Problem-Centered Design for Expressiveness and Facility in a Graphical Programming System

Clayton Lewis
John Rieman
Brigham Bell

CU-CS-479-90 June 1990

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Problem-Centered Design for Expressiveness and Facility in a Graphical Programming System

Clayton Lewis
John Rieman
Brigham Bell

CU-CS-479-90 June 1990

Institute of Cognitive Science
and
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430

This research was supported by the National Science Foundation, Grant No. IRI-8944178. Related research was supported by US West Advanced Technologies.

CONTENTS

Problem-Centered Design for Expressiveness and Facility in a Graphical Programming System.

Abstract

This paper presents a case study in the use of problems in design. Problems, concrete examples of proposed use, were used to describe the intended function of a graphical programming system and to manage the growth of the space of design alternatives for the system. They were also used to evaluate not only the expressiveness of alternative designs, the quality of solutions to problems supported by the designs, but also their facility, the ease with which hypothetical prospective users could find workable solutions to the problems. The problem-centered design process provides a view of design rationale in which the strengths and weaknesses of design alternatives in dealing with specific problems, rather than abstract connections among design issues, are central.

1. INTRODUCTION

This paper presents a design case study which illustrates new approaches to some important design questions in human-computer interaction. First, how can the rationale for large numbers of complex, interacting design alternatives be managed? Second, how can usability considerations be introduced into the design of programming languages? Third, how can the evaluation of design alternatives in programming languages take into account not only the nature of programs but the nature of the process by which programs must be produced?

Each of these issues is approached by placing *problems*, concrete examples of the intended use of the system being designed, at the center of the design process. Design alternatives are linked to problems which demonstrate concretely what an alternative does well or poorly. Design rationale is expressed in these linkages. The expression may be explicit, in cases where someone composes a statement of the abstract issues involved in applying an alternative to some problem, or it may be implicit, when the description of an alternative and an associated problem are left as they are as a concrete example of an advantage or disadvantage of the alternative.

Problems are also central to shaping the usability of the design as a programming system. Problems are used to weigh the *expressiveness* of a candidate design, that is, to determine whether solutions to problems can be expressed within the design. They are also used to assess what we call the *facility* afforded by a design. Two designs that permit similar solutions to a problem may yet differ greatly in how much mental work is needed to discover the solutions starting from the statement of the problem. A design offers superior facility if it is easy to get from the statement of a problem to a solution within the design.

We develop each of these points and their background in the balance of this introduction. We then describe the case study, showing in more detail the application of the ideas and the results obtained. We close with a discussion of the lessons we draw from the case study.

1.1 Problem-centered design.

Our use of problems is an outgrowth of earlier work suggesting the value of detailed analysis of examples of use in user interface design (Gould, 1988; Carroll and Rosson, 1990; see also Young, Barnard, Simon, and Whittington, 1989). Because of the complexity of the sequences of events that constitute successful use of a user interface, and because of the cost of using live user testing to explore the advantages and disadvantages of differing designs, these authors advocated the preparation of detailed *scenarios* that included each step a user would take in using a design. By preparing a scenario a designer could verify that a given task could actually be accomplished with a design, and by examining the scenario he or she could identify parts of the interaction that were complicated, imposed high memory demands, or were awkward in other ways. The cognitive walkthrough procedure (Lewis, Polson, Rieman, and Wharton 1990) is an example of the use of scenarios to examine the mental processes required in performing a task with a given design.

In our adaptation of these ideas we use the term *problem* to refer to a task description that is independent of any system that might be used to solve it. We reserve the term *scenario* for a description of how a problem might be solved using a particular design. Thus a problem could be used to compare two designs by developing scenarios for the problem with each design and comparing the scenarios. Using this terminology problems, not scenarios, turned out to be the chief organizing elements in our design process. This is because problems can be used to evaluate many different candidate designs, whereas scenarios are tied to particular designs or families of related designs.

1.2 Design rationale.

The papers in this special issue develop the arguments for giving design rationale an important place in the management of design. In our case study two reasons predominated. First, our design project was part of a larger project seeking to strengthen the role of cognitive theory in system design (Doane and Lemke, 1990), and as part of this we needed to keep track of the extent to which cognitive considerations did or did not play a role in the design. Second, and of more general relevance, we needed to manage a large, shifting set of design alternatives. We needed to recall why Idea A had been chosen over Idea B at some stage in design, because Idea B had resurfaced and looked attractive in connection with some new issue.

Our efforts to keep track of our design decisions and the reasons for them led us to give problems a key role for two reasons. First, we found that problems nearly always provided the basis for rejecting design alternatives. We seldom abandoned an idea on abstract grounds; rather, we composed a concrete problem that showed why the idea wouldn't work. Recording the reasons for such decisions was easy if we just described the problems, and very hard if we tried to present abstract characterizations of the design issues instead. Second, we found problems easy to think about, remember, and discuss. Design debates were punctuated with references to problems rather than abstract design issues.

1.3 Programming languages, expressiveness, and facility.

As Newell and Card (1985) pointed out, the field of programming language design has been untouched by any explicit consideration of usability. While the work of Brooks (1977), Soloway and colleagues (Soloway and Ehrlich 1984; Spohrer, Soloway, and Pope 1985), Green and colleagues (Sime, Green and Guest, 1977; Green, Bellamy and Parker, 1987), and others has produced insights into mental processes in programming, and how language features help or hinder them, language designers are apparently unaware of this work. Language designs are presented and evaluated mainly on the basis of the characteristics of programs rather than the characteristics of the mental processes needed to produce programs. In an excellent text MacLennan (1987) provides a clear summary of guiding principles of language design; it contains no reference to mental processes.

The clear reason for this state of affairs is that language designers have lacked the means to evaluate the mental processes involved in language use, or even a clear conceptual framework in which to pose the problem. Mackinlay's work on graphical representations (Mackinlay 1986) provides a useful start. He distinguishes *expressiveness* and *effectiveness* as evaluation criteria for representation schemes. A representation scheme meets the *expressiveness* criterion for a display task (for example, showing which states of the United States do and do not share borders) if it can be used to convey accurately the information required by the task. For example, using nested contours to represent states, under the interpretation that two states share a border if the contour for one is inside the contour for the other, fails this criterion, because "nested within" is a transitive relation and "shares a border with" is not. A representation scheme is *effective* if the resulting display can be processed easily by a human viewer. Thus a display which uses the areas of objects of different shapes to convey magnitudes which must be compared will not be effective because people are poor at comparing the areas of different shapes.

The expressiveness criterion can usefully be brought over into the programming domain. A language satisfies the expressiveness criterion for a problem if there is a program in the language that solves the problem. In practice it is useful to blur the expressiveness idea to permit us to reject programs that solve a problem but are unreasonably long or complex, since most languages include computationally universal features, or approximations to them. This gives us a graded notion of expressiveness: one language is more expressive than another for a problem if it permits a shorter or simpler program that solves the problem.

Mackinlay's effectiveness criterion is also relevant to programming languages, if one considers the important matter of the ability of people to understand what a program does. Our focus, however, has not been on the comprehensibility of programs but rather on the difficulty of writing them. We propose the term *facility* to describe the degree to which programs in a language are easy to write, not in the sense of physical effort but in the sense of mental effort. Given a statement of a problem, it is easy to write a program to solve the problem in a language with high facility for that problem. Using a language with low facility one would find that considerable problem solving, trying and rejecting alternative approaches, would be required.

In assessing facility we have adapted the cognitive walkthrough method mentioned earlier (Lewis et al 1990). In our approach one outlines the mental steps required to develop a program from a problem statement, and looks for steps that are unlikely to occur, given the user's knowledge. In making this assessment the designer can choose any sequence of mental steps he or she chooses. If all of the steps are straightforward this sequence is probable, and the language gets high facility marks. But if any steps do not seem likely the rating suffers.

In applying this idea it immediately became clear that no programming language is likely to show high facility without assuming some background knowledge on the part of users. This can be illustrated by Soloway's work on learning Pascal. Knowledge of the statements of Pascal is not adequate to write programs. Rather, one must know about *plans* that show how to combine statements into groupings that perform a function that can be linked to the problem. For example, the "accumulator variable" plan shows how to use iteration to calculate the sum of a collection of values. The plans can be seen as bridges that connect what the user wants to do (calculate the average of some rainfall measurements) to the statements of Pascal (assignment, iteration), often via knowledge of some mathematical abstractions (sum). A Pascal programmer who knows about sums, iteration, and assignments, but not about accumulator variables, has a long, frustrating problem solving episode ahead.

We call the background knowledge that must be available to use a language *doctrine*. Evaluating a language for facility really means evaluating the language and its doctrine. A language can show poor facility in two ways. First, it can require a lot of doctrine, because its features are hard to relate to the kinds of problems it is supposed to be used to solve. Second, on a particular problem, even with its doctrine, it may be impossible to develop a solution without extensive problem solving.

2. THE CASE STUDY

2.1 The system and its goals.

The ChemTrains system is a tool intended to permit users without programming background to construct animated graphical models of systems for which they have a qualitative behavior model, such as document flow in an organization or a Turing machine. The original design direction was shaped by three influences. First, work by one of us, CL, on solar flare forecasting had revealed a need for a graphical modeling system that could be used to produce animated models of solar activity. It appeared that a tool able to do this job would also be useful in many other domains. Second, the NoPumpG spreadsheet-based graphics system (Lewis in press, Wilde and Lewis 1990) had provided a satisfyingly simple way to control graphics, but only by constructing algebraic formulae to describe motions. Thus users had to erect a system of physical and mathematical abstractions to link their problem to its solution. This suggested the desirability of a graphics system controllable in qualitative, not quantitative terms. Finally, the work on qualitative physics by Forbus (1984) and others suggested the power of qualitative representations.

The initial design sketch, prepared by CL and Victor Schoenberg, provided an outline still recognizable despite the intervening design work described here. ChemTrains models were to show objects moving among places on the screen along visible or invisible paths which were specified graphically, not quantitatively. Objects participated in reactions, like chemical reactions, which could modify them or delete them. Like chemical reactions, ChemTrains reactions were thought of as occurring only when the objects involved appeared in the same place on the screen. The name ChemTrains was suggested by the role of these reactions and the role of paths, thought of as railroad tracks.

This early design was developed using two problems as illustrations of the intended functionality. The Office problem, of which a later version is shown in Table 1, showed how documents might flow in an organization, including copying and the use of distribution lists. The Bunsen Burner problem, also in Table 1, was a trivial qualitative physics problem showing how the control of a gas burner could affect the phase of water in a beaker.

2.2 Developing the design space.

This initial design direction obviously left many points to be investigated, including how movement along paths was to be controlled, how reactions would be specified, the nature of objects, and more. Informally, the design space began to grow in the following manner.

The Office and Bunsen Burner problems were the first examples of what we came to call *raw problems*. These are complete descriptions of things someone might want to do with the system and which the system is intended to support. Taken together, the raw problems represent the functional objectives of the system as it has evolved. Brief descriptions of the six raw problems are shown in Table 1. (The complete problem statements, together with complete documentation of other aspects of the case study, are in Rieman, Bell and Lewis , 1990.)

In attacking the raw problems with some version of a design, some aspects of the problem were easily dealt with while others caused trouble. We developed *micro problems* by pulling out the key parts of raw problems that illustrated trouble in the design. For example, matching a memo "to office" with a distribution list called "office" is a micro problem derived from the Office problem.

Table 1. Brief descriptions of the six raw problems.

(The descriptions used in the walkthroughs were more detailed.)

Bunsen Burner. Show how the flame of a bunsen burner responds when the user moves a control knob to the off, low, or high position. Show water in a beaker above the flame changing from ice to water to steam as the flame changes.

Office. Show three offices and a copy room. A memo addressed to any one of office holders travels to that person's office and is destroyed. A memo addressed "to office" travels to a copy room, where copies are made that travel to each of the office holders.

Tic-Tac-Toe. The simulation should play a non-losing game of tic-tac-toe against the user.

Maze. Simulate an intelligent mouse searching a maze for cheese. A technique is described by which a real mouse can keep track of its progress through the maze, using crumbs and string. The maze is specified, but the simulation should work if the maze is changed.

Petri Net. Show the operation of Petri Net transitions, which fire and generate a new token on their output place when tokens are found on both of their input places. A specific network of transitions is specified, but a general solution is required.

Turing Machine. Show the operation of a Turing machine, which includes a moving read/write head, a tape, an internal state variable, and a set of rules describing the head's actions. A specific task, changing a string of bits on the tape to even parity, is specified, but a general solution is required.

Consideration of micro problems led to the proposal of design alternatives. Where one design had trouble another design could be suggested that could deal with the trouble.

Comparison of design alternatives led to formulation of new micro problems. Design alternative B might outdo design alternative A on micro problem 1, but what about this new situation, micro problem 2? Micro problem 2 shows what's wrong with B and good about A.

Often these new, emergent micro problems could be embedded in an existing raw problem, either by focusing attention on some hitherto neglected aspect of one of them or by extending the scope of one. But other times the new micro problems seemed to bring out something the system should be able to do, but that no existing raw problem required. This led to the addition of new raw problems.

The Tic Tac Toe and Maze raw problems were added in this way, in response to the claims by one of us (BB) that the initial design was too limited. BB could not show the value of his proposed more powerful features on the trivial Office and Bunsen Burner problems, but was able to persuade the group that the system should be able to cope with the more complex Tic Tac Toe and Maze examples. Two other raw problems, Turing Machine and Petri Net, were added simply as a way to check that the evolving design could cover a range of applications, not to capture any particular kind of micro problem.

2.3 Examples of the evolution of the design space.

The space of design alternatives grew quite rapidly. Figure 1 gives a sketch of the alternatives, along with the raw problems and micro problems that gave rise to them. The growth processes just described can be illustrated using the alternatives shown.

Example 1: Micro problem derived from raw problem, design alternative spawned by micro problem.

The Turing Machine raw problem requires coordinating the movement of the read head of the machine with changes of state. In an early ChemTrains design movement was controlled by predicates, called *filters*, attached to the ends of paths. To make the read head move therefore required changing the object representing the head so that it would be accepted by the appropriate filter. There were two difficulties with this. First, it then became necessary to change the head object again when it got to its destination, so that it would not keep moving. Second, it was necessary to coordinate the action of filters with the reaction rules that changed the machine state and symbols on the tape. Together these difficulties meant that one transition rule for the Turing Machine had to be represented by several ChemTrains reaction rules. This constituted the *moving head* micro problem.

Contemplating the moving head problem led to the idea that movement control could be combined with reaction rules. Indeed, the micro problem made it obvious that some such combination would be necessary if Turing Machine transitions were to be economically represented. This line of thought led to the formulation of a new design alternative, the *anteroom* approach. In this approach rules could specify that objects should be placed in anterooms at the ends of paths, from which they would be moved along the associated paths.

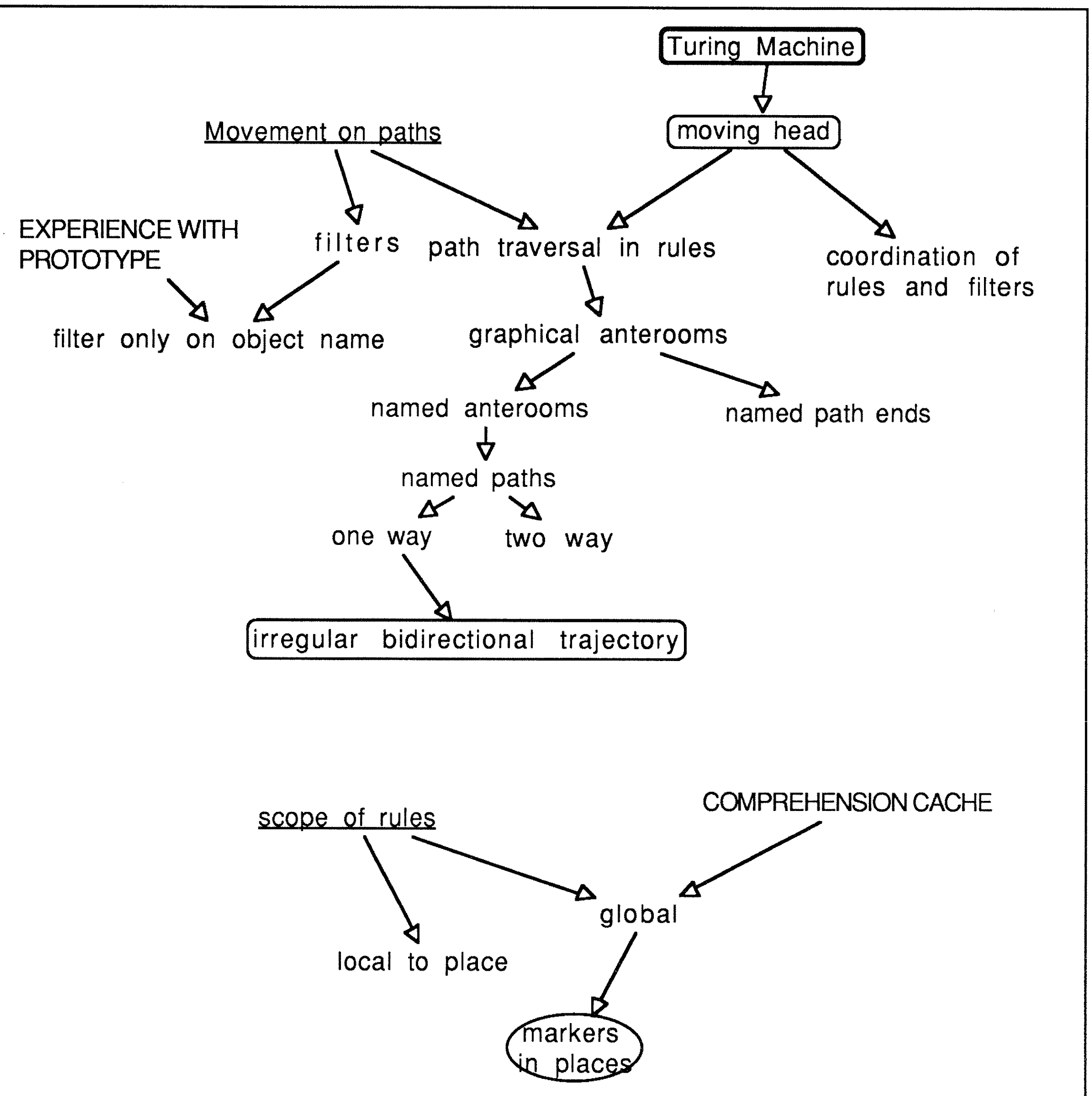
Example 2: Micro problem spawned to evaluate design alternatives.

Two kinds of internal structure were considered for objects. In the simpler view objects were simply bundles of values, while in the alternative they were bundles of attribute-value pairs. To show the weakness of the bundle of values approach, an example was proposed in which objects were to carry three numerical attributes, pressure, volume, and temperature. Without attribute names associated with the numerical values it would be impossible to determine which number represented which attribute.

Example 3: Design alternatives not rooted in problem, micro problem spawned by comparison of design alternatives.

Though raw problems and micro problems played a powerful part in producing design alternatives, not all of the alternatives shown in Figure 1 arose that way. The *before-after* vs. *operations* alternatives were not spawned in connection with any specific problem but rather arose from consideration of how rules might most naturally be written. In the *before-after* approach a rule would be specified by describing the relevant state of the model before the rule applied, and then showing the state that would be produced by the rule. In this approach one would not say explicitly that an object was to be deleted, for example, but rather would simply include it in the "before" description and omit it from the "after" description. In the *operations* approach the "before" conditions would be described as for before-after, but the action of the rule would be described as a collection of operations on objects, like "delete memo." This approach permits the user to say explicitly what is supposed to happen.

Though consideration of these contrasting approaches was sparked by concern for natural expression rather than by a specific problem, the



Key:

Items in bold boxes are raw problems. Items in plain boxes are micro problems. Plain text items are design alternatives. Items in ovals are solution methods using existing design alternatives. Underlined items are issues not derived from any specific problem. Uppercase items are background considerations brought into design deliberation. Some items appear in more than one panel of the figure. Connections show chronological development: the item at the tail of a connection led to consideration of the item at the head of the connection.

Figure 1. The ChemTrains design space.

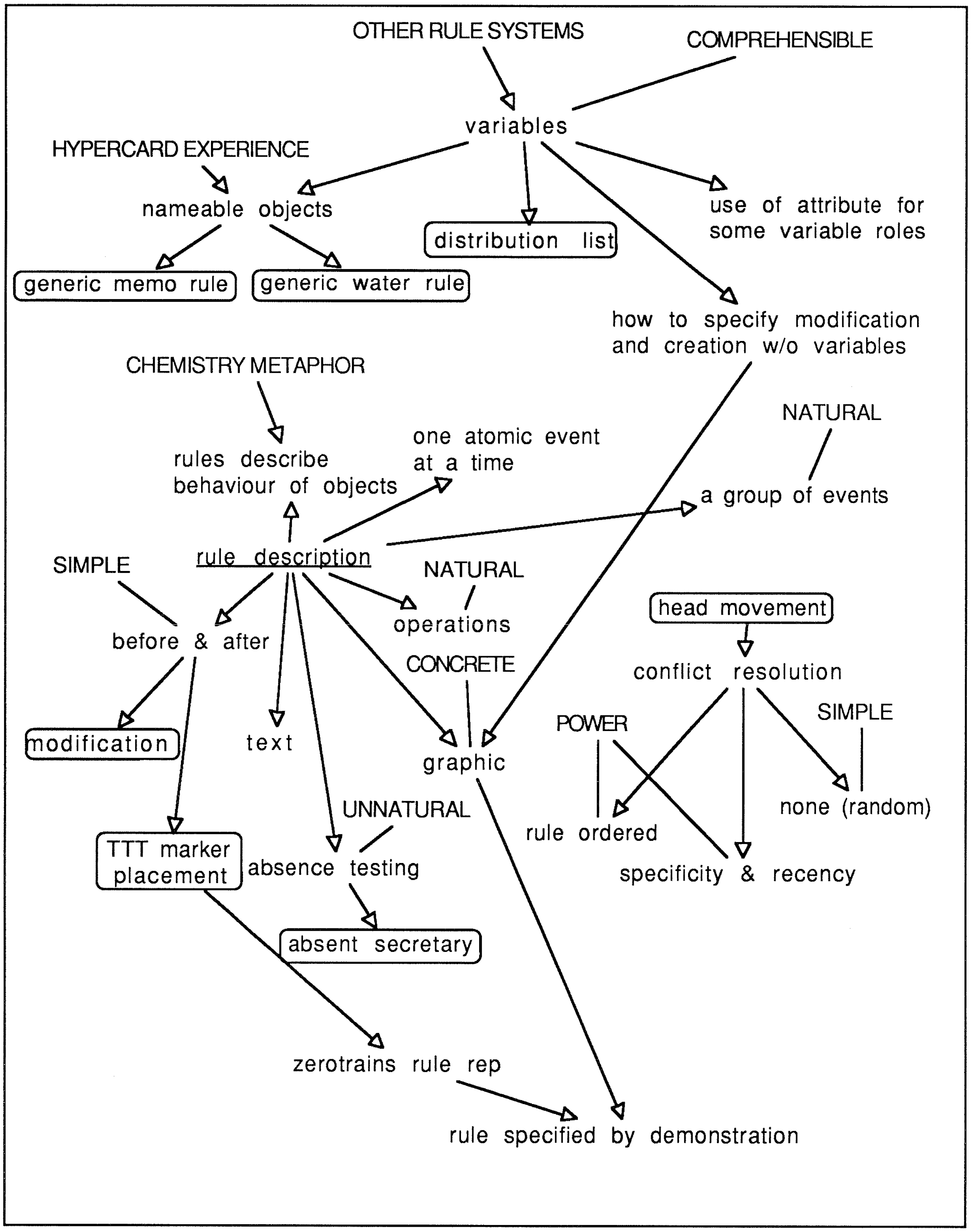


Figure 1. (continued)

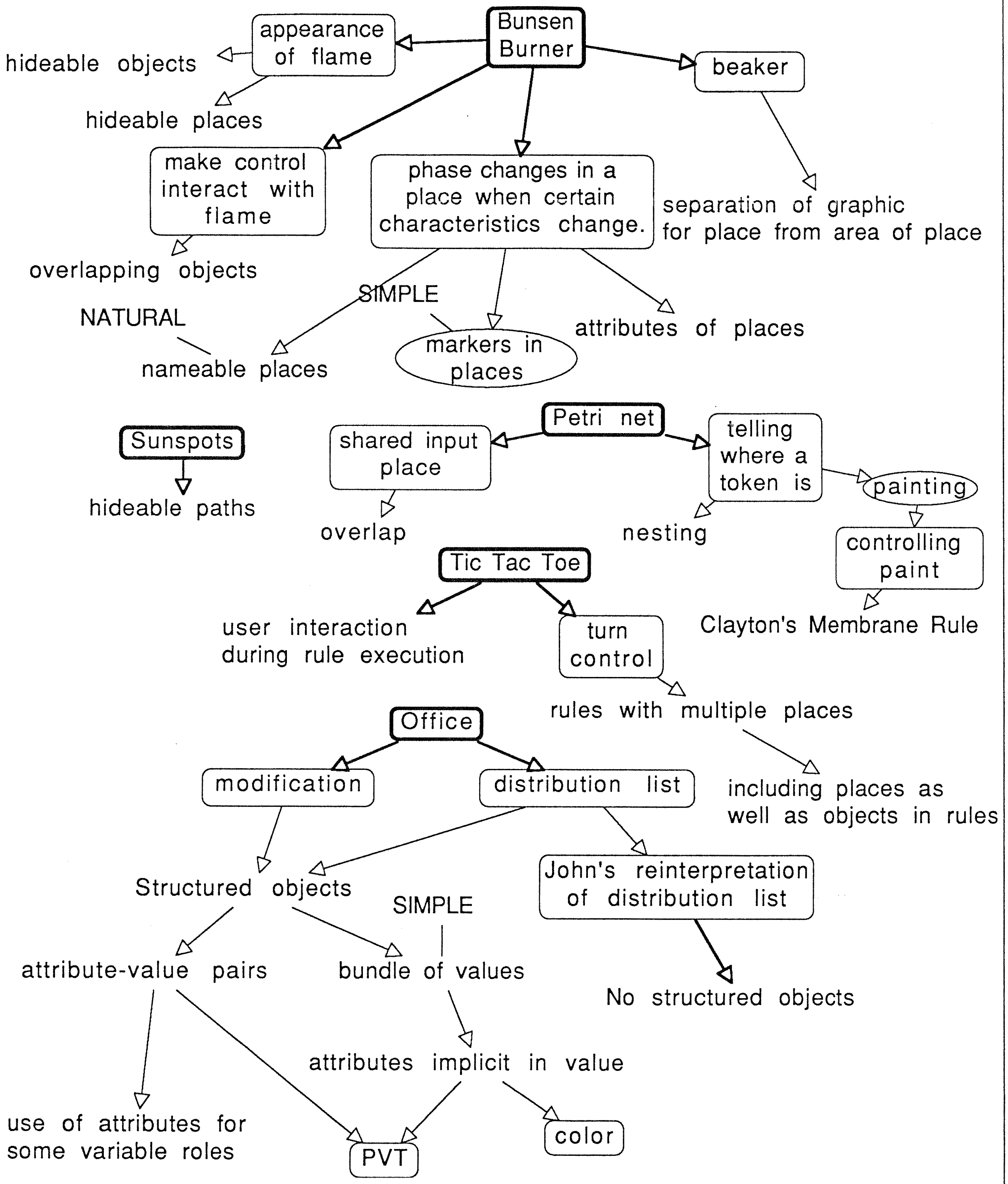


Figure 1 (continued further)

comparison spawned the formulation of a new micro problem, the *modification* problem, which was framed as a part of the Office raw problem. In the modification problem a memo is to have its destination changed for rerouting but carry the same content. In the *operations* approach to rules, this is easy to arrange. By specifying a modification operation to be applied to the memo one can easily indicate that only the destination is to be changed, while other aspects, including the content, should be left alone. But in the *before-after* approach there is trouble. The crux of the problem is the need to indicate that the memo appearing in the "after" description is really the same one that appears in the "before" condition, that is, to indicate that the original memo is being changed rather than that the original memo is to be deleted and a new, unrelated memo is to be created. The simplest before-after approach cannot make this distinction; some machinery must be added, such as explicit links between objects in "before" and "after", or the inclusion of variables in the "before" descriptions whose bindings can be used in the "after" description.

Example 4: Reframing a raw problem in response to design difficulties.

The Office raw problem also contained another micro problem, *distribution list interpretation*, that together with modification seemed to require the operations approach to rule specification, or to require that variables be used in the before-after approach. It was difficult to find an attractively simple presentation for rules that dealt with these micro problems.

One of the designers, JR, noted that modification and distribution list interpretation were not required subproblems of Office, but rather arose from one approach to solving it. JR pointed out that we were intent on building a model of distribution lists that would work for any memo and any distribution list, including ones created after the model was built, whereas the Office raw problem only required that we demonstrate how a typical distribution list worked. This goal could be achieved without treating memos and distribution lists as structured objects at all, just by providing rules that would produce the correct copying and routing for known memos and distribution lists. This recasting of the Office problem made it possible to ignore the modification and distribution list micro problems, if desired, and thus opened the way for a variety of simpler designs.

Example 5. Micro problem derived from raw problem, design alternative spawned by micro problem.

The original ChemTrains design prescribed that reaction rules could only operate when all participating objects were in the same place. This

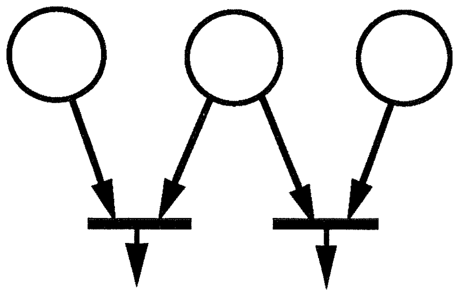


Figure 2a. A simple Petri net in which two transitions, shown as horizontal lines, share one of three input places, shown as circles.

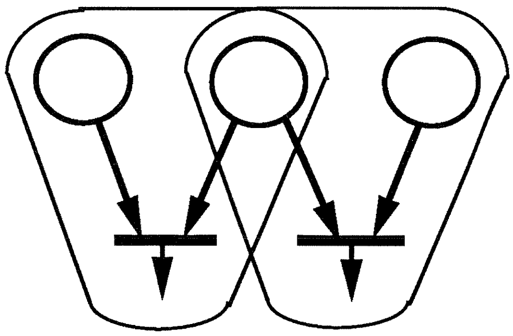


Figure 2b. Drawing ChemTrains places around the transitions and their associated input places permits tokens relevant to each transition to interact.

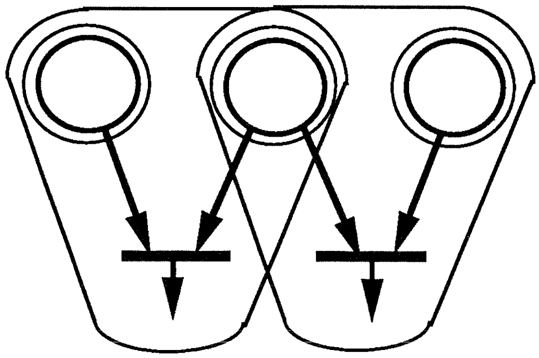


Figure 2c. Drawing ChemTrains places around each input place permits the position of tokens to be controlled.

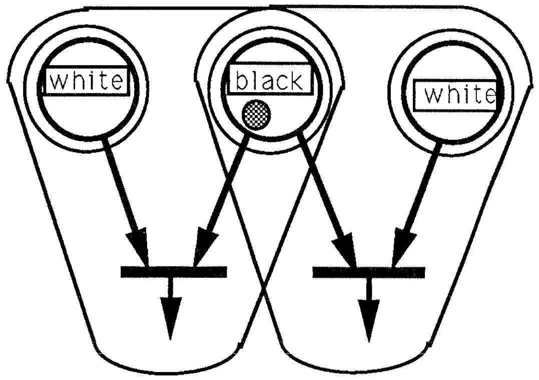


Figure 2d. If objects inside any place are allowed to interact, the token, shown as a shaded circle, can interact with any of the paint markers shown as boxed color names.

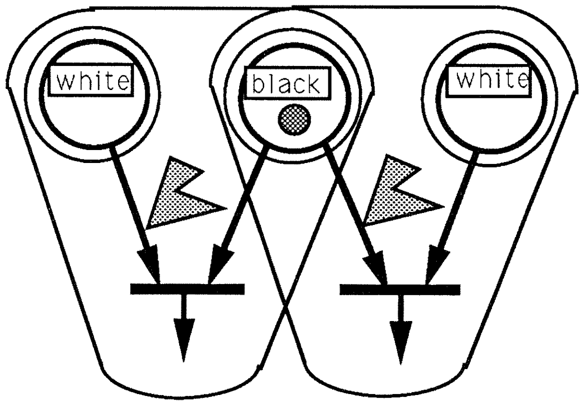


Figure 2e. By adding catalyst objects, shown as irregular polygons, and restricting interactions by Clayton's Membrane Rule (see text), rules which include the catalyst can permit tokens to interact within a place enclosing a transition, while rules which do not include the catalyst can restrict the interaction of tokens and paint markers.

Figure 2. Micro problems in the Petri Net raw problem.

limitation was motivated by CL's intuition that it would make rules simpler and easier to understand than if objects situated anywhere could freely interact.

This restriction on interaction caused immediate difficulty in handling the Petri Net raw problem. Figure 2a shows part of that problem. The circles in the diagram represent *input places* (not to be confused with ChemTrains places!) and the vertical lines represent *transitions*. The rules of operation for a Petri net specify that when one or more *tokens* are in each input place for some transition one token is absorbed from each input place and a single token is passed out of the transition (along the paths leading to the bottom of the diagram.) Notice that a token in the middle input place may trigger the firing of either transition, depending on where other tokens are.

In considering what ChemTrains places should be set up to model the Petri net there is obvious trouble. One would like to have a place enclosing both input places for each transition, since tokens in the input places need to interact to trigger the transition. But since the middle input place is shared by two transitions, these places would have to overlap, something not contemplated in the original ChemTrains design. Thus the *shared input place* micro problem, part of the Petri Net raw problem, directly spawned a new design alternative: permit ChemTrains places to overlap.

Example 6: Micro problem derived from raw problem, solution method suggested by micro problem, micro problem spawned by solution method, design alternative spawned by micro problem.

Continuing with the Petri Net problem, if one lays out overlapping ChemTrains places as shown in Figure 2b one now has the problem of distinguishing tokens in the two input places that are inside each ChemTrains place. This is necessary because the transition is triggered not by any two tokens but only by one token in each input place. A natural way to approach this, given that ChemTrains places can overlap, is to include smaller ChemTrains places inside the overlapping ones, one for each input place, as shown in Figure 2c. Tokens in these places can then be kept separate. But how can one detect when the two input places are both occupied? There was no direct way in the original ChemTrains design to specify that an object or objects had to be in specified places for a reaction rule to apply.

This micro problem of identifying the tokens' location spawned not a design alternative directly but an approach to a solution, which in turn led to design alternatives. The approach is called *painting*: the idea is that when a token is in a particular input place it is "painted" a color associated with that place. Assigning colors to input places in such a way that each transition has input places of two different colors, the rule that fires a transition can apply just when there is at least one token present of each of the two colors.

But how is painting to be controlled? Suppose a marker of the appropriate color is placed in each input place, and a rule is provided that modifies the color of any token in the presence of that marker. This paints too much: any token anywhere within the large place enclosing an entire transition will be in the same place as the marker in its input place and also the marker in the other input place. Thus the token in Figure 2d could get painted by the black marker or by the white marker since the token and both markers are enclosed in the outer place. This is the *controlling paint* micro problem.

It might seem possible to solve this problem by redefining when objects are considered to be in the same place. After all, the white marker in Figure 2d is not in the same inner place as the token. Perhaps rules should only apply when the participating objects are all in the same innermost place. But this throws the baby out with the bath: under this rule the painted tokens can no longer trigger the transition, because they are in different innermost places!

This difficulty is skirted in a new design alternative spawned by the *controlling paint* micro problem. Under *Clayton's Membrane Rule* any reaction must occur in a single place, all participating objects must be somewhere inside that place, and at least one participating object must be *immediately within* the place, that is, not further enclosed by any smaller place. So the token in Figure 2d can't be painted by the white marker, because neither the token nor the marker is immediately within any place that includes the other. To get the painted tokens to trigger the transition a new *catalyst* object is placed inside the outer place, as shown in Figure 2e, and included in the condition for the rule that fires the transition. Now the outer place includes painted tokens of the required colors, and immediately encloses the catalyst object, so the rule can apply.

Example *7*: Raw problems added to represent abstract concerns, micro problem motivates design alternative.

As mentioned earlier, one of us, BB, was concerned that the original ChemTrains designs and its derivatives were too limited to solve many realistic problems. He felt that features of existing rule-based systems, such as control of conflict resolution (what to do when more than one rule could apply) and structured representations of objects, places and paths would be needed. CL and JR rejoined that these features would be difficult to understand, and that it was not clear they would be needed for problems of interest. To develop the argument BB proposed two additional raw problems, Tic Tac Toe and Maze, and these were accepted as within the intended scope of ChemTrains. BB believed these problems would expose the limitations of existing ChemTrains designs.

The Tic Tac Toe problem quickly brought out a difficulty with the prohibition of interaction among objects in different places, the same design limitation that was dealt with, after some redesign, in the Petri Net problem. Since the Tic Tac Toe model plays a game against a user the model should make a move only when it is the model's turn to move, so there has to be some indication of whose turn it is. But since this turn indicator must interact with any other objects that are involved making a move, a complex arrangement of overlapping places would be required. For example, finding two O's and a blank in any row, column, or diagonal suffices to identify a win for O, but the places representing all of the pieces of the board would have to be extended to enclose the object representing the turn. BB showed that this problem could easily be avoided simply by allowing rules to test for the presence of objects in multiple places. Thus the micro problem *turn control* spawned the design alternative of permitting rules to test for objects in multiple places.

2.4 Collapsing the design space.

It became apparent that development of new micro problems and design alternatives could go on indefinitely. It was also clear that while we could produce designs that were *expressive* enough to solve the raw problems using many combinations of the features we had proposed, we could not assess the *facility* that different combinations of features would afford without examining the combinations in detail. Individual design features in themselves did not promise facility or the lack thereof; only by examining the process required to solve a problem with a complete design could we determine this.

We closed down the exploration of further design alternatives and defined three complete designs, ZeroTrains, ShowTrains, and OpsTrains, which represented distinct ways of choosing alternatives within the space. In ZeroTrains the simpler alternative was chosen whenever possible. Thus for example ZeroTrains retained the restriction that rules could apply only to objects in a single place. OpsTrains chose alternatives for power, influenced strongly by BB's experience with the OPS family of rule-based systems (see Forgy 1984). Thus OpsTrains views places as a kind of object, allows objects to be nested, and permits variables in rules. ShowTrains differs from the other two designs along a different dimension, concreteness. In ShowTrains rules are not expressed in any notation but rather are specified by demonstration, a technique suggested by Maulsby's work on specifying procedures by example (Maulsby, 1989). The ShowTrains user always

Table 2. Design features of the ChemTrains variants.

Z: ZeroTrains S: ShowTrains O: OpsTrains

Feature	Z	S	O
PLACES			
separation of graphic for place	F	T	T
attributes of places	F	F	T
hideable places	F	T	T
nameable places	F	T	F
overlapping places	T	F	T
nested places	T	F	T
OBJECTS			
structured objects	F	F	T
attribute-value pairs	F	F	T
nameable objects	F	T	F
hideable objects	F	T	T
PATHS			
named path ends	T	F	F
named paths	F	T	T
one way	F	T	T
two way	T	F	T
filters	F	F	F
path traversal in reactions	T	T	T
RULES			
global scope of rules	T	T	T
group of events in rule	T	T	T
places as well as objects in rules	F	T	T
Clayton's Membrane Rule	T	F	F
before&after specification	T	F	T
use of variables	F	F	T
graphic specification of modification and creation without variables	F	T	F
specification by demonstration	F	T	F
no conflict resolution	T	T	F
rule ordering	F	F	T
specificity and recency	F	F	F
absence testing	F	T	T

works within the context of the model, pointing out the objects that figure in the condition of a rule and then manually making the changes that should occur if the rule applies. Table 2 collects the design alternatives included in Figure 1 and shows which alternatives were chosen in each of these three designs.

As is evident in Table 2 the three designs we chose come nowhere near exhausting the possible combinations of design alternatives in the design space. But the designs do permit us to compare the consequences of designing by three distinct intuitions that we found increasingly in conflict in design discussions. We could not settle by discussion whether the power of OpsTrains would make it easier to apply than ZeroTrains, or whether the concreteness of ShowTrains would make it easier to understand and use than the other two designs.

2.5 Evaluating expressiveness.

The first step in evaluating the three designs was to attempt to find solutions to all six raw problems using each design. We were confident from discussion of the various design features that this would be possible, and some last minute adjustment to the designs was permitted in preparing the solutions. The decision to include Clayton's Membrane Rule in ZeroTrains, for example, was made only after some effort was put in confirming that good solutions not only to Petri Net but also Bunsen Burner depended on it. As expected reasonable solutions were found for all the problems for each design.

2.6 Development of doctrine and evaluation of facility.

The next step in the evaluation of the designs was to examine the mental steps required to develop solutions to the raw problems starting from the problem statements and drawing on a body of doctrine provided with the design. We began by preparing an initial statement of doctrine reflecting the steps that seemed to be involved in producing the solutions constructed in the expressiveness assessment. We then worked through each problem repeatedly, looking for points in the solution process where the doctrine provided inadequate or inaccurate guidance. The goal was to produce doctrine that as often as possible would provide a direct link between an aspect of a problem on the one hand, and a specific part of a solution in a design on the other.

As the doctrine developed it sometimes happened that a new solution to a problem was adopted. Some of the original solutions had reflected insight into a design that could not be conveyed readily in doctrine, or had resulted from a good deal of trial-and-error problem solving. Some of these could be replaced by solutions which could be derived more readily from a reasonable body of doctrine.

The Bunsen Burner solution in ZeroTrains illustrates this joint evolution of doctrine and solutions. In the original solution the place containing the flame was expanded to include both the beaker and the various places for the control. This is an economical solution to the problem of allowing the parts of the burner setup to interact under the ZeroTrains restriction that interacting objects must be in the same place. But it proved hard to formulate doctrine that could reliably indicate which of two or more places to expand in situations like this. After examining the requirements of this and other problems the doctrine shown as item D17 in Table A-1c was developed. It leads to a less economical solution, because a catalyst object is needed, but the doctrine seems to provide direct guidance in a wide range of cases.

We also found the need to revise the statements of the raw problems during this phase of the process. The original problem statements often permitted different approaches to the problems, which in turn led to different solutions. For example, the Maze problem as originally stated could be solved by following walls or by marking routes that have been explored. Different approaches could have been dictated by different doctrine for the designs, but they were not. We therefore selected one approach for each problem and specified it in the problem statement. This avoided differences in the solutions that were not traceable to differences in the design but just to the solution approach adopted.

The process of examining and modifying doctrine consisted of tracing out a path from a problem statement to a solution, using the doctrine at each step to determine what to do next. This analysis is similar in spirit to the cognitive walkthrough used for user interface evaluations by Lewis et al (1990). In the cognitive walkthrough the focus is on the adequacy of cues provided by the user interface to guide choices the user must make. In our analysis the focus is on the adequacy of the doctrine. We call our variant a "programming walkthrough." A programming walkthrough evaluates the facility of the underlying design indirectly: a design has high facility if there is a body of doctrine for it that guides each step in the solution process, and which is not large or complex.

The doctrine for the three designs is excerpted in Tables A-1a, A-1b, and A-1c. While differences in form and content make precise comparison impossible, it appears that three designs required about the same amounts of doctrine. Taking an item of doctrine to be a statement of a choice to make or action to take, together with some statement of the conditions in which it is relevant, OpsTrains has 21 items, ShowTrains 24, and ZeroTrains 26.

We concluded this phase of the design effort by producing “walkthroughs of record” for each of the six raw problems in all three designs. In these walkthroughs, the doctrine and design features were fixed across all problems. Each walkthrough took as its starting point a written description of a raw problem. The walkthrough then described the sequence of problem-solving activities and system interactions that the programmer would need to follow in arriving at a solution. At every point, the programmer’s critical decisions were justified by citations to specific items of doctrine and features of the problem. Since the designs and doctrine had been developed with these six problems as targets, the walkthroughs were a chance to exhibit each design in the best possible light.

2.7 Programming walkthroughs of the Bunsen Burner Problem

In this section we describe a walkthrough of record for each of the three designs, using a single raw problem, Bunsen Burner. The problem asks the programmer to simulate the operation of a bunsen burner, showing how the burner flame changes as a user drags a control knob into one of three positions: off, low, and high. The simulation must also show how water in a beaker above the flame changes phase, from ice to water to steam, as the flame changes. Table 3 shows the full problem statement. The full text of the walkthroughs describing the solution path are shown in Tables A-2a, A-2b, and A-2c, at the end of this report. Those tables refer to specific items of doctrine for each system, shown in Tables A-1a, A-1b, and A-1c.

Table 3. Statement of Bunsen Burner problem used in walkthroughs

---

Purpose: Show how changing the position of the control on a bunsen burner affects the state of the water in a beaker.

Task Description: Show a bunsen burner with a beaker of water on top of it. Also on screen, separated from the burner, show a control with positions high, low, and off. Allow the user to manipulate the control. When the control is in the high position, a large flame should be visible between the burner and the beaker, and the water should be shown as a cloud of steam (the cloud should stay in the beaker). When the control is in the low position, a small flame should be visible and the water should look like plain water. When the control is in the off position, no flame should be visible and the water should be shown as a cube of ice.

OpsTrains Solution Path. We present the OpsTrains walkthrough first because it is the easiest to understand. In OpsTrains, objects and places are not different elements: any object may contain other objects. An object is identified by its shape and by the objects it contains.

The doctrine for OpsTrains tells the programmer what to do as a first step, but no further order of action is specified. Doctrinal rules suggest appropriate actions to take in various situations. For the bunsen burner simulation, the doctrine tells the programmer to start by drawing the visible objects of the simulation. The programmer draws the bunsen burner, the beaker, and the control panel, which is made up of three separate rectangles in which the control may be located, marked by the text strings "off," "lo," and "hi." Drawing an object in OpsTrains causes the object to be created. Additional items of doctrine induce the programmer to draw a place for the flame to appear, enclosing an object whose graphic is the text string "off." Both the "off" object and the flame-place object are defined to be invisible when the simulation runs (Figure 3a).

An item of doctrine now suggests that the programmer specify the OpsTrains reaction rules that describe the activity in the simulation. The same item of doctrine describes the general procedure for specifying a reaction rule: set up the simulation window the way it will appear when the reaction is to occur, then copy-and-paste the relevant patterns of objects and paths into both the "condition" and "action" sides of a new reaction rule. Finally, modify the action side of the reaction rule to show its results, which may involve deleting objects, creating new objects, or sending objects along paths. Two additional items of doctrine map the general idea of reactions into the specific needs of the problem for deletion or creation.

Guided by this doctrine, the programmer begins to specify the reaction rule that causes the flame to change from "off" to a low-flame appearance when the control is moved into the low position. As the programmer is specifying the reaction rule, another item of doctrine becomes applicable. The appearance of the flame when the control is moved into the Low position is irrelevant; whether it is a high flame or the invisible "off" object, it should be transformed into a low-flame graphic. The doctrine advises the programmer to make the flame's appearance a variable in the condition of the rule. The programmer follows this advice, marking the flame with a graphic "V" that indicates its variability.

For the bunsen burner simulation, a total of six reaction rules are required. Three reaction rules change any flame object to a low flame, high flame, or the invisible "off" marker when the control is in the low, high, or off position. Three more reaction rules are needed to transform any phase of

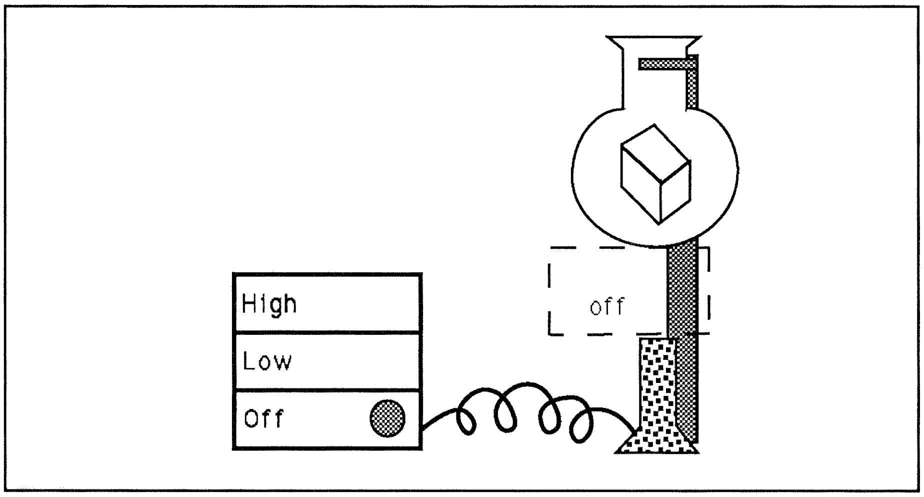


Figure 3a. OpsTrains layout

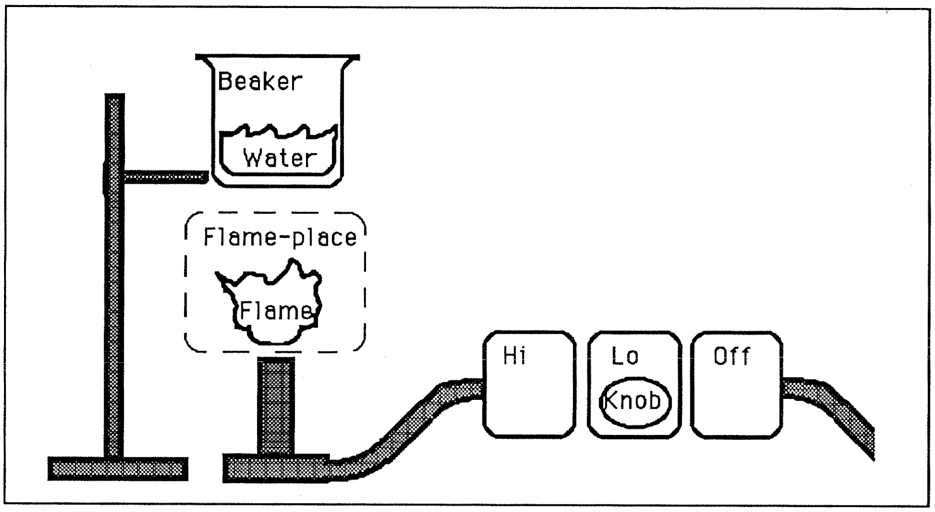


Figure 3b. ShowTrains layout

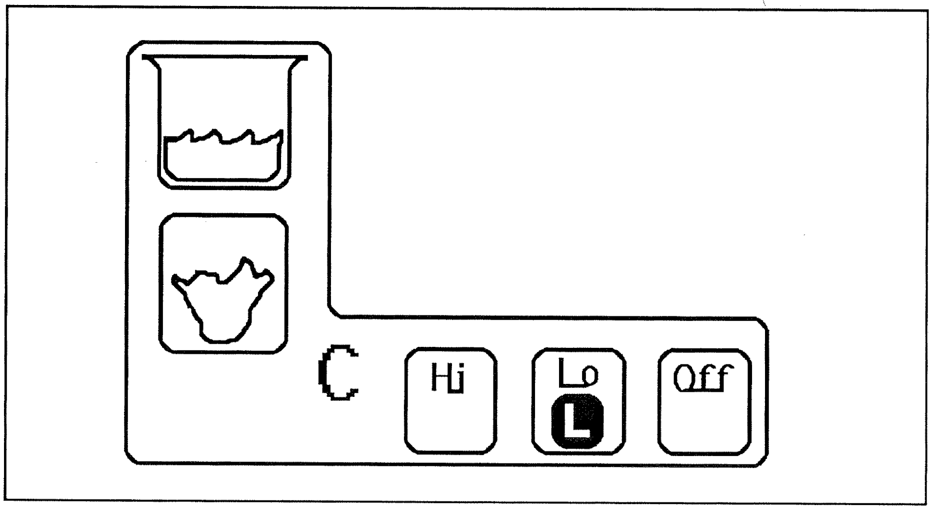


Figure 3c. ZeroTrains layout

Figure 3. Graphical layouts of Bunsen Burner solutions.

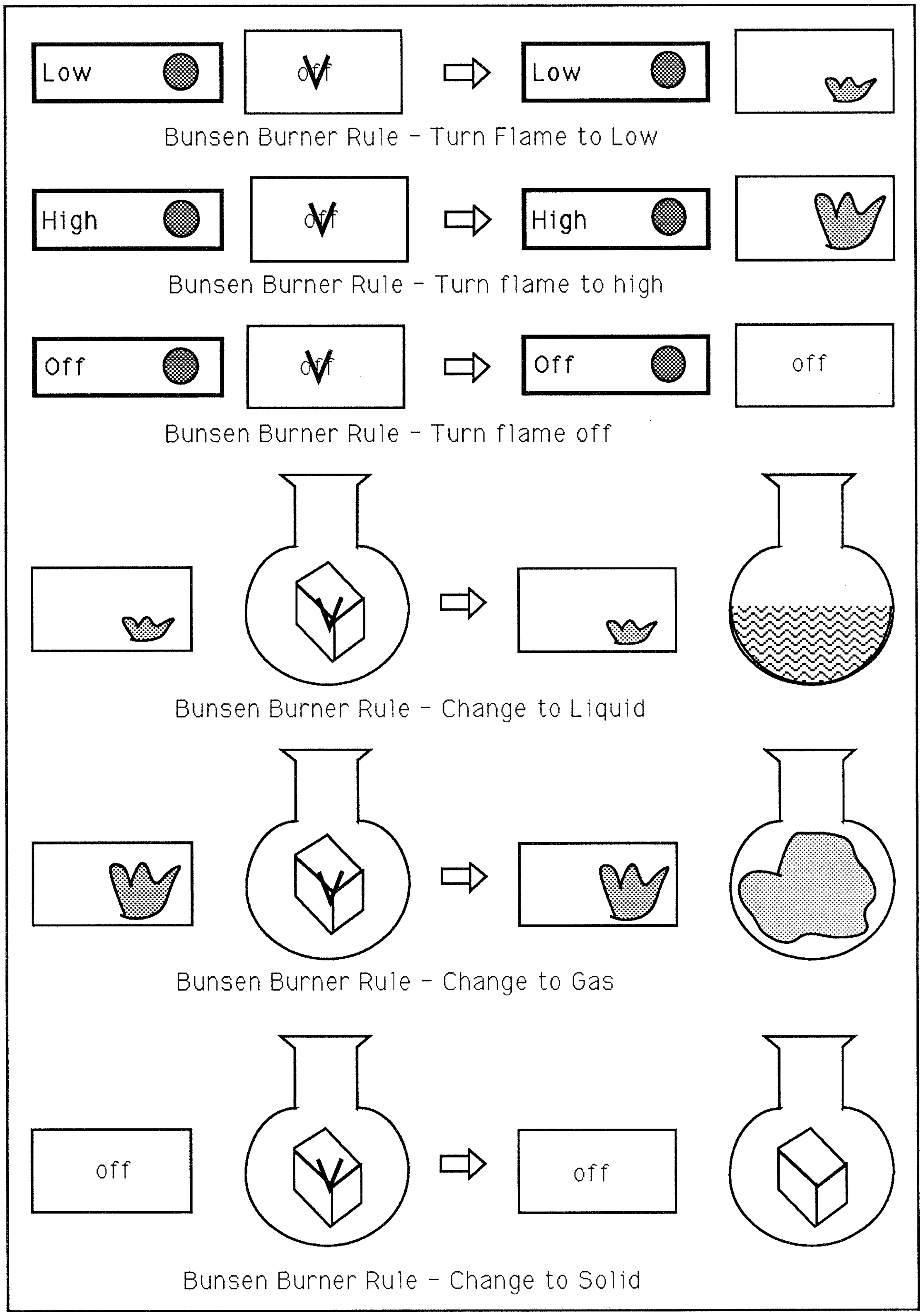

Low
off
Low
Bunsen Burner Rule - Turn Flame to Low
High
off
High
Bunsen Burner Rule - Turn flame to high
Off
off
Off
off
Bunsen Burner Rule - Turn flame off
Bunsen Burner Rule - Change to Liquid
Bunsen Burner Rule - Change to Gas
off
off
Bunsen Burner Rule - Change to Solid

Figure 4. Reaction rules in OpsTrains solution for Bunsen Burner.

the water to ice, liquid, or steam, depending on whether the object in the flame-place is “off,” low-flame, or hi-flame. The walkthrough shows that the programmer enters each of the reaction rules as a direct result of the application of doctrine to the problem statement. The graphic versions of the reaction rules are shown in Figure 4.

ShowTrains Solution Path. The final ShowTrains solution is similar to OpsTrains, but the walkthrough shows a solution path that requires many more decisions on the part of the programmer. In ShowTrains, elements of the simulation may be either objects, places, or paths. Where the OpsTrains programmer merely had to decide that some real-world thing would be an object, the ShowTrains programmer has to decide whether it will be a place or an object, and the walkthrough must show how the design, doctrine, and problem combine to justify each decision. ShowTrains is further complicated by the fact that objects, places, and paths have both a name and a graphic.

The ShowTrains doctrine advises a sequence of basic activities, then presents additional suggestions that the programmer should consider in case of difficulty. For the bunsen burner simulation, the ShowTrains programmer, guided by the first item of sequential doctrine, begins with a paper sketch of the simulation. The programmer then decides, again with help from doctrine, what should be places, objects, and paths, and creates these things in the simulation window (Figure 3b).

The objects needed for the bunsen burner simulation are the control knob, the flame, and the water. The places needed are the three positions for the control knob, the place representing the beaker, and the invisible place where the flame appears. The programmer must assign names to each object and place, and the walkthrough cites doctrine that guides the naming process, which is critical since the names will be tested by the reaction rules.

The next item of doctrine advises the programmer to specify the ShowTrains reaction rules that describe the simulation’s action. In ShowTrains, reaction rules are specified by demonstrating them in the simulation window itself. Variables in the OpsTrains sense are not available, but by highlighting or graying out items on the screen, the demonstration can specify that either the graphic, the name, or both the name and graphic of an element are to be tested in the reaction rule’s condition. It is also possible to indicate that objects must be contained together in some common place, but that any place will do. The doctrine describes, using examples, what should or should not be highlighted, and the system provides defaults that are intended to be appropriate for most situations. The walkthrough cites these items as justification for the programmer's demonstration of the first reaction rule for the bunsen simulation, which sets the flame to a low-flame graphic when the control is in Low. The demonstration proceeds roughly as follows:

The programmer sets up the screen as it would look when the control has just been moved from off to low. Now the programmer demonstrates the conditions that cause the reaction to occur by highlighting the name of the flame object (but not its graphic) and the name of the control-knob object (but not its graphic). The highlighting of the place where the objects reside is also significant, but the default highlighting is correct for the bunsen burner simulation. Finally the programmer demonstrates the result of the reaction: the null graphic of the flame object is to be replaced by a low-flame graphic.

As with OpsTrains, a total of six rules are needed, three to set the graphic of the flame depending on the position of the control, and three to set the graphic of the water based on the graphic of the flame. In the second three rules, it is the graphic of the flame that is significant, whereas in the first three it was the location of the control knob, a distinction that the doctrine induces the programmer to map into appropriate highlighting.

The final items of sequential doctrine drive the programmer to test the simulation and to draw graphic elements for inert objects that should appear in the simulation, including a ring stand, the burner body, and a hose between the burner and the control.

ZeroTrains Solution Path. In the ZeroTrains design, the elements from which a simulation must be built are places, objects, and paths. These elements are simple graphics, with no names. Places may be nested or overlapped, but objects can't contain other objects. Reactions have fewer options in ZeroTrains than in OpsTrains or ShowTrains, and the ZeroTrains final solution is significantly different from the solutions in the other designs.

The walkthrough of the ZeroTrains solution path begins similarly to the walkthrough for ShowTrains. Guided by sequential doctrine, the programmer first makes a sketch of the problem on paper. Using this sketch, the programmer makes an initial identification of what will be places, objects, and paths. The programmer then begins to list, on paper, the events that will take place in the simulation, such as: when there is no flame in the burner the water will turn to ice. These will eventually become the simulation's reaction rules.

Doctrine requires that the events be specified only in terms of objects (not places). Where existing objects are insufficient to describe an event, doctrine tells the programmer to create additional objects. In the Bunsen Burner problem, events must occur when the control knob is in different positions, but the positions refer to places, not objects. In the walkthrough, the programmer creates objects with the text strings as their graphics to identify each of the control locations.

Doctrine also guides the programmer through the steps required because ZeroTrains can't test for the absence of objects. The programmer creates a "noflame" object representing the absent flame, then lists the events that create and delete that object as the control is moved.

The ZeroTrains design adheres to the initial ChemTrains principle that things can interact only when they are together in the same place. Where the events list shows conditions in one place changing objects in another place, the programmer has additional work to do. For simple simulations, doctrine spells out the programmer's activity quite clearly, and the walkthrough shows that the bunsen problem fits within this category. The programmer creates a large place that overlaps the flame place, the beaker place, and the places where the control knob may rest. A "catalyst" object, with the text-string "C" as its graphic, is also created, and this is placed in the large place, outside of any other place (Figure 3c). The programmer adds the existence of the catalyst object to the conditions of each event that involves objects in more than one of the original places. This doctrinal guidance allows the programmer to use "Clayton's Membrane Rule", described earlier, without understanding it, at least in simple problems.

Finally doctrine guides the programmer through the last step of the solution: translating the events list into reaction rules in the ZeroTrains graphic reaction-editor window. This is fairly simple, since the events list is written in terms of objects occurring in the same place, exactly what the reactions require. However, nothing can be variablized in a ZeroTrains reaction, so events specifying "a or b" in their condition must be entered as two separate reaction rules. Twenty-four reaction rules are required for the final solution.

2.8 Comparing the programming walkthroughs

Comparison of the walkthroughs for the three designs revealed a consistent pattern of differences, clearly attributable to particular design decisions. The comparison does much to resolve the conflicts of intuition that bedevilled our efforts to evaluate these design decisions earlier in the development of the design.

Overall, the power-oriented OpsTrains design showed the greatest facility. The central ideas of the problem statement could be translated quite directly into OpsTrains rules guided by doctrine of about the same size and complexity as for the other designs. ShowTrains placed second, with many solutions reachable about as directly as OpsTrains, but others requiring considerable problem-solving even with the doctrine supplied. ZeroTrains placed last, with each solution requiring work directly attributable to getting around ZeroTrains' "simplifying" assumptions. Overall, the comparison provides no support for the intuition that the simplicity of the ZeroTrains design would pay back in simple, clear doctrine what it lost in power.

The differences exposed by the walkthroughs can be attributed to a few key design choices. ZeroTrains was hurt especially by two limitations. Since rules could involve only objects in a single place much work was required to contrive this situation, by adding new places and catalyst objects, as in the Bunsen Burner problem, or by sending "messenger" objects on a circuit of places to be processed.

ZeroTrains also suffered badly from the inability to test for the absence of objects. Tests for absence occurred naturally in a number of the problems, and in each case the ZeroTrains solver had to laboriously convert these tests into other, directly expressible tests.

Both ZeroTrains and ShowTrains lacked explicit variables, as were available in OpsTrains. This lack had minor impact on these small-scale problems, since it was not unreasonable to provide multiple similar rules to cover cases OpsTrains could cover in a single, variabilized rule. ShowTrains' distinction between the name of an object and the graphic for it permits different objects in a class to be handled by a single rule, but almost all ShowTrains solutions still required more rules than OpsTrains.

It is not surprising that variables permitted more economical solutions in OpsTrains. It is perhaps more surprising that the use of variables did not require substantial doctrine. The OpsTrains doctrine guides the user in an approach similar to that used in Query by Example (Zloof 1975), in which the user first writes a rule with no variables that handles a specific case. The user then replaces some constants by variables to produce a rule that handles other cases. As in Query by Example this approach appears to allow the OpsTrains user to create rules that use variables in a fairly complex way, for example to require that designated parts of two different objects must be the same, without really understanding much about how the variables work.

Our evaluation may overstate the advantages of OpsTrains' use of variables in one respect. Making effective use of variables requires representing objects in such a way that groups of related objects can be easily described. For example, in the OpsTrains Tic Tac Toe solution each cell in the first row of the board contains a "row 1" marker, each cell in the second row "row 2", and so on for each row, column and diagonal. By writing a rule that refers to three cells marked, say, "row 1", and then replacing "row 1" by a variable one obtains a rule that applies to any row, column or diagonal.

But how does one decide to place these markers? Related questions arise about the choice of names and graphics for ShowTrains objects. The doctrine for these systems provides general guidance, but we suspect that some problem-solving not required in ZeroTrains may still be needed.

Another OpsTrains feature greatly simplified the approach to one problem, the Maze. OpsTrains rules can test for configurations of paths, places, and objects, rather than just objects as in ZeroTrains or places and objects as in ShowTrains. This permits the OpsTrains user to test for the presence of a path leading to an unvisited place in the Maze problem. This test avoids the need to impose an order on the exploration of maze branches, which complicates the solutions in ZeroTrains and ShowTrains.

It is important to note the difference between the expressiveness evaluation of the designs, based just on the characteristics of the solutions supported by them, and this facility analysis which examines the process of arriving at solutions. Consider as an example the ZeroTrains requirement that rules involve only one place. It might appear that simply examining the solutions available with and without this restriction would suffice to establish its cost. But in fact the restriction does not make solutions much bigger. Only in the programming walkthroughs does it become obvious that meeting the ZeroTrains restriction requires a great deal of extra work. This can be clearly seen in the Bunsen Burner example.

The success of OpsTrains challenges the widely-held intuition (see for example MacLennan 1987, p. 547) that languages should contain a minimum number of concepts. Until the programming walkthroughs were complete it seemed quite likely to us that the power of OpsTrains would exact a cost, that understanding how to select among and apply its larger number of capabilities would require more doctrine or more problem solving. This did not happen, because the features of OpsTrains matched the naturally-arising statements of problems more directly than did the limited features of (especially) ZeroTrains. If added features in a language make it easier to relate the language to real problems they confer a significant advantage.

2.9 Evaluation using reserve problems.

The whole design process up to this point was strongly shaped by the set of raw problems. The growth of the design space and the evaluation of alternative complete designs for expressiveness and for facility relied on these problems. The economy and coherence gained by concentrating on a few problems is accompanied by a risk: what if the resulting designs were narrowly tuned to look good on these problems, but would fail on other problems that were within the intended scope of ChemTrains but were not included in the raw problem set?

To address this problem we held a number of raw problems in reserve, not using them in design discussions or in the evaluation process. These reserve problems included one of the original motivating problems for ChemTrains, modeling sunspot development, which was discussed briefly at the very start of the design process and then withdrawn from consideration. We added to this other problems that we encountered in the literature or in other work and that seemed within the intended scope of ChemTrains. Brief descriptions of the reserve problems are shown in Table 4.

After completing the expressiveness and facility evaluation of the three designs on the original raw problems we did walkthroughs for each design on each reserve problem. Neither the designs nor the doctrine used in the earlier evaluation were allowed to change, in keeping with the goal of detecting narrow focus in design or doctrine resulting from our concentration on the original raw problems earlier in the design process.

The analyses using the reserve problems produced results generally consistent with those from the original raw problems. The advantage of OpsTrains over the other designs in both expressiveness and facility was even greater than in the original raw problems because the reserve problems tended to be bigger. The lack of variables in ZeroTrains, and the limitations on variablization in ShowTrains, meant that many, many more rules were required under these designs than in OpsTrains, a fact already apparent in the earlier analyses but of more consequence for the bigger reserve problems.

While the analyses of the reserve problems did not change our assessment of the three designs very markedly they did reveal that some potentially important design issues were not adequately raised by the original raw problems. Three of the reserve problems, Doorbell, Sunspots, and Grasshopper, involved time-dependent processes, which had not figured in any of the original raw problems. None of the three designs had adequately direct ways of representing such processes.

Table 4. Brief descriptions of the four reserve problems.

(The descriptions used in the walkthroughs were more detailed.)

Doorbell. Simulate an electromechanical doorbell. At the push of a button, current generates a magnetic field, pulling on an arm that rings a bell and opens the circuit. A spring pulls the arm back and the cycle repeats.

Grasshoppers. Grasshoppers hatch, consume resources, change to adults, lay eggs, and die. Show how availability and importance of resources affects the population of adult grasshoppers over many generations. See Varley, Gradwell and Hassell (1973).

Sunspots. Show how a tube of magnetic field lines bends and rises above the surface of the sun, causing sunspots to appear. Then show how the tube rises even higher, the field lines separate and reconnect, and a solar flare is produced. Draws on Patrick McIntosh, personal communication (1990), Noyes (1982), Tandberg-Hanssen and Emslie (1988).

Copier. Show how an office photocopier operates. Actions to simulate include transfer of charge, transfer of toner particles conditional on charge, and coordinated motion of the paper and an internal belt. See Shrager (1984, 1987).

The reserve problems also exercised the ability to represent structured objects more fully than the original problems. In the Grasshopper problem, for example, it was natural to think of the state of a grasshopper as combining several components reflecting food and water needs, stage of maturation, and the like. This was difficult to represent in ShowTrains and ZeroTrains.

The ability in OpsTrains to move paths around was useful in the Doorbell problem to open and close a circuit. This feature was not used in any of the original raw problems.

None of the designs provided rotation of the graphics for objects. This is needed to portray the motion of segments in a moving belt in the Copier problem.

The doctrine for the three designs stood up quite well to the reserve problems. Even more than with the designs themselves we had been concerned that our doctrine might have been tailored to the requirements of the original raw problems, but there were few indications that different doctrine would have worked better for the reserve problems. Both ShowTrains and ZeroTrains had in fact included some "advanced" doctrine that was not needed in the original raw problems, but this was not needed for the reserve problems either.

All in all the reserve problems confirmed the conclusions drawn from the original raw problems, but did reveal that the original problems did not cover all aspects of the intended design space. This difficulty could be met in part by including more complex raw problems in the starting set for design development.

3. METHODOLOGICAL LESSONS FROM THE CASE STUDY

3.1 The role of problems in developing design alternatives.

Problems played three key roles in the growth of the design space for ChemTrains. Raw problems defined the functional scope of the intended design. Directing design effort at these problems kept the design activity focussed on issues that had to be dealt with for a design to succeed.

Micro problems played two roles. First, micro problems were used to evaluate or compare design alternatives by capturing difficulties associated with the use of some design alternative. In this role they can be seen as children of design issues. But micro problems are also the parents of design issues, in that they served to promote the formulation of new design alternatives to deal with the difficulties they represented.

In all of these roles two characteristics of problems seem to be critical. First, they are *concrete.* We do not think we could have prepared an abstract specification of the functional requirements of ChemTrains. Or rather, we could have, but only by beginning with a list of raw problems and trying to describe abstractly the characteristics apparently required in these problems. We simply lacked abstract concepts and categories adequate to describe what ChemTrains should do. Collecting raw problems was easy and natural.

The concreteness of micro problems was equally important. Occasionally an issue arose in abstract form, as in the case of conflict resolution, because it was familiar from previous work. But usually issues arose as difficulties whose nature needed to be clarified before we could describe it. A concrete micro problem was the natural way to clarify a difficulty, and to communicate it. Design discussions were punctuated by momentary suspensions in which someone would grope for a micro problem; the problem having been found and described, the discussion would leap forward.

Occasionally abstract formulations of issues emerged from noting the relationships among a family of micro problems. For some reason these issues never became an active focus for discussion, though they would be brought up from time to time. For example, we noted that overlapping places, overlapping objects, and messenger objects sent from one place to another all served related purposes in communicating across places. But despite occasional resolutions to do so we never explored this idea, but remained focussed on the issues embodied in concrete problems. Our behavior agreed with Boy's observation (submitted) that designers develop experiential cases but not formal theories.

A second key point about the problems we used, as noted by Ray McCall (personal communication, 1990), is that they all represent pieces of the intended scope of the ChemTrains design. This is obvious for the raw problems: we could not accept failure on any of the raw problems without changing the scope of the design. For micro problems the relationship is conditional, but still strong. As noted earlier, micro problems were children of current design alternatives, and embodied some difficulty for that alternative. Since micro problems were nearly always parts of raw problems, they could not be neglected without neglecting a raw problem or abandoning a design alternative.

In the context of the design process, then, both raw and micro problems represented work that had to be done. Debates about them were not in danger of being irrelevant unless some specific design alternatives eventually proved unworkable.

3.2 The role of problems in design rationale.

We first became aware of the role problems played in our design process when we attempted to record design rationale, as part of the overall project objective to explore the role of cognitive theory in system design. We initially attempted to record the issues under debate and the relationships between them in an IBIS-like way (see Fischer, Lemke, McCall and Morch and Conklin and Yakemovic, submitted). We tried to show issues deriving from issues, as in that framework, but we got bogged down. One problem was that it was not easy, as already noted, for us to give abstract statements of the issues we dealt with. Indeed, even preparing the diagram of the design space in Figure 1 required us to develop abstract descriptions of design alternatives that we did not use in the original design discussions and hence did not have ready to hand. But a second, related difficulty was that the problems kept getting in the way. We found that we thought about design alternatives and their strengths and weaknesses in terms of the micro problems they were connected to. But there was no direct way to incorporate this organizing material, the problems, into the issue structure. The same issue would arise in the Options, Questions and Criteria framework described by MacLean, Young, Belotti and Moran (submitted.)

As soon as we recognized the organizing role of problems it became very easy to map out our design discussions (though not always to think of terms that would be meaningful outside the context of the specific problems we dealt with). One of us (JR) created a Hypercard stack that captured much of the design space diagrammed in Figure 1.

The structure of problems and design alternatives we created contains the design rationale for ChemTrains in a way that is both implicit and concrete. We wrote no statements explaining that we chose alternative A for reason R. But by examining the micro problems associated with a design alternative one can reconstruct these reasons readily, in the form of concrete, specific examples of situations in which that alternative deals with difficulties encountered by other alternatives.

Despite the naturalness of capturing our design rationale in this form we did not keep any up-to-date record of it. Besides the obvious overhead associated with any recording process, we see two reasons for this. First, we found we could keep the rationale in our heads during design discussions. Even though these discussions took place over a long period of time and with many interruptions, we did not feel that we were losing our grasp of the issues or revisiting already plowed ground. The concreteness of the problems probably was important in allowing us to remember them and the alternatives tied to them.

A second reason we did not keep our rationale up to date was that we became confident of our ability to reconstruct the problem structure if we needed to. This is closely related to the point just made, but meant that even though we intended to produce a record of the design rationale we felt little pressure to produce it as we worked. In retrospect we are glad that we recorded as much material as we did, because in fact our recall of matter discussed following the creation of the Hypercard stack is not as complete as we could wish.

3.3 The role of problems in evaluating expressiveness and facility in programming languages.

The use of concrete problems in the ChemTrains design meant that we could not only compare the solutions afforded by different designs but also the processes involved in thinking of the solutions. We could address the facility as well as the expressiveness of the designs.

Like other walkthrough methods (Lewis, Polson, Rieman and Wharton 1990) the programming walkthrough can be seen as a partial substitute for, or preparation for, prototype evaluation. It can give an indication of probable weaknesses in a design without the expense and effort of building a prototype and testing it.

Of course the facility evaluation must be seen as tentative. Not having implemented even prototypes we cannot be sure that the designs could really work as intended. Further, our ability to map out the actual problem-solving processes of users is obviously limited. We nevertheless feel that the attempt to do so moves design debates about programming language design a step away from intuition and taste, where they have been. A designer can try to demonstrate that his or language supports a natural approach to a task by anatomizing the specific sequence of decisions needed to accomplish the task. A critic can point to a particular decision as unmotivated or unguided by knowledge plausibly possessed by users.

Our work also points up the importance of Soloway's concept of plan, not only to the analysis of the use of existing languages but also to the design of new languages. A language design should be viewed as including not just the features of the language but also the doctrine needed to put it to use. It appears to be practical to evaluate both of these elements of a design together, and indeed of limited value to evaluate the features without the doctrine.

A limitation of our facility evaluation is that it examines only the process of writing programs, and not understanding them, testing them, or modifying them, each of which has an important cognitive component. Certainly language design influences these other tasks as well. Further, in considering programming we have dealt only what might be called *shallow* programming, where sufficient knowledge is available (encoded as doctrine, in our case) to guide the writing of programs with very little search. Surely much programming is *deep* programming, where appropriate choices of representation and procedure are not obvious up front, and search and backtracking are required. Perhaps more limited designs would be easier to reason about, and would show an advantage in deep programming that does not appear in shallow programming.

On the other hand, it is likely that our evaluation actually understates the advantages of the less limited OpsTrains design. OpsTrains' relative performance was better on the reserve problems than on the original raw problems, because it could cope better with difficulties that had not appeared in the original problems. In actual use a less limited design offers more assurance that a new problem can be solved.

It is possible that consideration of other tasks, or deep programming rather than shallow, would change our evaluation of the designs we considered. Perhaps the intended simplicity of ZeroTrains would pay off in one of these settings, or the abstractions in OpsTrains would prove costly. We think the way to address these questions is to adapt the walkthrough approach to examine in detail the mental processes involved in these tasks.

3.4 Conclusion.

Though it was no part of our plans to make it so, the ChemTrains design process became a study in the use of concrete problems, as opposed to abstract principles and analysis, in design. Concrete problems entered the process as a way to define objectives and focus our efforts. Once on the scene they proliferated, and penetrated all aspects of design and evaluation. They helped to organize a complex design space. They also made it possible for us to evaluate not only the solutions afforded by the design but also the mental processes required to realize solutions.

Acknowledgments. Victor Schoenberg helped get the ChemTrains design started in early discussions and prototyping. Robert Weaver helped develop the programming walkthrough approach and clarified the role of doctrine in language evaluation. Ray McCall provided many suggestions, welcome encouragement, and keen insight in our efforts to understand and capture our design process.

Appendix: BUNSEN BURNER WALKTHROUGHS AND SUPPORTING DOCTRINE

Table A-1a. OpsTrains doctrine used in the Bunsen Burner walkthrough

The rules are separated into four sets: rules for creating objects, rules for creating paths, rules for building replacement rules, and rules for debugging. The following four rules are for creating objects.

RO1:	IF	starting,
	THEN	draw a picture of envisioned interface as it would initially appear to the user.
RO2:	IF	an object can move or can be placed in a particular area of the interface that is not drawn,
	THEN	draw an object that is big enough to contain the objects, and specify that the object is to be "Hidden in Simulation."
RO5:	IF	a place object is to be used to hold different types of object but never more than one at a time, and the place object is empty,
	THEN	create an object to denote that the object is empty, place it in the empty place object, and specify that the object is to be "Hidden in Simulation."

The following rules are for building replacement rules.

RR1:	IF	an object should be moved or deleted or a new object should be created based on specific conditions that may exist in the picture,
	THEN	enter the Replacement Rule Editor, copy all of the objects and paths relating to the condition and all the objects and paths to be modified from the main picture to the pattern picture of the rule, copy these objects and paths onto the result picture, modify the objects in the result picture appropriately, and give the rule a name that is appropriate for the task it does.
RR2:	IF	an object or path is to be deleted when a rule is executed,
	THEN	remove that object or path from the result picture.

RR3: IF an object or path is to be added when a rule is executed,
THEN create a new object or path or retrieve an existing one, and add it to the result picture.

RR5: IF an object in the pattern of a rule may match any object regardless of its display,
THEN specify that this object is a variable.
(a big V will be placed over the variable object in the pattern)

[OpsTrains doctrine also describes how to use paths, describes how to debug the simulation, and suggests some things to do in seemingly unusual cases. A few examples of these additional rules follow.]

RP1: IF an object can travel from one place object to another place object,
THEN make a path connecting the two objects that follows the intended route for the objects to travel.

RO4: IF an object or set of objects has a significant difference with other objects that have the same picture, and
the difference is not already shown,
THEN create a new object that can be used to identify these object(s), place a copy inside each object, and hide them if desired.

RD2: IF one rule is executing and causing a more appropriate rule not to execute,
THEN enter the rule order editor, and place the name of the more appropriate rule before the name of the rule that is currently executing.

Table A-1b. ShowTrains doctrine used in the Bunsen Burner walkthrough

---

[An introduction to the doctrine describes the simulation involving a cat, bird, and worm, which is referred to in some of the points of doctrine.]

[D01, D==Doctrine] Whenever you create a new simulation in ShowTrains, there are a number of decisions you have to make. For example, in the cat-and-bird simulation, you would have to decide that the cat, bird, and worm need to be ShowTrains objects, but the birdbath and the tree don't need to be. For a complicated simulation, there can be a lot of difficult decisions, which may interact in surprising ways.

[D02] To help you design a new simulation, we suggest the following guidelines. The basic guidelines (1 through 10) should initially be applied in order, although you will probably go back and forth among them as your simulation takes shape. If a guideline seems especially difficult to apply, you might check the guidelines for difficult situations (11 and following).

1. [D1] **Sketch a Snapshot.** On a blank piece of paper, sketch a "snapshot" of the running simulation — something like Figure 1. Don't spend a lot of time on this, but produce a rough graphic that shows how you expect the screen to look. Don't worry about marking things as Objects, Places, or Paths, but spend a few moments thinking about how you want to interact with the simulation as it is running. For example, will you want to have a button to click that creates more worms for the bird? This is also a good time to glance over the Guidelines for Difficult Cases, to see if any seem to apply to this simulation.

2. [D2 **Identify Objects.** Decide what's going to move or be created, deleted, or modified. These will be Objects. Create the Objects that should exist when the simulation begins to run. Give them meaningful names and graphics.

3. [D3] **Create Places for Objects That Aren't Moving.** Decide where where unmoving Objects are going to rest, and where moving Objects are going to stop, even for a moment. These will

be Places. Whenever an Object isn't moving, it must be in a Place — it can't just be floating on the screen. Create all the Places, and give them meaningful names. If the Places are object-like (a bottle, for example, or a chair or a building) then make them visible and create an appropriate graphic for them. If the Places are just areas within a larger area (for example, the spot where the bird stops to eat the worm), then make them invisible.

[D3a] **Hint: What should the Places be named?** Often this isn't a problem. A good name for the chair Place is obvious: "CHAIR." But sometimes it's useful to give many places the same name, so that a single rule can apply to all of them. For example, there might be several chair Places where the cat could sleep. If you name them all CHAIR, then you can have a single rule that says, "If the object named CAT is in a place named CHAIR, then change its graphic to a sleeping-cat."

4. [D4] **Create Places for Object and User Interaction.** Decide where Objects are going to be when they change or interact. Any change to an Object has to occur in a Place. For example, two airplanes can't collide while they are moving along paths. (However, two Objects can interact while they are in different Places — remember how the cat noticed the bird and woke up.) Users often interact with the simulation by clicking in a Place, or the User may move Objects from one Place to another while the simulation is running. Create any additional Places that are needed. Give these places names and graphics as appropriate.

5. [D5] **Define Paths.** Decide what routes Objects will follow when they are shown moving between Places. These will be Paths. Paths only allow travel in one direction, so Places will sometimes have two Paths between them. Like Places, Paths that represent object-like items in the real world (a pipe or a highway, for example) should be made visible with an appropriate graphic. (The graphic for a Path determines the appearance of a single segment of the Path.) Paths that represent motion within a larger area (the bird wandering through the garden, or an airplane flying from city to city) should be made invisible. Create the Paths you think you'll need, and give them meaningful names.

[D5a] **Hint: What should the Paths be named?** Some possible naming conventions are "TO-wherever," or "FROM-wherever," or "LEFT," "RIGHT," "UP," and "DOWN." In a complex simulation, it may be effective to give many Paths the same name, such as "TO ROME" or "WEST". Then a single rule can apply to objects in many different situations: "Rule: If the Object is named YOUNG MAN, then put it on the Path labelled WEST." Note, however, that Paths which aren't really needed can make your job a lot harder. Remember that the action of a Rule can move things invisibly between Places without a Path.

6. [D6] **Create Object-Changing Rules.** For each Place, decide what conditions will cause Objects to be created, modified, or deleted in that Place. Set up exactly those conditions, highlight the relevant items, and demonstrate to ShowTrains what should occur. Be sure to give each Rule a meaningful name — this will make it easier to revise the simulation.

[D6a] **Hint: What conditions should be highlighted?** For almost every simulation, it's important to highlight exactly the right things when you create a Rule. Stating the Rule you want in English may make it more obvious what to highlight. For example, in the cat-and-bird simulation, you might want a rule that says: "If the bird and the worm are together in any Place, delete the worm and change the bird's graphic to a bird-with-a-worm." This Rule refers to two Objects by name, the bird and the worm. That's a good indication that you want to highlight the names of those Objects, but not their graphics. On the other hand, the Rule talks about "any Place." That means you only want to highlight the outline of the Place in which you're demonstrating the Rule, not its name or graphic. You want the Rule to fire for *any* Place, not just the one in which you're demonstrating it.

[D6a continued] Similarly, if you have several Places named "CHAIR," but you only want the cat to sleep in overstuffed chairs, then the English version of the Rule would be: "If the cat is in any chair with the overstuffed-chair graphic, then change the cat's graphic to a sleeping cat." This rule refers to the cat by name, so you'd highlight the name of the cat Object, but not its graphic. The rule also refers to "any chair" Place, but only those with a specific appearance. So you would want to demonstrate the rule using a Place named "CHAIR" with an overstuffed-chair graphic, highlighting both the Place name and the graphic. (See "Creating Rules by Demonstration" in the ShowTrains manual for a more detailed description of how to create rules, including what to highlight in the Rule's action.)

7. [D7] **Create Object-Moving Rules.** For each path, decide what conditions will cause Objects to be moved onto the Path from the Place where the Path begins. Set up the conditions, highlight the relevant items, and demonstrate to ShowTrains which Objects should be placed onto Paths. If any Objects are to be moved instantaneously from one Place to another, create Rules to cause those moves as well. Be sure to give every Rule a meaningful name.

8. [D8] **Create User-Interaction Rules.** Set up Objects the way they will be after the User interacts with the running simulation, highlight the relevant items, and demonstrate to ShowTrains what changes or movements should occur. Users can interact with the simulation in two ways. They can move Objects from Place to Place, and they can click on Places, which creates a *click* (or *double-click*) object. The *click* or *double-click* object remains until every Rule has been tested, then it is automatically deleted.

9. [D9] **Try It! (And Fix What Doesn't Work).** Set up the Objects in the simulation's start state (make sure every Object is in a Place), and select Test Run from the Simulation menu. If the simulation isn't quite right, modify it and try again. This is another good time to look over the Additional Guidelines.
[D9a] **Caution! Caution! Caution!** Rules aren't applied in the order you wrote them. They aren't even applied in a consistent order each time a simulation is run. *This means that your simulation might work the way you want it one time, then work differently the next time!* When you're testing the simulation, always use the Test Run menu item instead of Run. When running under Test Run, ShowTrains will tell you whenever more than one Rule could fire, so you can examine the Rules and make sure the conflict won't cause a problem.

10. [D10] **Draw a Background.** Draw the Background graphic. It should show everything you haven't already shown as an Object,

Place, or Path. Rearrange the Places and Paths on the Background if necessary.

[ShowTrains also includes quite a bit of specialized doctrine, prefaced by a comment that these items should be considered if the programmer is having difficulty.. A couple of examples (not used in the Bunsen problem) follow.]

11. [D11] **Use Many Small Places.** When simulating a large real-world place, or a long real-world Path, it's often best to use several small ShowTrains Places, connected by Paths. For example, to show a detective moving around a room and looking for clues, you would need to have a small Place for every point at which the detective stopped. These Places can be invisible, and the Background can show the room itself.

20. [D20] **Use a Hidden Object to Control the Order of Rules.** You're writing a graphic simulation of the hiring process at your company. You want to filter each applicant through a series of increasingly difficult requirements, expressed as Rules. It's important that you test the Rules in a specific order: Does the applicant meet the stated requirements for education and experience? If they pass that Rule, then put them through the Initial Interview. If they pass that, send them to the Department Head, and so forth. How can you do this in ShowTrains, since you can't directly control the order in which Rules are tested? One way is to create a hidden Object, which in the hiring simulation you might call APPLICATION-STAGE. Give this a series of graphics that look like numbers. Then, make the graphic of the APPLICATION-STAGE a condition of each Rule, so the DEPARTMENT-HEAD-INTERVIEW Rule only fires when the APPLICATION-STAGE has a "3" graphic. To make sure that all Rules are tested before the APPLICATION-STAGE is changed, use the ALL-TESTED trick described in Appendix X.

Table A-1c. ZeroTrains doctrine used in the Bunsen Burner walkthrough.

D1. First draw a sketch showing roughly what you want to see on the screen for the model.

D2. Things that can move around in the picture, or that can be added to or removed from the picture, or that can change as the model runs will be represented by *objects*. Begin a list of objects for the problem, including different versions of objects that change. Things that can be moved or changed by the user are also represented by objects.

D3. Draw outlines on your sketch to show *places* where objects can appear, disappear, or change, or to which objects can move.

D4. Draw *paths* to connect places between which objects can move. Put names on the ends of the paths in such a way that the paths leaving any place all have different names. If a path will be used only one way you only need to give a name to the end objects will enter.

D7. Things that can happen in ZeroTrains are creation of new objects, deletion or modification of existing objects, or objects moving along a path. One or more of these things that should happen together is called an *event*. Make a list of events for your model. For each event say when it should occur, that is, what situation in the model triggers it.

D8. In describing the events try to specify them in terms of the objects you have already described. If necessary, add new objects toi your list so that you can describe everything that must happen in your model in terms of objects.

D9. Events can only be triggered by the presence of objects. You will find it necessary to rewrite many of your triggering conditions to get them in the right form for ZeroTrains. Here are some suggestions for doing this.

D13. If P is of the form "not A" rewrite it to test for some condition which will be true only if A is not true.

D14. If a predicate has the form "O is in P" for some object O and place P, rewrite it as a test for the presence of O-P, where O-P is a new object. Make a new object P to serve as marker for P and put it in P. Add events to change P to O-P in the presence of P. If O might move out of P make O-P be not a new object but a variant of O, and add events to change it back into O when it gets to its new destination.

D15. If an event should occur only in some certain place or places, make marker objects that identify these places and put them in the places. The marker objects can then be included in the trigger.

D17. The objects in a trigger condition for an event must all be in one place. If they are not, create a new place that encloses the places where the objects are found, and put a catalyst object C in the new place but outside all the original places. Include C in the trigger for the event.

D24. Write a *reaction* rule for each event. Begin by specifying the trigger condition: list the objects that must be present for the event to occur. Arrange the list of objects horizontally with vertical bars separating them.

D25. Then describe what should happen in the event. First, list the objects that should exist after the group of events under the objects in the condition. Put each one under the object in the condition that it should appear near. An object that should be unchanged will just be put under itself. An object that should be deleted will not be listed. A new object will be listed under an object it should appear near. An object to be modified will have a new version shown under it. Second, write a path name under any object that is supposed to move onto a path.

[ZeroTrains doctrine also includes items for more specialized situations. Some examples (not used in the Bunsen problem) follow.]

D6. If a place has many paths leaving it consider breaking it into smaller places each with one of the original paths leaving it. Connect these smaller places with new paths to form a ring.

D19. In some problems patterns that may occur in many places must be dealt with in a way that depends on some general state information. There are two approaches to this kind of problem, which can be adapted to variations of the basic problem. To describe the approaches, let {Pi} be the collection of of places in which the relevant condition may be satisfied.

D22. The *messenger* approach is as follows. Connect the Pi by paths in some order. Send a *messenger object* from place to place, and add the presence of the messenger to the condition tested in each place. If the problem requires processing just one case, delete the messenger when the condition is satisified, or change it to another form, so that the condition won't be tested after being satisfied once. If the messnger makes the round of all places then it can be used as an indicator that the condition was never satisfied, if this information is needed for subsequent processing.

Table A-2a. OpsTrains walkthrough of Bunsen Burner Problem.

[Rxx refers to the number of the doctrine item supporting the action.]

RO1:	draw a bunsen burner, beaker, and control panel, made up of three separate rectangles.
RO2:	draw a rectangle in the area that is to contain the flame.
RO5:	add a hidden object, the text string "off," inside of the flame holding rectangle.
RR1:	add a rule named "Turn Flame to Low"
	copy the "low" rectangle of the control panel and the rectangle holding the flame onto the pattern and result pictures.
RR2:	remove the "off" marker from the result picture.
RR3:	create an object that looks like a low flame and
	add it to the appropriate place in the flame holding rectangle.
RR5:	specify that the "off" object in the flame holder is a variable object.
RR1:	add a rule named "Turn Flame to high"
	follow the same steps as in making "Turn Flame to low." ...
RR1:	add a rule named "Turn Flame Off"
	copy the "off" rectangle of the control panel and the rectangle holding the flame onto the pattern and result pictures.
RR5:	specify that the "off" object in the flame holder is a variable object.
RR1:	add a rule named "Change to liquid"
	copy the beaker and the rectangle holding the flame onto the pattern and result pictures.
RR2:	remove the ice cube from the beaker.
RR3:	create an object that looks like liquid and add it to the beaker.
RR5:	specify that the ice cube object in the beaker is a variable object.
RR1:	add a rule named "Change to gas"
	follow the same steps as in making "Change to liquid" ...
RR1:	add a rule named "Change to solid"
	copy the beaker and the rectangle holding the flame onto the pattern and result pictures.
RR5:	specify that the ice cube in the beaker is a variable.

Table A-2b. ShowTrains walkthrough of Bunsen Burner problem.

[Numbers in brackets refer to items of doctrine]

1. [D02] P decides to follow the first 10 guidelines in order.

2. [D1] P sketches a rough graphic of the Bunsen burner, a beaker, and a control. The control is a lever with three positions.

3. [D2] P decides that the control lever will move, so P creates an Object CONTROL and gives it a knob-shaped graphic. P decides that the height of the flame and the state of the water will change, so P creates a FLAME Object and a WATER Object.

4. [D3] P decides that the flame and the water are unmoving objects, so P creates a Place for the flame and a Place for the water. Per the guideline, P makes the water place look like a beaker and names it BEAKER [D3a]. The flame Place is made invisible, with the name FLAME-PLACE [D3a].

5. [D4] The user has to interact with the control knob, so P makes three Places in which the knob can rest, naming them [D3a] OFF, LOW, and HIGH.

6. [D5] The only moving Object is the control knob. From [D5] P notes that the User can move the Object without a Path, so P doesn't create any Paths.

7. [D6] P identifies the beaker and the flame-place as places where changes occur, and demonstrates the following Rules.
   - If the knob is in the LO place, set the graphic of the flame to small-flame. Per guideline [D6a], P demonstrates the conditions for this Rule by highlighting the names of the knob, OFF place, and flame. The place outlines of the places in which the objects reside are automatically highlighted, and P does not change this.

   - If the knob is in the HIGH place, set the graphic of the flame to large-flame. Highlighting is similar to the previous Rule [D6a].

   - If the knob is in the OFF place, set the graphic of the flame to invisible. Highlighting is similar to the previous Rule [D6a].

   - If the flame has no graphic, set the graphic of the water to ice. Per guideline [D6a], P demonstrates the conditions for this Rule by highlighting the names of the flame and ice, and the graphic of the flame. The place outlines of the places in which the objects reside are automatically highlighted, and P does not change this.

- If the flame has the small-flame graphic, set the graphic of the water to water. Highlighting is similar to the previous Rule [D6a].
- If the flame has the large-flame graphic, set the graphic of the water to steam. Highlighting is similar to the previous Rule [D6a].

8. [D7] There are no Paths, so no Rules are needed for placing Objects on Paths. There is no instantaneous movement.
9. [D8] The user interaction has already been specified by the Rules.
10. [D9] P tries the simulation. It works.
11. [D10] P draws a hose from the control to the burner and a ring under the beaker.

Table A-2c. ZeroTrains walkthrough of Bunsen Burner problem.

[Numbers in brackets refer to items of doctrine.]

[D1] sketch...

[D2] objects

control
flamelo, flamehi
ice, water, steam

[D3] places: beaker, burner mouth, control settings off, lo, hi.

[D4] paths: none

[D7,8] events

what	when
water turns to ice	water, no flame
water turns to steam	water, flamehi
steam turns to water	steam, no flame or flamelo
ice turns to water	ice, flamelo or flamehi
flamelo or hi disappears	flamelo or hi, control in off
flamelo appears	no flame, control in lo
flamelo becomes flamehi	flamelo, control in hi
flamehi appears	no flame, control in hi
flamehi becomes flamelo	flamehi, control in lo

[D9, 14, 15] To deal with O in P tests add objects off, lo, and hi in control places; control-off, control-lo, control-hi

Add events:

what	when
control to control-off	control, off
control to control-lo	control, lo
control to control-hi	control, hi
control-off to control	control-off, lo or hi
control-lo to control	control-lo, off or hi
control-hi to control	control-hi, off or lo

old events modified:

what	when
water turns to ice	water, no flame

water turns to steam	water, flamehi
steam turns to water	steam, no flame or flamelo
ice turns to water	ice, flamelo or flamehi
flamelo or hi disappears	flamelo or hi, control-off
flamelo appears	no flame, control-lo
flamelo becomes flamehi	flamelo, control-hi
flamehi appears	no flame, control-hi
flamehi becomes flamelo	flamehi, control-lo

[D9,13] make an object flameno to represent no flame

add events

flamelo becomes flameno	flamelo, control-off
flamehi becomes flameno	flamehi, control-off

rewrite tests for no flame

events are now:

what	when
control to control-off	control, off
control to control-lo	control, lo
control to control-hi	control, hi
control-off to control	control-off, lo or hi
control-lo to control	control-lo, off or hi
control-hi to control	control-hi, off or lo
water turns to ice	water, flameno
water turns to steam	water, flamehi
steam turns to water	steam, flameno or flamelo
ice turns to water	ice, flamelo or flamehi
flamelo becomes flameno	flamelo, control-off
flamehi becomes flameno	flamehi, control-off
flamelo appears	flameno, control-lo
flamelo becomes flamehi	flamelo, control-hi
flamehi appears	flameno, control-hi
flamehi becomes flamelo	flamehi, control-lo

[D17] Now need to consider cases where objects are not in same place. Need water objects in same place as flameobjs, and flame objs in same place as control objs. Add a place which encloses everything and add catalyst C to it. Add test for C to rules that involve both control-X objects and flame or water.

[D24] Now write reaction rules that implement all of the listed events.

References

Boy, G. (--) Is there a rationale for storing and retrieving design rationale? <*Human-Computer Interaction*, special issue on Design Rationale, submitted>

Brooks, R. (1977) Towards a theory of cognitive processes in computer programming. *International Journal of Man-Machine Studies*, **9**, 737-751.

Carroll, J. and Rosson, M.B. (1990) Human-Computer Interaction scenarios as a design representation. In *Proc. 23d Annual Hawaii International Conference on System Sciences, Software Track.* Los Angeles: IEEE Computer Society Press, 555-561.

Conklin, J. and Yakemovic, K. (--) A process-oriented approach to design rationale. <*Human-Computer Interaction*, special issue on Design Rationale, submitted >

Doane. S. and Lemke, A. (1990) Using cognitive simulation to develop user interface design principles. In *Proc. 23d Annual Hawaii International Conference on System Sciences, Software Track.* Los Angeles: IEEE Computer Society Press, 547-554.

Fischer, G., Lemke, A., McCall, R. and Morch, A. (--) Making argumentation serve design. <*Human-Computer Interaction*, special issue on Design Rationale, submitted >

Forbus, K. (1984) Qualitative process theory. *Artificial Intelligence*, **24**, 85-168.

Forgy, C.L. (1984) OPS5 User's Manual. Technical Report CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University, July, 1984.

Gould, J. (1988) How to design usable systems. In M. Helander (Ed.) *Handbook of Human-Computer Interaction.* New York: North Holland.

Green, T.R.G, Bellamy, R.K.E, and Parker, J.M. (1987) Parsing and gnisrap: A model of device use. In G. Olson, S. Sheppard, and E. Soloway (Eds.) *Empirical studies of programmers: Second workshop.* Norwood, NJ: Ablex, 132-146.

Lewis, C. (in press) Creating interactive graphics with spreadsheet machinery. In E.P. Glinert (Ed.) *Visual programming environments.* Los

Angeles, CA: IEEE Computer Society Press.

Lewis, C., Polson, P., Rieman, J. and Wharton, C. (1990) Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In Proceedings of CHI90 (Seattle, WA, April 1-5, 1990) New York: ACM, 235-242.

MacKinlay, J. (1986) Automating the design of graphical presentations of relational information. ACM Transactions on Graphics, **5**, 110-141.

MacLean, A, Young, R., Belotti, V. and Moran, T (--) Options, questions, and criteria: Elements of a design rationale for user interfaces. <*Human-Computer Interaction*, special issue on Design Rationale, submitted >

MacLennan, B. (1987) *Principles of Programming Languages.* New York: Holt, Rinehart and Winston.

Maulsby, D. and Witten, I. (1989) Inducing programs in a direct-manipulation environment. In *Proceedings of CHI89* (Austin, TX, April 30- May 4, 1989) New York: ACM, 57-62.

McCall, R. Personal communication, Boulder, CO, June 18, 1990.

McIntosh, P. Personal communication, Boulder, CO, June 19. 1990.

Newell, A. and Card, S. (1985) The prospect for psychological science in human-computer interaction. *Human-Computer Interaction*, **1**, 209-242.

Noyes, R. (1982) *The sun, our star.* Cambridge, MA: Harvard.

Rieman, J., Bell, B. and Lewis, C. (1990) ChemTrains Design Study Supplement. Technical Report CUCS 480-90, Department of Computer Science, University of Colorado, Boulder CO.

Shrager, J. (1984) Notes of the ARIA modeling system. Intelligent Systems Laboratory, Xerox Palo Alto Research Center, Palo Alto, CA.

Shrager, J. (1987) Issues in the pragmatics of qualitative modeling: lessons learned from a xerographics project. *Communications of the Association for Computing Machinery, 30* 1036-1047.

Sime, M.E., Green, T.R.G., and Guest, D.J. (1977) Scope marking in computer conditionals- A psychological evaluation. *International Journal of Man-Machine Studies*, **9**, 107-118.

Soloway, E. and Ehrlich, K. (1984) Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, **SWE-10**, 595-609.

Spohrer, J., Soloway, E. and Pope, E. (1985) A goal/plan analysis of buggy Pascal programs. *Human Computer Interaction*, **1**, 163-207.

Tandberg-Hanssen, E. and Emslie, A. (1988) *The physics of solar flares*. New York: Cambridge University Press.

Varley, C.G., Gradwell, G. and Hassell, M. (1973) Insect population ecology: an analytic approach. Berkeley: University of California Press.

Wilde, N. and Lewis, C. (1990) *Spreadsheet-based interactive graphics: From prototype to tool*. In Proceedings of CHI90 (Seattle, WA, April 1-5, 1990) New York: ACM, 153-159.

Young, R.M., Barnard, P., Simon, T. and Whittington, J. (1989) How would your favorite user model cope with these scenarios? *ACM SIGCHI Bulletin*, **20**, 51-55.

Zloof, M. (1975) Query by example. *AFIPS Conference Proceedings*, **44**, 431-432.