches and drawings to electronic media for design development once the conceptual design work has been done. This way of working imposes a time-consuming discontinuity in the design process between conceptual exploration and design development that is largely irreversible: you cannot move back and forth easily between sketch and computational representation. As a consequence, computer-based tools that might be useful to designers are unavailable in the earliest and most formative stages of design. They can only be employed during design development after the designer has made the most critical decisions.

The Electronic Cocktail Napkin supports freehand drawing on a digitizing tablet, a whiteboard, and a PDA, and it allows several users drawing on different devices to work together on a drawing. It simulates various drawing instruments (pencil, pen, marker, brush) and media (tracing paper, sketchbook). At first glance, the program appears merely to combine the features of a simple paint program (users draw freehand) and a drawing program (they select, resize, move, and delete elements). But beyond emulating physical media and conventional drawing program software, the Electronic Cocktail Napkin also provides trainable symbol recognition, parses more complex configurations of symbols and spatial relations, and can match similar figures.

The Napkin supports recognition of simple glyphs, or symbols. It delimits individual symbols by the length of time the pen is lifted; a symbol need not be restricted to a single stroke. It recognizes a symbol by comparing its features—pen path, number of strokes and corners, and aspect ratio—with a library of stored templates. Each user works with a personal library of glyphs, which allows idiosyncratic drawing styles. The Napkin supports on-the-fly training: a user adds to the library of templates just by drawing a few examples and identifying the symbol.

The Napkin identifies spatial relationships among the elements of a diagram—for example, left-of, above, contains, or connects. It recognizes user-defined con-

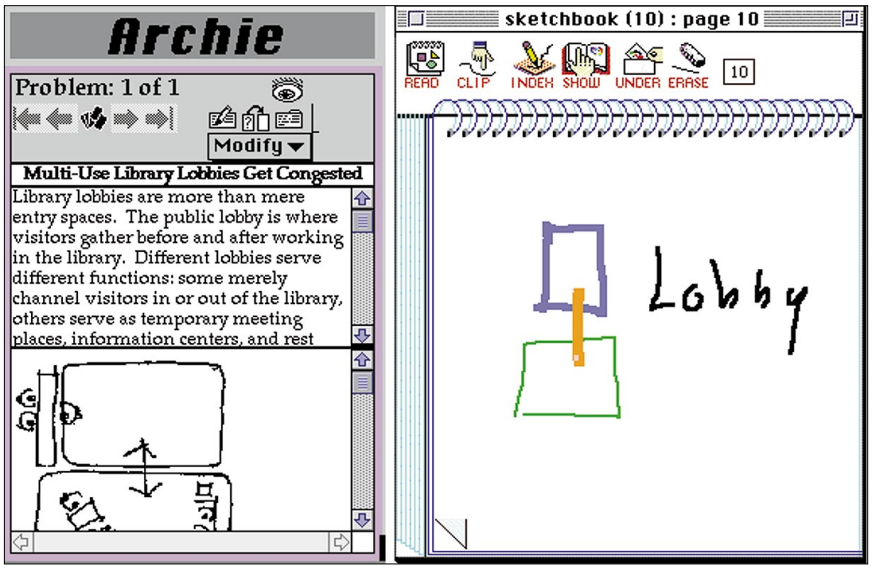


Figure 1. A diagram is used to index Archie, a case base of building design information.

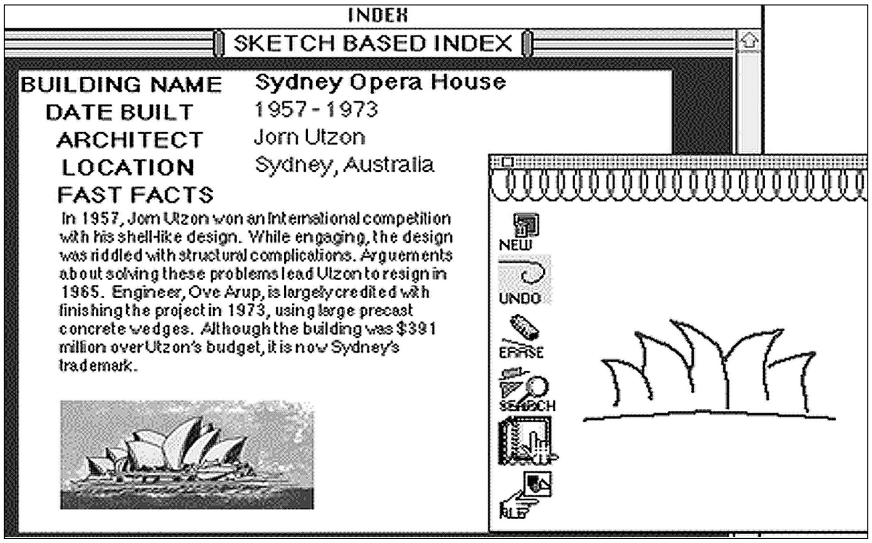


Figure 2. The sketch of the Sydney Opera House serves as a bookmark into a visual library of famous buildings.

figurations of elements arranged in certain spatial relationships; for example, a tree diagram defined as "circles or rectangles, one vertically above the other, connected by line segments." A user defines a configuration recognizer by drawing examples; the Napkin offers a description of the configuration, and the user refines the description. As with simple glyphs, training a new configuration takes place on the fly, not as a separate programming step.

With these two recognition capabilities (symbols and configurations) the Electronic Cocktail Napkin lets users graphically and interactively define a visual-language grammar, which the program can then use to parse freehand drawing input. In addition to parsing compound configurations, the Napkin can also diagnose the similarity of two diagrams, comparing the numbers and types of elements and the spatial relations in each.

Applications

Our program has served as a platform to build prototype interfaces to a variety of applications. We have built a visual indexing and retrieval scheme, a freehand drawing interface to simulation programs, an HTML layout application, and a simple constraint-based diagram editor.[4,5]

Visual bookmarks. We used the Napkin's similarity matching to develop a visual bookmarking scheme. A designer, browsing a digital library of images or design information, makes a diagram to index an item of interest. Later, drawing a similar diagram retrieves the database entry or image. We built a diagram index for a case library of architectural design problems and solutions (see Figure 1), for a published CD-ROM library of famous buildings (see Figure 2), and for URLs in the World Wide Web. In each case, the designer draws a diagram on the Napkin's sketchbook page, which also stores linking information for various external databases and applications. When the designer draws a diagram to index a Web page, Napkin asks the browser for the current URL and stores that address with the diagram in the sketchbook. To retrieve the page, after finding a matching diagram in the sketchbook the program sends the browser a "go to URL" message.

Interface to simulations. We used the Napkin as an interface to drive two interactive simulation tools. In the first example, using a local area network design tool called ProNet, the designer draws a freehand diagram of a network to be simulated; the Napkin parses the diagram and sends instructions to ProNet to construct and run the simulation. The second simulation tool, IsoVist, helps architects compute and visualize the *isovist*, or territory visible from a given vantage (see Figure 3). The designer draws the floor plan on the Napkin, which parses and exports it to the IsoVist calculator, which performs the IsoVist calculation and displays the result.

HTML editing. Our WebStyler prototype explored using freehand diagrams to lay out a Web page. The designer draws the page layout using graphic symbols to position titles, text, and pictures. After the designer attaches specific text and pictures to the page, WebStyler generates appropriate HTML code to position the items correctly in the layout (see Figure 4).

Discussion

The same diagram might have rather different meanings in different domains. In surveying whiteboards at my university, we found that across a wide range of departments the diagrams contained mostly the same symbols and spatial relations, although they quite clearly were talking about different subjects. Context plays a large role in forming a correct interpretation of a diagram, as it does in natural-language understanding. For example, a curlicue line means "inductance" in the context of analog circuits, but in a mechanics diagram it means "spring," and in a child's drawing of a person, "hair." It's

sometimes possible for the Napkin to recognize context by a unique symbol or configuration that appears in the diagram (for example, a treble clef indicates that the diagram is about music). Once the context is recognized, interpretation of symbols might change (curlicues become inductances or springs), as well as the set of recognizable configurations.

We're currently redesigning the Electronic Cocktail Napkin interface so that designers can use it as a tracing paper overlay on top of other applications. In this interaction mode, they would see through the Napkin's window and interact with the application by drawing freehand marks and diagrams. For example, an architect could interact with a modeling program, sketching on top of the 3D forms to edit them. Or the architect could edit a text document the same way you work on paper, using a pen to insert, delete, and move sections of text. This will require specifying a standard way of translating marks made on the Napkin to commands for the back-end application. It will also require enabling the back-end application to take control of the user's diagram.

In the Right Tool at the Right Time project, we're looking at whether the interface might be able to identify what the designer is doing just by looking at the drawing, and then provide an appropriate tool for the task at hand. For example, observing that an architect is drawing light rays and computer monitors, the Right Tool manager might proffer a lighting simulation program or advice in a case base about how to avoid glare in work areas.

Drawing is most often used in conjunction with other communication modalities. A drawing doesn't stand alone, but is embedded in a social context. The whiteboard drawing, the back-of-the-envelope diagram, and the napkin sketch are part of a conversation. We don't expect that interacting with computers will be any different in this regard than with people: the real payoff will come by integrating freehand drawing interfaces with other interaction modes, including conventional structured interfaces, speech, and text. As a first step in this direction, we capture and store the spoken conversation made during drawing, and tag the audio track with the elements of the drawing. The designers can touch an element in the drawing and play back the conversation that was happening when the element was drawn or referenced.

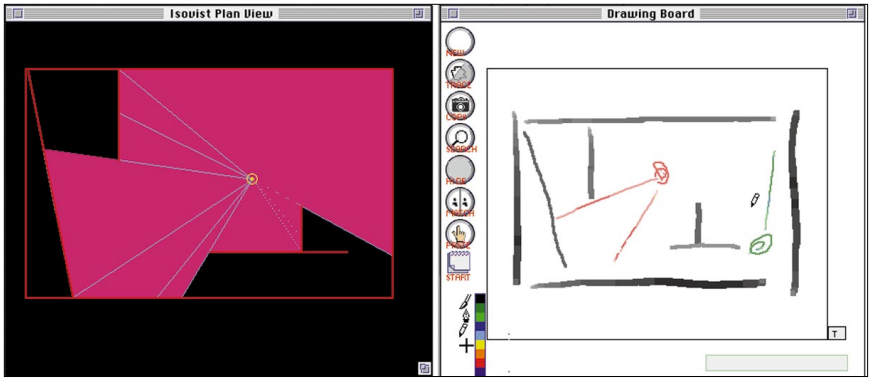


Figure 3. The Napkin sketch at right drives the IsoVist simulator at left, which computes and displays the viewshed in a floor plan.

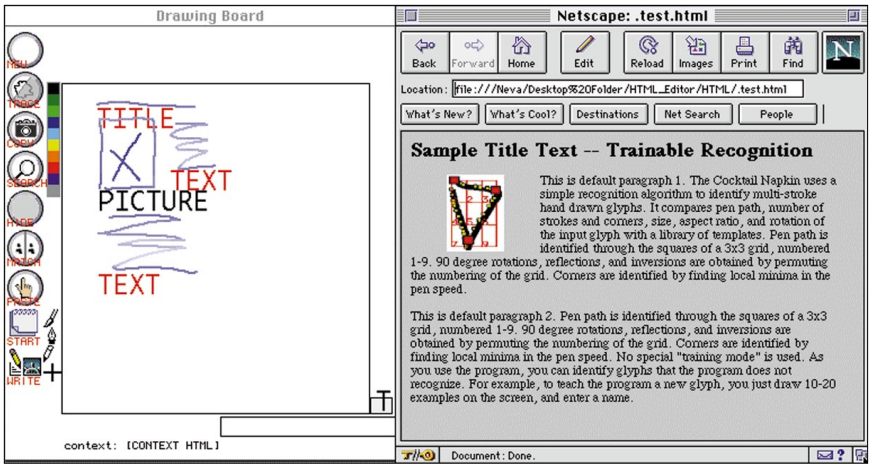


Figure 4. WebStyler. The Web page at right was laid out using the Napkin sketch at left.

There's a chicken-and-egg problem here. It's not clear that an informal interface to a formal application is a good idea: if an application requires precision, freehand diagrams might be the wrong approach. Today's structured interfaces to design applications affect the role we expect the software to play, and thus the design of the software itself. For example, most energy-estimating tools for architects require precise input and deliver precise output. They are therefore less useful in the early stages of design, although that is when the design is most malleable. Perhaps the advent of informal interfaces will foster a new generation of sketchy applications, software that is more qualitative and less demanding of precision, and thereby more useful during early, conceptual design.

Acknowledgments

Support came from NSF grant IRI 96-19856 (the Back of an Envelope), and from a grant (with Wayne Citrin and Steve Laufmann) by the Colorado Advanced Software Institute and USWest Advanced Technologies. Ellen Yi-Luen Do, Adrienne Warmack, Kyle Kuczun, Jen Lewin, Paul Hamill, and Kristin Anderson made valuable contributions to various parts of the project. Thanks also to the L³D group at the University of Colorado for the use of ProNet and to Janet Kolodner, Eric Domeshek, and Craig Zimring at Georgia Tech for the use of Archie.

References

1. G. Polya, *How to Solve It*, Princeton Univ. Press, Princeton, N.J., 1945, p. 97.
2. D. Ullman, S. Wood, and D. Craig, "The Importance of Drawing in the Mechanical Design Process," *Computer Graphics*, Vol. 14, No. 2, 1990, pp. 263–274.
3. I.E. Sutherland, "Sketchpad: A Man-Machine Graphical Communication System," *Proc. Spring Joint Computer Conf.*, Amer. Fed. Information Processing Socs., Montvale, N.J., 1963, pp. 329–346.
4. M.D. Gross and E.Y.-L. Do, "Ambiguous Intentions," *Proc. ACM Symp. User Interface Software and Technology (UIST '96)*, ACM Press, New York, 1996, pp. 183–192.
5. K. Kuczun and M.D. Gross, "Local Area Network Tools and Tasks," *ACM Conf. Designing Interactive Systems 1997 (DIS '97)*, ACM Press, 1990, pp. 215–221.

Informal user interfaces for natural human-computer interaction

James A. Landay, Department of Electrical Engineering and Computer Science, University of California at Berkeley

Computers have found their greatest success in applications where they have replaced humans in tasks at which humans are poor, such as performing complex, redundant mathematical calculations, or storing and searching large amounts of data. As computers grow more powerful, less expensive, and more ubiquitous, we begin to expect them to assist us in tasks that humans do well, such as writing, drawing, and designing. Many of these tasks involve creativity and communication, two activities that lie close to the heart of what makes us human. How to design computer systems to help humans do better at what they already do well is a problem that is far from solved.

Unfortunately, the historical strengths of computers have led to a design bias towards support of precise computation, and away from the more human properties of ambiguity, creativity, and communication. Consequently, user interfaces today are designed to facilitate structured data input rather than natural human communication, which consists of the imprecise modes of speaking, writing, gesturing, and sketching. We use the term *informal user interfaces* to describe user interfaces designed to support natural, ambiguous forms of human-computer interaction.

Sketching and gesturing are two modes of informal interaction that are especially valuable for creative design tasks.[1–3] For designers, the ability to rapidly sketch ambiguous objects—those with uncertain types, sizes, shapes, and positions—is very important to the creative process. Ambiguity encourages the designer to explore more ideas in the early design stages without being burdened by concern for inappropriate details such as colors, fonts, and precise alignment. At this phase, ambiguity also improves communication, both with collaborators and the target audience of the designed artifact. For example, an audience examining a sketched user-interface design will be inclined to focus on important issues, such as the overall structure and flow of the interaction, while not being distracted by the details of the look, such as colors, fonts, and alignment.[4–7] When the designer is ready to move past this early stage and focus on the details, the interface can be recreated in a more formal and precise way.

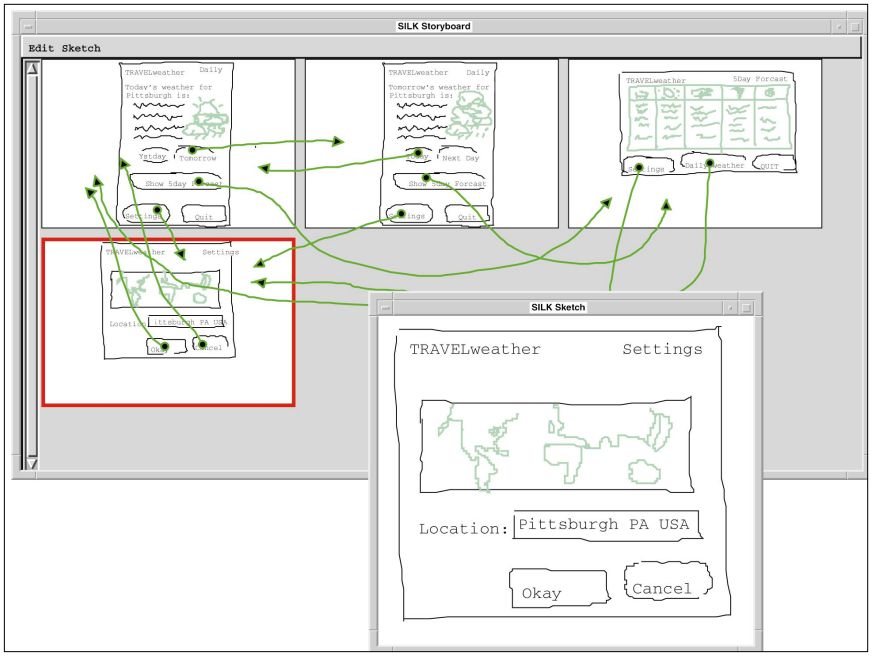


Figure 5. A Silk sketch and storyboard for a weather application. A designer created these sketches during the Silk usability testing.

The primary tenet of our work in informal user interfaces is to bend computers to people's mode of interaction, not the other way around. We believe that any application that requires interaction between humans and computers, and in which absolute precision is not required, could benefit from an informal approach. For some domains, the formal approach imposed by conventional user interfaces clearly presents an obstacle to effective task performance. One such domain that we have investigated is user-interface design itself. In the remainder of this essay, I describe a promising approach to solving this problem and its embodiment in Silk, a tool we have developed for user-interface design. We also describe directions for future work in informal user interfaces.

Silk

Professional user-interface designers are aware of the advantages of ambiguity; consequently, when they first start thinking about a visual interface to an application or a Web site, they often sketch rough pictures of the screen layouts. These sketches help them explore the overall layout and structure of interface components, rather than to refine the detailed look-and-feel. In addition to sketching the various interface screens, designers often build up storyboards from these sketches. By numbering the screens, drawing arrows on them, and attaching annotations, a designer can describe the major transitions that occur between screens when a user manipulates the interface. The storyboard window at the top of Figure 5 illustrates a simple sketched storyboard.

Designers need interactive tools that give them the freedom to sketch rough design ideas quickly, the ability to test the designs by interacting with them, and the flexibility to fill in the design details as choices are made.[8] In contrast, commercial interface and Web page construction tools inhibit creativity by focusing on the look of the *final* designs, not on allowing the rapid exploration of design ideas. Silk combines the advantages of paper-based sketching and storyboarding with those of computer-based construction tools. In describing the major components of Silk, I will use an extended example to illustrate how a designer would use Silk to sketch and edit designs, specify and test interactive behaviors using storyboards, and, when satisfied with the design, automatically transform an interface to a more finished design.

Sketching interfaces. Silk lets the designer move quickly through several iterations of a design by using gestures to edit and redraw portions of the sketch. Silk can be used with a mouse or graphics tablet, but is designed for use with an integrated display tablet.

Widget recognition. Silk recognizes two types of pen strokes, *primitive components*

and *editing gestures*. The primitive components are single-stroke shapes that, when combined, make up a widget. For example, sketching a rectangle and then a squiggly line inside of it created the buttons illustrated in both the Sketch and Storyboard windows of Figure 5. The order in which they were sketched does not matter. Later, the designer replaced the squiggly line with a textual label. The other primitive components recognized by Silk are circles and lines.

Allowing designers to sketch on the computer, rather than on paper, has many advantages, including ease of editing, duplication, communication, and reuse. Several of these advantages cannot be realized without software support for recognizing the interface widgets in the sketch. Having a system that recognizes the drawn widgets gives the designer a tool that can be used for designing, testing, and eventually producing a final application interface. Silk's recognition engine identifies individual user-interface components as they are drawn, rather than after an entire sketch has been completed. This way, the designer can test the interface at any point without waiting for the entire sketch to be designed and recognized. This is key for a tool that supports iterative design. We often observed participants trying out new ideas and immediately testing them during our usability testing sessions.

Because Silk may offer multiple interpretations for a widget, which can be chosen from later in the design process, a Silk design can capitalize on the ambiguity in the sketch. For example, radio buttons and check boxes look very similar when drawn. Silk maintains both interpretations for such a sketch. When testing a design, the designer can easily switch between these two interpretations as this design choice becomes concrete. The other basic widgets Silk recognizes include buttons, text fields, menu bars, scroll bars, scrolling windows, and palettes. The system also recognizes vertical and horizontal sequences of some of these widgets as panels (such as radio button panels).

The widget-recognition algorithm is broken into three phases. First, Silk tries to recognize the primitive components in the sketch as they are drawn. The recognition engine uses Rubine's gesture-recognition algorithm[9] to identify the primitive components that make up an interface widget. After recognizing a primitive component, the system looks for spatial relationships between the new component and other components in the sketch. These spatial relationships include containment, nearness, alignment, and sequencing of objects. Finally, the result is passed to a rule system, which tries to combine the new component with related components to form a more complex widget. The rule system contains basic knowledge of the structure and make-up of user interfaces.

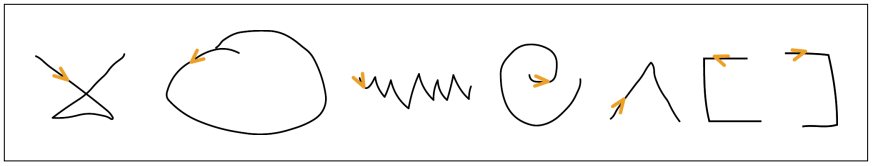


Figure 6. Editing gestures for delete, group, ungroup, change inference, and three gestures for text editing. The arrows indicate a recommended direction to draw these gestures for best recognition performance.

Editing and annotating sketches. One advantage of interactive sketches over paper sketches is the ability to quickly edit them. When the user holds down the button on the side of the stylus, Silk interprets strokes as editing gestures. These gestures go to a different recognizer than the one used for recognizing primitive components. The power of gestures comes from the ability to specify, with a single mark, a set of objects, an operation, and other parameters.[10] For example, deleting a section of the drawing is as simple as making an **X**-shaped stroke with the stylus.

Silk supports gestures for changing inferences, deleting, grouping, and ungrouping primitive components or widgets. There are also text-editing gestures (see Figure 6).

Silk also supports a mode for annotating sketches with drawn, written, or typed comments. The annotation layer can be displayed or hidden, as desired. Practicing designers have found that the annotations of design sketches serve as a diary of the design process, which are often more valuable to the client than the sketches themselves.[11]

Specifying behavior. Easing the specification of the interface layout and structure solves much of the design problem, but a design is not complete until the behavior has also been specified. Silk allows a designer or end-user to test a sketched interface design. For example, as soon as Silk recognizes one of the buttons shown in Figure 5, the designer can switch to Run mode and operate the button by selecting it with the stylus or mouse. The button will highlight when held down. Although only objects recognized by Silk offer built-in feedback, arbitrary sketched objects might still need to have interactive behaviors attached to them.

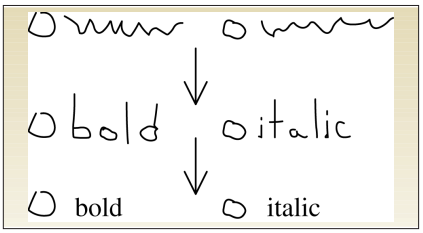


Figure 7. Iterative refinement of text labels: squiggle to handwritten to typed. The designer makes these transitions from the informal to the formal, using the text-editing gestures illustrated in Figure 6.

Unfortunately, the behavior of individual recognized widgets is also insufficient to test a working interface. For example, Silk knows how a button operates, but it cannot know what interface action should occur *after* a user presses the button. These behaviors are specified using storyboards.

Storyboards use the sequencing of screens to allow the specification of the dynamic behavior between widgets and the basic behavior of new widgets or application-specific objects, like a dialog box appearing when a button is pressed. Screen sequencing is expressed by drawing arrows on the storyboard from objects to screens that should appear when the mouse clicks on the object (see Figure 5). Storyboards are a natural representation, are easy to edit, and can easily be used to simulate functionality without worrying about how to implement it.

Iterative refinement. The philosophy behind Silk is that designers are never forced to give more detail than they wish, but they can add more detail as these specifics become known. For example, button labels often start out as simple text squiggles. This is only sufficient for a short time. Eventually, the designer might wish to specify handwritten labels and later, typed labels. Figure 7 illustrates this progression. No OCR is done in this example; the designer first sketches squiggles, then replaces those with written labels using the bracket gestures, and later replaces the written labels with typed labels using the caret gesture. Silk recognizes all three versions as text labels and properly infers that they are part of a radio button panel.

Figure 8. Transformed version of the sketched screen shown in Figure 5.

Transformation. When the designer is satisfied with the interface, Silk can create a new window that contains real widgets and graphical objects that correspond to those in the rough sketch. These objects can take on the look-and-feel of a specified standard graphical-user interface. Silk currently outputs both Visual Basic 5 code and the Motif look-and-feel under Common Lisp in the Garnet User Interface Development Environment.[12]

The transformed interface is only partially finished; the designer still needs to finalize the details of the interface—for example, colors, alignment, and any textual labels that have not yet been specified. At this point, programmers can add callbacks and constraints that include the application-specific code to complete the application. Figure 8 illustrates the transformed version of the sketched screen illustrated in Figure 5.

Design range. Silk is oriented towards the design of the user interfaces found in graphical editors and simple forms-based applications. Therefore, it recognizes and supports the interactive behavior of the standard user-interface widgets (buttons, text fields, scroll bars, and menus). However, Silk is not limited to this style of interface, as Silk's storyboarding component allows screen transitions to occur when the mouse clicks arbitrary graphical objects. A designer can use this mechanism to simulate interfaces that are not dominated by either the structure or behavior of the standard widgets. During our usability testing, several designers used this technique to support new widgets that were not built into Silk. Because Silk is mainly a tool for exploring and communicating interface ideas in the early design stages, it need not support all possible interactive behaviors and events.

Future directions for informal user interfaces

Silk is just one example of an informal user interface that is appropriate for a specific creative task. Other communication and creation tasks might also benefit from this approach. Here at Berkeley, we are working on several tools that have this same flavor. For example, SilkWeb uses an informal approach for Web site design. This tool can generate the working code for a Web site design from a sketch. It can also read in existing designs (in HTML) to allow a user to redesign pages in an informal way. We are exploring how designers mix representations, both formal and informal, in the same design. Like Silk, this tool lets designers quickly prototype some of the more interactive portions of Web pages (forms, for example).

We have also been exploring how informal user interfaces can improve systems for capturing notes. Most handwriting-recognition systems try to recognize writing as the user writes. This might be useful for taking down someone's phone number, but begins to get in the way when the user is trying to focus on a more involved task, such as taking notes in a talk or writing up a new idea. The recognition system forces the user to engage in a dialog that has nothing to do with the immediate goals of his or her task. Deferred recognition is one way of dealing with this problem. We have built a tool for small PDAs such as the PalmPilot that treats electronic ink as a first-class type and lets users write as they naturally do on paper.[13]

We are also exploring the use of informal user interfaces in the development of presentation tools for informal venues such as research group meetings and class lectures (as opposed to formal conference and marketing presentations). Some early work indicates that informal presentations generate more feedback and interaction with the audience. Our tools will allow users to create these presentations, and the rough drawings and animations that go with them, in an amount of time commensurate with the formality of the task. In general, we believe informal interfaces will let users concentrate on performing their tasks.

References

1. V. Goel, *Sketches of Thought*, MIT Press, Cambridge, Mass., 1995.
2. J.A. Landay, *Interactive Sketching for the Early Stages of User Interface Design*, PhD dissertation, CMU-CS-96-201, Carnegie Mellon Univ., Computer Science Dept., Pittsburgh, 1996, pp. 242.
3. M.D. Gross and E.Y.-L. Do, "Ambiguous Intentions: A Paper-Like Interface for Creative Design," *ACM Symp. User Interface Software and Technology*, ACM Press, New York, 1996.
4. A. Black, "Visible Planning on Paper and on Screen: The Impact of Working Medium on Decision-Making by Novice Graphic Designers," *Behaviour & Information Technology*, Vol. 9, 1990, pp. 283–296.
5. Y.Y. Wong, "Rough and Ready Prototypes: Lessons from Graphic Design," *Short Talks Proc. CHI '92: Human Factors in Computing Systems*, 1992, pp. 83–84.
6. M. Rettig, "Prototyping for Tiny Fingers," *Comm. ACM*, Vol. 37, 1994, pp. 21–27.
7. J.A. Landay and B.A. Myers, "Interactive Sketching for the Early Stages of User Interface Design," *Human Factors in Computing Systems Conf.*, ACM Press, New York, 1995, pp. 43–50.
8. A. Wagner, "Prototyping: A Day in the Life of an Interface Designer," in *The Art of Human-Computer Interface Design*, B. Laurel, ed. Addison-Wesley, Reading, Mass., 1990, pp. 79–84.
9. D. Rubine, "Specifying Gestures by Example," *Computer Graphics*, Vol. 25, 1991, pp. 329–337.
10. W. Buxton, "There's More to Interaction Than Meets the Eye: Some Issues in Manual Input," in *User Centered Systems Design: New Perspectives on Human-Computer Interaction*, D.A. Norman and S.W. Draper, eds., Lawrence Erlbaum Associates, Hillsdale, N.J., 1986, pp. 319–337.
11. D. Boyarski and R. Buchanan, "Computers and Communication Design: Exploring the Rhetoric of HCI," *Interactions*, Vol. 1, 1994, pp. 24–35.
12. B.A. Myers et al., "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces," *Computer*, Vol. 23, 1990, pp. 71–85.
13. R.C. Davis et al., *NotePals: Lightweight Note Taking by the Group, for the Group*, Report CSD-98-997, CS Division, UC Berkeley, EECS Dept., Berkeley, Calif., 1998.

The engineering sketch

Thomas F. Stahovich, Department of Mechanical Engineering, Carnegie Mellon University

Drawings are ubiquitous in everyday life because they are so versatile. They can be used for the rather mundane task of communicating information, as in subway maps; they can be an inspiring work of art, such as the drawings of Escher; and they can serve as a powerful reasoning tool for problem solving and design, as in the back-of-the-envelope sketch. Furthermore, drawings are often the best tools we have for certain tasks: try describing a complicated device to someone who hasn't seen it before without drawing a picture. As a mechanical engineer, my interest lies in harnessing the drawing, or the sketch to be more imprecise, as a tool for interacting with computer-based design software.

Drawings have always been an important tool for engineers. The oldest known technical drawing is a ground plan of the ziggurat (a kind of pyramid) at Ur, which was built around 2100 BC.[1] The drawing was made to scale and carved in stone (thus beginning a long tradition of design specifications being set in stone). By the 15th century, Leonardo da Vinci had perfected many of the drawing techniques that are now in common practice, including crosshatching, sectioning, pictorial sketching, and isometric sketching. Today, we have CAD tools that can produce photo-realistic, 3D, geometric models of everything from simple parts to complete aircraft. But surprisingly, engineers still prefer to attack the early stages of a design problem as they have for hundreds of years—with paper and pencil.

To gain insight into the reasons behind this, it helps to contrast this with another problem where computers *are* the preferred tool for the early design stages: paper writing. Many people find they cannot write effectively without a word processor. One of the more obvious reasons is speed: with even mediocre typing skills, you can type much faster that you can write. Also, the editing capabilities of a word processor lets you quickly and easily rearrange the text fragments as the ideas develop so that you can focus on the content rather than the process. There are also psychological factors at work: the word-processed document always looks neat, which can impart a sense that something has been accomplished.

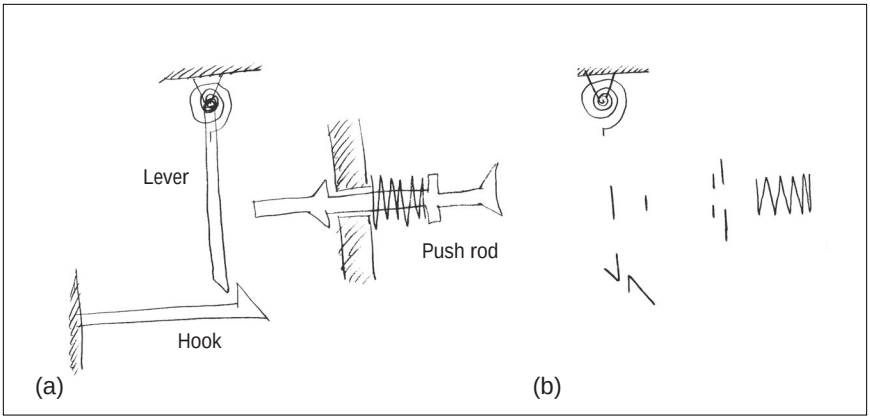


Figure 9. A circuit breaker: (a) a pencil sketch; (b) nonfunctional parts stripped away.

But, when it comes to making a simple drawing, the computer can become a hindrance. What might take 60 seconds to sketch on paper might take 15 minutes to create with traditional drawing software. At its worst, the process can be quite tedious, as you must switch from one drawing tool for lines to another for arcs, and perhaps yet another set of tools to edit the objects once you have created them. Paradoxically, despite the increased precision afforded by a computer, hastily created computer-drawn sketches usually don't look as "nice" as freehand sketches. The lines might be straighter and the arcs rounder, but the elements usually don't have the right proportions or connect smoothly. Putting the pieces together correctly can require painstaking manipulation of numerous control points and parameters.

Thus, at first glance it might appear that all of the difficulties have to do with the clumsiness of the user interface. If this were so, pen-based interfaces would be a nearly complete remedy. However, experience with these systems has proven otherwise. The crux of the problem, I believe, has to do with what is preserved as the design is refined. When writing a paper, some authors first write an outline, then list key phrases under each heading, and finally weave these into cohesive text. After much polishing, this first draft becomes the finished document. Interestingly, words from any step in the process—the outline, the key phrases, the first draft—might end up, in their original form, in the final document.

This is rather unlike the process that occurs as rough sketches are transformed into finished designs. The sketch serves as a tool for investigating the relationships between the parts. Once the relationships are understood, the designer typically creates new geometry that preserves these relationships while satisfying other requirements, such as aesthetic and engineering requirements. The relationships between the elements, much more than the elements themselves, are what are preserved in the transformation.

Consider asking an engineer to turn the sketch of the circuit breaker in Figure 9a into a working design. She would begin by examining the important relationships, which, in this case, are the ways that the parts interact with each other mechanically. The understanding she would extract would be at the level of "When the hook bends out of the way, the lever gets pushed by its spring until it bumps into the push rod. Once the hook relaxes again, pressing the push rod will push the

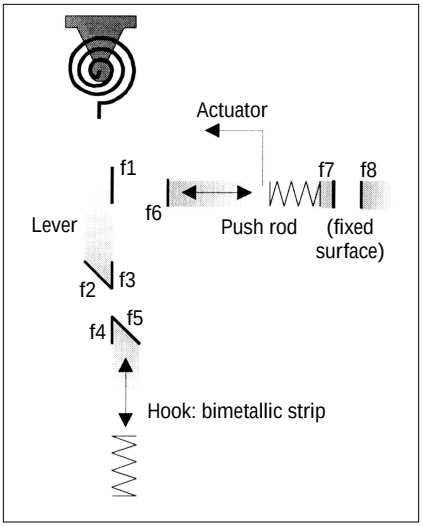


Figure 10. A stylized sketch of the circuit breaker, illustrating the kind of input a designer currently gives to SketchIT. Interacting faces are bold lines. The actuator applied to the pushrod represents the reset motion imparted by the user.

lever into the hook, and the sloped faces will allow the lever to slip by and get trapped by the back of the hook."

With this understanding, the designer would then create new geometry for the hook, lever, and push rod to make these interactions actually happen. Even if the geometry as drawn would not have worked, she can still see how the parts should have interacted, and from this can generate a new geometry that does work. The process goes from geometry (the sketch) to the interactions and then back to geometry. The final geometry might actually look very different from the original sketch, but it will still work the same way the sketch did (or should have)—the interactions will be the same.

Generalizing from this example, we see that the first step in building software that can work from sketches is to develop techniques for recognizing and understanding the important relationships between the elements in the sketch. To be sure, the kinds of relationships that are important depend on the problem domain. For example, in sketches of mechanical devices, the important relationships involve the mechanical interactions between the parts. But in an architectural sketch, the important relationships might have to do with the relative sizes of the spaces and the connections between them.

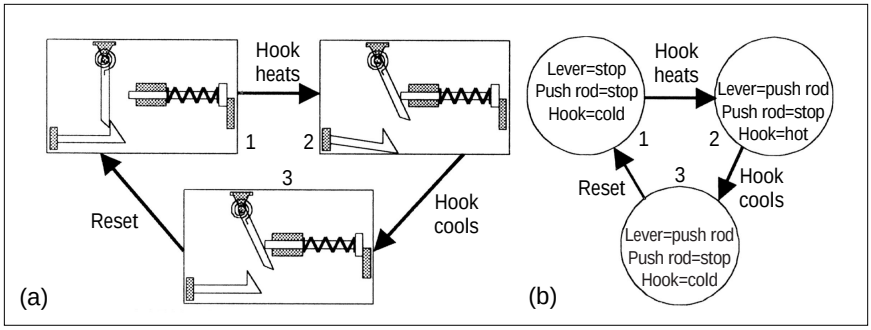


Figure 11. The desired behavior of the circuit breaker: (a) physical interpretation; (b) state-transition diagram. In each of the three states, the hook is either at its hot or cold neutral position.

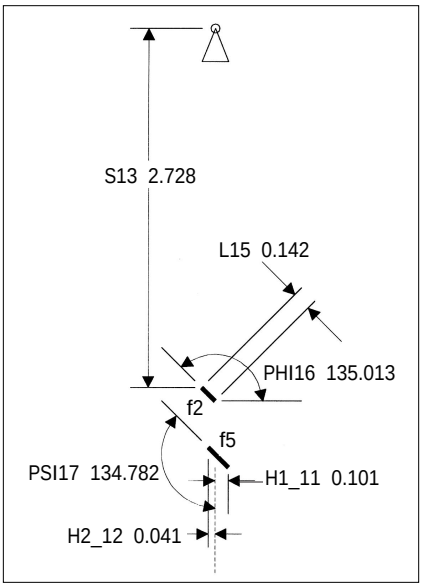
Once the software has recognized the key relationships in the sketch, it faces a second, more difficult task: it must translate these relationships into constraints on the geometric elements in the sketch. Doing this correctly requires a form of generalization. One reason sketches are so useful as a reasoning tool is that they provide a concrete instance of a problem to think about; thinking concretely is almost always easier than thinking abstractly. But a sketch is just one example of the idea the designer had in mind. Therefore, when enumerating the constraints imposed by the desired relationships, it is important to distinguish between those properties of the geometry that were intended and those that were an accident of the particular example used to explore the ideas.

Solving these problems will likely require different reasoning techniques and representations for each different problem domain. In our work at Carnegie Mellon, we have focused on the problem of transforming sketches of mechanical devices into working designs. We have built a system called SketchIT, which I will now use as an example of the challenges in working from sketches.

SketchIT

The SketchIT program can transform a rough sketch of a mechanical device into working geometry and can generalize the original sketch to produce multiple new design alternatives.[2] The program first uses a novel representation called qualitative configuration space (qc-space) to identify the mechanical interactions between the parts of the sketch. It then uses a library of implementations to map from the identified interactions back to geometry that implements them, thus producing new, working designs. Each new design is actually a family of implementations, which SketchIT represents as a behavior-ensuring parametric model ("BEP-Model"): a parametric model augmented with constraints that ensure the geometry produces the desired behavior (a parametric model is a geometric model in which a set of parameters controls the shapes).

Here I use the design of circuit breaker to illustrate SketchIT in action. Figure 9a shows a pencil sketch of a circuit breaker. In normal use, current flows from the lever to the hook and out to the rest of the circuit.




```
H1_11 > 0    H2_12 > 0    S13 > H1_11
L15 > 0      PHI16 > 90   PHI16 < 180
PSI17 > 90   PSI17 < 180
0 > R14/TAN(PSI17) + H2_12/SIN(PSI17)
R14 = SQRT(S13^2 + L15^2 -
           2*S13*L15*COS(PHI16))
```

Figure 12. Output from the program (a BEP-Model). Top: The parametric geometry; the decimal number next to each parameter is the current value of that parameter. Bottom: The constraints on the parameters. For clarity, only the parameters and constraints for faces f2 and f5 are shown.

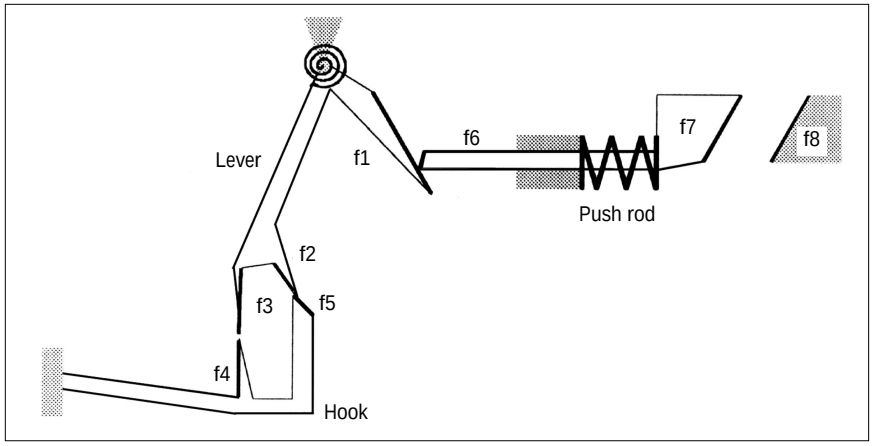


Figure 13. Another solution to the BEP-Model of Figure 12. This solution shows that neither the pair of faces at the end of the lever (f2 and f3) nor the pair of faces at the end of the hook (f4 and f5) need be contiguous. In the position shown, the push rod is pressed so that the hook is on the verge of latching the lever.

Current overload causes the bimetallic hook to heat and bend, releasing the lever and interrupting the current flow. After the hook cools, pressing and releasing the pushrod resets the device.

SketchIT is concerned with only the functional parts of the sketch: springs, actuators, kinematic joints, and the faces where parts meet and through which force and motion are transmitted. Figure 9b shows what is left when we peel away all but the functional parts of the pencil sketch. SketchIT's input is actually a stylized sketch that contains just this information.

Our software currently provides a mouse-driven interface for creating these kinds of stylized sketches. Figure 10 shows the stylized sketch of the circuit breaker that the designer created with this interface. Line segments serve for part faces, and icons serve for springs, joints, and actuators. We are working on using an electronic whiteboard to build a better sketching interface. The designer will use a marker to draw the usual sort of sketch (such as the one in Figure 9a) on a whiteboard and will then annotate the sketch to indicate the functional parts.

Because the device in the original sketch might or might not work, for the program to understand the sketch, it has to know what the device was supposed to have done. The designer describes the desired behavior to SketchIT using a state-transition diagram. Each node in the diagram is a list of the pairs of faces that are engaged and the springs that are relaxed. The arcs are the external inputs that drive the device. Figure 11b, for instance, describes how the circuit breaker should behave in the face of heating and cooling the hook and pressing the reset push rod. Currently, the designer creates the state-transition diagram directly, but our eventual goal is to work from a sequence of sketches as in Figure 11a.

Figure 12 shows a portion of one of the BEP-models that SketchIT derives from the stylized sketch of the circuit breaker and the desired behavior. The top of the figure shows the parameters that define the lever's sloped face (f2) and the hook's sloped face (f5). The bottom shows the constraints that ensure this pair of faces works properly—that is, the constraints ensure that moving the lever clockwise pushes the hook down until the lever moves past the point of the hook, whereupon the hook springs back to its rest position.

These constraints generalize the design beyond the single example (the sketch) the designer used to examine the interactions between the parts of the circuit breaker. Any other geometry that satisfies these constraints is an equivalent design. For example, Figure 13 shows a rather unusual geometry that satisfies these constraints, and thus is guaranteed to work properly.

Figures 12 and 13 show members of just one of the families of designs SketchIT creates as it maps the circuit breaker's qc-space back to geometry. In this mapping, the program can actually generate a wide variety of new design families that implement the interactions depicted in the original sketch.

References

1. R.F. Steidel Jr. and J.M Henderson, *The Graphic Languages of Engineering*, John Wiley & Sons, New York, 1983
2. T.F. Stahovich, R. Davis, and H. Shrobe, "Generating Multiple New Designs from a Sketch," *Proc. AAAI-96*, AAAI Press, Menlo Park, Calif., 1996, pp. 1022–1029.

How to Reach Us

Writers

For detailed information on submitting articles, write for our Editorial Guidelines (m.davis@computer.org), or access http://computer.org/intelligent/edguide.htm.

Letters to the Editor

Send letters to

Managing Editor
IEEE Intelligent Systems
10662 Los Vaqueros Circle
Los Alamitos, CA 90720

Please provide an e-mail address or daytime phone number with your letter.

On the Web

Access http://computer.org/intelligent/ for information about *IEEE Intelligent Systems*.

Subscription Change of Address

Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *Intelligent Systems*.

Membership Change of Address

Send change-of-address requests for the membership directory to directory.updates@computer.org.

Missing or Damaged Copies

If you are missing an issue or you received a damaged copy, contact membership@computer.org.

Reprints of Articles

For price information or to order reprints, send e-mail to m.davis@computer.org or fax (714) 821-4010.

Reprint Permission

To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at whagen@ieee.org.

Intelligent SYSTEMS & their applications